CSC-2410 Computer Systems

Lab 3: Exploring Bitwise Operations and Bit Shifting in C

Introduction

Bitwise operations are fundamental in low-level programming, enabling efficient data manipulation and optimization. This lab exercise aims to develop your understanding of bitwise operations and bit shifting in the C programming language. You will implement various functions that perform these operations, enhancing your ability to manipulate data at the bit level—a crucial skill in systems programming.

Learning Objectives

By the end of this lab, you will be able to:

- **Understand and implement basic bitwise operations:** AND, OR, XOR, and NOT.
- **Utilize bitwise shifts:** Left shift (<<) and right shift (>>).
- **Apply bitwise operations for practical tasks:** Bit masking, setting, clearing, toggling bits, packing and unpacking data.
- **Implement advanced bit manipulation techniques:** Circular bit shifting.

Part 1: Bitwise Operations and Truth Tables

Before diving into the implementation, it's essential to understand the fundamental bitwise operations and their corresponding truth tables.

1. Bitwise AND (&)

- **Description:** Performs a binary AND operation, where each bit in the result is 1 only if the corresponding bits in both operands are 1.

  Truth Table:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

2. Bitwise OR (|)

- **Description:** Performs a binary OR operation, where each bit in the result is 1 if at least one of the corresponding bits in the operands is 1.

  Truth Table:

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

3. Bitwise XOR (^)

- **Description:** Performs a binary XOR (exclusive OR) operation, where each bit in the result is 1 only if the corresponding bits in the operands are different.

  Truth Table:

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

4. Bitwise NOT (~)

- **Description:** Performs a binary NOT operation, where each bit in the result is the inverse of the corresponding bit in the operand.

  Truth Table:

A	~A
0	1
1	0

Part 2: Bit Shifting

Bit shifting allows you to move bits to the left or right within a binary number, effectively multiplying or dividing the number by powers of two.

1. Left Shift (<<)

- **Description:** Shifts the bits of a number to the left by a specified number of positions. Each shift left effectively multiplies the number by 2.

2. Right Shift (>>)

- **Description:** Shifts the bits of a number to the right by a specified number of positions. For unsigned integers, each shift right effectively divides the number by 2.

Part 3: Practical Applications

In this section, you will apply the bitwise operations and shifts to solve practical problems such as bit masking, setting, clearing, toggling bits, packing and unpacking data, and implementing circular bit shifts.

Lab Instructions

You are provided with a C source file, `bitwise_lab.c`, containing function definitions with comments explaining what each function should accomplish. Your task is to implement these functions according to the provided specifications.

1. **Implement the Functions:** Complete each function in `bitwise_lab.c` based on the comments provided within the function bodies.
2. **Test Your Code:** Use the `main` function to test your implementations with various input values. Ensure that all edge cases are handled correctly.
3. **Submit Your Work:** Once all functions are implemented and tested, submit your completed `bitwise_lab.c` file along with a brief report detailing your approach and any challenges faced during implementation.

Starter Code (`bitwise_lab.c`)

