islington college
(इस्लिङ्टन कलेज)

LONDON METROPOLITAN UNIVERSITY

Module Code & Module Title

CC4051NA Fundamentals of Computing

Assessment Weightage & Type

60% Individual Coursework

Year and Semester

2019-20 Autumn

Student Name: Bijay Bharati

Group: L1C4

London Met ID: 19030824

College ID: NP01CP4A190041

Assignment Due Date: 8th June 2020

Assignment Submission Date: 3rd June 2020

I confirm that I understand my coursework needs to be submitted online via Google Classroom under the relevant module page before the deadline for my assignment to be accepted and marked. I am fully aware that late submissions will be treated as non-submission and a mark of zero will be awarded.

Contents

Figures

Tables

1 Introduction

The project is based on simulating the behavior of a digital circuit. Python is implemented to create a byte adder by implementing logic gates.

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. (Kuhlman, 2013)

It is very flexible and versatile programming language; numerous preexisting libraries make python powerful and reliable. Python is becoming more and more preferred for automation tasks, data handling, machine learning, etc.

This project is just beginner level python programming aimed as an introduction to python.

We will be developing an 8-bit binary adder which takes decimal input from users, validates the user input, converts the input to binary and gives the sum. We are trying to show what happens inside a chip when we add 2 numbers by implementing logic gates but in a program.

Algorithm, flowchart, pseudocode and testing will be clearly demonstrated.

2 Model

Half adder and full adder circuits are implemented in the model of the 8-bit adder. Further information is given below.

2.1 Half adder

Half adder works by taking in two inputs and produces sum and carry out as output. Truth table and logic circuit diagram for half adder is presented below for better understanding.

Table 1: Half adder truth table

A	B	Sum	Cout
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

To find sum let's take all the operation instances where 1 occur as result for Sum in the truth table.

Sum= $(A * \overline{B}) + (\overline{A} * B)$

$= A \oplus B$

To find Cout let's take all the operation instances where 1 occur as result for Cout in the truth table.

Cout= A*B

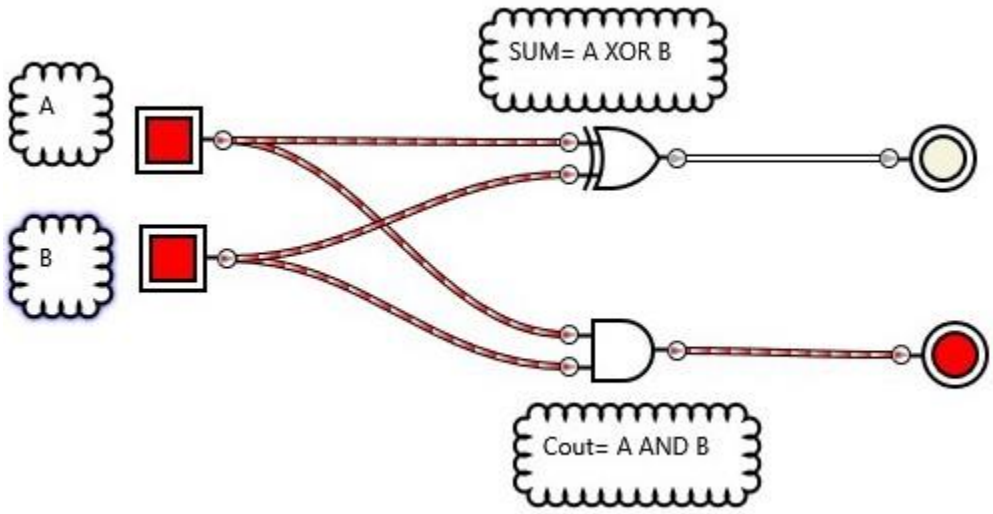

Figure 1: Half adder circuit diagram

2.2 Full adder

Full adder can take in 3 inputs including carry in, the carry in is previous operation's carryout. Full adder also gives sum and carryout as output.

