Charles Misback, Declan Casey, and Nathan Loria

12/1/20

Dr. Mohan

Team 5 Project: Final Report

Introduction

Our group has developed a data convertor and calculator. The converter is able to convert to and from binary notation, decimal notation, and hexadecimal notation, and the calculator has been developed using only low-level hardware solutions. As a late addition, we also implemented a bitwise calculator that utilizes recursion and bitwise operators as a demonstration of higher level logic to simulate the arithmetic operators. We developed these specific programs because we wanted to examine and learn how exactly data is expressed at the hardware level, as well as how this data is communicated between a human and a computer. Understanding these concepts will help us become better C developers, and gain a better grasp efficient programming.

We have also implemented a CLI that allows the users to work with the operators and converters as they please. Throughout our development and research we have continued to learn that seemingly simple operations, such as addition, subtraction, multiplication and division, are actually high level operators, as well as how these high-level operators can be used to implement much larger mathematical projects and functions.

Data Converter

Working on the data converter posed several challenges as well as providing many learning experiences. Although the six total data conversions are kept in the converter.c file, two

of the conversion functions, decimal-to-binary and binary-to-decimal are also kept in the calculator.c file. This is because a lot of the code was originally written in calculator.c, and also because the two conversions are still used for many of the basic operations in the calculator. One goal of our project is to show how essential these hardware-level operators are to all operations in low-level programming. Creating several functions and classes that are all built upon the same operations does just this. The converter also uses the getBits() function, which is imported calculator, which helps us in finding several conversions, such as the decimal to binary conversion.

The getBits() function is very straightforward. It reads in a number in decimal notation of type int and stores that in a temporary variable, while another variable “bits” is declared with an initial value of zero. We used a while loop to keep dividing the temporary variable by 2 and then to increment the bits variable to represent how many bits the current temp variable contains. This bits variable is then returned, making ready the total number of bits to be used by other functions such as the addition and subtraction.

Each of the notations has their own unique way of being converted, so there must be a unique implementation for each type of conversion; this makes reusing and recycling code a bit difficult. What we have to do instead is reverse-engineer the methods that correspond to each other, for example the hexToDec and decToHex. One conversion in particular was a bit more difficult and tedious than others to code, that being the hexadecimal-to-binary function (which is the longest conversion in terms of lines of code). Since letters in hexadecimal notation directly correspond to a binary number, it was easiest to hard-code these individual digit/number conversions. This was done using a while loop and switch statement, where each case corresponds to the binary equivalent of a hex digit.

One issue that has occurred with this switch block, however, was the need to add conditional logic in each case so that there were not any trailing zeros for the binary numbers that contained them, these numbers being 0 through 7. For example, the decimal notation for 7 is 0111, which in a much longer hexadecimal digit would be correct, as long as it's not the first bit. If it were the first, then we would only use 111 followed by whatever other digits are left. This bug in the code was caught early on, but allowed us to make our code more robust.

Outside of this function, they were a bit more straightforward as far as implementation goes, as we were able to use much less lines of code using conditional logic, rather than hard-coding each individual conversion. The conditional logic can still be a bit tricky to wrap one's head around, so we'll discuss a bit more in depth what ours accomplishes.

Converting from to decimal to binary and from binary to decimal were very similar, as you have to reverse engineer the two functions given that the respective formula to find each is just the other one backwards. For binary to decimal, you have to start with the most significant bit from left to right and repeatedly multiply by 2 and add to each bit per iteration. We accomplished this conversion using a for loop and a nested if statement, where the for loop finds the binary digit to be added and multiplied, and the if statement does the multiplication and addition. This function uses the pow() function imported from math and several other operators. This is important to note because this part of the project does not deal with using low-level operations to build high-level operations, only with data conversion.

For the decimal to binary conversion, the general formula, if doing it by hand, is to repeatedly divide the given decimal input by 2 and to keep the remainders, which will always be either one or zero. You continue to do this until the quotient equals 0. To accomplish this, we used a do-while statement with a nested if statement. The do-while statement continues to loop

until the quotient is zero, and the if statement handles the rest of the division and remainder retention. Again, one can see how these processes are reversed.

Mathematical Operators

As stated in our initial project proposal, our group implemented algorithms for addition, subtraction, multiplication, and division. These algorithms are intended to mirror the same methods that computers use at a mechanical level in order to perform such operations. In this project, we developed working functions for addition, subtraction, multiplication, and division.

The first method that we developed was addition. The reason that this was the first method developed was because in order to develop a method for multiplication (repeated addition), an addition operation is needed. Another reason why we implemented this method first was because it was the easiest to visualize what needed to be done and allowed us to establish a framework for what we were going to do moving forward in the projects, as well as establish a strong base understanding of how we are storing data (malloc arrays). The addition method works by using logical operators in C and combining two binary values. The method takes in binary arrays and returns an integer. The method contains a carry in value that allows for a true addition implementation. As with computers, when the bits being added are 1 & 1, the output is 0 with a carry; 1 & 0, 1; 1 & 1 & 1, 1 with a carry 1. The values of each addition are then stored in a result array which is converted to a decimal using our binaryToDecimal() method and returned.

