2025 Introduction to Computer Software Systems Lab Assignment #1

Doyoung Kim

September 8, 2025

This assignment focuses on understanding the integer datatypes of the C programming language and utilizing bitwise operations. To achieve this, it requires the implementation of the following five functions:

`bitNor` Implement a bitwise NOR operator using only the bitwise NOT and AND operators.

`isZero` Determine if a given argument is zero, using a maximum of two operators.

`addOK` Verify if the addition of two integers incurs an overflow.

`absVal` Compute the absolute value of an integer using a restricted set of bitwise and arithmetic operators.

`logicalShift` Implement a logical right shift operation without using type casting to an unsigned integer.

This report will discuss the implementation details and the underlying principles of two's complement integer arithmetic as it applies to computer systems and the C programming language. The semantics of individual bitwise operators will not be detailed, as they are covered comprehensively in the course lecture notes and textbook.

1 Implementation

1.1 Function `bitNor`

This task requires the implementation of a bitwise NOR operator. This is accomplished by applying De Morgan's laws, which state that $\neg(P \vee Q) \iff (\neg P \wedge \neg Q)$. The bitwise equivalent is `~(x | y) == ~x & ~y`.

```
int bitNor(int x, int y) { return ~x & ~y; }
```

1.2 Function `isZero`

This function must check if an integer is zero, returning `1` if true and `0` otherwise. The logical NOT operator (`!`) in C performs this exact operation, yielding the most direct implementation.

```
int isZero(int x) { return !x; }
```

1.3 Function `addOK`

In this task, an integer addition must be checked for overflow using only bitwise and a limited set of other operations. The key observation is that an addition between two integers with different signs will not result in an overflow. If the two operands share the same sign, an overflow occurs if and only if the sign of the sum differs from the sign of the operands.

This implementation extracts the sign bit of the two operands and their sum. It first checks if the operand signs differ. If they are identical, it then compares the sign of the sum to the sign of the operands to detect an overflow.

```
int addOK(int x, int y) {
    int x_sign = x >> 31 & 1;
    int y_sign = y >> 31 & 1;
    int sum_sign = (x + y) >> 31 & 1;

    return (x_sign ^ y_sign) | !(sum_sign ^ x_sign);
}
```

1.4 Function `absVal`

This function computes the absolute value of an integer. The strategy leverages the behavior of the arithmetic right shift on signed integers. Shifting a signed integer by 31 bits generates a mask of all ones (`0xFFFFFFFF`) for a negative number and all zeros (`0x00000000`) for a non-negative number.

This mask can then be used to select between two values: the original number `x` (if non-negative) or its two's complement negation `~x + 1` (if negative).

```
int absVal(int x) {
    int mask = x >> 31;
    return (mask & (~x + 1)) | (~mask & x);
}
```

1.5 Function `logicalShift`

The right shift operator in C performs an arithmetic shift on signed datatypes. The objective here is to implement a logical right shift. The strategy is to perform an arithmetic shift and then clear the most significant `n` bits, which may have been filled with ones if the operand was negative.

To achieve this, a mask for the upper `n` bits is generated using the expression `(1 << 31) >> (n - 1)`. A notable edge case is `n = 0`, where a right shift by -1 would occur, invoking undefined behavior. To address this, a secondary mask is created to bypass the operation entirely when `n` is zero, returning the original value `x`.

```
int logicalShift(int x, int n) {
    int mask = (1 << 31) >> (n + ~0);
    int is_zero = !!n << 31 >> 31;
    return (~is_zero & x) | (is_zero & ((x >> n) & ~mask));
}
```

2 Result and Discussion

The execution of the `driver.pl` script confirms that all implemented functions pass the required correctness and performance tests.

```
[d0319@programming2 datalab]$ ./driver.pl
...
Score = 22/22 [12/12 Corr + 10/10 Perf] (37 total operators)
```

Although the implementations are correct and satisfy all constraints, they possess a significant flaw: the code relies heavily on the assumption that the `int` datatype is 32 bits wide. This is evident in the use of hardcoded constants like ‘31‘ for shift operations.

The C standard only guarantees that an `int` is a signed integer with a size of at least 16 bits; it does not mandate a 32-bit width. This dependency compromises the portability of the code, as it would produce incorrect results on systems where `sizeof(int)` is not 4.

This limitation is fundamental under the assignment's constraints, which forbid the use of the `sizeof` operator or other mechanisms for determining the bit-width of the datatype at runtime.

3 Conclusion

This assignment was successfully completed, with all functions passing the required correctness and performance benchmarks. The exercise provided practical experience in low-level data manipulation, reinforcing key concepts such as two's complement arithmetic, bitwise operations, and the behavior of signed integers in C.

A critical insight from this lab is the distinction between a correct implementation and a portable one. The solutions, while functional within the assumed 32-bit environment, are inherently non-portable due to hardcoded bit-width dependencies. This underscores a fundamental challenge in systems programming: creating robust software that is abstracted from the specifics of the underlying hardware.

Ultimately, this assignment demonstrates not only how to manipulate data at the bit-level but also illuminates the critical importance of platform assumptions in software development. The strict constraints highlight the trade-offs that engineers face between performance, elegance, and portability.