Kozell YON-BROOKS

5727

51319

boolean logic circuit simulator

AQA Computer Science NEA

[Date]

# Contents

[Contents 2](#_Toc102735880)

[Analysis 4](#_Toc102735881)

[The Problem 4](#_Toc102735882)

[End Users 4](#_Toc102735883)

[Overview of Logic Gates 4](#_Toc102735884)

[Features Needed for GCSE Students 5](#_Toc102735885)

[Analysis of existing systems 7](#_Toc102735886)

[Acceptable Limitations 8](#_Toc102735887)

[Hardware and software constraints for Highdown 8](#_Toc102735888)

[Futureproofing for mobile development 8](#_Toc102735889)

[Platform, Programming Language, and Modules 9](#_Toc102735890)

[Kivy Framework 9](#_Toc102735891)

[Overview 9](#_Toc102735892)

[Kivy Language 9](#_Toc102735893)

[Objectives 9](#_Toc102735894)

[General 9](#_Toc102735895)

[Specific 10](#_Toc102735896)

[Design 12](#_Toc102735897)

[Overview 12](#_Toc102735898)

[Hierarchy Chart 12](#_Toc102735899)

[Input, Output, Process, Storage 13](#_Toc102735900)

[Object Oriented Design 13](#_Toc102735901)

[Logic Class Diagram 14](#_Toc102735902)

[User Interface 15](#_Toc102735903)

[Main Interface 15](#_Toc102735904)

[Gate Canvas 16](#_Toc102735905)

[Toolbar 16](#_Toc102735906)

[Menubar for adding components 16](#_Toc102735907)

[Menubar for changing tool 16](#_Toc102735908)

[Truth Table Popup 17](#_Toc102735909)

[Components 18](#_Toc102735910)

[Nodes 19](#_Toc102735911)

[Data Structures 20](#_Toc102735912)

[Logic Components 20](#_Toc102735913)

[Truth Table 21](#_Toc102735914)

[Algorithms 22](#_Toc102735915)

[Truth Table Generator: 22](#_Toc102735916)

[File Structure 25](#_Toc102735917)

[Technical Solution 26](#_Toc102735918)

[Truth Table 26](#_Toc102735919)

[Logic Board 28](#_Toc102735920)

[Logic Components 29](#_Toc102735921)

[GUI (Python) 37](#_Toc102735922)

[GUI (kvlang) 54](#_Toc102735923)

[Testing 62](#_Toc102735924)

[Code Testing 62](#_Toc102735925)

[Video Testing 63](#_Toc102735926)

[Code Testing Code 66](#_Toc102735927)

[Evaluation 67](#_Toc102735928)

# Analysis

## The Problem

GCSE Computer science is where a student’s introduction to logic gates may happen, this is a part of the curriculum but also helps to further a student’s logical thinking which can help in other areas of the subject, such as programming. A way to help me better understand logic gates were logic gate simulators, these simulators allowed me to create simple circuits using basic AND, OR, XOR, and NOT gates virtually. A simulator would allow the students to create circuits that give automatically give outputs without needing to be traced. However, the current simulators are too complex for what is needed for GCSE computer science and lack the teaching tools that my proposed system will include.

## End Users

The primary users of the logic gate simulators will be GCSE students studying computer science. However, as the computer science teachers will most likely introduce the students to this program and use it as part of their lessons, they will be considered as end-users as well.

## Overview of Logic Gates

Logic gates are a model of computation that take one or two inputs and returns a single output based on the gate's logical operation / Boolean function, they are the fundamentals of logical circuits and physical logic gates made of diodes and transistors are what allow computers to work. Logic gates can be combined to produce a certain output based on the inputs of the circuit. An AND gate (Figure 1) for example will take two inputs and return a True output if both inputs are True, and a False output otherwise. Inputs and outputs can be True or False as they are Boolean, this is usually represented as a 1 and 0 for True and False, respectively. Certain gates such as the Not gate will only need one input. There are other parts to the logic circuits besides the gates; switches, constant inputs, and clocks can provide initial inputs. Output can be handled by a simple ‘bulb’ that is on/lit for True and off for False. A more complex output such as a 4-bit digit would produce an integer output based on a binary sequence from 4 Boolean inputs.



Figure 1: AND Logic Gate

## Features Needed for GCSE Students

The truth table below (Figure 2) displays all the possible inputs and outputs in tabular form for the Logic operations AND, OR, XOR, and NOT. These are the only gates that are needed for the AQA course as per the specification (Figure 3).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** |  | **Output** | | | |
| A | B | AND | OR | XOR | NOT (Input only A) |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Figure 2: Example of Truth Table

Table

Description automatically generated

Figure 3: AQA GCSE Specification for Boolean Algebra 3.4.2

As these are the only gates needed for the AQA course, they will be the only gates included, simplifying the program for the students. My solution would be more focused on teaching, specifically what is needed for the AQA GCSE specification. I had a short conversation with Mr Flynn about what features would make my program relevant to GCSE students. These included:

* Abstracting logic gates that are not needed.
* Including a checklist to determine if the user has used each of the gates at least once.
* A truth table generator to display the results of a circuit as a truth table.
* Converting a written Boolean expression into a truth table.
* Displaying the Boolean expression for the created circuit.
* Allowing individuals to save and load circuits to and from their computer’s local storage.
* Comparing a user-made circuit to a given Boolean expression to check if they have created the circuit correctly

These are all the features I believe are needed for the student to understand and solve GCSE level questions such as the one shown in Figure 4 below.

A picture containing diagram

Description automatically generatedTable

Description automatically generated

Figure 4: Logic Gate question from AQA GCSE Computer Science June 2019 (8020/1)

## Analysis of existing systems

Figure 5, Figure 6, and Figure 7 below are examples of existing logic gate simulators.

Diagram

Description automatically generated

Figure 5: <https://academo.org/demos/logic-gate-simulator/>

A screenshot of a computer

Description automatically generated with medium confidence

Figure 6: <https://logic.ly/demo/>

Graphical user interface

Description automatically generated

Figure 7: <http://www.richardbowles.co.uk/resources/digital/tools/sim/sim.html>

|  |  |  |
| --- | --- | --- |
|  | **Advantages** | **Disadvantages** |
| **Figure 5** | * Lines between gates clearly show the connections of gates. | * Gates are behind a drop-down menu, making them harder to find. * No function to generate a truth table from the circuit created. |
| **Figure 6** | * Lines connecting gates are coloured to show the value of the gates they are coming out of. | * Gate selection is too complex for a GCSE student. |
| **Figure 7** | * Lines connecting gates are coloured to show the value of the gates they are coming out of. * Gates are clearly shown in the menu bar at top of the page. | * No dragging of gates around the canvas; makes repositioning difficult. * States of components do not dynamically update; the run button must be pressed to update them when a change is made. * Wires are placed manually; this is tedious and makes repositioning components difficult. |

## Acceptable Limitations

### Hardware and software constraints for Highdown

The program must run on school computers and therefore not be resource-intensive due to their performance. The school uses Windows 10 as its operating system, so the program currently does not need to run on any other OS.

### Futureproofing for mobile development

Because I would like the program to be run on mobile devices such as phones, the interface must be interacted with by only one button – the left-click on the mouse. This means each input must be a variation of a mouse click, release, drag, double-click, etc. This should allow easy transposing into touch inputs when altering the program for mobile development.

## Platform, Programming Language, and Modules

The program would run as either a web application or a windows desktop application. These choices are suitable as Highdown uses windows as the operating system on its computers, and all its computers have a browser and constant access to the internet.

A desktop application is a type of software that is directly installed onto the hard drive of the computer. It can be launched whenever, independent of other applications, i.e., it does not need a web browser to run within, like a web app. They also work regardless of internet connection to access it (unless the program itself requires it).

A Web application is a type of software application that is used through the internet via a web browser. The files are stored on a remote server, the backend processing is done remotely, and the application is only accessible via an internet connection and browser.

As a desktop application will provide all the features needed for the program, I have chosen it over the web app as it does not need a server to be stored/run on nor require a constant internet connection to access.

The programming language I use will somewhat depend on the platform that the program will be run on. For a web app, the programming languages I would use would be Python, JavaScript, and HTML; I would also need to be familiar with CSS. For a windows application, many languages would be suitable, including python. As python is a language that I am already familiar with, it is my top choice for programming language, because the program will be a desktop application only python is needed. Python version 3.9 is the minimum version of python that I will use.

For the GUI, python has an array of frameworks to help build user interfaces. These include PyQt5, tkinter, Kivy, wxPython, and PySimpleGUI. Kivy is currently my choice as it has many notable features and allows for the possibility of deploying the program on multiple platforms (Windows, macOS, iOS, Android). <https://kivy.org/#home>

## Kivy Framework

### Overview

Kivy is a free and open-source Python framework for developing mobile apps and other multitouch application software with a natural user interface. It is distributed under the terms of the MIT License, and can run on Android, iOS, Linux, macOS, and Windows. – [Wikipedia](https://en.wikipedia.org/wiki/Kivy_(framework)).

### Kivy Language

Kivy has its own declarative language, KvLang, which dedicated to describing the interface of the program. This means the python file does not need all the code for static interface items, this makes the python code shorter and easier to read.

## Objectives

### General

The general objective is to create a program that will allow a user to create logical gate circuits using draggable components that will evaluate and return an output based on the circuit. The program will also show a truth table of an expression either taken from a component or input manually. The program should be responsive, easy to use, compatible with the school computers, and have the gates that are on the AQA GCSE computer science specification.

