Charle Misback, Declan Casey, and Nathan Loria

12/1/20

Dr. Mohan

CS 200: Team 5 Project Report

Introduction

As previously mentioned in our project proposal, our group has been developing 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. 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.

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 has posed several challenges as well as provide many learning experiences. Although the six total data conversions as well as the getBits() function 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 and getBits() function 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.

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. 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 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.

Mathematical Operators

As stated in our initial project proposal, our group is planning to implement algorithms for addition, subtraction, multiplication, and division. These algorithms are intended to mirror the same method that computers use at a mechanical level in order to perform such operations. So far in the implementation, we have been able to develop working functions for add, subtract, and multiply. Moving forward, we plan to develop the division algorithm before the submission deadline.

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 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 integer values 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 final 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.

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 implemented the arithmetic functions from scratch.

Another complicated implementation was the bitwise operators that would simulate the arithmetic operators for the converter. The first part of this is complete but they still need to be implemented into the converter. The point of this is so we can say that our 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.