Computer Systems 2022
Lab Assignment 1: Bitwise operations
Due: Wednesday, September 14, 23:59.

Student 1, Student 2

1 Introduction

The goal of this lab is threefold. First, we aim to demonstrate the versatility of bit vectors as a mean for data representation. Second, we show the importance of different types of data encoding (e.g., integers, unsigned integers, floating point numbers), and how they can impact the way we read and write code. Third, and final, we aim show how basic computational operations can be replaced with bit-wise operators. Although cumbersome, this transformation can be very important for the performance of applications.

This report aims to describe the process of solving these puzzles and to reflect on data representation. Specifically, we describe the general challenges this assignment has posed and the generic approach/process taken towards solving the puzzles. We further dive into the details of the puzzles themselves. Finally, we summarize our findings and lessons learned.

2 Background

In this section we discuss the background knowledge necessary to solve the puzzles. Specifically, data types, data encoding, and operators are briefly described.

Data encoding and data types

Q1 (0.75p): What are the integer and floating point representations you used in this assignment? What is their correlation with the data types you have used? What are the main differences between them?

Integers are represented using 32-bit 2s complement
Floating points are represented the IEEE-754 floating-point standard
The datatypes int and float have been used, both of which are also 32 bits long, but float has a much larger range than int. This is because it uses exponents to approximate a number, instead of representing the exact number

Operators

Q2 (0.75p): What were the operators you have used for solving the puzzles and what is their behavior? Are they useful for anything else than these puzzles?
$\sim$: one's compliment, turns every 0 to 1 and every 1 to 0 in a row of bits.
|: bitwise or, 1-0, 0-1 and 1-1 in the same position in two rows of bits become 1s
!: not operator, 0 becomes 1, non-zero becomes 0
<<: left shift, shift the ones and zeroes in a row of bits to the left a certain number of times. Extra zeroes get added on the right and all ones and zeroes that leave the row of bits on the left (since the row of bits doesn't become any longer) disappear.
>>: right shift, works like left shift except for the fact that the ones and zeroes move to the right and instead of zeroes it adds whatever value the MSB (Most Significant Bit) has on the right.
+: it adds two rows of bits
^: xor, it works like or, but 1 and 1 gives 0 instead of 1.
&: bitwise and, when you have two rows of bits, 1 and 1 becomes 1, but 0 and 0/1 becomes 0.
-: it subtracts two rows of bits.
<: it evaluates whether one number is less then the other, if the lesser number is on the left, it returns true.
>: same as the one above, except the lesser number should be on the right.
>=: same as the one above, except also succeeds if two numbers are equal.
All these operators are very useful outside the puzzles for optimizing programs and calculations, they are the major building blocks of almost all code that is written .

3 General approach

In this section we describe the general strategy we have devised to approach the puzzles, and the main challenges we have identified early on.

Q3 (1.0p): Please describe the general approach you took/wanted to take when solving all the puzzles. Were there any immediate challenges emerging from this strategy?

To begin solving the puzzles we first tried to find a solution on paper using examples of 8 bits. We would think of a possible approach or possible intermediate situations we wanted to reach. We would write out different cases using different bits, for example representing 0, -128, 127 and some other random numbers, and think of the different outcomes these had. Then we tried to achieve the solution by trying different combinations of operators. After it succeeded on paper, we would translate our method to code in C.

4 Example solutions

We present here a detailed analysis of two of our solutions by means of two examples - an integer puzzle and a floating point puzzle.

Integer Puzzles

Q4 (1.5p): Explain your solutions for two integer puzzles you have solved. Please choose the two solutions that used the most operators from the ones you have solved. Please describe your approach and the challenges you encountered in solving these puzzles. You may use pseudo-code or code snippets, pictures, examples, etc.

isGreater(int x, int y):
We first thought for what it meant for x to be greater than y. If x is greater than y, y-x is < 0. We could then simply return the sign of the result. We also figured out a way to write the - operator, which was redundant since it was covered in the lecture. After implementing our approach in C, we found out it didn't yet work for certain cases involving 0 and combinations of positive/negative numbers, partly due to overflow. To fix this, we had to find a way to check for the signs of x and y. x and y were both positive or both negative, y-x 0 ¡ 0 worked. If y is negative and x is not, x will always be larger than y. If y is positive and x is not, x will always be smaller than x. If x is zero and y is negative, it returns 0. Implementing this gave:
$return(greater\&same\_sign)|(sign\_y\&!same\_sign)$