### Specific

1. The program must create and evaluate logic gate circuits.
   1. Can have gates, switches, and/or outputs (the three referred to as components).
      1. Each component will have a state of either true or false.
      2. The gates will be AND, OR, NOT, and XOR (from GCSE spec).
      3. The components can connect to each other.
         1. The components will have input and output nodes that point to their connected gates.
      4. The components can return their state as an output.
      5. The gates can calculate their state based on the gate’s Boolean operator and its inputs through an ‘evaluate’ method.
      6. The switches can be flipped, changing their state from false to true or vice-versa.
      7. Each component stores the expression for the circuit up to where it is in the circuit.
         1. The expression will update when components are connected and disconnected
         2. The components operator will be correctly added to the expression.
   2. There will be a board class containing the components.
      1. The board will store the created components in an array.
      2. The board will tell the gates to connect and disconnect to and from each other.
      3. The board will remove the components from its array and disconnect everything it connected to.
2. There will be a truth table generator­.
   1. It will produce a truth table for a given expression.
      1. It can use the expression of an output component.
      2. It can use an input Boolean expression string.
   2. It will produce a list of input combinations based on the number of inputs.
   3. It will substitute each combination of inputs into the expression to get an output.
   4. It will list the outputs alongside the input combinations in a table.
3. There will be a graphical user interface.
   1. It will allow for the dynamic placing of the components onto a canvas.
      1. It will use mouse inputs to drag the components and move them about the canvas.
      2. Connection lines will be added between connected components.
         1. They will show the state of the component they’re exiting through their colour.
         2. They will not be straight lines; they will have a 90-degree bend at its horizontal centre to keep the canvas tidy and help with finding where the lines lead to.
   2. It will have a component toolbar that will have buttons that can add components to the canvas.
      1. The gates will have an indicator showing whether they have been used in the current session.
   3. It will have a tool toolbar that will determine what will happen when the components are interacted with depending on the tool selected.
      1. Connect tool: the program will tell the board to connect the selected components.
      2. Disconnect tool: the program will disconnect the selected components from each other.
      3. Move tool: the selected component can be moved about the canvas.
      4. Delete tool: the selected components will be deleted.
      5. Clear tool: All components will be deleted.
      6. Truth Table tool: the program will show a truth table based on the expression of the selected component.
         1. It will show a popup with a truth table and text input box that the user can enter their own expression into
   4. It will have a menu bar with options:
      1. Save, which saves the current circuit to a JSON file.
      2. Load, which loads a circuit from a JSON file.
      3. Quit, which exits the program.
         1. When clicked, a popup will appear asking if you are sure you want to exit.
   5. The user interface will be resizable with scrollable menus to allow access but keep the text legible.

# Design

## Overview

This design section will show how the system will look and operate, and demonstrate how the objectives will be implemented.

I have structured the project so that the logic of the circuits are independent of the user interface, this allows the logic code to be used in different project and to be tested without the whole program.

The main window will house everything the user can interact with. The gate canvas will have a board object, this will control the logic component objects. The board will tell them to connect and disconnect, but the components will store who they are connected to themselves. A logic component will be created when a visual component is created and stored in the component array of the board. When the canvas interacts with the visual component, the canvas will ask for its logic component reference and pass that to the board. The board will use that reference to get the logic component from its component array for use.

## Hierarchy Chart

Diagram

Description automatically generated

## Input, Output, Process, Storage

As Kivy can be used to develop for multiple operating systems, and I would like the program to be easily converted into an Android or iOS app, the user should be able to interact with the program with a single mouse button being pressed or released, and its movement around the screen, as this simulates what can be done on a touch screen.

|  |  |
| --- | --- |
| **Input** | **Output** |
| Mouse Inputs: Left Click Down, Mouse Move, Left Click Up. These inputs will be used for moving the components around a canvas and connecting or disconnecting them through dragging or clicking. (3.1.1)  Button Presses: Buttons around the user interface are what will control most of the program functionality, such as adding gates, selecting tools, and opening the truth table menu. (3.2, 3.3, 3.4)  Boolean expression input box: The truth table menu will display a truth table created for a user-entered Boolean expression; this will be a string entered into a text box which the truth table generator will then use as the input expression. (2.1.2) | Graphical Indicators: Most of the outputs will be done through the GUI, changing colours of components and lines based on their states, buttons showing as depressed/coloured to indicate when they are pressed, and borders around components when they are selected or being interacted with. (3.1)  Truth Table: A multiline string representation of a truth table produced by the truth table generator function will be displayed. The string will only be printed onto the screen. (2.4) |
| **Process** | **Storage** |
| Truth Table Generator: Will process a Boolean expression into a truth table. (2)  Boolean Evaluation: Each gate will use the values of their inputs with their respective Boolean operators to produce their output value. (1.1.5) | Saving and loading files to a centralised system is a stretch goal for this project and may not be implemented.  The primary goal is a simulation tool to support students investigating simple logic gate circuits they will develop as part of their studies. |

## Object Oriented Design

The program will be written with an object-oriented design pattern. The program will be split into modules for the UI and Logic. Within these modules classes are used to define the components of the program such as the individual logic components and the widgets of the kivy framework. The program will make use of object-oriented concepts such as classes, abstraction, inheritance and polymorphism.

## Logic Class Diagram

Figure 8 below shows the class diagram for the logic side of the program. The 4 gates (and, or, xor, not) will inherit from a parent class called Gate, the subclasses will contain the same attributes as the parent class but will overwrite some to fit the gates purpose, e.g. And\_Gate will overwrite the Gate classes currently empty attribute *type* with the string “and”. The gate subclasses will also overwrite the evaluate method so that it uses the correct boolean operator. As the not gate is different to the other 3 gates because of its single input rather than 2, it will overwrite other methods to account for only having 1 input.

There are also classes for switch and output objects, these have methods similar to the gate class, but with slight changes as they have only one node and no boolean evaluation method. Switch also has a flip method which will *flip* its state.

The switch, output and all gate classes all have a relationship with the Board class through composition. The board class will have 2 attributes, board\_id; an integer, and gates; an array of all components that currently exist. It will also have methods to add, remove, connect, and disconnect gates, a method to clear the board which will remove all components.Graphical user interface

Description automatically generated

Figure 8: Class diagram for circuit components

## User Interface

### Main Interface

When the program is initially opened, the first thing seen will be the full interactable program, allowing the user to start creating their circuits right away. The user interface should be simple and intuitive to use, no features should be hidden away or unclear on what their use is. Figure 9 shows a draft of how the main interface will look and the sections it will be split into.

Diagram

Description automatically generated

Figure 9: Main Graphical User Interface Design

### Gate Canvas

This is the canvas that will display Gates and Connection Lines. It will allow the gates two be placed anywhere within the canvas and the connection lines will dynamically update their positions to match where the components they are connected to are positioned.

### Toolbar

This will house the buttons to save, load, and open the truth table popup interface. It also has a quit button to quit the program, this will also bring up a popup asking for confirmation that the user wants to quit and that anything unsaved will be lost.

### Menubar for adding components

This sidebar will contain the components that the user can add to the canvas, it will have buttons that when pressed adds the component to the canvas. Each button will also have an indicator that shows whether the component has been used in the current session. There will also be labels alongside the components to show what they are but this isn’t shown in Figure 9.

### Menubar for changing tool

This is a sidebar with 5 buttons, the three top buttons are toggles that the user can choose between (one and only one of the three will always be on). The bottom two buttons, delete and clear, are not toggles and will immediately have an effect. Below is an explanation of each tool.

|  |  |
| --- | --- |
| **Tool** | **What it does.** |
| **Connect** | When toggled, the gate canvas will show the nodes of each component, the user may then drag from a node of one component to the node of another to connect them. This will create a connection line on the canvas linking the two components. The order of the connecting nodes (OUT 🡪 IN or IN 🡪 OUT) should not matter, the program will handle connecting them correctly. Connecting an out-node to an out-node or in-node to an in-node will not work. The components should not allow multiple connections to their in-nodes but should allow multiple connections from their out-nodes. |
| **Disconnect** | When toggled, the gate canvas will display the nodes of each component, the user may then click a node of a component and it will disconnect that component from the component that its node is connected to. When the components are disconnected, the connection lines connected to the clicked node will be removed. This means if the user clicks an out-node with multiple connections out of it, it will remove all connections. |
| **Move** | When toggled, the nodes are hidden, the user may click a component to select it, multiple components can be selected, click-and-drag a component to move it around the canvas, and double click the switch to flip its state. |
| **Delete** | When pressed, this will remove any selected components from the canvas. It will only work if the move tool is toggled. If not, it will toggle the move tool but will not remove the components. |
| **Clear** | When pressed, all components on the canvas will be deleted. Will toggle move tool when gate canvas is cleared. |

### Truth Table Popup

Truth Table Popup: This is where the user can generate truth tables. The expression input box will allow the user to input a Boolean expression, when the user presses the enter key or clicks the generate button a truth table for that expression will display in the truth table area. If a component is selected when the truth table button is pressed in the toolbar of the main interface, the expression box will automatically be populated with the expression of that component. Only one component must be selected, or it will default to empty on opening. There is also a cancel button to close the popup.

A picture containing diagram

Description automatically generated

Figure 10: Truth Table popup UI

### Components

These are all the icons for the components. The menu icons for switch and output will only appear in the menu bar for adding components. The gate components will only have one icon as they do not need to show their state. Switch and output will have Green and Red versions to show their states, output will also have an extra white icon for when its state is None (when the circuit is incomplete). They are all PNGs as they benefit from lossless compression and have transparency.

|  |  |
| --- | --- |
| **Component** | **Icons** |
| **Output:**  **(None, On, Off, Menu)** | **Shape, circle  Description automatically generated Shape, circle  Description automatically generated Shape, circle  Description automatically generated Chart, pie chart  Description automatically generated** |
| **Switch:**  **(On, Off, Menu)** | **Shape  Description automatically generated** |
| **XOR Gate** |  |
| **OR Gate** | **A picture containing icon  Description automatically generated** |
| **NOT Gate** | **Icon  Description automatically generated** |
| **AND Gate** | **Shape  Description automatically generated** |

Nodes: Component inputs and outputs are split into nodes, an AND gate for example will have 3 nodes, input\_node\_1, input\_node\_2, and out\_node, whereas a SWITCH or OUTPUT will have only an out\_node or in\_node\_1, respectively. The nodes are how the user will connect and disconnect components. The nodes will only appear when either the connect or disconnect tool is selected.

|  |  |
| --- | --- |
| **And Gate** | **And Gate with visible Nodes** |
| Shape  Description automatically generated | Icon  Description automatically generated |

### Nodes

Component nodes will be visually linked by connection lines; these lines will be drawn onto the canvas and will move with the components so that they always start on the out\_node of one component and end on the correct in\_node of the other component. Figure 11 below shows how the line’s colour will show the state of the gate it’s exiting, green for true, red for false, and black/white for an incomplete or invalid circuit. The lines will also shape themselves to always be made up of only horizontal or vertical parts. This will help with following the lines and organising the canvas.

Diagram, schematic

Description automatically generated

Figure 11: 3 Logic Circuits

## Data Structures

### Logic Components

The logic components in a circuit will be treated as a binary tree. The outputs or last component in a circuit would act as the root node, each node would represent a component and would have either 2 children (AND, OR, XOR), 1 child (NOT) or no children (SWITCH). This conforms to the binary tree structure.