Table 2: Full adder truth table

A	B	Cin	Sum	Cout
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

To find sum let's take all the operation instances where 1 occur as result for Sum in the truth table.

Sum= $(A * B * C) + (A * \overline{B} * \overline{C}) + (\overline{A} * B * \overline{C}) + (\overline{A} * \overline{B} * C)$

$= A * (B * C + \overline{B} * \overline{C}) + \overline{A} * (B * \overline{C} + \overline{B} * C)$

$= A * (\overline{B \oplus C}) + \overline{A} * (B \oplus C)$ {let, $B \oplus C = D$}

$= A * \overline{D} + \overline{A} * D$

$= A \oplus D$

$= A \oplus (B \oplus C)$

To find Cout, taking all the instances where 1 occur as result for Cout in the truth table.

Cout= $(A * B * C) + (A * B * \overline{C}) + (A * \overline{B} * C) + (\overline{A} * B * C)$

$= A * B * (C + \overline{C}) + C * (A * \overline{B} + \overline{A} * B)$

$= A * B * 1 + C (A \oplus B)$

$= C(A \oplus B) + AB$

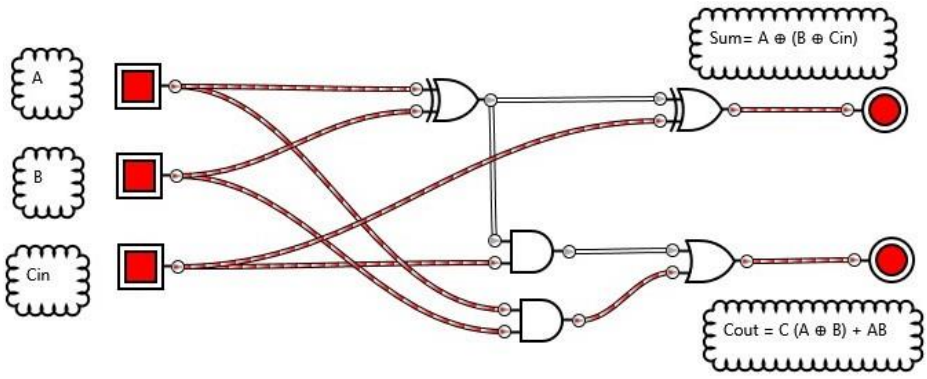

Figure 2: Full adder circuit diagram

2.3 Bit adder model

There is an 8-bit adder implemented in the project. It uses both half adder and full adder. First the half adder takes a bit and provides sum and carry out as output. The carry out is passed as carry in for full adder along with other 2 bits. full half adder then produces sum and carry out.

