1. Pointers-On-C.pdf 5.8 question 12
   **How can you determine whether a right shift of a signed value is performed on your machine as an arithmetic or a logical shift?**

   a. To determine whether a right shift (>>) of a signed value on our machine is arithmetic or logical, we can use the following C program:

```c
//whether a right shift of a signed value is performed on
//this machine as an arithmetic or a logical shift
#include<stdio.h>

int main(){

    int num = -1;
    int shifted_num = (num >> 1);

    if( num == shifted_num){
        printf("\nArithmatic_shift\n");
    }
    else printf("Logical Shift\n");

    return 0;
}
```

   b. output:

```
Arithmatic_shift
PS C:\Users\Admin\Documents\bdcom_coding_zone\bitwise_operation>
```

2. **Computer-Systems-A-Programmers-Perspective.pdf Practice Problem 2.16**

Fill in the table below showing the effects of the different shift operations on single-byte quantities. The best way to think about shift operations is to work with binary representations. Convert the initial values to binary, perform the shifts, and then convert back to hexadecimal. Each of the answers should be 8 binary digits or 2 hexadecimal digits.

| a | | a << 2 | | Logical a >> 3 | | Arithmetic a >> 3 | |
|---|---|---|---|---|---|---|---|
| Hex | Binary | Binary | Hex | Binary | Hex | Binary | Hex |
| 0xD4 | | | | | | | |
| 0x64 | | | | | | | |
| 0x72 | | | | | | | |
| 0x44 | | | | | | | |

a.

| X | | a << 2 | | Logical( a>>3) | | Arithmetic(a>>3) | |
|---|---|---|---|---|---|---|---|
| Hex | Binary | Binary | Hex | Binary | Hex | Binary | Hex |
| 0xD4 | 11010100 | 01010000 | 0x50 | 00011010 | 0x1A | 11111010 | 0xFA |
| 0x64 | 01100100 | 10010000 | 0x90 | 00001100 | 0x0C | 00001100 | 0x0C |
| 0x72 | 01110010 | 11001000 | 0xC8 | 00001110 | 0x0E | 00001110 | 0x0E |
| 0x44 | 01000100 | 00010000 | 0x10 | 00001000 | 0x08 | 00001000 | 0x08 |

3. Computer-Systems-A-Programmers-Perspective.pdf Practice Problem 2.13
**Fill in the missing code below. Hint: Write C expressions for the operations bis and bic**

```
/* Declarations of functions implementing operations bis and bic */
int bis(int x, int m);
int bic(int x, int m);

/* Compute x|y using only calls to functions bis and bic */
int bool_or(int x, int y) {
    int result = _________;
    return result;
}

/* Compute x^y using only calls to functions bis and bic */
int bool_xor(int x, int y) {
    int result = _________;
    return result;
}
```

---

a. bis(x, m) equivalent to Boolean OR. A bit is set in z if either this bit is set in x or is set in m. bic(x, m) is equivalent to **(x & ~m)** that means a bit is set in z if either bit is set in x not in m or bit is set in m not in x,
**we can write x^y = (x & ~y) | (y & ~x),**

```
1   /* Declarations of functions implementing operations bis and bic */
2   int bis(int x, int m);
3   int bic(int x, int m);
4   /* Compute x|y using only calls to functions bis and bic */
5   int bool_or(int x, int y) {
6       int result = bis(x, y);
7       return result;
8   }
9   /* Compute x^y using only calls to functions bis and bic */
10  int bool_xor(int x, int y) {
11      int result = bis(bic(x, y), bic(x, y));
12      return result;
13  }
```

4. Pointers-On-C.pdf 5.9 question 4

Write a set of functions that implement an array of bits. The functions should have the following prototypes:

```
void set_bit( char bit_array[],
      unsigned bit_number );

void clear_bit( char bit_array[],
      unsigned bit_number );

void assign_bit( char bit_array[],
      unsigned bit_number, int value );

int test_bit( char bit_array[],
      unsigned bit_numlber );
```

