Lab 12
Final Project Report

Physics 111A
Nicholas Kotsianas
Lab Partner: Meiji Nguyen

Due: August 13, 2016
Last compiled: August 13, 2016

Abstract

For our final project, we built a 4 bit (signed) adder with a LabVIEW interface. This project attempts to emulate a modern processor by accepting an arbitrarily long (syntactically correct) input of additions, subtractions, and parentheses, then, using LabVIEW, parses and orders the input and sends each individual computation to the calculator, which then sends the result back to the VI. The process repeats until there are no more operations to be done, yielding the final answer. The calculator produces results admirably, provided that the input and outputs are kept between -8 and $+7$, with an estimated individual computation time of a rather high 10 ms.

Contents

Introduction

The initial idea for the project was to build a simple 4 bit adder/subtracter entirely out of BJT NAND gates. However, this turns out to be exceedingly tedious, and not demonstrative of as many principles we learned in the class as a project should. Thus, a single NAND gate was constructed to demonstrate the principle, and the circuit was built with integrated logic chips, and a LabVIEW portion was added. The motivation for the LabVIEW section was to attempt to more accurately represent how full-adders and similar simple circuits would be used in, say, a modern processor. That is, to break down a complex calculation into a series of easier computations, then returning that result.

Some trade-offs between what should be done with software and what should be done in the circuit had to be made. For example, an original idea included a 4 bit full adder which could also perform subtraction with an additional $\pm$ bit and a two's complement calculator for the second input (see Figure 1). However, this produced problems with parsing the user inputs, since $A + B$ and $A - B$ are well handled, but inputs of the form $-A + B$ and $-A - B$ need some more complicated hardware, an additional bit for A's $\pm$, and/or more pre-processing of the inputs in the software. This solution was considered less favorable since LabVIEW's conversion from decimal to binary (to send the inputs to the calculator) automatically converts negative numbers to the two's complement, therefore only requiring addition to be done on the hardware, which does not care about the representation of the inputs.

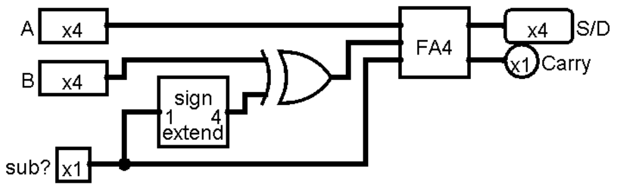


Figure 1: A 4 bit adder/subtracter, able to compute $A \pm B$, with subtraction indicated by flipping the "`sub?`" bit to 1. "`FA4`" is a 4 bit full adder. (Not implemented in final design.)

Thus, the project is in two parts: the 4 bit (signed) adder, and the LabVIEW software. The LabVIEW software interprets a user input, such as the string "$3 + (5 - 1) + (-8 - (-4 - 2))$", re-orders the operations into Reverse Polish Notation, sends each computation to the calculator, then reads back the result into LabVIEW, and loops until there are no more operations to be completed. A more in-depth discussion of the LabVIEW VIs is given later in the report, along with representative screenshots of the block diagrams.

Circuit Description

High-Level

The circuit, a 4 bit ripple-carry adder, is shown in Figure 2. The high-level logic of this circuit is that it consists of 4 single-bit full adders, which feed their output carry into the input carry of the next full adder (the input carry of the first full adder is hard-wired to ground, and the output carry of the last full adder is ignored). A diagram of this high-level circuit is shown in Figure 3. Each single-bit full adder is built from 2 XOR gates, 2 AND gates, and 1 OR (or XOR) gate as shown in Figure 4 in order to produce the truth table that corresponds to addition. This configuration corresponds to addition since the Sum output bit, formed by two consecutive XORs of the inputs, is high only if exactly one or all three of the inputs are high (i.e. the output should be odd). And the Carry output bit is formed with AND gates so that it is only high if two (or more) of the inputs are high. Note that a XOR can be used for the carry since the first XOR between A and B prevents the output of both AND gates to be high at the same time. (This behavior is expected since the sum of three bits cannot carry over into a third bit.)

Detailed Components

In the picture of the circuit (Figure 2), each column is a single-bit full adder. Each of these full adders was built using four integrated logic chips. From top to bottom, these chips are NAND, AND, NAND, and NAND. Since the integrated NAND chips contain four individual NAND gates each, they are used to construct other gates. The first and third chips are NANDs configured as XOR gates, which is accomplished with the circuit diagram shown in Figure 5. The second chip contains four AND gates, two of which are used to be the two ANDs required for the full adder. The last chip is a NAND configured as an OR, which is accomplished with three NAND gates in the configuration shown in Figure 6. Thus, There are 2 XORs, 2 ANDs, and one OR for each full adder. The carry can be seen in the green wire going from one OR to the XOR of the next bit. (The most significant bit is on the left.) The capacitors placed across the +5 V and ground lines are only used for eliminating parasitic oscillations. The only other components of the circuit are the indicator LEDs. These are configured to be on when the corresponding bit is "on". The blue and red LEDs indicate the state of the input bits, and the green LEDs indicate the result of the sum. Resistor values of 1k were found to be sufficient not to burn out the red and blue LEDs, and 200 Ω was used for the green LEDs to increase brightness slightly.

I/O

As for input and output, the DAQ provided has only 8 digital I/O ports, which means that if both 4 bit inputs are going to be sent to the calculator with the digital I/O, then the output of the calculator must be sent back to the PC by a different means. A number of different options were considered, such as sending the inputs to the calculator through the analog output of the DAQ and then using an integrated ADC, or doing the reverse, converting the output of the calculator to analog with a DAC and again using the analog inputs of the DAQ. However, since only 4 additional signal needed to be sent back to the PC, it was considered most convenient to send the output of the calculator to the PC using 4 separate analog inputs on the DAQ. Thus, the VI must read in 4 analog voltages and compare them to an appropriate voltage (2 V was used in this case) to determine if the value is high or low. This allows all information to be sent to and from the calculator without recourse to additional chips.

