

CALIFORNIA STATE POLYTECHNIC
UNIVERSITY, POMONA

ECE 480:
Software Engineering
Professor Chandra
Winter 2014

Project 1:
Combinational Logic Simulator
with Python

Omar Ramirez
John Valera
Avis Kirakosyan
Hang Zhao
Jaran Francis

Table of Contents

1. Abstract	Page 2
2. “How did we do it?”	Page 3
3. Source Code	Page 7
4. 3-Case Studies	Page 13
5. Limitations	Page 16
6. Accountability	Page 17
7. Conclusion	Page 18
8. References	Page 19

Abstract -

The following Python program was created to simulate a combinational logic gate circuit. Our goal was to create a logic simulator to read a combinational logic description file (LDF) and output the circuit in ascending gate number and finally generate the truth table for the user-selected outputs. Our requirements to complete the project go as follow:

1. The number of primary inputs is limited to 26.
2. The number of user-selected outputs is limited to 26.
3. The Fan-in for each gate is 8.
4. The following gate types are supported: NOT, AND, OR, XOR, NAND,
NOR, XNOR.
5. The gate count in each circuit is limited to 50.
6. Each line of the LDF is formatted like so: <Gate Number> <Name>
<Type> [<Input1>....<Input8>]/n
7. The gate numbers should be continuous, beginning with 1.

Our program was created successfully with the help of the powerpoints and notes provided by our instructor. The colossus of documentation on Python.org played a significant part as well. The procedure for our success is in the following pages.

How did we do it? -

- 1.** We began by reading a logic description file (LDF) and sorting the gates in the format desired.
- 2.** We created an array to store each line in the LDF into a separate element inside our array.

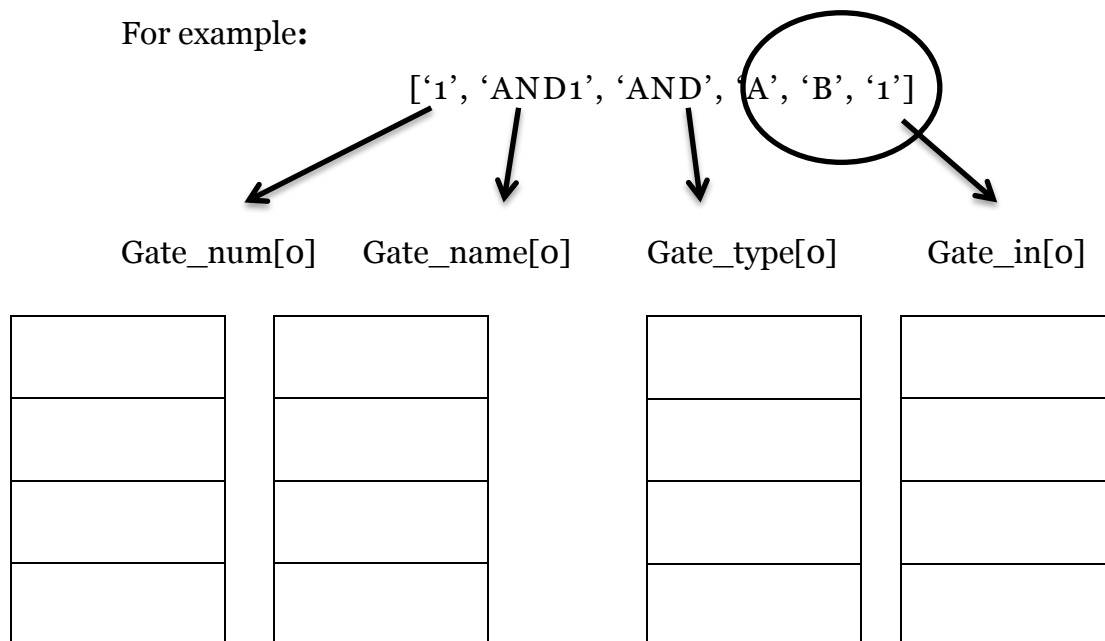
For Example:

file_in [0]

['1', 'XOR1', 'XOR', 'A', 'B']
['2', 'SUM', 'XOR', '1', 'C']
['3', 'AND2', 'AND', 'A', 'C']
['4', 'AND1', 'AND', 'A', 'C']
.....