a.

```c
#include <stdio.h>

#define BYTE_SIZE 8  // Number of bits in a byte

// Set a specific bit to 1
void set_bit(char bit_array[], unsigned bit_number) {
    unsigned byte_number = bit_number / BYTE_SIZE;
    unsigned bit_offset = bit_number % BYTE_SIZE;

    bit_array[byte_number] |= (1 << (bit_offset));
}

// Clear a specific bit (set to 0)
void clear_bit(char bit_array[], unsigned bit_number) {
    unsigned byte_number = bit_number / BYTE_SIZE;
    unsigned bit_offset = bit_number % BYTE_SIZE;

    bit_array[byte_number] &= ~(1 << (bit_offset));
}

// Assign a specific bit: 0 for clear, 1 for set
void assign_bit(char bit_array[], unsigned bit_number, int value) {
    if (value)
        set_bit(bit_array, bit_number);
    else
        clear_bit(bit_array, bit_number);
}

// Test a specific bit (returns 1 if set, 0 otherwise)
int test_bit(char bit_array[], unsigned bit_number) {
    unsigned byte_number = bit_number / BYTE_SIZE;
    unsigned bit_offset = bit_number % BYTE_SIZE;

    return (bit_array[byte_number] & (1 << bit_offset)) ? 1 : 0;
}
```

```c

// Function to print bits of the array (for debugging)
void print_bits(char bit_array[], int size) {
    for (int i = 0; i < size * BYTE_SIZE; i++) {
        printf("%d", test_bit(bit_array, i));
        if (i % BYTE_SIZE == BYTE_SIZE - 1) printf(" ");  // Space every byte
    }
    printf("\n");
}

// Test the functions
int main() {
    char bit_array[2] = {0};  // 16-bit array (2 bytes)

    set_bit(bit_array, 3);     // Set bit 3
     // Print bit array
     printf("Bit array after set_bit 3: ");
     print_bits(bit_array, 2);

    clear_bit(bit_array, 3);   // Clear bit 3

     // Print bit array
     printf("Bit array after clear bit 3: ");
     print_bits(bit_array, 2);

    assign_bit(bit_array, 7, 1);  // Assign bit 7 to 1

     // Print bit array
     printf("Bit array after assign_bit 7 for value 1: ");
     print_bits(bit_array, 2);

    // Check bit states
    printf("Bit 7: %d\n", test_bit(bit_array, 7));  // Should be 1
    printf("Bit 3: %d\n", test_bit(bit_array, 3));  // Should be 0

    return 0;
}

```

b. output:

```
Bit array after set_bit 3: 00010000 00000000
Bit array after clear bit 3: 00000000 00000000
Bit array after assign_bit 7 for value 1: 00000001 00000000
Bit 7: 1
Bit 3: 0
PS C:\Users\Admin\Documents\bdcom_coding_zone\bitwise_operation>
```

5. Pointers-On-C.pdf 5.9 question 5

. Write a function that will store a given value into specified bit positions of an integer. It should have this prototype:

```
int store_bit_field( int original_value,
      int value_to_stor,
      unsigned starting_bit, unsigned ending_bit );
```

Assume that the bits in an integer are numbered from right to left. Thus, the starting bit position may not be less than the ending bit position.

To illustrate, this function should return the following values:

| Original Value | Value to Store | Starting Bit | Ending Bit | Returned Value |
|---|---|---|---|---|
| 0x0 | 0x1 | 4 | 4 | 0x10 |
| 0xffff | 0x123 | 15 | 4 | 0x123f |
| 0xffff | 0x123 | 13 | 9 | 0xc7ff |

a.

```
2
3  // Function to store a value into specified bit positions of an integer
4  int store_bit_field(int original_value, int value_to_store, unsigned starting_bit, unsigned ending_bit) {
5      // Step 1: Construct the mask for the bit field
6      unsigned mask = ((1 << (starting_bit - ending_bit + 1)) - 1) << ending_bit;
7
8      // Step 2: Clear the specified bits in the original value
9      original_value &= ~mask;
10
11     // Step 3: Shift the new value left so that it is aligned in the field.
12     value_to_store = (value_to_store << ending_bit);
13
14     // Step 4: AND the shifted value with the mask to ensure that
15     //         it has no bits outside of the field.
16     value_to_store &= mask;
17
18     // Step 5: OR the resulting value into the original integer
19     return original_value | value_to_store;
20 }
21
22
```

```
22
23
24  // Test function
25  int main() {
26      // Test cases
27      int result1 = store_bit_field(0x0, 0x1, 4, 4);
28      int result2 = store_bit_field(0xFFFF, 0x123, 15, 4);
29      int result3 = store_bit_field(0xFFFF, 0x123, 13, 9);
30
31      // Print results
32      printf("Result 1: 0x%X (Expected: 0x10)\n", result1);
33      printf("Result 2: 0x%X (Expected: 0x123F)\n", result2);
34      printf("Result 3: 0x%X (Expected: 0xC7FF)\n", result3);
35
36      return 0;
37  }
38
```

output:

```
PS C:\Users\Admin\Documents\bdcom_coding_zone\bitwise_operation> cd "c:\Users\Ad
Result 1: 0x10 (Expected: 0x10)
Result 2: 0x123F (Expected: 0x123F)
Result 3: 0xC7FF (Expected: 0xC7FF)
```