Because each component can have multiple output connections and therefore each node can have multiple parents, the overall structure does not fully match a binary tree. However when the logic is evaluated the logic will work from the root component and will not need access to any output components of the components.

A preorder traversal recursive method will check if each component has all its input connections, if not it will return its current boolean output. If the component does have both input connections it will call a preorder traversal method on the left child/1st input. When the left child has returned some data, it will store it and then repeat for the right child/2nd input if there is one. Once all of its children have returned data, the component will use it in an evaluate method to get a boolean value and return that.

Below is a pseudocode representation of a node that has two inputs.

CLASS Node2:

instance.data = [None, None]

instance.left = None

instance.right = None

FUNCTION set\_data(data):

instance.data = data

FUNCTION set\_left(left\_node):

instance.left = left\_node

FUNCTION set\_right(right\_node):

instance.right = right\_node

FUNCTION preorder\_traverse():

IF (instance has no left or right) THEN:

RETURN instance.data

ELSE

instance.data[0] 🡨 instance.left.preorder\_traverse()

instance.data[1] 🡨 instance.right.preorder\_traverse()

RETURN instance.data

### Truth Table

The truth table will be stored using a dictionary. The dictionary will have keys that are strings, which is valid as strings are hashable, and values which are arrays of integers. The final key value pair will store the outputs of the expression.

Expression = “A or (B and C)”

Example\_Dictionary: dict[key, list[int]] = {

“A”:[0,0,0,0,1,1,1,1],

“B”:[0,0,1,1,0,0,1,1],

“C”:[0,1,0,1,0,1,0,1],

“OUT”:[0,0,0,1,1,1,1,1]

}

This translates into a truth table where each dictionary entry is a column of the table:

|  |  |  |  |
| --- | --- | --- | --- |
| **A or (B and C)** | | | |
| **A** | **B** | **C** | **OUT** |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

## Algorithms

### Truth Table Generator:

FUNCTION generateTruthTable(expression):

input\_expression 🡨 expression

expression 🡨 expression.lower()

expression 🡨 expression.replace("not", "2+~")

expression 🡨 expression.replace("and", "&")

expression 🡨 expression.replace("xor", "^")

expression 🡨 expression.replace("or", "|")

operators 🡨 ["2+~", "&", "^", "|", "(", ")"]

variables 🡨 []

temp\_exp 🡨 expression.replace("(", "")

temp\_exp 🡨 temp\_exp.replace(")", "")

temp\_exp 🡨 temp\_exp.split(" ")

FOR(i IN temp\_exp):

IF (i NOT IN operators) AND (i NOT IN variables):

variables.append(i)

num\_variables 🡨 len(variables)

temp 🡨 []

FOR(i 🡨 0 TO num\_variables):

temp.append(0)

FOR(i 🡨 0 TO num\_variables):

temp.append(1)

p 🡨 get\_permutations(temp, num\_variables)

variable\_permutations 🡨 []

FOR(i IN p):

IF i NOT IN variable\_permutations:

variable\_permutations.append(i)

dictionary = DICTIONARY()

FOR i,j IN ENUMERATE(variables):

FOR ii IN variable\_permutaions:

dictionary[j] = i[ii]

final\_dictionary 🡨 getTruthTableOutputs(expression, dictionary, LEN(variable\_permutations))

Explanation:

The function is given a Boolean expression as a string. The first 7 lines will format the string into an expression that can be used by the getTruthTableOutputs function.

An array called variables is created and populated with each variable in the expression, they are found by removing any brackets and splitting the string by its spaces into an array. Then each item is checked if it’s in the list of operators and if not, added to variables.

The number of variables is found from the length of the variables array and a temporary array will be created with that amount of 0s followed by the same amount of 1s. This is for the get\_permutations function to work correctly.

The get\_permutations will return a list of the permutations of the temp array as tuples of size num\_variables. E.g., for an expression with 2 variables:

get\_permutations([0,0,1,1], 3) will return

[(0, 0), (0, 1), (0, 1), (0, 0), (0, 1), (0, 1), (1, 0), (1, 0), (1, 1), (1, 0), (1, 0), (1, 1)]

The duplicated tuples are then removed to make the variable\_permuations array:

[(0, 0), (0, 1), (1, 0), (1, 1)]

This method was used as it produced a nicely ordered list of all possible value combinations used in a truth table.

A dictionary is then made with the variables as keys, and an array of their ordered values, found by combining the values with the same index in the variable\_permutations array tuples.

E.g. with variables A and B:

{‘A’:[0,0,1,1], ‘B’:[0,1,0,1]}

The formatted expression, dictionary, and the number of tuples in the variable\_permutations will be given to the getTruthTableOutputs function which will return the same dictionary but with an extra OUT key with its values.

A tuple of the final dictionary and the original expression will then be returned.

FUNCTION getTruthTableOutputs(expression, dictionary, num\_permutations):

truth\_dictionary 🡨 dictionary

list\_of\_outputs 🡨 []

FOR (i 🡨 0 TO num\_permutations):

expression\_with\_variables 🡨 expression

FOR (variable in truth\_dictionary):

value 🡨 INT\_TO\_STRING(dictionary[variable][i]))

expression\_with\_variables 🡨 expression\_with\_variables.replace(variable, value)

TRY:

output 🡨 EVALUATE(expression\_with\_variables)

EXCEPT SyntaxError as e:

return "Error"

list\_of\_outputs.append(output)

truth\_dictionary['OUT'] 🡨 list\_of\_outputs

RETURN truth\_dictionary

Explanation:

The function is given a formatted expression, a dictionary with a key-value pair of each variable and the ordered array of its values, and num\_permutations which is the number of values the variables will have.

The function creates a copy of the given dictionary called truth\_dictionary and an array called list\_of\_outputs. Within a for loop counting from i = 0 to the num\_permutations, a copy of the expression called expression will be created, a for loop iterating through the truth\_dictionary variables will then produce a value by converting the value at index i of the dictionary value with the variable as key to a string. All occurrences of that variable in the expression\_with\_variables string are replaced the current value the for loop has. This is done for all other variables so that the expression is now solveable. An evaluate function will treat the expression as code. If the expression cannot be correctly evaluated, most likely due to an invalid expression the function will return ‘ERROR’ and not continue. If the expression is evaluated, the output of it is added to a array called list\_of\_outputs. The process is repeated for all other values that the variable can be. The now completed list\_of\_outputs array is now added to the truth dictionary under the key ‘OUT’.

The updated truth\_dictionary is returned.

## File Structure

| CODE ¬

| Logic ¬

| TruthTable.py

| Components.py

| Board.py

| Gate Icons ¬

| (All component icon PNGs)

| GUI.py

| GUI.kv

This is how the file structure will be laid out for the program code. The GUI.py will import the code from each file in the Logic folder and access the icons of the component from the Gate Icons folder.

# Technical Solution

## Truth Table

from itertools import permutations

from os import system as sys

from typing import Union, Tuple

def get\_tt\_outputs(expression: str, dictionary: dict[str, list], num\_permutations: int) -> dict[str, list]:

    '''Adds OUT variable and values to incomplete truth table dictionary using by replacing variables in given expression.'''

    truth\_dictionary = dictionary

    list\_of\_outputs = []

    for i in range(num\_permutations):

        expression\_with\_variables = expression

        for variable in truth\_dictionary:

            expression\_with\_variables = expression\_with\_variables.replace(variable, str(dictionary[variable][i]))

        try:

            output = eval(expression\_with\_variables)

        except SyntaxError as e:

            print(e, expression\_with\_variables)

            return "Error"

        list\_of\_outputs.append(output)

    truth\_dictionary['OUT'] = list\_of\_outputs

    return truth\_dictionary

def generate\_truth\_table(expression: str) -> Union[Tuple[dict[str, list], str], str]:

    '''Returns a truth table in dictionary form'''

    input\_expression = expression

    expression = expression.lower()

    expression = expression.replace("not ", "2+~ ")

    expression = expression.replace("not", "2+~ ")

    expression = expression.replace("and", "&")

    expression = expression.replace("xor", "^")

    expression = expression.replace("or", "|")

    operators = ["2+~", "&", "^", "|", "(", ")"]

    variables = []

    temp\_exp = expression.replace("(", "").replace(")", "")

    for var in temp\_exp.split(" "):

        if var not in operators and var not in variables:

            variables.append(var)

    num\_variables = len(variables)

    temp = [0 for \_ in range(num\_variables)] + [1 for \_ in range(num\_variables)]

    perms = permutations(temp, num\_variables)

    variable\_permutations = []

    for perm in perms:

        if perm not in variable\_permutations:

            variable\_permutations.append(perm)

    dictionary = dict()

    for i, key in enumerate(variables):

        dictionary[key] = [perm[i] for perm in variable\_permutations]

    final\_dictionary = get\_tt\_outputs(expression, dictionary, len(variable\_permutations))

    if final\_dictionary == "Error":

        return "Invalid Input"

    else:

        return (final\_dictionary, input\_expression)

if \_\_name\_\_ == '\_\_main\_\_':

    '''For Testing'''

    sys('clear')

    exp = input('Here: ')

    print(generate\_truth\_table(exp))

    print()

## Logic Board

from logic.gates import \*

from logic.truth\_table import \*

class Board:

    '''Logic Board class.'''

    ID = 0

    def \_\_init\_\_(self) -> None:

        self.id = Board.ID

        Board.ID += 1

        self.gates = []

    def add\_gate(self, gate: Gate) -> None:

        '''Adds given component to gate array.'''

        self.gates.append(gate)

    def remove\_gate(self, gate: Gate) -> None:

        '''Removes given component from gate array.'''

        gate.disconnect\_all()

        self.gates.remove(gate)

    def connect\_gate(self, in\_gate: Gate, out\_gate: Gate, node: int) -> bool:

        '''Connects two given components together'''

        if node == 1:

            if in\_gate.connect\_node(out\_gate, 1):

                return True

        if node == 2:

            if in\_gate.get\_gate\_type() != "not" or in\_gate.get\_gate\_type() != "output":

                if in\_gate.connect\_node(out\_gate, 2):

                    return True

        return False

    def disconnect\_gate(self, in\_gate: Gate, out\_gate: Gate) -> bool:

        '''Disconnects two given components from eachother'''

        return in\_gate.disconnect\_node(out\_gate)

    def clear\_board(self) -> None:

        '''Removes all components from gate array'''

        for gate in self.gates[:]:

            self.remove\_gate(gate)

## Logic Components

from typing import Union # For type hinting

class Gate:

    '''Logic Gate component class. Parent class for logic gates.'''