- 3.** We used a series of Arrays to distinguish and store each specific element of each line in the input logic description file.

For example:



For Gate_in[0], we stored the letters and numbers from the 3rd element to the end of the arrays. Gate_in[0] elements will be inserted directly into our Gates.

We also created an array to store only the letters from the 3rd element to the end of the arrays. The number of letters will determine their corresponding values.

For example:

['1', 'AND1', 'AND', 'A', 'B', '1']



Letters[0]

4. Once we calculated the number of inputs from our Letters[o], we assigned binary values to each letter to create a truth table using a binary sequence generator and the dictionary function.

For Example:

BINARY SEQUENCE

[0, 0, 0]

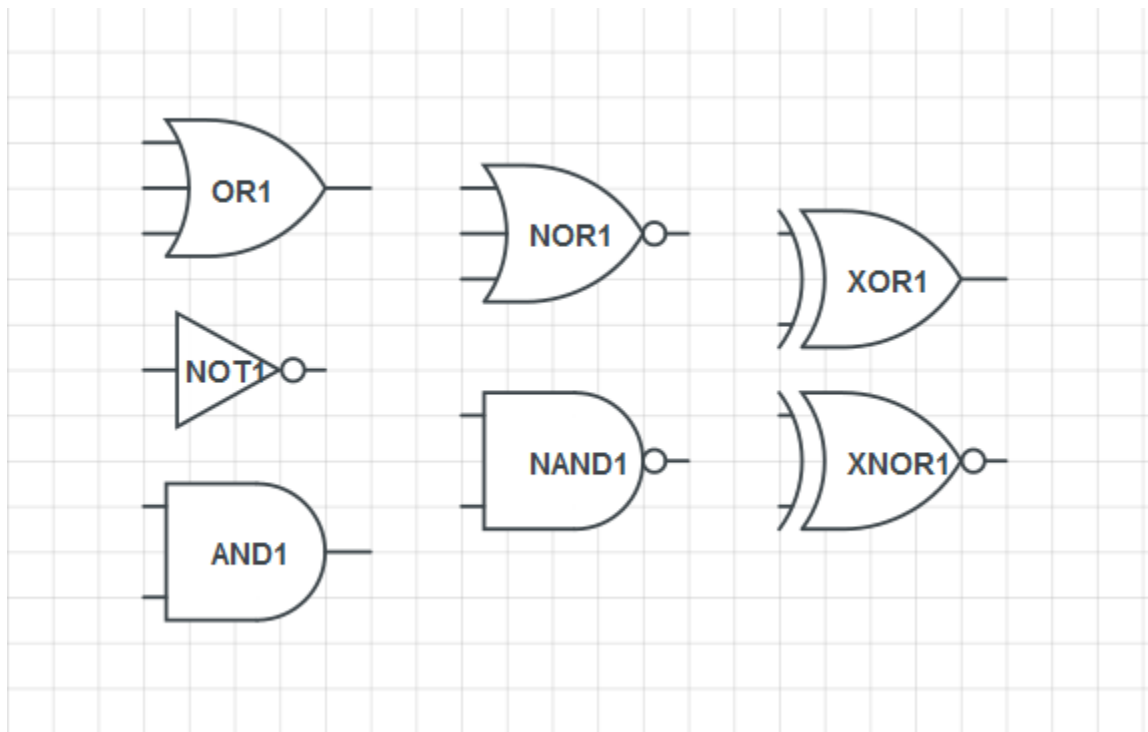
[0, 0, 1]

[0, 1, 0]

Letters [o]

A = 0
B = 0
C = 0
....

5. We created the 7 logic gates and tested the outputs.



6. We used the Dictionary Function in Python to map the gate types the corresponding functions. We instantiated each gate and connected them an element of the output array.