Below is the starter code with function definitions. Each function contains comments describing its intended behavior. Implement each function accordingly.

```c
#include <stdio.h>

// Function Definitions

// Performs the bitwise AND operation between two integers.
// Returns the result of a & b.
int andOperation(int a, int b) {
    // TODO: Implement the AND operation
    return 0;
}

// Performs the bitwise OR operation between two integers.
// Returns the result of a | b.
int orOperation(int a, int b) {
    // TODO: Implement the OR operation
    return 0;
}

// Performs the bitwise XOR operation between two integers.
// Returns the result of a ^ b.
int xorOperation(int a, int b) {
    // TODO: Implement the XOR operation
```

```
    return 0;
}

// Performs the bitwise NOT operation on an integer.
// Returns the result of ~a.
int notOperation(int a) {
    // TODO: Implement the NOT operation
    return 0;
}

// Shifts the bits of 'num' to the left by 'shift' positions.
// Returns the result of num << shift.
int leftShift(int num, int shift) {
    // TODO: Implement the left shift operation
    return 0;
}

// Shifts the bits of 'num' to the right by 'shift' positions.
// Returns the result of num >> shift.
int rightShift(int num, int shift) {
    // TODO: Implement the right shift operation
    return 0;
}

// Sets the bit at 'position' in 'num' to 1.
// Returns the modified number.
int setBit(int num, int position) {
    // TODO: Implement the set bit operation
    return 0;
}

// Clears the bit at 'position' in 'num' (sets it to 0).
// Returns the modified number.
int clearBit(int num, int position) {
    // TODO: Implement the clear bit operation
    return 0;
}

// Toggles the bit at 'position' in 'num'.
// Returns the modified number.
int toggleBit(int num, int position) {
    // TODO: Implement the toggle bit operation
    return 0;
}

// Packs two 4-bit numbers 'a' and 'b' into a single 8-bit number.
// 'a' occupies the higher 4 bits, and 'b' occupies the lower 4 bits.
// Returns the packed 8-bit number.
int packBits(int a, int b) {
    // TODO: Implement the pack bits operation
    return 0;
```

```c
}

// Unpacks an 8-bit number 'packed' into two 4-bit numbers.
// Stores the higher 4 bits in '*a' and the lower 4 bits in '*b'.
void unpackBits(int packed, int *a, int *b) {
    // TODO: Implement the unpack bits operation
}

// Performs a circular left shift on 'num' by 'shift' positions.
// Returns the result of the circular left shift.
int circularLeftShift(int num, int shift) {
    // TODO: Implement the circular left shift operation
    return 0;
}

// Performs a circular right shift on 'num' by 'shift' positions.
// Returns the result of the circular right shift.
int circularRightShift(int num, int shift) {
    // TODO: Implement the circular right shift operation
    return 0;
}

int main() {
    int a = 12; // Example value
    int b = 10; // Example value

    // Perform and display results for each bitwise operation
    printf("AND Operation: %d\n", andOperation(a, b));
    printf("OR Operation: %d\n", orOperation(a, b));
    printf("XOR Operation: %d\n", xorOperation(a, b));
    printf("NOT Operation: %d\n", notOperation(a));

    int shift = 2;
    printf("Left Shift: %d\n", leftShift(a, shift));
    printf("Right Shift: %d\n", rightShift(a, shift));

    // Bit manipulation examples
    int position = 2;
    printf("Set Bit: %d\n", setBit(a, position));
    printf("Clear Bit: %d\n", clearBit(a, position));
    printf("Toggle Bit: %d\n", toggleBit(a, position));

    // Packing and unpacking
    int packed = packBits(a, b);
    printf("Packed Bits: %d\n", packed);
    int unpackedA, unpackedB;
    unpackBits(packed, &unpackedA, &unpackedB);
    printf("Unpacked Bits A: %d\n", unpackedA);
    printf("Unpacked Bits B: %d\n", unpackedB);

    // Circular shifts
```

```
    printf("Circular Left Shift: %d\n", circularLeftShift(a, shift));
    printf("Circular Right Shift: %d\n", circularRightShift(a, shift));

    return 0;
}
```

Lab Questions:

1. What is the difference between the bitwise AND (`&`) and the logical AND (`&&`) operators in C?
2. If you perform the bitwise OR operation (`|`) on the numbers 5 (`0101` in binary) and 3 (`0011` in binary), what is the result? Explain how you arrived at the answer.
3. What does the bitwise NOT (~) operation do to the binary number `00001111`?
4. What would be the result of performing an XOR (^) operation on any number with itself, like `a ^ a`? Why?
5. If you have an 8-bit number `11010010`, how can you clear the third bit (counting from the right, zero-indexed) using a bitwise operation?
6. Why is the bitwise OR operation useful when you want to set specific bits in a number without changing the others?
7. Can you explain in your own words what a “bit mask” is and how it can be used in combination with bitwise operations?

Submission Guidelines

- **Code Implementation:** Ensure all functions in `bitwise_lab.c` are correctly implemented as per the function descriptions.
- **Testing:** Thoroughly test your implementations using the `main` function with various input values, including edge cases.
- **Report:** Prepare a brief report (1-2 pages) explaining:
    - Your approach to implementing each function.
    - Any challenges or obstacles you encountered.
    - Insights or interesting findings during the lab.

Submit both the completed `bitwise_lab.c` file and your report on Blackboard.