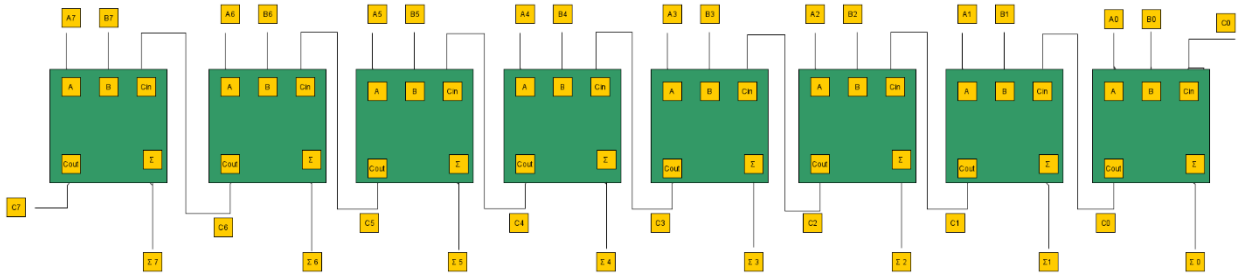

Figure 3: Parallel adder circuit

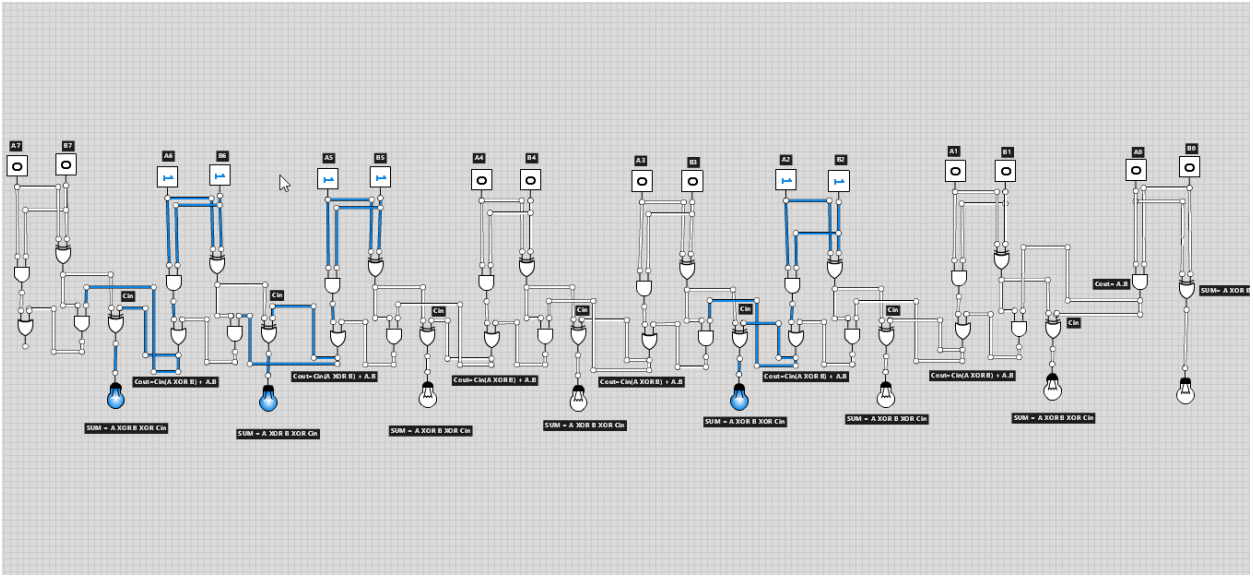

Figure 4: 8-bit adder representation

Each Cout becomes Cin for next adder circuit.

The circuit represented by the figure is an 8-bit adder circuit. The 8-bit adder adds two 8-bit binary user input. The adder was created by implementing both half and full adders. There is one half adder and seven full adders.

The design is simple to implement. First single half adder and full adder were developed, and the rest were copied, and connections were made. First the half adder will take the least significant bit LSB (the least significant bit is the lowest bit in a series of numbers in binary. It is either the leftmost or rightmost bit in a binary number, depending on the computer's architecture. If the LSB is on the right, the architecture is called "little-endian." If the LSB is on the left, the architecture is called "big-endian." For example, in a little-endian architecture, the LSB of binary number 00000001 is 1.) then

the half adder will give sum and carry out. The carry out is then passed to subsequent adder. Then the full adder now has A, B and Cin which are the required parameters for it to function as described previously.

In the figure above, an 8-bit adder performs binary addition on decimal (100+100) i.e. binary 01100100 + 01100100 and gives result 01100100 as output. The bulbs represent the sum 1 is bulb on and off is 0.

The half adder takes in 0,0 and gives sum and Cout. Then the full adder takes 0,0 and 0(Cin) produced from the half adder and gives sum and Cout. This process keeps on repeating and all the inputs and outputs generated by the adder circuit can be clearly seen in the figure.

3 Algorithm

An algorithm is a finite series of well-defined, computer-implementable instructions to solve a specific set of computable problems. It takes a finite amount of initial input(s), processes them unambiguously at each operation, before returning its outputs within a finite amount of time. (Math Vault, 2020).

We develop algorithms to break down and focus on our problem to develop an easy to understand and implement solution. The algorithm for the project is given below.