NAND from First Principles

Lastly, a TTL NAND gate was constructed with BJTs in order to demonstrate the working principles of logical gates from first principles. The NAND gate is shown in Figure 7. If both BJTs are set to conduct current through the collector (that is, if $V_{\text{BE}} \approx 0.6\,\text{V}$), then V_{CE} is fairly low (on the order of a couple dozen millivolts), and power supply voltage must fall almost entirely over the 4.7k resistor. On the other hand, if at least one of the BJTs does not conduct, no current can flow from the power supply through the resistor, making the voltage at the output the same as the input (i.e. 5 V). This demonstrates that in the case for both inputs "on", the output is "off", and in any other case, the output is "on", which is the ideal behavior of the NAND gate.

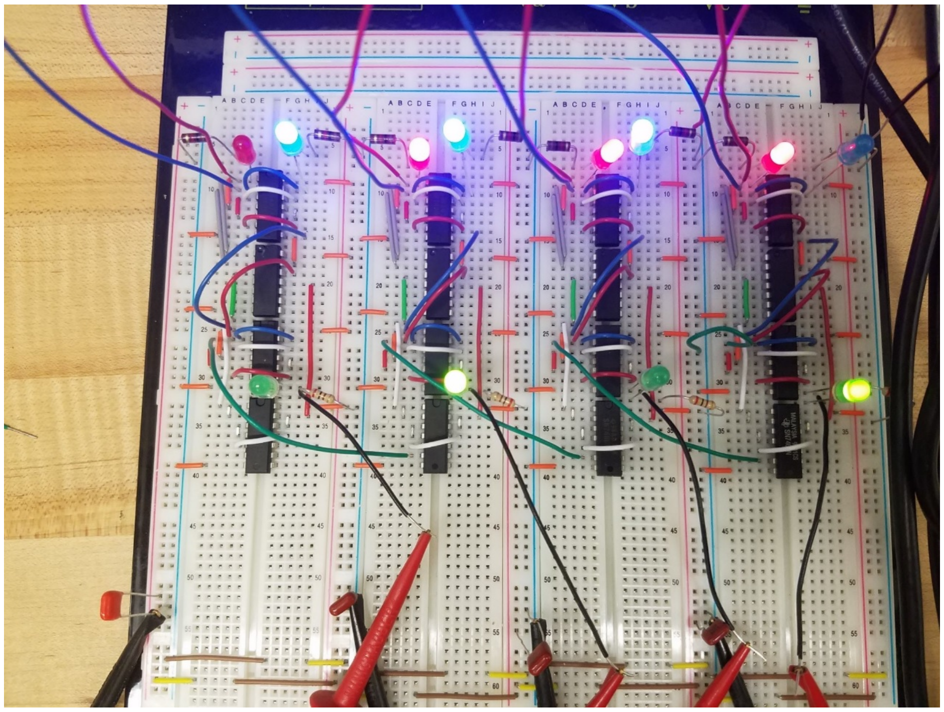


Figure 2: The 4 bit (signed) adding circuit, composed of four ripple-carry full adders. LEDs indicate the state of the input, output, and result bits. The calculator is currently computing 7 (red) plus −2 (blue), giving 5 (green).

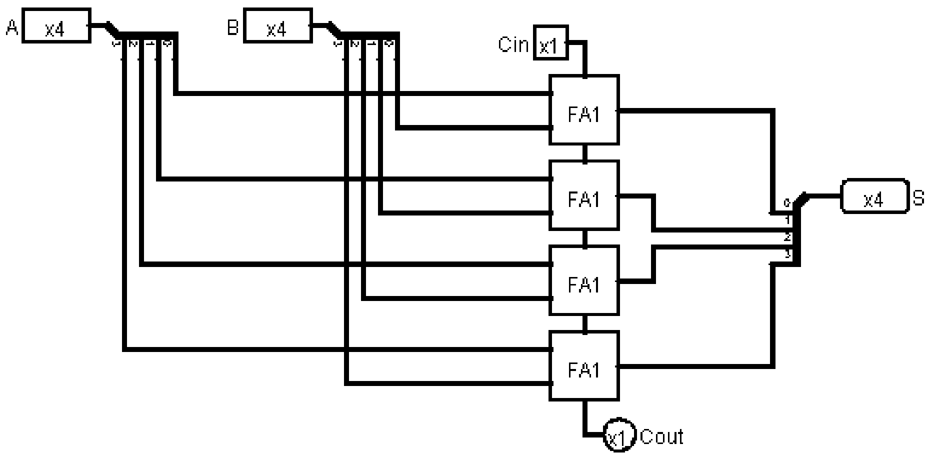


Figure 3: Circuit diagram of the high-level logic of the 4 bit adding circuit. "`FA1`" represents the single-bit full adder, shown in Figure 4.

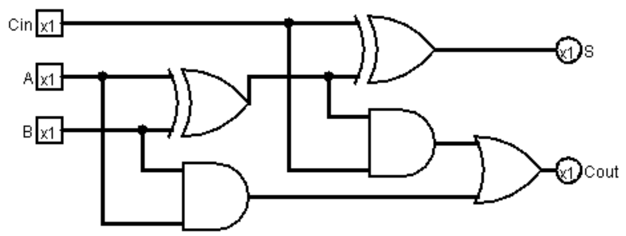


Figure 4: Circuit diagram for a single-bit full adder. Composed of 2 half adders and a pass-through OR or XOR for the carry.

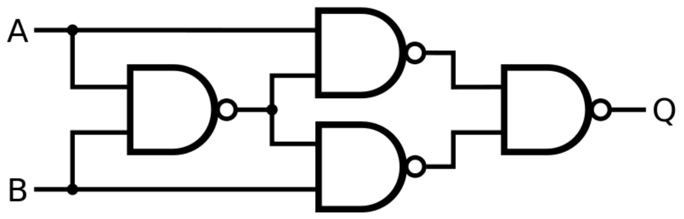


Figure 5: Configuration of NAND gates to produce a XOR gate.

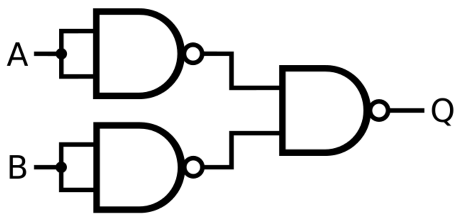


Figure 6: Configuration of NAND gates to produce an OR gate.