bitCount(int x, int y):
We originally had the idea to simply shift each 1 from right to left to the position of the LSB and add them up that way. That would mean cycling through the row of bits one by one and even though it would work, it was very inefficient and impossible to do using the allowed number of operators. So based on our original idea, we came up with an alternative strategy. We decided to shift all the 1s in our row of bits to the desired position in a number of steps which would allow us to shift multiple bits to the correct position at once. The idea would work like this in practice:
10010111, first right shift every bit in position 1, 3, 5, etc. once→
01010110, then right shift every bit in position 1, 2 and 4, 5 twice→
00100011, then right shift every bit in position 1-4 four times→
00000101, this would give you the number of 1s. This also works on a row with 32 bits, except you need more steps.
To do this we needed to add up, the bits in the positions that were shifted and the bits in the positions the bits were shifted to. To add these up and only shift part of the bits, we needed certain hexadecimal numbers to help us shift the right bits. 0x55555555 was needed for the first step for example, to not move the numbers in the even positions. After finding all the needed hexadecimal numbers and confirming our program worked, we found out that we were only allowed to use 0x00-0xFF. This meant that we had to create these numbers manually, we figured out how to do this using two left shifts and adding the binary rows, for example:
0x55 + (0x55 $<<$8) = 0x5555
0x5555 + (0x5555 $<<$16) = 0x55555555
With this we had everything we needed and the puzzle was finished!

Note: In case none of the puzzles you have attempted to solve succeeded, please document here what have you tried, how far you've gotten, and what errors you encountered.

Floating-point Puzzle

Q5 (1.5p): Explain your solution for the floating-point puzzle you have solved. Please describe your approach and the challenges you encountered in solving it. Focus on the differences between this puzzle and the integer ones. You may use pseudo-code or code snippets, pictures, examples, etc.

After going over in class how to convert floating points in a tiny float to bits, we looked up the IEEE-754 floating-point standard which is used in C. We then implemented the same method in C, only using a different amount of bits for the exponent and fracture. Using shifting and the operator we acquired the exponent, and we took the last 23 bits as the fracture. After implementing the general method, we had to add several if statements to check for edge cases in the IEEE method, such as infinity, nan and numbers that were out of bounds. A difference with this puzzle and the integer ones, is that the starting number wasn't signed and thus the sign didn't get handled automatically. Since using large numbers such as 0xfffffffff was allowed, fewer shift operators had to be used. There were also more cases in which different operations were needed. Hence the need in our code for if statement, for example when deciding when to shift the num left or right.

Note: In case none of the puzzles you have attempted to solve succeeded, please document here what have you tried, how far you've gotten, and what errors you encountered.

5 Lessons Learned

We present here the main lessons we have learned from this lab. We focus mostly on the content-wise elements.

Q6 (1p): What are your main findings, overall, about converting regular operations into bit-wise operations?

Some regular operations are surprisingly complicated when converted to bit-wise operations. For some implementations a low of bit-wise operations are needed

Q7(0.5p): What was the role of limiting the number of operators and the size of the variables? How did this influence your approach and/or what is the impact of such a restriction on your solution?
It forced us to really come up with computationally efficient solutions and think creatively, since you can't just brute-force a solution.

6 Conclusion and Future Work

Q8 (1.0p): What have you learned during this lab? Focus on your own knowledge and experience, do not go for "standard" statements.
I'm now much more conscious about the binary workings of a computer and it's really added another level of understanding about the workings of operators and code in general. I've also learned to work with bitwise operators and think in a way that allows me to come up with solutions based on binary. 0 One of the reasons for which we use bit-wise operators and integer numbers is to increase performance.

Q8 (0.5p): What do you think are the advantages and disadvantages of using these low-level operations in real applications?

The advantage is that they are about as precise as you get, so you can use them to really optimize a program as far as possible. The downside is that they can also break programs, think about the Ariane 5 rocket failure. They are also low-level operations, so it is very time-consuming and complicated to write a big program using only these operations. So they are good for performance, but bad for the time it takes to code using them.

Q10 (0.75p): Do you think your solutions will have better or worse performance than the ones using traditional operators? Argument your choice.

We think that most of our solutions will have slightly worse performance than the ones using traditional operators. Traditional operators probably work using bitwise operators in the most optimal way and while we are confident in our solutions, we are not sure they are the most optimal solutions possible. There might be ways to use less operators and maybe use less storage

Q11 (0.75p): What would you recommend to be done next to confirm/infirm your Q10 estimate? Are the findings generalizable in any way?

Test the solutions against traditional operators using a bunch of different test cases. For example, space and time complexity could be compared to see which solution is optimal. These tests are generalizable since they (should) work the same way for every number that is used.