Step 1: start
Step 2: take input from user number1, number2
Step 3: check if the inputs are valid integer between 0 and 255
Step 4: if input is invalid go t0 step 2 and take another input
Step 5: convert the inputs to binary
Step 6: create list bit=[]
Step 7: create conversion_number, conversion_number=user input
Step 8: create remainder, remainder=conversion_number mod 2
Step 9: add remainder to bit
Step 10: update conversion_number, conversion_number//2
Step 11: repeat steps 6-10 until conversion_number=0
Step 12: store the updated list
Step 13: create string final=””
Step 14: take the length of list from step 12
Step 15: if the length is not 8 add 0 to the list until length is 8
Step 16: take the list in reverse order and add each element to final
Step 17: the string produced in step 16 is final binary number
Step 18: repeat steps 6-17 for number1 and number2
Step 19: store the strings in step 18 as binary1_final and binay2_final for number1 and number2
Step 20: create half adder
Step 21: half adder takes in 2 bits (a,b)
Step 22: implement logic sum=a^b
Step 23: implement logic carry=a and b
Step 24: return sum and carry form step 22 and 23
Step 25: create full adder
Step 26: full adder takes in 3 bits (carry_in, a, b)
Step 27: take carry from step 23 as carry_in
Step 28: implement logic sum=(A XOR (B XOR C))
Step 29: implement logic carry out= C(A XOR B) +A AND B
Step 30: create binary_adder
Step 31: implement half adder and full adder
Step 32: create lists from the binary numbers in step 19
Step 33: create list result=[]
Step 34: create string final_sum=""

Step 35: iterate through first list from step 32
Step 36: take each elements from lists 32
Step 37: find carry and sum using full adder
Step 38: add sum to result
Step 39: while length of result is<8 add carry to list
Step 40: finally obtain result
Step 41: add each element of result from step 40 to final_sum
Step 42: final_sum obtained in step 41 is the result of binary addition
Step 43: display the final_sum of step 42 as output
Step 44: ask if the user would like to perform another addition
Step 45: if user wants to perform another addition go to step 2
Step 46: if user says to quit, end the program

The algorithm gives the general idea of how the coding is done, it provides the outline for development and usage of the program.

4 Pseudocode

Pseudocode gives the outline for the program; it gives a rough idea of how the codes are written and how the final program will be like. Pseudocode for different modules implemented in the project are given below.

4.1 Pseudocode for input_validation module

```
FUNCTION input_check(message):
    while true:
        try:
            Value=int (input(message))
        exception:
            Print(try again)
        if(value<0):
            Print(enter num>=0)
        end if
        if(value>255):
            Print(enter num<=255)
        end if
        else:
            break
        end else
        end try
    return value
END FUNCTION
```

4.2 Pseudocode for binary_conversion_1 module

```
FUNCTION binary_conversion (conversion_number):
    bit= []
    while conversion_number!= 0:
        remainder= conversion_number % 2
        bit.append(remainder)
        conversion_number=conversion_number//2
```

```
    return Bit
END FUNCTION
```

4.3 Pseudocode for binary_conversion_final module

```
FUNCTION final_binary(binary_value):
    final=’’’’
    count=len(binary_value)
    while count != 8:
        binary_value.append(0)
        count=count+1
    for i in range(len(binary_value)-1,-1,-1):
        final=final + str(binary_value[i])
    end for
    return final
END FUNCTION
```

4.4 Pseudocode for binary_adder module

```
FUNCTION
    FUNCTION half_adder(a,b):
        sum = a^b
        carry = a and b
        return carry,sum
    END FUNCTION
    FUNCTION full_adder(cin,a,b):
        carry1,sum1=half_adder(cin,a)
        carry2,sum=half_adder(sum1,b)
        carry=carry1 or carry2
        return carry,sum
    END FUNCTION
```

```
    FUNCTION binary_adder(a,b):
        a_ist=list(int(x,2) for x in list(a))
        b_list=list(int(x,2) for x in list(b))
        result=[]
        carry=0
        final_sum=""
        for i in range(len(a_list) -1,-1,-,1):
            carry,sum = full_adder(carry, a_list[i],b_list[i])
            result.insert(0,sum)
        if len(result)<8:
            result.insert(0,carry)
        end if
        end for
        fina_sum=''.join(str(x) for x in result)
        return final_sum
    END FUNCTION
END FUNCTION
```