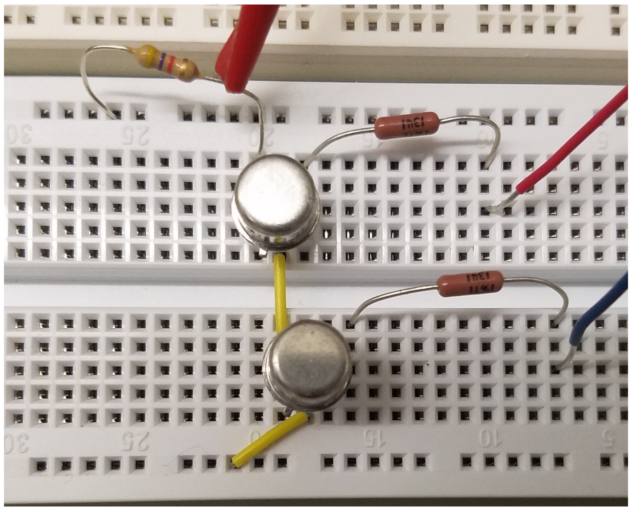


Figure 7: A TTL NAND gate constructed from 2N2219 NPN BJTs. The resistor values at the bases of the BJTs are both $10\,\mathrm{k\Omega}$, while the resistor between the power supply and the collector of the upper BJT is $4.7\,\mathrm{k\Omega}$. The power supply voltage was $5\,\mathrm{V}$.

LabVIEW Description

The LabVIEW program is responsible for interpreting the user input, deciding the order of the computations, and sending and receiving each computation from the calculator. As such, the software is subdivided into four parts, named “`Tokenizer.vi`”, “`RPN.vi`”, “`Calculator.vi`”, and “`Main.vi`”. The next few sections describe each VI and give a motivation for their existence and flow. The author will also present equivalent code for the tokenizer and RPN converter in Python (Figure 10) at the end of this section that should that be easier to follow. It also should be noted that the author is happy to distribute these VIs to anyone for any reason.

Tokenizer

The purpose of the tokenizer is to transform the string input into individual recognizable pieces, in this case “numbers” and “operators” is sufficient notation. A screenshot of the VI is shown in Figure 8. The heavy lifting in this VI is done by the regular expression search method. “Numbers” are characterized by the regular expression “`-?([0-9]+(\.[0-9]+))|(\.[0-9]+)`”, which matches any real number (while intentionally ignoring inputs such as ‘.’ and ‘x.’), while “operators” are characterized by “`[\-\+\*\^\(\)/]`”, which matches the included characters.

Searching these regular expressions would be sufficient, however there is ambiguity with the “$-$” sign, since it can mean subtraction (an operation) or it can simply indicate negation as part of a number. Thus, we must be extra careful when parsing inputs such as “$4 - 1 + (-1 + 2)$” that the first minus sign is treated as an operation, while the second is lumped in with the number. The solution used in this implementation is to always assume minus signs are part of numbers, then go back and insert a “+” sign wherever two numbers are adjacent without an operation between them. This is sufficient for this project (especially considering that it is less effort to send signed numbers to the adder than to pre-process all numbers to be positive and indicate the operation in another way), however it is certainly not the most efficient or most general method. Please also see the equivalent Python solution in Figure 10.

Note also that this tokenizer will parse expressions with many more operations than are supported by the calculator. These are included for debugging, future expandability, and because it is trivial to do so.

RPN

The most involved VI was certainly the Reverse Polish Notation converter, as it contains many nested case structures. It can be seen in the Python implementation (Figure 10) that the bulk of the script is `if` and `else-if` statements, thus the VI shown in Figure 9 shows only the deepest case, instead of showing every possible combination (although this can also be provided). Not much more explanation of the RPN algorithm is necessary, as there are numerous examples online, including Wikipedia, on which this implementation is based. That being said, the purpose of the RPN conversion is to put the numbers and operations into an order such that only a left-to-right scan of the queue is necessary to produce the correct output while accounting for grouping parentheses and the order of operations. Note also that this RPN converter was written to support many more operations than the calculator, although this generalization was not as trivial as for the tokenizer.

The RPN VI can also identify parentheses errors. However, this functionality is not used to a great advantage in this implementation.

Calculator

The Calculator VI is fairly straightforward in that it only sends two digitized (binary) inputs to the calculator, then reads the result. The VI contains a stacked sequence structure for timing, and so is shown with an inset in Figure 11. It can be seen that frame 1 converts the decimal inputs to 4 bit binary and sends the results to the digital outputs of the DAQ. Frame 2 is not shown, but contains an optional wait time if it is necessary to pause to let the calculator perform the computation it has been sent before reading the result in. This wait time was found to be not necessary, with calculations being accurate for a 0 wait time. Frames 2–5 read the output of the calculator into the DAQ analog inputs. Four separate analog inputs were used since the DAQ did not have enough digital inputs. It is suspected that the time required to read a value from one analog input, switch it off, and read another value from the next is the greatest contributor to the relatively slow computation time of the calculator. However, this is discussed more in the Experiments: Timing section.

Once the analog inputs are read in, they are compared with the value 2, which was found to be adequate to separate low-levels from high-levels, then the resulting boolean array converted back into decimal and returned. One extra step is required if the result of the calculator's computation was negative, and we must force LabVIEW to perform a two's complement inversion instead of reading the leading bit as an eight's place. An alternative solution would be to have fed the result back into the calculator (which would perform 0 minus the result) to return a positive result, and simply insert a minus sign before the final result is returned from the VI. However, it is important to mention that a value of -8 returned by the calculator would not be able to be transformed into a 4 bit signed positive number, thus leading possibly to errors or infinite loops.

It is also important to note that the user must be careful not only to input values between -8 and $+7$, but also to ensure that every intermediate calculation returned by the calculator also does not fall outside of this range. It may, however, be the case that errors due to exceedingly large numbers may be canceled out if a similarly large number is subtracted again, due to the way bits are truncated and sent to the calculator. For example, "$4 + 29 - 28$" may still correctly evaluate to 5 since the most significant bits are ignored. However, this has not been tested with the two's complement and should not be considered a feature.