The second method that we developed was multiplication. The reason for this, as previously stated, is that multiplication requires addition. This method takes in binary arrays and returns an integer value just as all of the other methods. The method works by iterating through a series of actions 8 times. The reason for this is because as of right now, the result is only

designed to hold 8 bits (a max integer value of 255). In the future, we plan to increase this number to allow for any size of result that is allowed within a long long value in C. Each time the multiplication algorithm repeats, there is a sequence of steps that are executed. These steps are as follows; (1) check if *Multiplier[0] == 0*. If this condition is true, then (2) the multiplicand register is left shifted and the multiplier register is shifted right. If this condition is not true (Multiplier[0] == 1), then the same step 2 happens from condition one, but after the multiplicand is added to the product and the result is placed in the project register. As of right now, our algorithm produces a correct output for values up to 255.

The next algorithm that we have implemented up to this point is the subtraction algorithm. This algorithm takes in two binary arrays as inputs and returns an integer value as a result. Much like in the addition algorithm, there is a carry in value and the same rules are held true. Unlike the addition method, however, instead of both operands being positive, one binary value is negated using 2's complement. With 2's complement, 1's and 0's are reversed and a 1 is added to the binary value. After this, the result of the addition (num1 + -num2) without the final carryout value is the result of the subtraction. The method also supports negative numbers and returns the correct answer for any set of inputs.

The final algorithm that was implemented by the team was for division. The division algorithm works just like the algorithm that was discussed in the class slides. Initially, dynamic arrays named A, Q, and M are created and initialized. Q and M are initialized with their respective inputs (arg0 & arg1), and A is initialized with all zeros. After this, the number of bits in Q is used within a for loop to ensure the algorithm is run the proper amount of times. For each bit in Q (N) the algorithm performs a left shift on A and Q. After this, A is set equal to A - M and the most significant bit is evaluated. If this bit is high (1) then that means that A < 0. When this

is true, Q[0] is set to 0 and M is added back to A in order to reset it's value. If this is false, then Q[0] is set to 1 and A remains as it is. After all of the iterations, the algorithm produces the result (stored in Q) and the remainder (stored in A).

Parser

At first glance, the parser seems easy enough to implement: display a menu to the user, get their input with the function scanf(), and call the according functions written to either convert or calculate then display the result. When one actually goes to program this there is much more that you have to keep in mind. The idea for the actual implementation started with determining what we needed our CLI to do.

First, the user should be prompted to either go to the converter or the calculator, any incorrect input from the user will be discarded and they will be asked again. This was probably the most simple part. Next we had to determine the syntax of both the converter and the calculator. For the converter we went with a simple " 'base' 'num' 'base'. The first 'base' described the number system that the user's input 'num' was (bin/dec/hex). The second 'base' is the number system that the user wants their 'num' to be converted to. Assuming the user always has perfect inputs, we would've been done. Sadly, this is not the case. Any invalid bases or numbers entered should result in an error and ask the user for their input again. Invalid numbers would be anything other than 0 or 1 for binary, 0 - 9 for decimal, and 0 - 9 as well as a - f for hexadecimal.

The calculator had similar obstacles similar to this to get over. The syntax for it was 'base' num' 'operator' 'base' 'num'. The 'base' in front of each of the 'num's is exactly the same as how the 'num' works in the converter and the operator is one of the 4 arithmetic

operators: +, - , *, and /. With this unique syntax, the user can enter their numbers in any number system that they want (bin/dec/hex) and add, subtract, multiply, and divide them. Because the calculator allows any number system, the converter has to come into play. Instead of doing the conversion inside of the calculator, it's done in the CLI to make it more straightforward when we implement the arithmetic functions from scratch.

Another complicated implementation was the bitwise operators that would be used for the bitwise. Recursion seemed to be a very clean way to demonstrate the algorithm is not very complicated but very straightforward. One can see that the difference between addWise and subWise is just one difference, the one's complement of the minuend is AND'd with the subtrahend instead of just the normal minuend for addition. This shows that subtraction is just the addition of the complement. The point of this was to emphasize that a calculator to converter/calculator does not use any arithmetic operators in it and demonstrate that everything that a calculator or converter does is possible just using logic gates.

The biggest difficulty that was experienced in making the CLI was making sure that the user's number that they entered matched their desired base and making sure that the user did not enter invalid input. Another issue that we encountered was the two numbers input by the user had to be converted into a binary integer array. The method of simulating the arithmetic operators with logical operators was entirely through manipulating and utilizing right shift and left shift type methods which is much easier on arrays as opposed to integers. The method that we used to get around this was using the converter to first make sure that input number was converted into a decimal number, and then converted again to a binary array. The method that one uses for converting from decimal to binary works perfectly with output into an array.

Conclusion

Overall we believe that this project challenged each of us in unique and different ways while implementing our separate programs. As it turned out, we mostly worked on our parts independently for the first week or so; when we were forced to bring the three main components together closer to the final due date there was a lot of communication and teamwork to assure each other's parts worked together like a well oiled machine. Charles said his error checking and user interface was irritating to program at times but rewarding to see input correctly identified as either valid or invalid. Nate had trouble implementing the mathematical operators at certain points because of the number of bits each value stored, but he was able to overcome this and store values that were within the scope of the algorithms that were learned in class. Declan said that he was frustrated at certain points working with hexadecimal but after time was put in he was finally able to develop a working solution. In the end, this project was very rewarding and allowed all of us to deepen our understanding of computers and how they operate.