4.5 Pseudocode for main module

```
FUNCTION
    Import input_validation
    Import binary_conversion_part
    Import binary_conversion_final
    Import binary_adder
    continue_looping=true
    while continue_looping=true:
        number1=input_validation.input_check("message: ")
        number2= input_validation.input_check("message: ”)
        if (number1+number2)>255:
            print(9-bit operation)
        end if
        binary1=binary_conversion_1.binary_conversion(number1)
        binary2=binary_conversion_1.binary_conversion(number2)
        binary1_final=binary_conversion_final.final_binary(binary1)
        binary2_final=binary_conversion_final.final_binary(binary2)
        sum=binary_adder.binary_adder(binary1_final,binary2_final)
        print(sum)
        quit=input("Press q to quit”)
        if:
            quit=q or Q
            break
        end if
END FUNCTION
```

5 Flowchart

Flowchart is just the graphical representation of algorithm. Flowcharts are made to explain the workings of a program in a way everyone can understand. Different shapes are used in a flowchart to indicate start, input, process, decisions, output, stop etc. The flowchart for the program implemented in the project is given below.

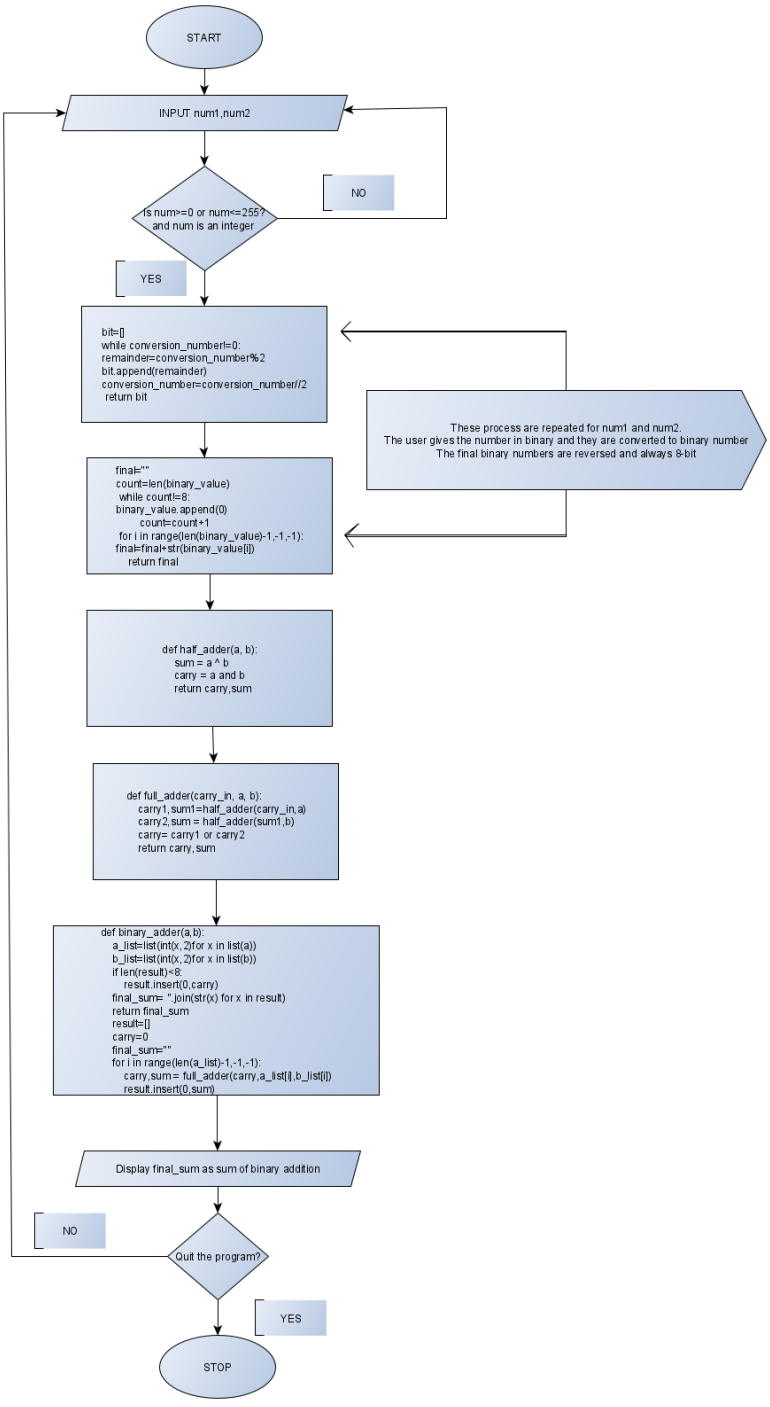

Figure 5: Flowchart

6 Data Structures

Data structures are basically just that - they are structures which can hold some data together. In other words, they are used to store a collection of related data. There are four built-in data structures in Python - list, tuple, dictionary and set. (Chitlur, 2020)

String, int, float, boolean etc. are can also be called data structures but are better and more accurately defined as data types. Data Structures than can be found in this project are briefly described below.

- int – int is used to store the input given by the user. In the case of the program, int will store user input 0-255.

- str – str is used to store hold a string of characters. All the outputs in the project is converted to string. Items from lists are also extracted and stored in string