Main

The VI which brings all of these functions together is the Main VI. This VI accepts a string input from the user, tokenizes and RPN-converts it, then searches through the resulting array for an operation. Once the first operation is found, the two previous numbers (along with possibly the operation itself for a more sophisticated calculator) are sent to the calculator. Once the result is returned, the two numbers and the operation are deleted, and the new result is put in their place. This continues until the evaluation queue has only one element left, which is guaranteed to be the result (assuming a syntactically correct input). The Main VI is shown in Figure 12. A sequence of screenshots of the front panel showing `Main.vi` in operation can be seen in Figure 13.

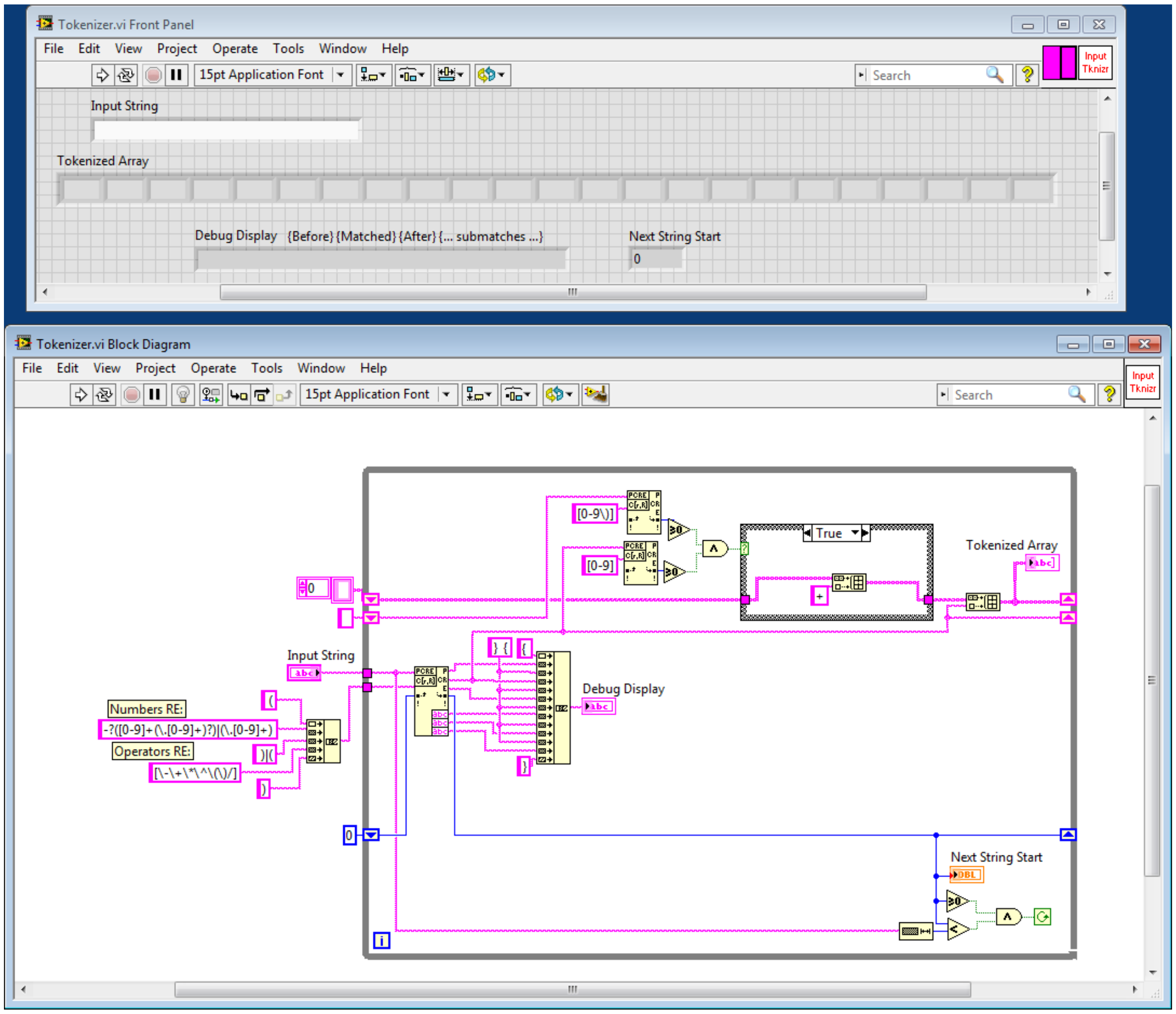


Figure 8: The tokenizer VI, which transforms the input string into recognizable pieces of numbers and operators (tokens).