    ID = 0

    def \_\_init\_\_(self) -> None:

        self.\_input\_nodes = [None, None]

        self.\_output\_nodes = []

        self.\_output = None

        self.\_type = ''

        self.\_expression = None

        self.name = ''

        self.id = Gate.ID

        Gate.ID += 1

    def evaluate(self) -> None:

        '''Not implemented in parent class.'''

        pass

    def has\_input(self, node=0) -> bool:

        '''Checks if nodes are connected to other components.'''

        if node == 0:

            return bool(self.\_input\_nodes[0] and self.\_input\_nodes[1])

        elif node == 1:

            return bool(self.\_input\_nodes[0])

        elif node == 2:

            return bool(self.\_input\_nodes[1])

    def connect\_node(self, gate, node) -> bool:

        '''Connects given component to given node of self'''

        if gate in self.\_input\_nodes:

            return False

        if node == 1:

            if not self.has\_input(1):

                self.\_input\_nodes[0] = gate

                gate.connect\_node(self, -1)

                self.update\_expression()

                return True

            else:

                return False

        elif node == 2:

            if not self.has\_input(2):

                self.\_input\_nodes[1] = gate

                gate.connect\_node(self, -1)

                self.update\_expression()

                return True

            else:

                return False

        elif node == -1:

            self.\_output\_nodes.append(gate)

            return True

    def disconnect\_node(self, gate) -> bool:

        '''Disconnects given component from self.'''

        if self.\_input\_nodes[0] is gate:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

            self.update\_expression()

            return True

        elif self.\_input\_nodes[1] is gate:

            self.\_input\_nodes[1].disconnect\_node(self)

            self.\_input\_nodes[1] = None

            self.update\_expression()

            return True

        elif gate in self.\_output\_nodes:

            self.\_output\_nodes.remove(gate)

            return True

        else:

            return False

    def disconnect\_all(self) -> None:

        '''Disconnects all inputs and outputs from self.'''

        if self.\_input\_nodes[0] != None:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

            self.update\_expression()

        if self.\_input\_nodes[1] != None:

            self.\_input\_nodes[1].disconnect\_node(self)

            self.\_input\_nodes[1] = None

            self.update\_expression()

        for gate in self.\_output\_nodes:

            gate.disconnect\_node(self)

    def \_process(self) -> None:

        '''Uses evaluate method to set output of self. If both input nodes empty, or any input nodes output is None, output is set to None.'''

        if self.has\_input():

            var1 = self.\_input\_nodes[0].get\_output()

            var2 = self.\_input\_nodes[1].get\_output()

            if var1 == None or var2 == None:

                self.\_output = None

            else:

                self.\_output = self.evaluate(var1, var2)

        else:

            self.\_output = None

    def update\_expression(self) -> None:

        '''Updates the boolean expression of self. Used when connecting or disconnecting components to/from self.'''

        if self.has\_input():

            if self.\_input\_nodes[0].get\_gate\_type() == 'switch':

                exp1 = self.\_input\_nodes[0].get\_expression()

            else:

                exp1 = f"({self.\_input\_nodes[0].get\_expression()})"

            if self.\_input\_nodes[1].get\_gate\_type() == 'switch':

                exp2 = self.\_input\_nodes[1].get\_expression()

            else:

                exp2 = f"({self.\_input\_nodes[1].get\_expression()})"

            self.\_expression = f"{exp1} {self.\_type} {exp2}"

        else:

            self.\_expression = None

    def get\_expression(self) -> str:

        '''Returns boolean expression of self.'''

        return self.\_expression

    def get\_gate\_type(self) -> str:

        '''Returns the type of component that self is.'''

        return self.\_type

    def get\_name(self) -> str:

        '''Returns the name of self. Name is string: type\_id.'''

        return self.name

    def get\_output(self) -> Union[int, None]:

        '''Calls process method, then return output of self.'''

        self.\_process()

        return self.\_output

class And\_Gate(Gate):

    '''Logic And\_Gate component class, child of Gate class.'''

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        self.\_type = 'and'

        self.name = f"{self.\_type}\_{str(self.id)}"

    def evaluate(self, var1, var2) -> int:

        '''Return the result of an and comparison bewtween two variables.'''

        return int(var1 and var2)

class Or\_Gate(Gate):

    '''Logic Or\_Gate component class, child of Gate class.'''

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        self.\_type = 'or'

        self.name = f"{self.\_type}\_{str(self.id)}"

    def evaluate(self, var1, var2) -> int:

        '''Return the result of an or comparison bewtween two variables.'''

        return int(var1 or var2)

class Xor\_Gate(Gate):

    '''Logic Xor\_Gate component class, child of Gate class.'''

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        self.\_type = 'xor'

        self.name = f"{self.\_type}\_{str(self.id)}"

    def evaluate(self, var1, var2) -> int:

        '''Return the result of an xor comparison bewtween two variables.'''

        return int((var1 and not var2) or (not var1 and var2))

class Not\_Gate(Gate):

    '''Logic Not\_Gate component class, child of Gate class.'''

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        self.\_type = 'not'

        self.name = f"{self.\_type}\_{str(self.id)}"

    def evaluate(self, var1) -> int:

        '''Return the result of a not operation on a variable.'''

        return int((not var1))

    def has\_input(self) -> bool:

        '''Checks if input node is connected to another component.'''

        return bool(self.\_input\_nodes[0])

    def \_process(self) -> None:

        '''Uses evaluate method to set output of self. If has no input node, or input nodes output is None, output is set to None.'''

        if self.has\_input():

            var1 = self.\_input\_nodes[0].get\_output()

            if var1 == None:

                self.\_output = None

            else:

                self.\_output = self.evaluate(var1)

        else:

            self.\_output = None

    def connect\_node(self, gate, node) -> bool:

        '''Connects given component to given node of self.'''

        if gate in self.\_input\_nodes:

            return False

        if node == 1:

            if not self.has\_input():

                self.\_input\_nodes[0] = gate

                gate.connect\_node(self, -1)

                self.update\_expression()

                return True

            else:

                return False

        elif node == -1:

            self.\_output\_nodes.append(gate)

            return True

    def disconnect\_node(self, gate) -> bool:

        '''Disconnects given component from self.'''

        if self.\_input\_nodes[0] is gate:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

            self.update\_expression()

            return True

        elif gate in self.\_output\_nodes:

            self.\_output\_nodes.remove(gate)

            return True

        else:

            return False

    def disconnect\_all(self) -> None:

        '''Disconnects all inputs and outputs from self.'''

        if self.\_input\_nodes[0] != None:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

            self.update\_expression()

        for gate in self.\_output\_nodes:

            gate.disconnect\_node(self)

    def update\_expression(self) -> None:

        '''Updates the boolean expression of self. Used when connecting or disconnecting components to/from self.'''

        if self.has\_input():

            self.\_expression = str(f"{self.\_type}({self.\_input\_nodes[0].get\_expression()})")

        else:

            self.\_expression = None

class Switch:

    '''Logic Switch component class.'''

    ID = 0

    def \_\_init\_\_(self) -> None:

        self.\_output = 0

        self.\_output\_nodes = []

        self.\_type = 'switch'

        self.id = chr(Switch.ID+97)

        Switch.ID += 1

        self.name = (f"{self.\_type}\_{self.id}")

    def connect\_node(self, gate, node) -> bool:

        '''Connects given component to given node of self.'''

        if gate not in self.\_output\_nodes and node == -1:

            self.\_output\_nodes.append(gate)

            return True

        return False

    def disconnect\_node(self, gate) -> bool:

        '''Disconnects given component from self.'''

        if gate in self.\_output\_nodes:

            self.\_output\_nodes.remove(gate)

            return True

        return False

    def disconnect\_all(self) -> None:

        '''Disconnects all inputs and outputs from self.'''

        for gate in self.\_output\_nodes:

            gate.disconnect\_node(self)

    def flip(self) -> None:

        '''Changes output of self from 0 to 1 or 1 to 0.'''

        self.\_output = int(not(self.\_output))

    def get\_name(self) -> str:

        '''Returns the name of self. Name is string: type\_id.'''

        return self.name

    def get\_gate\_type(self) -> str:

        '''Returns the type of self that self is.'''

        return self.\_type

    def get\_expression(self) -> str:

        '''Return character id of self.'''

        return self.id

    def get\_output(self) -> Union[int, None]:

        '''Returns output of self.'''

        return self.\_output

class Output:

    '''Logic Output component class.'''

    ID = 65

    def \_\_init\_\_(self) -> None:

        self.\_output = None

        self.\_input\_nodes = [None]

        self.\_type = 'output'

        self.id = chr(Output.ID)

        Output.ID += 1

        self.name = (f"{self.\_type}\_{self.id}")

    def \_process(self) -> None:

        '''Sets output of self to output of input node. If input node is empty, output is set to None.'''

        try:

            self.\_output = self.\_input\_nodes[0].get\_output()

        except AttributeError as e:

            print(e, len(self.\_input\_nodes))

            self.\_output = None

    def connect\_node(self, gate, node) -> bool:

        '''Connects given component to given node of self.'''

        if self.\_input\_nodes[0] == None:

            self.\_input\_nodes[0] = gate

            gate.connect\_node(self, -1)

            return True

        else:

            return False

    def disconnect\_node(self, gate) -> bool:

        '''Disconnects given component from self.'''

        if self.\_input\_nodes[0] is gate:

            self.\_input\_nodes[0] = None

            gate.disconnect\_node(self)

            return True

        return False

    def disconnect\_all(self) -> None:

        '''Disconnects all inputs and outputs from self.'''

        if self.\_input\_nodes[0] == None:

            pass

        else:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

        self.\_process()

    def get\_expression(self) -> str:

        '''Return boolean expression of input component.'''

        return self.\_input\_nodes[0].get\_expression()

    def get\_name(self) -> str:

        '''Returns the name of self. Name is string: type\_id.'''

        return self.name

    def get\_gate\_type(self) -> str:

        '''Returns the type of component that self is.'''

        return self.\_type

    def get\_output(self) -> Union[int, None]:

        '''Calls process method, then return output of self.'''

        self.\_process()

        return self.\_output

## GUI (Python)

import sys

import platform

import kivy

kivy.require('2.0.0')

from kivy.app import App

from kivy.clock import Clock

from kivy.config import Config

from kivy.core.window import Window

from kivy.graphics import Color, Line

from kivy.lang import Builder

from kivy.properties import ListProperty, StringProperty

from kivy.uix.behaviors import DragBehavior

from kivy.uix.floatlayout import FloatLayout

from kivy.uix.image import Image

from kivy.uix.popup import Popup

from kivy.uix.widget import Widget

from kivy.utils import platform

from kivy.utils import rgba

from tabulate import tabulate

from logic.board import Board

from logic.gates import \*

from logic.truth\_table import \*

Window.maximize()

class ExitPopup(Popup):

    '''Popup that opens when quit button is pressed.'''

    def close\_window(self):

        '''Exits program.'''

        sys.exit()

class TruthPopup(Popup):

    '''Popup for truth table generator.'''