For example:

`Outputs [Gate_num[0]] = Gate[Gate_type [0]] (Gate_in [0])`

`Outputs ['1'] = Gate ['AND'] (['A', 'B' , '1'])`



`Outputs = {'1': Outputs [0], '2': Outputs [1], '3': Outputs [2]}`



`Gate = {'AND': AND}`

7. The final program asks the user for the input file and the desired outputs to be tracked. The program prints out the truth table and circuit listing into the console and into an output file.

Source Code -

Project1.v8

```
from collections import deque
#####

inputs = [0 for _ in range(50)] #input for each letter 1-26
output = [0 for _ in range(50)] #output for each gate 1 - 50

numbers = [] #numbers from 1-50 (number of gates)
for i in range(50):
    numbers.append(str(i+1))

letter_input = {}

alphabet = 'A B C D E F G H I J K L M N O P Q R S T U V W X Y Z'.split() #make a list of alphabet

for i in range(len(alphabet)): # map letters 'A' - 'Z' to element 0 - 25 of an array
    letter_input[alphabet[i]] = inputs[i]

outputs = {}

for i in range(50): # map output of each gate into an output array. Outputs 1-50 to Elements 0 - 49
    outputs[str(i+1)] = output[i]

file_in = [] # store each line of file in this array (going to be a 2-D array)

letters = [] # store the letters from 'A' - 'Z' array to keep track of inputs (used for controlling input values)

gate_in = [[] for _ in range(50)] #store inputs letters ('A','B'...) and input numbers ('1','2'...) into this array
                                   #to input into each gate (used for making gates)

out_list = [[] for _ in range(50)] # 2-D array that stores the output truth table in each element (each
                                   #element is the gate output of each gate)

binary_seq = [] #binary sequence of inputs ex. 000 - 111

#####

def AND(inputa): #and function
    if inputa[0] in numbers:
        l = outputs[inputa[0]]
    else:
        l = letter_input[inputa[0]]
    for i in range(1,len(inputa)):
        if inputa[i] in numbers:
```



```

        l = bool(l) & bool(outputs[inputa[i]])
    else:
        l = bool(l) & bool(letter_input[inputa[i]])
    return int(l)

def NAND(inputa): #nand function
    if inputa[0] in numbers:
        l = outputs[inputa[0]]
    else:
        l = letter_input[inputa[0]]
    for i in range(1,len(inputa)):
        if inputa[i] in numbers:
            l = bool(l) & bool(outputs[inputa[i]])
        else:
            l = bool(l) & bool(letter_input[inputa[i]])
    return int(not l)

def OR(inputa): #or function
    if inputa[0] in numbers:
        l = outputs[inputa[0]]
    else:
        l = letter_input[inputa[0]]
    for i in range(1,len(inputa)):
        if inputa[i] in numbers:
            l = bool(l) | bool(outputs[inputa[i]])
        else:
            l = bool(l) | bool(letter_input[inputa[i]])
    return int(l)

def NOR(inputa): #nor function
    if inputa[0] in numbers:
        l = outputs[inputa[0]]
    else:
        l = letter_input[inputa[0]]
    for i in range(1,len(inputa)):
        if inputa[i] in numbers:
            l = bool(l) | bool(outputs[inputa[i]])
        else:
            l = bool(l) | bool(letter_input[inputa[i]])
    return int(not l)

def XOR(inputa): #xor function
    if inputa[0] in numbers:
        l = outputs[inputa[0]]
    else:
        l = letter_input[inputa[0]]
    for i in range(1,len(inputa)):
        if inputa[i] in numbers:

```

```

        l = bool(l) ^ bool(outputs[inputa[i]])
    else:
        l = bool(l) ^ bool(letter_input[inputa[i]])
    return int(l)

def XNOR(inputa): #xnor function
    if inputa[0] in numbers:
        l = outputs[inputa[0]]
    else:
        l = letter_input[inputa[0]]
    for i in range(1,len(inputa)):
        if inputa[i] in numbers:
            l = bool(l) ^ bool(outputs[inputa[i]])
        else:
            l = bool(l) ^ bool(letter_input[inputa[i]])
    return int(not l)

def NOT(inputa): #Not function
    if inputa[0] in numbers:
        return int(not outputs[inputa[0]])
    else:
        return int(not letter_input[inputa[0]])

#####
#tie gates to each function
gate = {'AND': AND, 'NAND': NAND, 'OR': OR, 'NOR': NOR, 'XOR': XOR, 'XNOR': XNOR, 'NOT':NOT}

#####

filename = input("Enter a filename: ").strip()

if filename[-4:len(filename)] != '.txt': #if you leave out .txt it adds it in
    filename = filename + '.txt'

try:
    file = open(filename, "r") # (textfile, read)
except IOError:
    print('FILE DOES NOT EXIST')

lines = [line.upper() for line in file if line.strip()] #nevermind the white space

file.close()
lines.sort()

#-----
def int2bin(intvalue, digit): #turns an integer into a binary string (used to make input sequence for 'A'-'Z')
    binstr = ""; val = intvalue
    while val>0:

```

```

    if val % 2 == 0:
        binstr = '0' + binstr
    else:
        binstr = '1' + binstr
    val = val >> 1
    if len(binstr) < digit:
        binstr = '0' * (digit - len(binstr)) + binstr
    return binstr

def transpose(m): #transposes a matrix
    m1length=len(m)
    temp=[x for elem in m for x in elem] #flatten matrix
    m2length=len(temp)
    num = m2length//m1length #number of rows
    trans_m = [[x[i] for x in m] for i in range(num)] #transpose matrix
    return trans_m

#-----
file_in = [line.split() for line in lines] #tokenize each line and put it in a 2-D array

gate_num = [None for _ in range(len(file_in))] #initialize array length of gate_num with the amount of
                                                lines in file
gate_name = [None for _ in range(len(file_in))] #initialize array length of gate_name with the amount of
                                                lines in file
gate_type = [None for _ in range(len(file_in))] #initialize array length of gate_type with the amount of
                                                lines in file

for i in range(len(file_in)): #populate letters, gate_num, gate_name, gate_in, gate_type lists
    for j in range(3, len(file_in[i])):
        gate_in[i].append(file_in[i][j])
        if file_in[i][j] not in letters:
            if file_in[i][j] not in numbers:
                letters.append(file_in[i][j])
    gate_num[i] = file_in[i][0]
    gate_name[i] = file_in[i][1]
    gate_type[i] = file_in[i][2]

bin_seq = [list(tuple(int2bin(i, len(letters)))) for i in range(2**len(letters))] # conversion integer -->
                                                                                    binary string --> tuple --> list
bi_seq=[x for elem in bin_seq for x in elem] #flatten the list

bi_seq = deque(bi_seq) #turn flattened list into a double ended queue
bi_seq = deque(bi_seq) #turn flattened list into a double ended queue

```

```

truth_in = [[] for _ in range(len(letters))] #make a 2-D list to store input sequence per input

user_out = {}
for i in range(2**len(letters)):
    for k in range(len(letters)):
        truth_in[k].append(int(bi_seqt.popleft())) #pop inputs to the input sequence 2-D list
        letter_input[letters[k]] = int(bi_seq.popleft()) #change input values in each gate, changing outputs
    for m in range(len(file_in)):
        outputs[gate_num[m]] = gate[gate_type[m]](gate_in[m]) #create each gate
        out_list[m].append(outputs[str(m+1)]) #create a 2-D list to store output sequence per output
        user_out[gate_name[m]] = out_list[m] #tie in gate_name with that gate's output

print('-----Circuit Listing-----')
for w in lines:
    print(w,end="")
print()

##### USER INPUT #####
user_input = input('Enter the outputs you want to track seperated by a comma: ')
print()
tracked_out = []

user_input = user_input.upper().strip() #strip the user input and uppercase it
temp1 = user_input.split(',') #split the string based on commas and put it in the temp1 list
tracked_out = [i.strip() for i in temp1 if i.strip() in gate_name] #check if the user_inputs are valid

truth_matrix = truth_in #initialize truth_matrix list with the truth_in list

for i in tracked_out: #add each tracked output to the truth_matrix 2-D list
    truth_matrix = truth_matrix + [user_out[i]] #combine lists

in_out = letters + tracked_out #labels for the truth table

##### TRUTH TABLE #####
for i in in_out:
    print("%10s" %i, end="")
print()
print()

transposed = transpose(truth_matrix)
for i in range(len(transposed)):