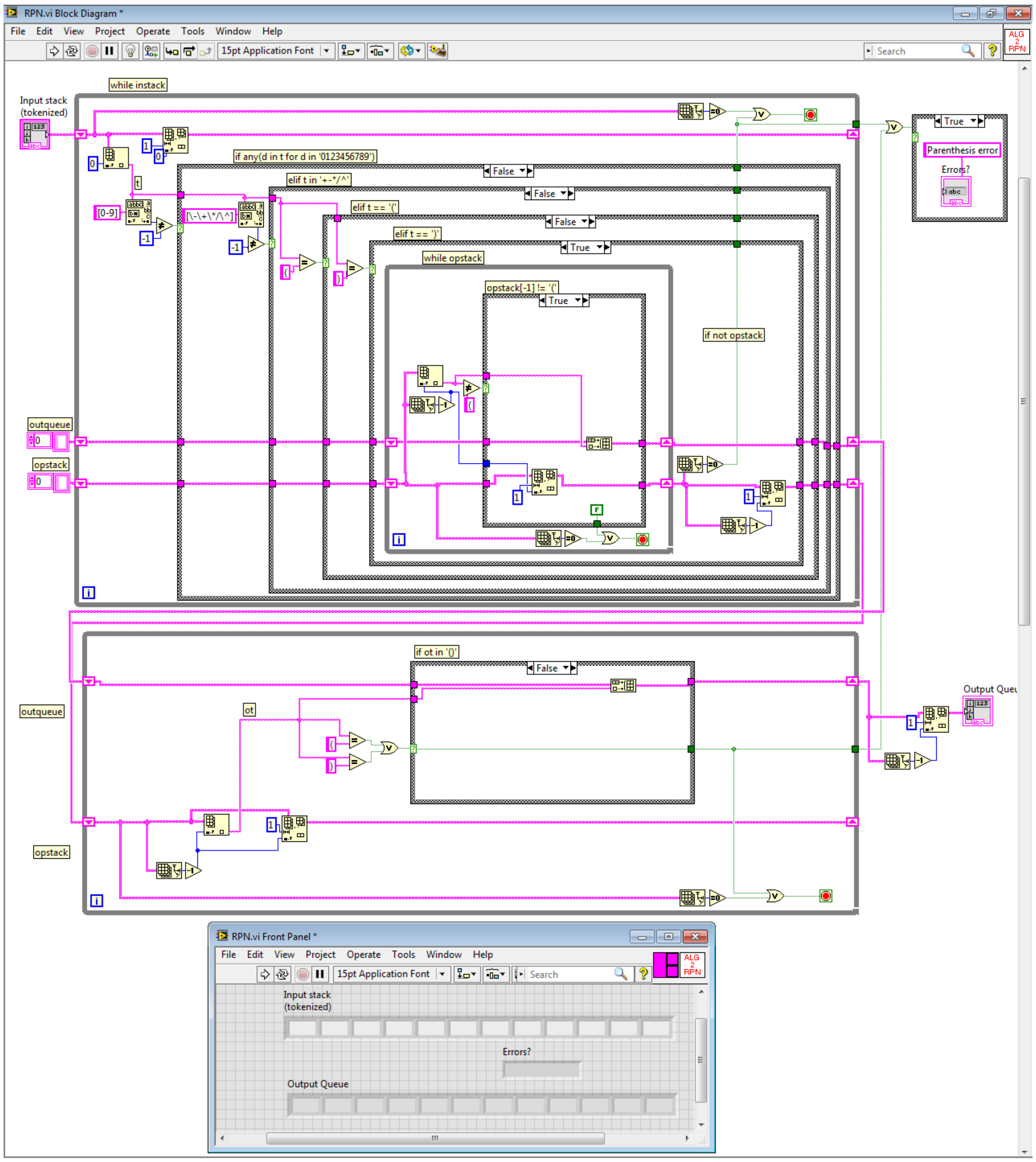
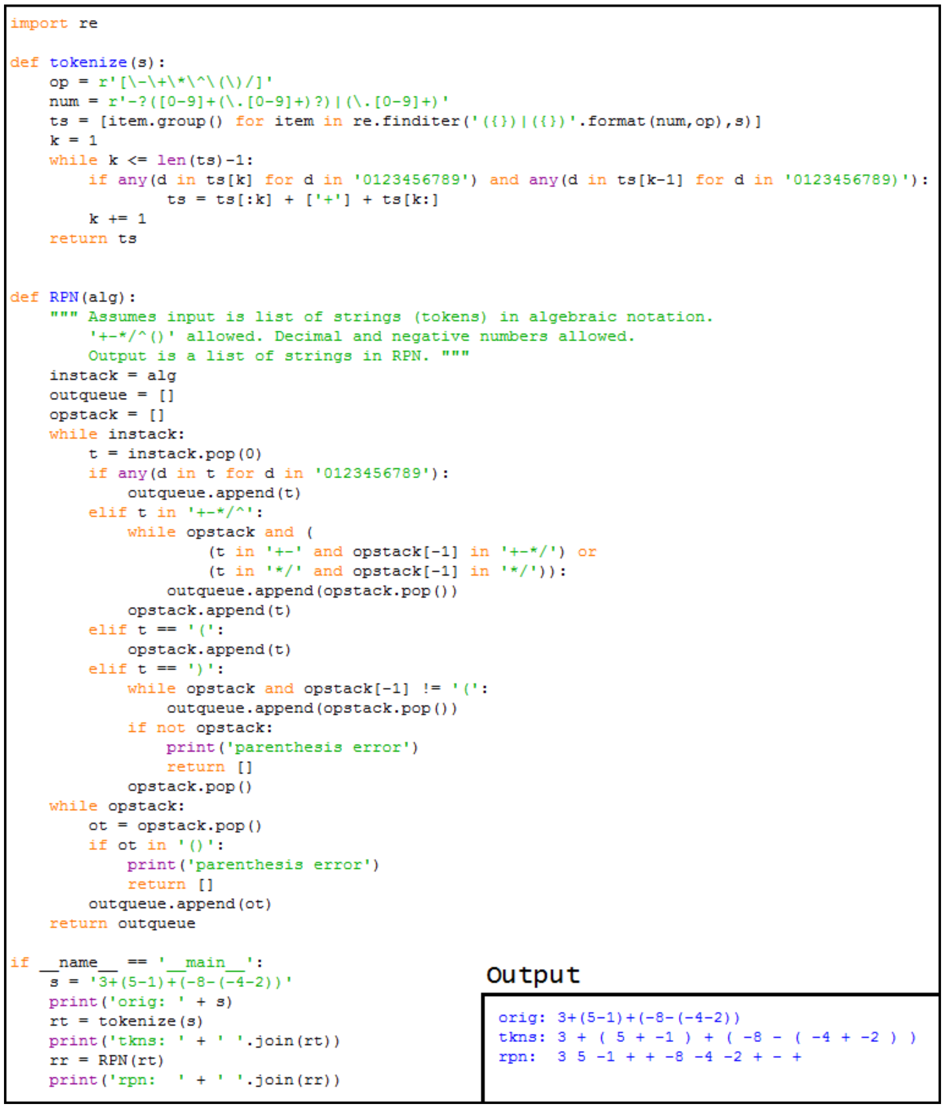