    FONT = StringProperty()

    def \_\_init\_\_(self, \*args, \*\*kwargs):

        super().\_\_init\_\_(\*args, \*\*kwargs)

        if platform == 'linux':

            self.FONT = "FreeMonoBold"

        else:

            self.FONT = "LUCON"

    def open(self, root, \*args, \*\*kwargs):

        '''Overwrite of kivy method, inserts selected components expression into input field.'''

        super().open(\*args, \*\*kwargs)

        selected\_gate = root.ids["gateCanvas"].get\_selected\_gates()

        if len(selected\_gate) != 1:

            text = 'Enter Boolean Expression Here'

        else:

            text = selected\_gate[0].logic\_gate.get\_expression()

        self.ids["truth\_input"].text = text

        self.generate()

    def generate(self):

        '''Using expression in input field, generates a truth table and diplays it.'''

        input\_expression = self.ids["truth\_input"].text

        out = generate\_truth\_table(input\_expression)

        if out == "Invalid Input":

            if input\_expression == "Enter Boolean Expression Here":

                self.ids["truth\_label"].text = "Truth table will appear here"

            else:

                self.ids["truth\_label"].text = "Invalid Input."

        else:

            out2 = tabulate(out[0], headers=out[0].keys(), tablefmt="pretty")

            self.ids["truth\_label"].text = f"{out[1]}\n{out2}"

class ConnectionLine(Widget):

    '''Class for lines that connects components together'''

    color\_state = ListProperty()

    def \_\_init\_\_(self, out\_gate, in\_gate, in\_node, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.color\_states = {None:(1,1,1,1), 0:(1,0,0,1), 1:(0,1,0,1)}

        self.color\_state = (1,1,1,1)

        self.out\_gate = out\_gate

        self.in\_gate = in\_gate

        self.in\_node = in\_node

        self.points = []

        self.state = None

        self.inline = Line(points=self.points, width=1.5, cap='round', joint='round')

        self.outline = Line(points=self.points, width=4, cap='round', joint='round')

        self.canvas.add(self.inline)

        self.canvas.before.add(Color(rgba=(0,0,0,1)))

        self.canvas.before.add(self.outline)

        self.update\_pos()

        self.update\_state()

    def get\_gates(self):

        '''Returns tuple: (component line is exiting, component line is entering)'''

        return (self.out\_gate, self.in\_gate)

    def get\_nodes(self):

        '''Returns tuple: (component line is exiting, node its exiting), (component line is entering, node its entering))'''

        return ((self.out\_gate, -1), (self.in\_gate, self.in\_node))

    def get\_turn\_points(self, out\_node\_pos, in\_node\_pos):

        '''Returns the extra points to make the line the shape it is <<<<<'''

        x1 = out\_node\_pos[0]

        x2 = in\_node\_pos[0]

        y1 = out\_node\_pos[1]

        y2 = in\_node\_pos[1]

        x\_mid = (x1 + x2)/2

        turn1 = [x\_mid, y1]

        turn2 = [x\_mid, y2]

        return turn1 + turn2

    def update\_state(self):

        '''Updates colour of line to match the output of the component its exiting.'''

        self.state = self.out\_gate.get\_state()

        self.color\_state = self.color\_states[self.state]

    def update\_pos(self):

        '''Updates the points of the line to match the position of the components its connect to.'''

        out\_node\_pos = self.out\_gate.get\_node\_pos(-1)

        in\_node\_pos = self.in\_gate.get\_node\_pos(self.in\_node)

        self.points = out\_node\_pos + self.get\_turn\_points(out\_node\_pos, in\_node\_pos) + in\_node\_pos

        self.outline.points = self.points

        self.inline.points = self.points

class GateCanvas(FloatLayout):

    '''Canvas where components and connection lines are placed onto'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.connection\_lines = []

        self.in\_connection = None

        self.out\_connection = None

        self.gates = []

        self.tool = "move"

        self.board = Board()

        self.gate\_dict = {

            "and":DragAndGate,

            "or":DragOrGate,

            "not":DragNotGate,

            "xor":DragXorGate,

            "switch":DragSwitch,

            "output":DragOutput,

            "clock":DragClock

        }

    def add\_connection\_line(self, out\_gate, in\_gate, in\_node):

        '''Creates connection line.'''

        line = ConnectionLine(out\_gate, in\_gate, in\_node)

        self.connection\_lines.append(line)

        self.add\_widget(line)

    def update\_connection\_lines(self):

        '''Updates the postions of all connection lines on the canvas'''

        for line in self.connection\_lines[:]:

            line.update\_pos()

    def update\_states(self):

        '''Updates the states of all components and connection lines on the canvas'''

        for gate in self.gates:

            gate.update\_state()

        for line in self.connection\_lines:

            line.update\_state()

    def set\_tool(self, tool):

        '''Sets the tool, can be move, connect, disconnect'''

        self.tool = tool

        if tool is "move":

            for gate in self.gates[:]:

                gate.hide\_nodes()

        else:

            for gate in self.gates[:]:

                gate.show\_nodes()

                self.deselect\_gates()

    def on\_touch\_down(self, touch):

        '''Checks if there is a touch down within the canvas. If touch is double click, set tool to move. Calls a method based on the current tool if single click.'''

        if self.collide\_point(\*touch.pos):

            if touch.is\_double\_tap and self.tool != "move":

                self.root.set\_tool("move")

                return True

            if self.tool == "connect":

                return self.connect\_down(touch)

            if self.tool == "disconnect":

                return self.disconnect\_down(touch)

            if self.tool == "move":

                return self.move\_down(touch)

    def on\_touch\_up(self, touch):

        '''Checks if there is a touch up within the canvas, Calls a method based on the cuurent tool if there is'''

        if self.collide\_point(\*touch.pos):

            if self.tool == "connect":

                return self.connect\_up(touch)

            if self.tool == "disconnect":

                pass

            if self.tool == "move":

                pass

        return super().on\_touch\_up(touch)

    def connect\_down(self, touch):

        '''Checks for node collision on touch down. If collision replace either out\_connection or in\_connection attribute with (node, gate) tuple'''

        self.deselect\_gates()

        for gate in self.gates[:]:

            if node := gate.get\_node\_collide(touch):

                if node == -1:

                    self.out\_connection = (node, gate)

                else:

                    self.in\_connection = (node, gate)

                gate.select\_node(node)

                return True

        return True

    def connect\_up(self, touch):

        '''Checks for node collision on touch up. If collision, replace either out\_connection or in\_connection attribute with (node, gate) tuple.

        If out\_connection and in\_connection both have values, connects the component nodes together'''

        for gate in self.gates[:]:

            if node := gate.get\_node\_collide(touch):

                if node == -1:

                    self.out\_connection = (node, gate)

                else:

                    self.in\_connection = (node, gate)

                gate.select\_node(node)

                if self.in\_connection is None:

                    print("Can't connect output to output")

                elif self.out\_connection is None:

                    print("Cant connect input to input")

                else:

                    in\_gate = self.in\_connection[1]

                    out\_gate = self.out\_connection[1]

                    in\_node = self.in\_connection[0]

                    if self.board.connect\_gate(in\_gate.get\_logic\_gate(), out\_gate.get\_logic\_gate(), node=in\_node):

                        self.add\_connection\_line(out\_gate, in\_gate, in\_node)

                    self.update\_states()

        for gate in self.gates[:]:

            gate.deselect\_nodes()

        self.out\_connection = None

        self.in\_connection = None

        return True

    def move\_down(self, touch):

        '''Calls on\_touch\_down method for children. If canvas clicked instead of a child, deselects all gates.'''

        for child in self.gates[::-1]:

            if child.dispatch('on\_touch\_down', touch):

                return True

        if not self.collide\_point(\*touch.pos):

            return False

        self.deselect\_gates()

        return True

    def disconnect\_down(self, touch):

        '''Checks for node collision on touch down. If collision, disconnect clicked node from any nodes its connected to and remove connection lines.'''

        self.deselect\_gates()

        for gate in self.gates[:]:

            if node := gate.get\_node\_collide(touch):

                if node == -1:

                    for line in self.connection\_lines:

                        gate\_node\_tuple = line.get\_nodes()

                        if gate\_node\_tuple[0][0] == gate:

                            gate2 = gate\_node\_tuple[1][0]

                            self.board.disconnect\_gate(gate2.get\_logic\_gate(), gate.get\_logic\_gate())

                            self.connection\_lines.remove(line)

                            self.remove\_widget(line)

                elif node == 1 or node == 2:

                    for line in self.connection\_lines:

                        gate\_node\_tuple = line.get\_nodes()

                        if gate\_node\_tuple[1] == (gate,node):

                            gate2 = gate\_node\_tuple[0][0]

                            self.board.disconnect\_gate(gate.get\_logic\_gate(), gate2.get\_logic\_gate())

                            self.connection\_lines.remove(line)

                            self.remove\_widget(line)

        self.update\_states()

        return True

    def get\_selected\_gates(self):

        '''Returns a list of all components that are selected.'''

        return [child for child in self.gates[:] if child.is\_selected()]

    def deselect\_gates(self):

        '''Deselects all selected components.'''