```

```

    for j in transposed[i]:
        print('%10s' %j,end='')
    print()

#-----WRITING TO OUTPUT FILE-----

file = open("Output.txt", "w") #(new_textfile, write)

file.writelines('-----Circuit Listing-----\n')
for w in lines:
    file.writelines(w)

file.writelines('\n')

for i in in_out:
    file.writelines("%10s" %i)
    file.writelines(' ')
file.writelines('\n\n')

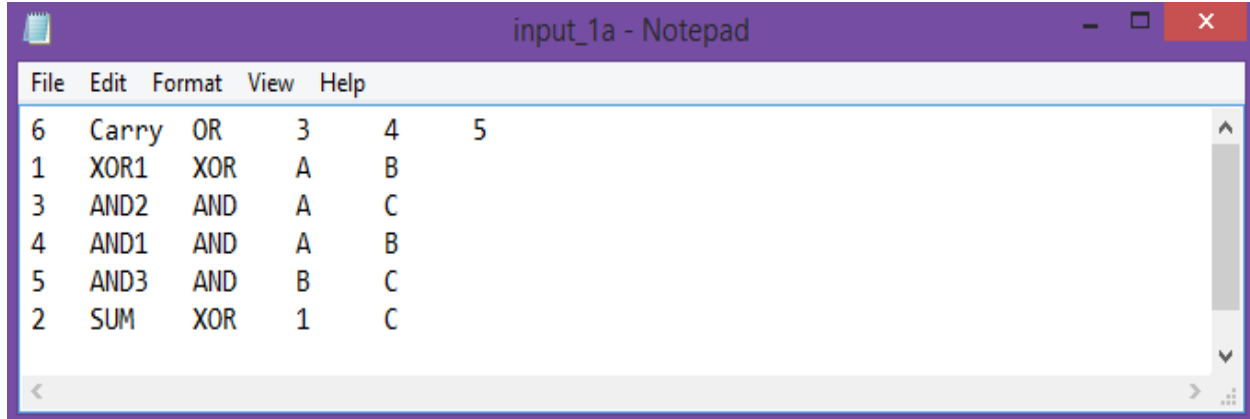
for i in range(len(transposed)):
    for j in transposed[i]:
        file.writelines('%10s ' %j)
    file.writelines('\n')

file.close()