Figure 9: The Reverse Polish Notation VI, which converts the tokenized input form algebraic "infix" notation to RPN.



```python
import re

def tokenize(s):
    op = r'[\-\+\*\^\(\)/]'
    num = r'-?([0-9]+(\.[0-9]+)?)|(\.[0-9]+)'
    ts = [item.group() for item in re.finditer('({})|({})'.format(num,op),s)]
    k = 1
    while k <= len(ts)-1:
        if any(d in ts[k] for d in '0123456789') and any(d in ts[k-1] for d in '0123456789)'):
                ts = ts[:k] + ['+'] + ts[k:]
        k += 1
    return ts


def RPN(alg):
    """ Assumes input is list of strings (tokens) in algebraic notation.
        '+-*/^()' allowed. Decimal and negative numbers allowed.
        Output is a list of strings in RPN. """
    instack = alg
    outqueue = []
    opstack = []
    while instack:
        t = instack.pop(0)
        if any(d in t for d in '0123456789'):
            outqueue.append(t)
        elif t in '+-*/^':
            while opstack and (
                    (t in '+-' and opstack[-1] in '+-*/') or
                    (t in '*/' and opstack[-1] in '*/')):
                outqueue.append(opstack.pop())
            opstack.append(t)
        elif t == '(':
            opstack.append(t)
        elif t == ')':
            while opstack and opstack[-1] != '(':
                outqueue.append(opstack.pop())
            if not opstack:
                print('parenthesis error')
                return []
            opstack.pop()
    while opstack:
        ot = opstack.pop()
        if ot in '()':
            print('parenthesis error')
            return []
        outqueue.append(ot)
    return outqueue

if __name__ == '__main__':
    s = '3+(5-1)+(-8-(-4-2))'
    print('orig: ' + s)
    rt = tokenize(s)
    print('tkns: ' + ' '.join(rt))
    rr = RPN(rt)
    print('rpn:  ' + ' '.join(rr))
```

Output

```
orig: 3+(5-1)+(-8-(-4-2))
tkns: 3 + ( 5 + -1 ) + ( -8 - ( -4 + -2 ) )
rpn:  3 5 -1 + + -8 -4 -2 + - +
```

Figure 10: Python versions of the tokenizer and RPN converter, which may be easier for the reader to follow.

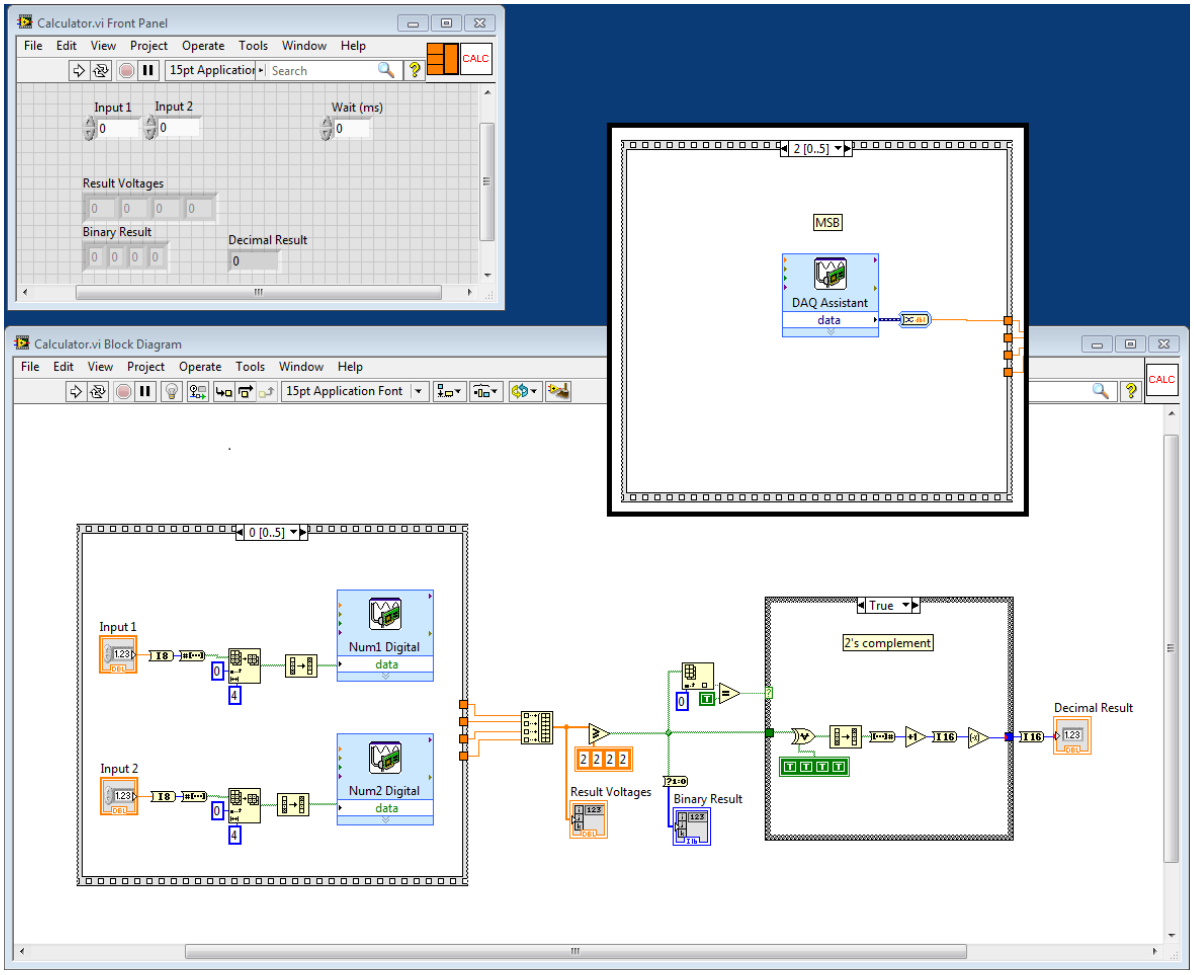
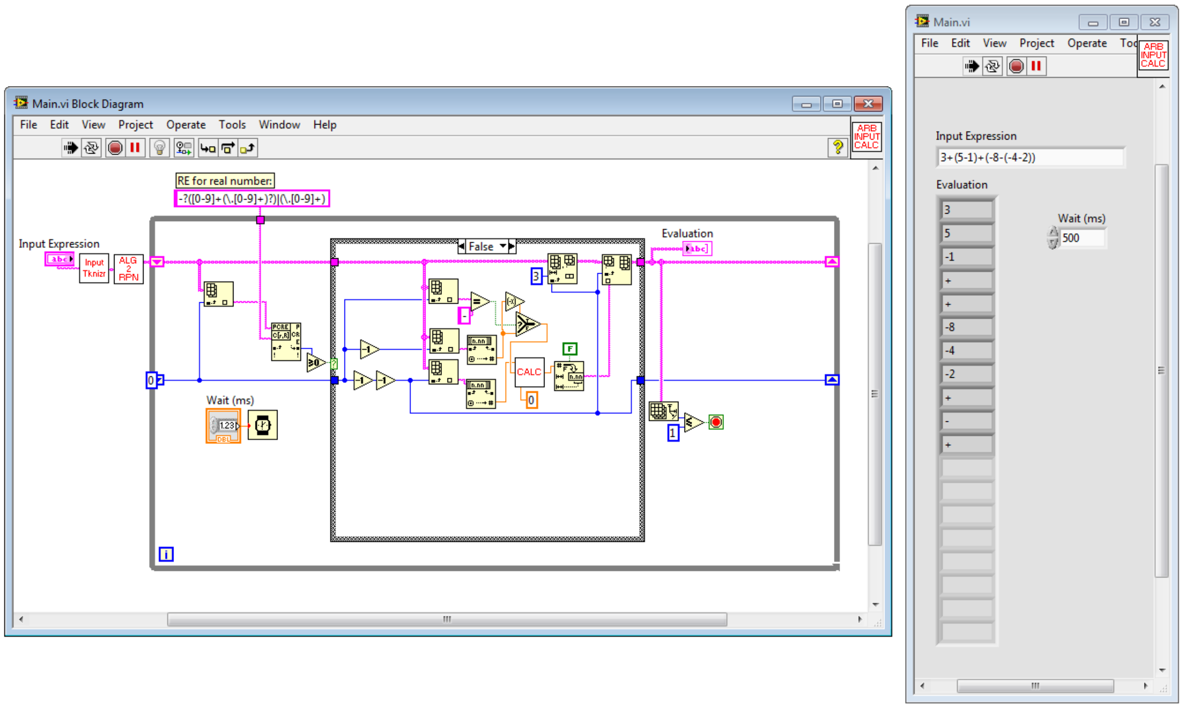