        [child.deselect() for child in self.gates[:]]

    def add\_gate(self, gate\_type):

        '''Creates new visual component and adds it to canvas, adds logic component to logic board.'''

        new\_gate = self.gate\_dict[gate\_type](parent\_rect=(self.x, self.y, self.width, self.height))

        self.add\_widget(new\_gate)

        self.gates.append(new\_gate)

        new\_gate.root = self.root

        self.board.add\_gate(new\_gate.get\_logic\_gate())

        new\_gate.update\_state()

    def delete\_gate(self):

        '''Deletes selected components from canvas and logic board, also removes any connection lines they are connected to.'''

        if not self.tool == "move":

            self.root.set\_tool("move")

        else:

            for gate in self.get\_selected\_gates():

                #This is probably a terrible way to do this, so find a better one

                for line in self.connection\_lines[:]:

                    if gate in line.get\_gates():

                        self.connection\_lines.remove(line)

                        self.remove\_widget(line)

                        del line

                self.board.remove\_gate(gate.get\_logic\_gate())

                self.remove\_widget(gate)

                self.gates.remove(gate)

                del gate

                self.update\_states()

    def clear\_canvas(self):

        '''Deletes all components from canvas and logic board'''

        self.board.clear\_board()

        self.clear\_widgets()

        self.gates = []

class DragGate(DragBehavior, FloatLayout):

    '''Visual components parent class. DOCSRINGS NOT DONE'''

    def \_\_init\_\_(self, parent\_rect, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        '''Sets kivy properties'''

        self.drag\_rectangle = parent\_rect

        self.drag\_timeout = 500

        self.drag\_distance = 5

        self.allow\_stretch = True

        self.size\_hint = None, None

        self.size = (100,100)

        self.pos = (parent\_rect[0]+parent\_rect[2])/2, (parent\_rect[1]+parent\_rect[3])/2

        '''Sets image of component'''

        self.img = Image(pos = self.pos, size\_hint = (1,1))

        self.add\_widget(self.img)

        '''Initialises the nodes of the component and adds them to node array'''

        self.nodes = []

        self.nodes\_init()

        '''Creates the selection border and adds to component.'''

        self.border = Line(rounded\_rectangle = (self.x, self.y, self.width, self.height, 10))

        self.canvas.add(Color(0,0,0,1))

        self.canvas.add(self.border)

        '''Creates a parallel logic version of the component. None in parent class.'''

        self.logic\_gate = None

        self.state = None

        '''Creates dragged attribute and selects the component.'''

        self.dragged = False

        self.select()

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.in\_node\_1 = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.in\_node\_2 = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.out\_node = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.nodes.append(self.in\_node\_1)

        self.nodes.append(self.in\_node\_2)

        self.nodes.append(self.out\_node)

        self.add\_widget(self.in\_node\_1)

        self.add\_widget(self.in\_node\_2)

        self.add\_widget(self.out\_node)

        self.hide\_nodes()

    def get\_node\_pos(self, node):

        '''Return the coordinates of the given node.'''

        return self.get\_node(node).center

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.in\_node\_1.center=(self.x, self.center\_y+26)

        self.in\_node\_2.center=(self.x, self.center\_y-26)

        self.out\_node.center=(self.right, self.center\_y+2)

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.in\_node\_1:

                    return 1

                elif node == self.in\_node\_2:

                    return 2

                elif node == self.out\_node:

                    return -1

        return False

    def select\_node(self, node\_index):

        '''Selects the given node.'''

        node = self.get\_node(node\_index)

        node.color = (0.3, 0.7, 0.7, 1)

    def deselect\_node(self, node\_index):

        '''Deselects the given node.'''

        node = self.get\_node(node\_index)

        node.color = (1, 1, 1, 1)

    def deselect\_nodes(self):

        '''Deselects all node of component.'''

        for node in self.nodes:

            node.color = (1, 1, 1, 1)

    def get\_node(self, index):

        '''Returns the node object for given node index.'''

        if index == 1:

            return self.in\_node\_1

        if index == 2:

            return self.in\_node\_2

        if index == -1:

            return self.out\_node

    def on\_pos(self, \*args, \*\*kwargs):

        '''Listens for change of kivy pos property. Updates position of image, nodes and selection border. Also tells board to update position of connection lines.'''

        try:

            self.border.rounded\_rectangle = (self.x, self.y, self.width, self.height, 10)

            self.img.pos = self.pos

            self.update\_nodes()

            self.parent.update\_connection\_lines()

        except AttributeError as e:

            print("Gate not finished initalising", e)

    def is\_selected(self):

        '''Returns True if component is currently selected.'''

        return self.selected

    def show\_nodes(self):

        '''Shows the components nodes.'''

        for node in self.nodes:

            node.opacity = 1

            node.disabled = False

        self.update\_nodes()

    def hide\_nodes(self):

        '''Hides the components nodes.'''

        for node in self.nodes:

            node.opacity = 0

            node.disabled = True

    def select(self):

        '''Selects the component.'''

        self.selected = True

        self.border.width = 2

    def deselect(self):

        '''Deselects the component'''

        self.selected = False

        self.border.width = 0.0001

    def on\_touch\_down(self, touch):

        '''Checks if user has clicked node, sets dragged attribute to false.'''

        if self.collide\_point(\*touch.pos):

            self.dragged = False

        return super().on\_touch\_down(touch)

    def on\_touch\_move(self, touch):

        '''Checks if the user drags the component when clicked. Sets dragged attribute to true, selectes component updates position if user drags within the drag\_timeout time and a distance greater than the drag\_distance distance.'''

        if touch.grab\_current is not self:

            return super().on\_touch\_move(touch)

        self.select()

        self.dragged = True

        if self.x < self.parent.x+5:

            self.x = self.parent.x+5

        elif self.y < self.parent.y+5:

            self.y = self.parent.y+5

        elif self.right > self.parent.right-5:

            self.right = self.parent.right-5

        elif self.top > self.parent.top-5:

            self.top = self.parent.top-5

        else:

            return super().on\_touch\_move(touch)

    def on\_touch\_up(self, touch):

        '''When user releases click, checks dragged attribute, if true component is deselected, if false and the component is already selected the component is also deselected, if both false component is selected.'''

        if not self.collide\_point(\*touch.pos):

            return super().on\_touch\_up(touch)

        if self.dragged:

            self.deselect()

        elif self.selected:

            self.deselect()

        else:

            self.select()

        return super().on\_touch\_up(touch)

    def get\_logic\_gate(self):

        '''Returns the logic component object of this visual component object.'''

        return self.logic\_gate

    def update\_state(self):

        '''Updates the state of component to match state of logic component.'''

        x = self.logic\_gate.get\_output()

        self.state = x

    def get\_state(self):

        '''Returns the components state.'''

        return self.state

class DragSwitch(DragGate):

    '''Switch visual component. User sets state of component'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.logic\_gate = Switch()

        self.states = {1:"GateIcons/switch\_on.png", 0:"GateIcons/switch\_off.png"}

        self.img.source = self.states[self.logic\_gate.get\_output()]

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.out\_node = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.add\_widget(self.out\_node)

        self.nodes.append(self.out\_node)

        self.hide\_nodes()

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.out\_node.center=(self.right, self.y+(self.height//2))

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.out\_node:

                    return -1

        return False

    def on\_touch\_up(self, touch):

        '''When user releases click, checks dragged attribute, if false and the touch is within the set area, the switch is flipped and dragged is set as true, then super of method is called.'''

        if not self.dragged:

            if (touch.pos[0] < self.right - 30) and (touch.pos[0] > self.x + 10) and (touch.pos[1] < self.top - 20) and (touch.pos[1] > self.y + 20):

                self.logic\_gate.flip()

                self.update\_state()

                self.parent.update\_states()

                self.dragged = True

        return super().on\_touch\_up(touch)

    def update\_state(self):

        '''Updates the state of component to match state of logic component.'''

        x = self.logic\_gate.get\_output()

        self.state = x

        self.img.source = self.states[self.state]

class DragClock(DragGate):

    '''Clock visual component. Kivy scheduler flips state after an interval. (Uses switch logic component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.logic\_gate = Switch()

        self.states = {1:"GateIcons/clock\_on.png", 0:"GateIcons/clock\_off.png"}

        self.img.source = self.states[self.logic\_gate.get\_output()]

        Clock.schedule\_interval(self.clock\_flip, 1)

    def clock\_flip(self, \*args):

        '''Flips state of logic component, if AttributeError, the clock has been deleted so it stop the kivy schedule.'''

        self.logic\_gate.flip()

        self.update\_state()

        try:

            self.parent.update\_states()

        except AttributeError as e:

            print("Clock must have been deleted")

            Clock.unschedule(self.clock\_flip)

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.out\_node = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.add\_widget(self.out\_node)

        self.nodes.append(self.out\_node)

        self.hide\_nodes()

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.out\_node.center=(self.right, self.y+(self.height//2))

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.out\_node:

                    return -1

        return False

    def update\_state(self):

        '''Updates the state of component to match state of logic component.'''

        x = self.logic\_gate.get\_output()

        self.state = x

        self.img.source = self.states[self.state]

class DragOutput(DragGate):

    '''Output visual component. Displays its state.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.logic\_gate = Output()

        self.states = {1:"GateIcons/output\_on.png", 0:"GateIcons/output\_off.png", None:"GateIcons/output\_empty.png"}

        self.update\_state()

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.in\_node\_1 = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.add\_widget(self.in\_node\_1)

        self.nodes.append(self.in\_node\_1)

        self.hide\_nodes()

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.in\_node\_1.center=(self.x, self.center\_y)

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.in\_node\_1:

                    return 1

        return False

    def update\_state(self):

        '''Updates the state of component to match state of logic component.'''

        x = self.logic\_gate.get\_output()

        self.state = x

        self.img.source = self.states[self.state]

class DragAndGate(DragGate):

    '''And Gate visual component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.img.source = "GateIcons/and.png"

        self.logic\_gate = And\_Gate()

class DragOrGate(DragGate):

    '''And Or visual component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.img.source = "GateIcons/or.png"

        self.logic\_gate = Or\_Gate()

class DragXorGate(DragGate):

    '''And Xor visual component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.img.source = "GateIcons/xor.png"

        self.logic\_gate = Xor\_Gate()

class DragNotGate(DragGate):

    '''Not Gate visual component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.img.source = "GateIcons/not.png"

        self.logic\_gate = Not\_Gate()

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.in\_node\_1 = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.out\_node = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.add\_widget(self.in\_node\_1)

        self.add\_widget(self.out\_node)

        self.nodes.append(self.in\_node\_1)

        self.nodes.append(self.out\_node)

        self.hide\_nodes()

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.in\_node\_1.center=(self.x, self.center\_y+2)

        self.out\_node.center=(self.right, self.center\_y+2)

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.in\_node\_1:

                    return 1

                elif node == self.out\_node:

                    return -1

        return False

class MainWindow(Widget):

    '''Main Window class'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.ids["gateCanvas"].root = self

        self.toggles = ["connectToggle", "moveToggle", "disconnectToggle"]

    def set\_tool(self, tool):

        '''Calls Gate Canvases set\_tool method.'''

        self.ids["gateCanvas"].set\_tool(tool)

        toggle = f"{tool}Toggle"

        if self.ids[toggle].state == 'down':

            pass

        else:

            for tog in self.toggles:

                if tog == toggle:

                    self.ids[tog].state = 'down'

                else:

                    self.ids[tog].state = 'normal'

    def clear\_canvas(self):

        '''Calls Gate Canvases clear\_canvas method.'''

        self.ids["gateCanvas"].clear\_canvas()

        self.set\_tool("move")

    def add\_gate(self, gate\_type):

        '''Calls Gate Canvases add\_gate method.'''

        self.ids["gateCanvas"].add\_gate(gate\_type)

    def delete\_gate(self):

        '''Calls Gate Canvases delete\_gate method.'''

        self.ids["gateCanvas"].delete\_gate()

        self.set\_tool("move")

kv = Builder.load\_file("LogicSim.kv")

'''Loads the kivy file'''

class LogicGateSimulator(App):

    '''Kivy App class.'''