- boolean – boolean is a data type used to hold true of false values used most notably in the project to continue looping

- list – lists are used to store a collection of data. In this project, lists are widely used to digits of binary values. For loop is primarily used to go through the list. The elements of the list are also converted to string. Big brackets ([]) are used to create lists and each element in a list is separated by a comma (,) lists have different methods but list.append(position) and list.insert(i,j) are the only methods used in the project. We give a value i in .append to extract element from certain index of the list and .insert(position, element) is used to insert elements in desired position.

7 Testing

Table 3: Test 1

Test 1	
Objective	To check if invalid inputs are accepted
Action	• Invalid inputs are given • Valid inputs are given
Expected Results	• User should be prompted to enter a valid input if the input is invalid • Program should run after valid inputs are given
Actual Results	• User was prompted to give input again • Problem continued as intended when valid inputs were given
Conclusion	Test is successful

```
Enter first number: -1
Please enter an integer >= 0


Enter first number: 256
Please enter an integer <= 255


Enter first number: 123

Enter second number: 45
The result of binary addition is: 10101000


Press q to quit the program, anything else will continue the program:
```

Figure 6: Test 1

As seen in the figure, the program will prompt the user to give the input again if input less than 0 or more than 255 is entered and once valid inputs are given, the program will continue.

Table 4: Test 2

Test 2	
Objective	to check if the program will give correct binary addition
Action	• Inputs are given, for our test we will give 89 and 33 as input
Expected Results	• The program should give 01111010 as result • This result was pre-calculated for testing
Actual Results	• The output was 01111010
Conclusion	Test is successful