Figure 11: The Calculator VI, which sends the computation to be completed to the calculator and reads back the result. A two's complement conversion may also need to be completed if the result is negative.



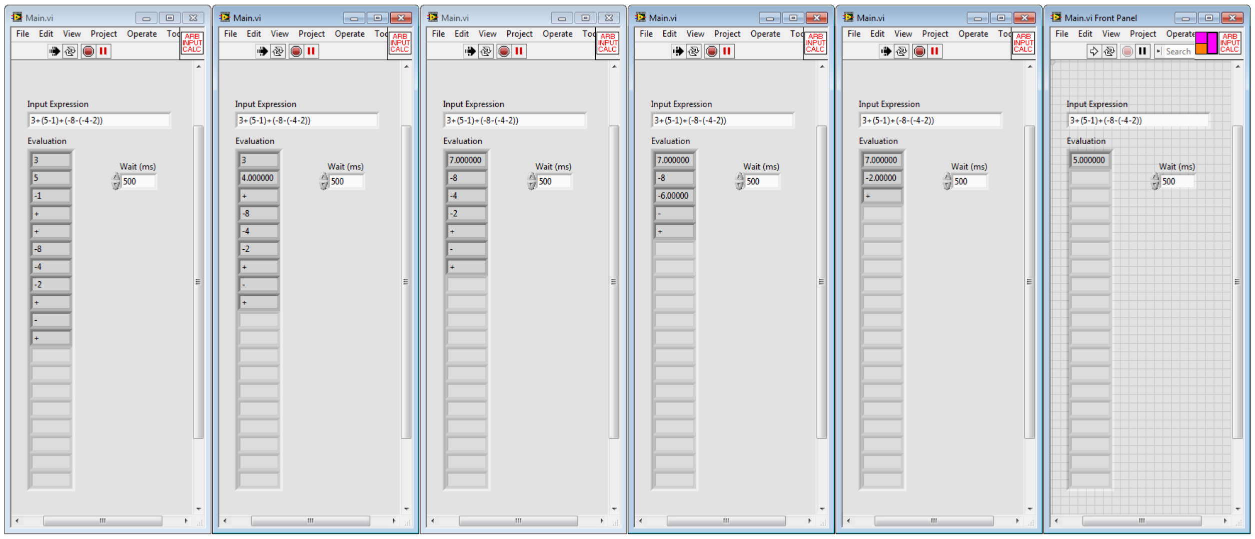
RE for real number:
-?([0-9]+(\.[0-9]+)?)|(\.[0-9]+)

Input Expression: 3+(5-1)+(-8-(-4-2))

Evaluation: 3, 5, -1, +, +, -8, -4, -2, +, -, +

Wait (ms): 500

Figure 12: The Main VI, which incorporates all other VIs and manages the sequence of computations.



Input Expression: 3+(5-1)+(-8-(-4-2))

Wait (ms): 500

Evaluation: 3, 5, -1, +, +, -8, -4, -2, +, -, +

Evaluation: 3, 4.000000, +, -8, -4, -2, +, -, +

Evaluation: 7.000000, -8, -4, -2, +, -, +

Evaluation: 7.000000, -8, -6.00000, -, +

Evaluation: 7.000000, -2.00000, +

Evaluation: 5.000000

Figure 13: The Main VI front panel at various times during its iteration, showing the progressive simplification of the calculation.

Experiments: Timing

Aside from demonstrating that the circuit and software work correctly, we wanted to time how long the calculator takes to perform one computation. Figure 14 shows the VI used to time the Main VI. Unfortunately, the VI only managed to solve an expression of 5 operations about 100 times in 5 seconds. This implies an average individual computation time of about 10 ms. However, it is reasonable to believe that the slow behavior does not lie in the calculator, but rather in the LabVIEW implementation. Note that the Calculator VI contains no wait time between sending the inputs to the calculator and reading the results back into the VI. The delay is only the amount of time it takes LabVIEW to advance a frame in a stacked sequence structure and prepare the first DAQ Assistant for reading. This is where it is believed that the majority of the computation time is taking place. Since not enough digital I/O was available, four different analog inputs were used for the calculator output. It is reasonable to believe that it takes LabVIEW on the order of some milliseconds to prepare the DAQ to read a value, then stop reading and shut down that DAQ Assistant, then prepare the next one, four times over. It is known, for example, that each DAQ Assistant had to be placed in a separate frame because LabVIEW was unable to read from multiple DAQ Assistants simultaneously and would return an error. Additionally, attempting to read four inputs with one DAQ Assistant proved unsatisfactory, as the DAQ Assistant would return a single scalar instead of a four element array, as would be expected.