    RED = StringProperty('FF0000')

    DARK\_GREY = StringProperty('222222')

    BLUE = StringProperty('748bb4')

    LIGHT\_GREY = StringProperty('DCDCDC')

    ORANGE = StringProperty('f1ae80')

    WHITE = StringProperty('FFFFFF')

    BLACK = StringProperty('000000')

    def build(self):

        '''Builds the kivy app.'''

        self.icon = "GateIcons/and.png"

        return MainWindow()

if \_\_name\_\_ == '\_\_main\_\_':

    LogicGateSimulator().run()

## GUI (kvlang)

#:kivy 2.0

#:import Factory kivy.factory.Factory

<GridLayout>:

    canvas.before:

        Color:

            rgba: rgba(app.DARK\_GREY)

        Rectangle:

            pos: self.pos

            size: self.size

<MyGridLayout@GridLayout>:

    canvas.before:

        Color:

            rgba: rgba(app.BLUE)

        Rectangle:

            pos: self.pos

            size: self.size

<ConnectionLine>:

    canvas:

        Color:

            rgba: self.color\_state

<ActionButton>:

    font\_size:15

    color: rgba(app.WHITE)

<GateButton@Button>:

    used: False

    size\_hint: None, None

    background\_normal: ''

    background\_color: rgba(app.LIGHT\_GREY)

    canvas:

        Color:

            rgba: rgba(app.RED) if not self.used else rgba(app.DARK\_GREY)

        Line:

            width: 2

            rectangle: self.x, self.y, self.width, self.height

<ToolToggleButton@ToggleButton>:

    font\_size: 15

    color: rgba(app.BLACK)

    size\_hint: None, None

    allow\_no\_selection: False

    background\_normal: ''

    background\_color: rgba(app.LIGHT\_GREY)

    canvas:

        Color:

            rgba: rgba(app.DARK\_GREY)

        Line:

            width: 2

            rectangle: self.x, self.y, self.width, self.height

<ToolButton@Button>:

    font\_size: 15

    color: rgba(app.BLACK)

    size\_hint: None, None

    background\_normal: ''

    background\_color: rgba(app.LIGHT\_GREY)

    canvas:

        Color:

            rgba: rgba(app.DARK\_GREY)

        Line:

            width: 2

            rectangle: self.x, self.y, self.width, self.height

<GateLabel@Label>:

    font\_size: 25

    color: rgba(app.BLACK)

<ExitPopup>:

    auto\_dismiss: True

    title\_color: rgba(app.WHITE)

    title\_size: 18

    title: "Are you sure you want to quit?\nAny unsaved progress will be lost"

    size\_hint: None, None

    size: 350,150

    BoxLayout:

        spacing: 10

        Button:

            color: rgba(app.BLACK)

            text: 'Cancel'

            size\_hint: 0.5, None

            height: 50

            background\_normal: ''

            background\_color: rgba(app.LIGHT\_GREY)

            on\_release: root.dismiss()

        Button:

            color: rgba(app.BLACK)

            color:

            text: 'Quit'

            size\_hint: 0.5, None

            height: 50

            background\_normal: ''

            background\_color: rgba(app.LIGHT\_GREY)

            on\_release: root.close\_window()

<TruthPopup>:

    auto\_dismiss: True

    title\_color: rgba(app.WHITE)

    title\_size: 18

    title: "Truth Table Generator"

    size\_hint: 0.6, 0.8

    BoxLayout:

        orientation: "vertical"

        padding: 10

        ScrollView:

            size\_hint: 1, 1

            do\_scroll\_x: False

            effect\_cls: "ScrollEffect"

            Label:

                id: truth\_label

                size\_hint\_y: None

                text\_size: self.width, None

                height: self.texture\_size[1]

                halign: "center"

                bold: True

                font\_name: root.FONT

                font\_size: 20

                color: rgba(app.WHITE)

        GridLayout:

            size\_hint: 1, None

            height: 60

            rows: 1

            columns: 3

            padding: 5

            spacing: 10

            TextInput:

                id: truth\_input

                text: "Enter Boolean Expression"

                font\_size: 30

                size\_hint: 1, 1

                multiline: False

                on\_text\_validate: root.generate()

            Button:

                color: rgba(app.BLACK)

                text: 'Cancel'

                on\_release: root.dismiss()

                size\_hint: None, 1

                width: 100

                background\_normal: ''

                background\_color: rgba(app.LIGHT\_GREY)

            Button:

                color: rgba(app.BLACK)

                text: 'Generate'

                on\_release: root.generate()

                size\_hint: None, 1

                width: 100

                background\_normal: ''

                background\_color: rgba(app.LIGHT\_GREY)

<MainWindow>:

    GridLayout:

        size: root.width, root.height

        rows:2

        orientation: "tb-lr"

        ActionBar:

            height: 40

            pos\_hint: {"top":1}

            ActionView:

                padding: [0,0,10,0]

                ActionPrevious:

                    with\_previous: False

                    font\_size: 50

                    markup: True

                    title: "[b]Logic Gate Simulator[/b]"

                    app\_icon: ""

                    app\_icon\_width: 0

                ActionButton:

                    text: "Truth Table"

                    on\_release: Factory.TruthPopup().open(root)

                ActionButton:

                    text: "Save"

                ActionButton:

                    text: "Load"

                ActionButton:

                    text: "Help"

                ActionButton:

                    text: "Quit"

                    on\_release: Factory.ExitPopup().open()

        GridLayout:

            size: self.parent.width, self.parent.height

            orientation: "tb-lr"

            cols: 3

            padding: [10,0,10,10]

            spacing: 10

            ScrollView:

                size\_hint: None, 1

                width: 200

                effect\_cls: "ScrollEffect"

                canvas.before:

                    Color:

                        rgba: rgba(app.BLUE)

                    Rectangle:

                        pos: self.pos

                        size: self.size

                MyGridLayout:

                    size\_hint\_y: None

                    height: 900

                    cols: 1

                    rows: 2

                    padding: 5

                    Label:

                        size\_hint: 1,0.1

                        markup: True

                        font\_size: 40

                        color: rgba(app.BLACK)

                        text: "[b]Gates[/b]"

                    MyGridLayout:

                        cols: 2

                        size: self.parent.size

                        GateLabel:

                            text: "And"

                        GateButton:

                            id: And

                            on\_press:

                                root.add\_gate("and")

                                self.used = True

                            Image:

                                source: 'GateIcons/and.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Or"

                        GateButton:

                            id: Or

                            on\_press:

                                root.add\_gate("or")

                                self.used = True

                            Image:

                                source: 'GateIcons/or.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Not"

                        GateButton:

                            id: Not

                            on\_press:

                                root.add\_gate("not")

                                self.used = True

                            Image:

                                source: 'GateIcons/not.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Xor"

                        GateButton:

                            id: Xor

                            on\_press:

                                root.add\_gate("xor")

                                self.used = True

                            Image:

                                source: 'GateIcons/xor.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Switch"

                        GateButton:

                            id: Input

                            on\_press:

                                root.add\_gate("switch")

                                self.used = True

                            Image:

                                source: 'GateIcons/switch\_menu.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Output"

                        GateButton:

                            id: Output

                            on\_press:

                                root.add\_gate("output")

                                self.used = True

                            Image:

                                source: 'GateIcons/output\_menu.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Clock"

                        GateButton:

                            id: Clock

                            on\_press:

                                root.add\_gate("clock")

                                self.used = True

                            Image:

                                source: 'GateIcons/clock\_menu.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

            GateCanvas:

                id: gateCanvas

                canvas.before:

                    Color:

                        rgba: rgba(app.LIGHT\_GREY)

                    Rectangle:

                        pos: self.pos

                        size: self.size

            ScrollView:

                size\_hint: None, 1

                width: 110

                effect\_cls: "ScrollEffect"

                canvas.before:

                    Color:

                        rgba: rgba(app.ORANGE)

                    Rectangle:

                        pos: self.pos

                        size: self.size

                GridLayout:

                    canvas.before:

                        Color:

                            rgba: rgba(app.ORANGE)

                        Rectangle:

                            pos: self.pos

                            size: self.size

                    orientation: "tb-lr"

                    size\_hint\_y: None

                    width: 110

                    height: 700

                    cols: 1

                    padding: 5

                    spacing: 10

                    Label:

                        color: rgba(app.BLACK)

                        markup: True

                        font\_size: 30

                        size\_hint\_y: None

                        text: "[b]Tools[/b]"

                    ToolToggleButton:

                        id: connectToggle

                        group: "tools"

                        text: "Connect"

                        on\_press: root.set\_tool("connect")

                    ToolToggleButton:

                        id: disconnectToggle

                        group: "tools"

                        text: "Disconnect"

                        on\_press: root.set\_tool("disconnect")

                    ToolToggleButton:

                        id: moveToggle

                        state: "down"

                        group: "tools"

                        text: "Move"

                        on\_press: root.set\_tool("move")

                    ToolButton:

                        text: "Delete"

                        on\_press: root.delete\_gate()

                    ToolButton:

                        text: "Clear"

                        on\_press: root.clear\_canvas()

# Testing

The testing tables below will refer to the specific objectives in the analysis section. The testing will be split into two sections.