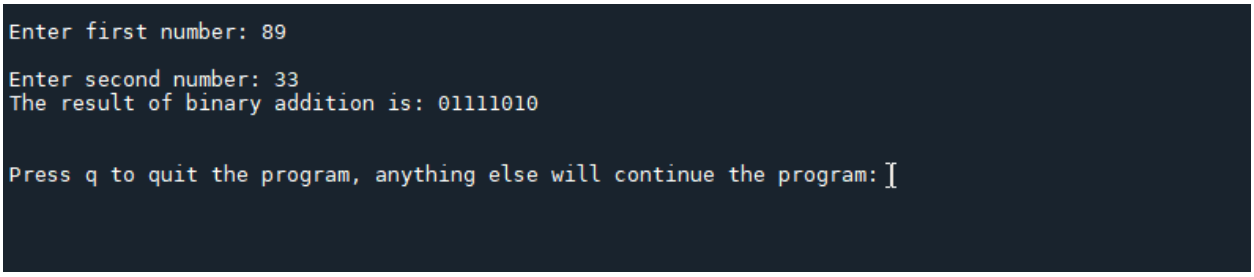

Figure 7: Test 2

The program performs as intended when valid inputs are given

Table 5: Test 3

Test 3	
Objective	To check if the program can be used continuously at a time
Action	• When prompted to exit, we chose not to exit
Expected Results	• Program should ask for new inputs when we specify, we don't want to quit the program
Actual Results	• The program asked for new inputs when chosen not to quit the program
Conclusion	Test is successful

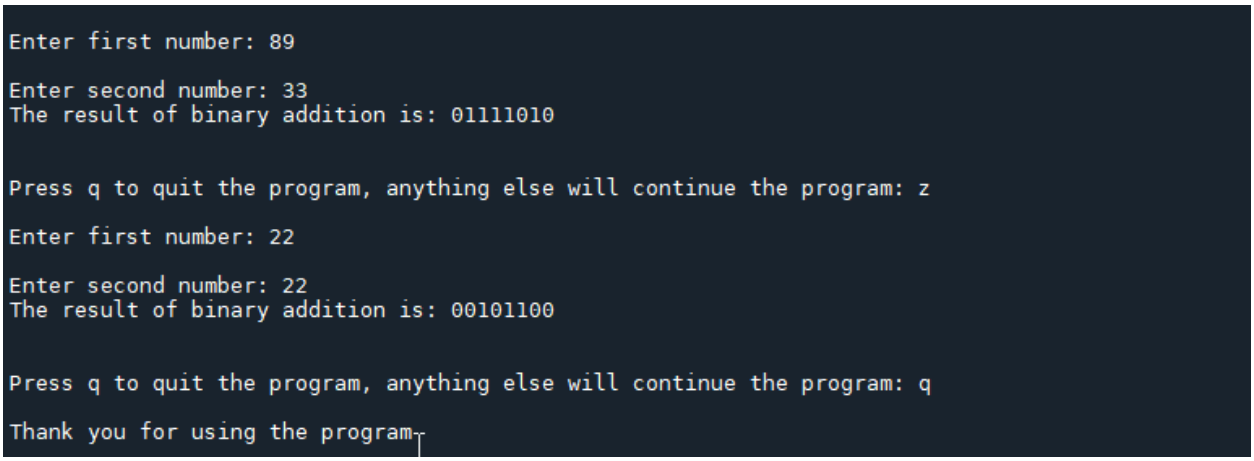

Figure 8: Test 3

Until the user presser q when prompted to exit, the program will keep on executing.

Table 6: Test 4

Test 4	
Objective	To check exception handling
Action	• Strings and floating-point variables were given as input
Expected Results	• User should be prompted to enter the value again
Actual Results	• User was prompted to re-enter the value
Conclusion	Test is successful

```
Enter first number: wubbalubbadubdub
Oops! invalid input found, please try again.


Enter first number: 1

Enter second number: 1.5
Oops! invalid input found, please try again.


Enter second number: 1
The result of binary addition is: 00000010


Press q to quit the program, anything else will continue the program:
```

Figure 9: Test 4

The program runs even if string or float is given instead of int, this is because of exception handling. Without exception handling, the program would crash as shown below.

```
Enter first number: 0.5
Traceback (most recent call last):

  File "B:\STUDY\FUNDAMENTALS OF COMPUTING\course work\bkup\main.py", line 15, in <module>
    number1=input_validation.input_check("Enter first number: ")

  File "B:\STUDY\FUNDAMENTALS OF COMPUTING\course work\bkup\input_validation.py", line 3, in
input_check
    value=int(input(message))

ValueError: invalid literal for int() with base 10: '0.5'
```