The timing VI was run again with `Calculator.vi` duplicated in `Main.vi` so that it would be called twice for each time that it would normally be called once. This resulted in exactly half as many runs within 5 seconds as before, which narrows the timing issue to `Calculator.vi`.

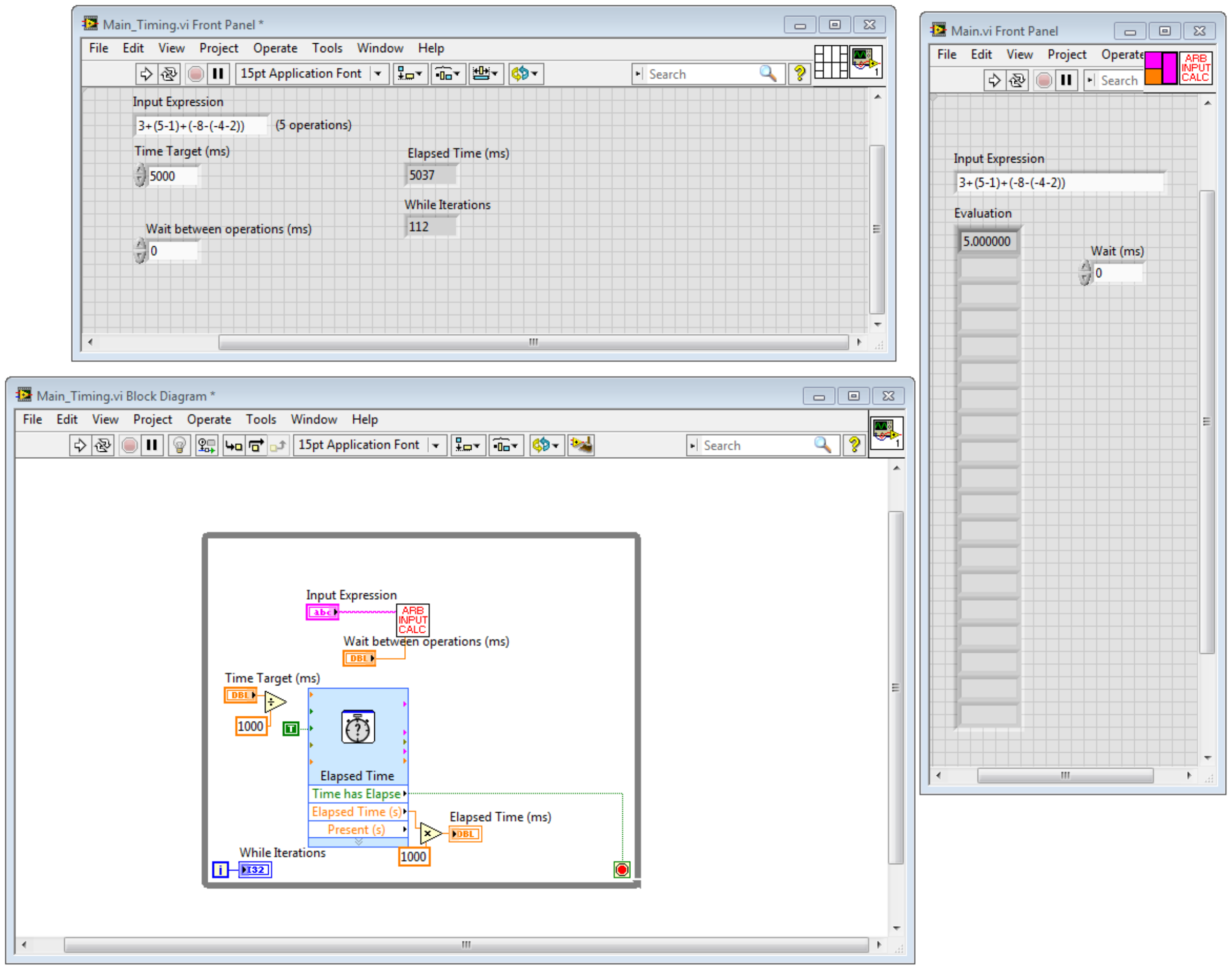
Unfortunately, more sophisticated tests were not run, but the above analysis seems to indicate that the circuit is behaving properly (with computation times ideally in the microsecond range, if not less), and that the issue is with the software, with a number of avenues to explore to decrease computation time.

Conclusion

Considering this project a success, with the exception of the slow execution time, a lot has been learned about the fundamentals of digital circuits and LabVIEW programming. Ideas for future improvements include fixing the timing issue, expanding the bit width of the calculator, expanding the number of operations available on the calculator, and finally, transposing as much of the logic of the LabVIEW software as possible down to hardware.

Acknowledgments

The author would like to thank Professor Reinsch, as this class would not have been possible or nearly as enjoyable without him. Especially his thoughtful lectures and dedication to listen to all of the questions his students had. Thank you also to the GSIs for which much the same can be said. Thanks also to Dr. Fajans for his work on the lab manuals during the course, the improvements to which helped immensely. Thanks also to Don who, while we did not see him much, is known to be ever present in the lab. And thanks finally to Carl Burch who produced and developed the free and open-source software Logisim, which is a fantastic tool and was used to do much of the modeling for the project.



Main_Timing.vi Front Panel:

Input Expression: 3+(5-1)+(-8-(-4-2)) (5 operations)

Time Target (ms): 5000

Elapsed Time (ms): 5037

While Iterations: 112

Wait between operations (ms): 0

Main.vi Front Panel:

Input Expression: 3+(5-1)+(-8-(-4-2))

Evaluation: 5.000000

Wait (ms): 0

Main_Timing.vi Block Diagram: Input Expression, Wait between operations (ms), Time Target (ms), Elapsed Time (ms), While Iterations

Figure 14: The VI used to run `Main.vi` for a certain amount of time and count the number of times it ran.