```

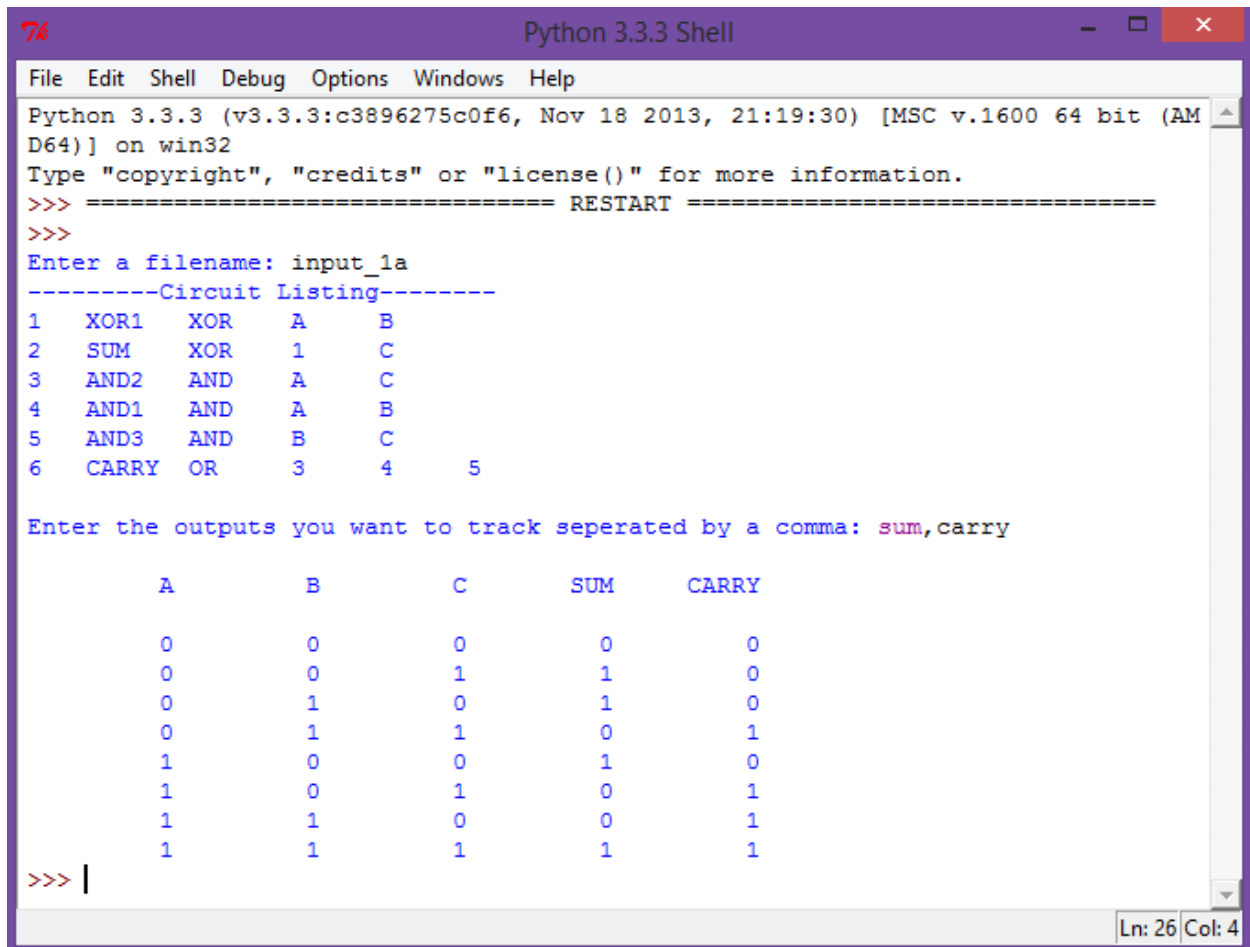
3-Case Studies -

1st LDF:



```
File Edit Format View Help
6 Carry OR 3 4 5
1 XOR1 XOR A B
3 AND2 AND A C
4 AND1 AND A B
5 AND3 AND B C
2 SUM XOR 1 C
```

Output:



```
Python 3.3.3 Shell
File Edit Shell Debug Options Windows Help
Python 3.3.3 (v3.3.3:c3896275c0f6, Nov 18 2013, 21:19:30) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter a filename: input_1a
-----Circuit Listing-----
1 XOR1 XOR A B
2 SUM XOR 1 C
3 AND2 AND A C
4 AND1 AND A B
5 AND3 AND B C
6 CARRY OR 3 4 5

Enter the outputs you want to track seperated by a comma: sum,carry

      A      B      C      SUM      CARRY
      0      0      0      0      0
      0      0      1      1      0
      0      1      0      1      0
      0      1      1      0      1
      1      0      0      1      0
      1      0      1      0      1
      1      1      0      0      1
      1      1      1      1      1

>>> |
```

2nd LDF:

```
File Edit Format View Help
3 Y0 AND 1 2
4 Y1 AND 1 B
5 Y2 AND 2 A
6 Y3 AND A B
1 Nota NOT A
2 Notb NOT B
```

Output:

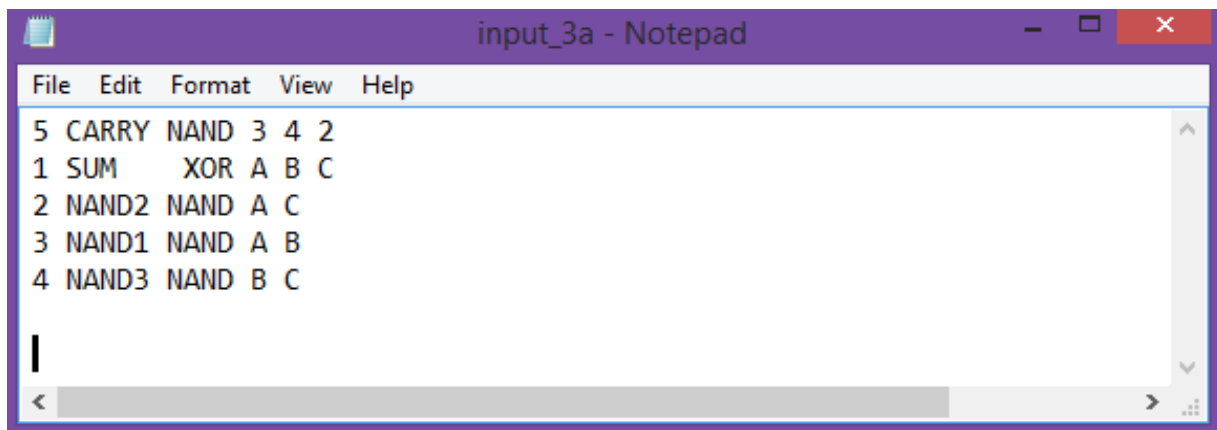
```
Python 3.3.3 Shell
File Edit Shell Debug Options Windows Help
Python 3.3.3 (v3.3.3:c3896275c0f6, Nov 18 2013, 21:19:30) [MSC v.1600 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter a filename: input_2a
-----Circuit Listing-----
1  NOTA  NOT  A
2  NOTB  NOT  B
3  Y0    AND  1  2
4  Y1    AND  1  B
5  Y2    AND  2  A
6  Y3    AND  A  B

Enter the outputs you want to track seperated by a comma: y0,y1,y2,y3

      A      B      Y0      Y1      Y2      Y3
      0      0      1      0      0      0
      0      1      0      1      0      0
      1      0      0      0      1      0
      1      1      0      0      0      1

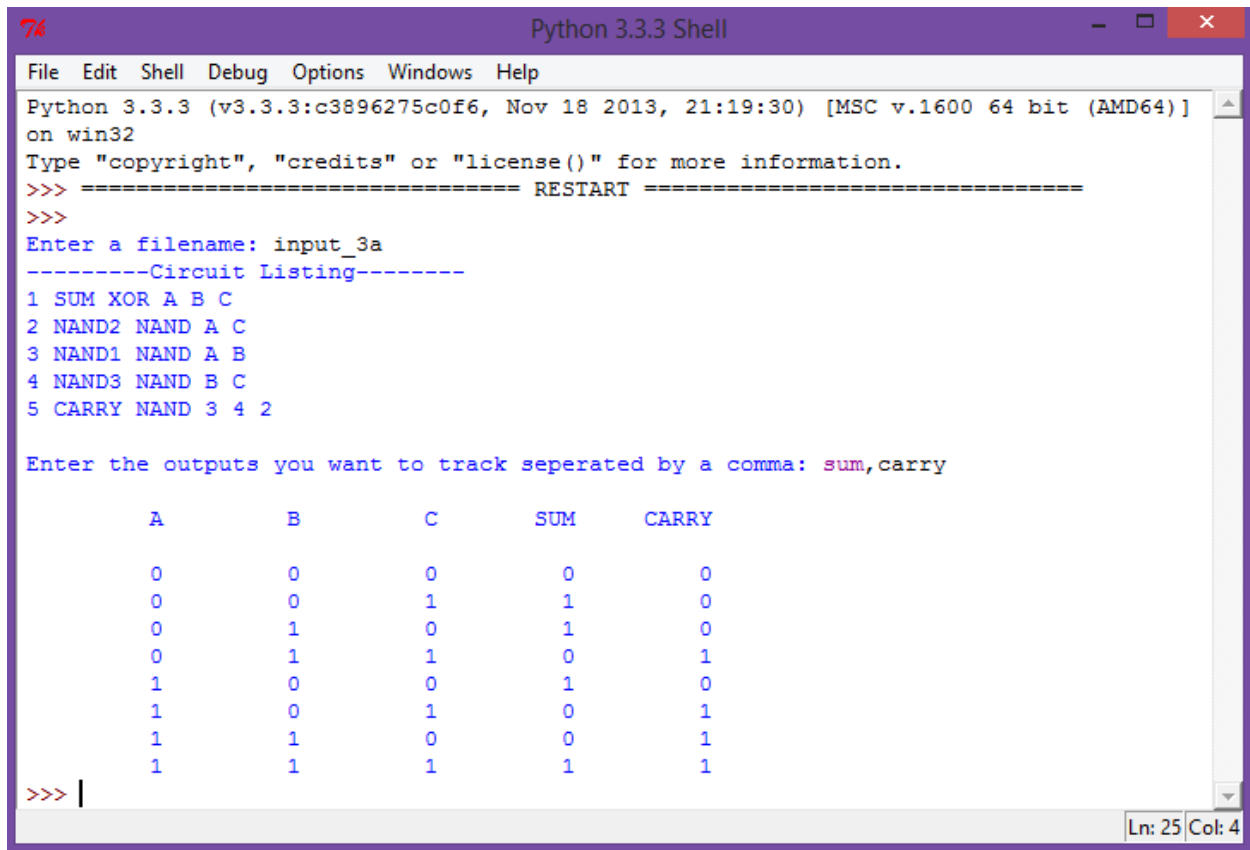
>>> |
```