Figure 10: No exception handing

Table 7: Test 5

Test 5	
Objective	To check what will happen if user tries to perform a 9-bit operation
Action	• 255 and 255 were given as input, whose result of binary addition is 111111110 which is a 9-bit operation
Expected Results	• User should be notified about the limitations of the program • If the result is displayed, it should be 8-bit
Actual Results	• User was notified about the limitations of the program • Only 8-bit result was displayed
Conclusion	Test is successful

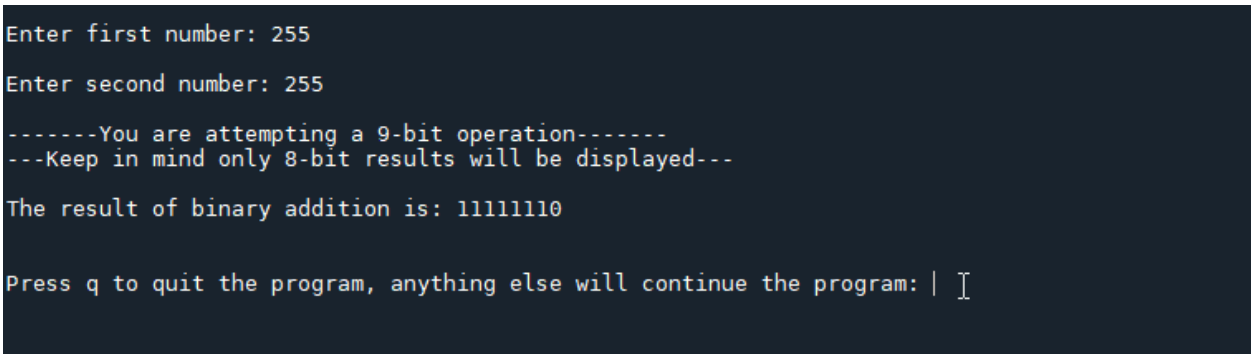

Figure 11: Test 5

The user is notified that he is attempting a 9-bit binary addition and that only 8-bits will be displayed, and only 8-bit result is given by the program.

8 Conclusion

The project was completed successfully. During the research phase of the project many things were learnt about bit adder model and electric circuits I got to know that in the project, we were simulating an actual logic circuit, before that, I just thought this project was to convert two numbers to binary and add them, but it turned out to be much more interesting.

The project proved useful in helping to understand that there are many ways to manipulate data. Integers can be converted to lists, lists to string, string to lists, there are multiple possibilities to handle data.

Different modules are implemented in the project, each module handles a part of the whole program. There is a module to validate input, extract 0 and 1, reverse the 0s and 1s into proper binary form, add the binary numbers and a main module that integrates all the modules.

The project encouraged us to solve the program, in fact any problem by breaking the problem into smaller pieces as demonstrated by the modules present in the submission. This approach is not only easy to implement but also reusable, less prone to errors, easy to maintain and less tiring for the person implementing it.

The section to develop algorithm and flowchart encouraged us to show how we thought about our solution and how to confidently present our ideas for others to see.

The research in python in general showed that python is very versatile allowing object oriented as well as functional and procedural programming among others.

The research also showed me the possibilities of python and the projects and articles that I came across during my research were very informative and helped better my understanding of python.

Bibliography

Chitlur, S. (2020) *Gitbook* [Online]. Available from: https://python.swaroopch.com/data_structures.html [Accessed 5 May 2020].

ComputerHope. (2019) *ComputerHope* [Online]. Available from: https://www.computerhope.com/jargon/l/leastsb.htm [Accessed 01 June 2020].

Kuhlman, D. (2013) *A Python Book: Beginning Python, Advanced*.

Math Vault. (2020) *Math Vault* [Online]. Available from: https://mathvault.ca/math-glossary/#algo [Accessed 01 June 2020].