3rd LDF:



```
File Edit Format View Help
5 CARRY NAND 3 4 2
1 SUM XOR A B C
2 NAND2 NAND A C
3 NAND1 NAND A B
4 NAND3 NAND B C
```

Output:



```
Python 3.3.3 Shell
File Edit Shell Debug Options Windows Help
Python 3.3.3 (v3.3.3:c3896275c0f6, Nov 18 2013, 21:19:30) [MSC v.1600 64 bit (AMD64)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Enter a filename: input_3a
-----Circuit Listing-----
1 SUM XOR A B C
2 NAND2 NAND A C
3 NAND1 NAND A B
4 NAND3 NAND B C
5 CARRY NAND 3 4 2

Enter the outputs you want to track seperated by a comma: sum,carry

      A      B      C      SUM      CARRY
      0      0      0      0      0
      0      0      1      1      0
      0      1      0      1      0
      0      1      1      0      1
      1      0      0      1      0
      1      0      1      0      1
      1      1      0      0      1
      1      1      1      1      1
>>> |
```

Ln: 25 Col: 4

Limitations -

Our program's limitations are as follows:

1. We are limited to 50 gates in the circuit.
2. We are limited to 26 inputs because of the length of the alphabet.

Note: This can easily be extended to distinguish Capital/lower letters.

3. We are not able to simulate flip flops due to sequential logic.

Accountability -

The lead software engineer for this project was John Valera followed by Omar Ramirez, Hang Zhao, Avis Kirakosyan, and Jaran Francis. The coding was done in a span of 3 weeks. The first week involved mostly brainstorming and planning by the entire group. The second week was reserved for coding our implementations and testing our program. For the third week, we focused on debugging our program and making it more efficient. We created 8 different revisions of our code throughout this process.

Conclusion -

Our biggest struggle was our lack of familiarity with the Python language. We had several ideas on how to approach the project, but we struggled when it was time to code our implementations. Once we planned our objectives and visually determined an approach, we were able to gain enough momentum to become confident in our programming. As we progressed in our project, we came to appreciate Python much more. We can improve our program by using the GUI modules in python to simulate our circuits graphically. We can also introduce error-checking. With more time, we can better debug our program to have it run more efficient. In conclusion, we found coding this project quite enjoyable.

References -

1. PYTHON.ORG
2. DR. CHANDRA'S LECTURES AND POWERPOINTS
3. JOSEPH TRAN'S "ZERO-DELAY COMBINATIONAL LOGIC SIMULATOR"
4. ELECTRIC-PYTHON.PYTHONGBLOGS.COM