Section 1 will use python unit tests to test that the truth table, logic components and board work correctly and achieve their objectives. The testing is automated using python’s unit tests module. The tests are split into test case classes and test methods which I will write to set up the test correctly then using an assert method compare the output to the correct output I have given it. The testing code can be seen at the end of this section.

Section 2 will be a video test that shows that the GUI works as it should and that it achieves its objectives. The video is timestamped for each test. The video will have repeats of tests in section one but using the GUI and tests which are not directly referencing the objectives.

## Code Testing Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test No.** | **Description** | **Objective** | **Reference** | **Passed/Failed** | **Notes** |
| 1 | When a component is created it is the correct type and has no inputs or outputs. | 1.1, 1.1.3.1 | Y |  |  |
| 2 | A component with no inputs has an output None, unless the component is a switch then output is default to 0. | 1.1.5 | Y |  |  |
| 3 | The switch components flip method flips the state of the switch. | 1.1.6 | Y |  |  |
| 4 | Each components evaluate method return the correct output. | 1.1.5 | Y |  |  |
| 5 | When a board is created it has an empty gates array. | 1.2 | Y |  |  |
| 6 | Components can be added to the board’s gates array. | 1.2.1 | Y |  |  |
| 7 | Components can be removed from the board. | 1.2.3 | Y |  |  |
| 8 | The board connects components together. | 1.2.2, 1.1.3 | Y |  |  |
| 9 | The board disconnects components. | 1.2.2, 1.1.3 | Y |  |  |
| 10 | When the board removes a component from it array, it disconnects everything connected to it. | 1.2.3 | Y |  |  |
| 11 | Connecting components together updates their expressions. | 1.1.7 | y |  |  |
| 12 | A component with two valid inputs will have the correct state. | 1.1.4, 1.1.5 | Y |  |  |
| 13 | Changing the state of an input will correctly alter the state of the component. | 1.1.4, 1.1.5 | Y |  |  |
| 14 | The truth table generator will produce a truth table in the form of a dictionary for a valid expression. | 2.1.1 | Y |  |  |
| 15 | The truth table generator will return an error for an invalid expression. | 2.1.1 | Y |  |  |
| 16 | The truth table correctly produces input values for each variable it’s given. | 2.2 | Y |  |  |
| 17 | The truth table produces an out column that is accurate to the inputs and expression. | 2.3, 2.4 | Y |  |  |

## Video Testing Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test No.** | **Description** | **Objective** | **Reference** | **Passed/Failed** | **Notes** |
| 18 | Window opens to main GUI. |  |  |  |  |
| 19 | Window is maximised on opening. |  |  |  |  |
| 20 | Window is scalable. Side menus stay the same width with the canvas size decreasing. |  |  |  |  |
| 21 | If a is window too small to fit full side menus, they allow the user to scroll the menu. |  |  |  |  |
| 22 | Buttons in above toolbar is put into drop down menu when window is too small. |  |  |  |  |
| 23 | Gate menu consists of all components in objectives. |  |  |  | Menu also contains a clock component which was not an objective. |
| 24 | Tool Menu consists of tools: Connect, Disconnect, Move, Delete and Clear. |  |  |  |  |
| 25 | Move toggle tool is initially selected. |  |  |  |  |
| 26 | Connect, Disconnect and Move are toggle buttons which can only have one active at a time. |  |  |  |  |
| 27 | Clicking Delete and clear sets toggle tool to move if it is not already selected. |  |  |  |  |
| 28 | Gate menu buttons start with a red outline to show the component has not been used. |  |  |  |  |
| 29 | Clicking a component button adds that component to the canvas and changes the button outline to black to show it has been used in the current session. |  |  |  |  |
| 30 | A component added to the canvas is initially selected. |  |  |  |  |
| 31 | A component that is clicked will be selected or deselected. |  |  |  | A Switch will need to be clicked outside of the switch’s toggle area. |
| 32 | Multiple components can be selected at once. |  |  |  |  |
| 33 | A component can only be selected if the move tool is toggled. |  |  |  |  |
| 34 | Clicking and dragging a component selects a component and moves it with the cursor. |  |  |  |  |
| 35 | Clicking the canvas will deselect any selected gates. |  |  |  |  |
| 36 | Clicking a component without holding selects a component. |  |  |  |  |
| 37 | Components can only be selected when the move tool is active. |  |  |  |  |
| 38 | Component nodes are only visible if the Connect or Disconnect tool is toggled. |  |  |  |  |
| 39 | With the connect tool select tool selected, dragging from an input node to an output node will connect the nodes. |  |  |  |  |
| 40 | Trying to connect an output node to an output node or input node to an input node will not work. |  |  |  |  |
| 41 | If the connection works a connection line will be added to the canvas connecting the two nodes. |  |  |  |  |
| 42 | The connection line will be coloured to reflect the state of the component it is exiting. |  |  |  | White = None  Green = True  Red = False |
| 43 | The connection line will automatically move with the components and adjust its bend to its centre. |  |  |  |  |
| 44 | Multiple connections can come from an output node. |  |  |  |  |
| 45 | Only one connection to an input node is allowed. |  |  |  |  |
| 46 | With the disconnect tool selected, clicking a node will disconnect it from any nodes it is connected to. |  |  |  |  |
| 47 | When components are disconnected, connection lines between them are removed. |  |  |  |  |
| 48 | Using the delete tool will remove any already selected components. |  |  |  |  |
| 49 | The delete tool will only work if the current toggle tool is move. |  |  |  |  |
| 50 | The clear tool will remove every component from the canvas. |  |  |  |  |
| 51 | Output component displays the state of the component it’s connect to. |  |  |  | White = None  Green = True  Red = False |
| 52 | Gate components work correctly |  |  |  | Tested each gate and some more complicated circuits. |
| 53 | The truth table button in the toolbar opens a popup for making truth tables. |  |  |  |  |
| 54 | If no component is selected, when the truth table popup is opened there is no truth table, and the input box says enter expression here. |  |  |  |  |
| 55 | If a component is selected that has valid inputs, the truth table opens with a truth table for that component already created, the expression in the input box. |  |  |  |  |
| 56 | If more than one component is selected the truth table opens in the same state as test 49. |  |  |  |  |
| 57 | If a component that has invalid input is selected the truth table opens in the same state as test 49. |  |  |  |  |
| 58 | The truth table produces the correct truth table for the given expression. |  |  |  |  |
| 59 | The truth table is generated when the generate button is pressed. |  |  |  |  |
| 60 | The truth table popup is closed when the cancel button is pressed. |  |  |  |  |
| 61 | The save button in the toolbar opens the saving menu. |  |  | Failed |  |
| 62 | The load button in the toolbar opens the loading menu. |  |  | Failed |  |
| 63 | The help button in the tool bar opens a popup with info on how to use the program. |  |  | Failed |  |
| 64 | The quit button in the toolbar opens a popup asking, ‘Are you sure you want to quit?’. |  |  |  |  |
| 65 | Pressing cancel in the quit popup will close it and return to the program. |  |  |  |  |
| 66 | Pressing quit in the quit popup will close the whole program. |  |  |  |  |

## Code Testing Code

from logic.gates import \*

from logic.board import \*

from logic.truth\_table import generate\_truth\_table

import unittest

'''Testing the logic side of the components.'''

class TestCreatingComponents(unittest.TestCase):

    '''Test: When components are made they are of the correct type and have empty outputs and inputs.'''

    def test\_and(self):

        and\_gate = And\_Gate()

        self.assertIsInstance(and\_gate, And\_Gate)

        self.assertIsInstance(and\_gate, Gate)

        self.assertListEqual(and\_gate.\_input\_nodes, [None, None])

        self.assertListEqual(and\_gate.\_output\_nodes, [])

        self.assertEqual(and\_gate.get\_output(), None)

    def test\_or(self):

        or\_gate = Or\_Gate()

        self.assertIsInstance(or\_gate, Or\_Gate)

        self.assertIsInstance(or\_gate, Gate)

        self.assertListEqual(or\_gate.\_input\_nodes, [None, None])

        self.assertListEqual(or\_gate.\_output\_nodes, [])

        self.assertEqual(or\_gate.get\_output(), None)

    def test\_xor(self):

        xor\_gate = Xor\_Gate()

        self.assertIsInstance(xor\_gate, Xor\_Gate)

        self.assertIsInstance(xor\_gate, Gate)

        self.assertListEqual(xor\_gate.\_input\_nodes, [None, None])

        self.assertListEqual(xor\_gate.\_output\_nodes, [])

        self.assertEqual(xor\_gate.get\_output(), None)

    def test\_not(self):

        not\_gate = Not\_Gate()

        self.assertIsInstance(not\_gate, Not\_Gate)

        self.assertIsInstance(not\_gate, Gate)

        self.assertListEqual(not\_gate.\_input\_nodes, [None, None])

        self.assertListEqual(not\_gate.\_output\_nodes, [])

        self.assertEqual(not\_gate.get\_output(), None)

    def test\_switch(self):

        switch = Switch()

        self.assertIsInstance(switch, Switch)

        self.assertListEqual(switch.\_output\_nodes, [])

        self.assertEqual(switch.get\_output(), 0)

        switch.flip()

        self.assertEqual(switch.get\_output(), 1)

    def test\_output(self):

        output = Output()

        self.assertIsInstance(output, Output)

        self.assertListEqual(output.\_input\_nodes, [None])

        self.assertEqual(output.get\_output(), None)

class TestGateEvaluateMethods(unittest.TestCase):

    '''Test: The evaluate methods of each gate returns the correct result.'''

    def test\_and(self):

        and\_gate = And\_Gate()

        self.assertEqual(and\_gate.evaluate(0,0), 0)

        self.assertEqual(and\_gate.evaluate(0,1), 0)

        self.assertEqual(and\_gate.evaluate(1,0), 0)

        self.assertEqual(and\_gate.evaluate(1,1), 1)

    def test\_or(self):

        or\_gate = Or\_Gate()

        self.assertEqual(or\_gate.evaluate(0,0), 0)

        self.assertEqual(or\_gate.evaluate(0,1), 1)

        self.assertEqual(or\_gate.evaluate(1,0), 1)

        self.assertEqual(or\_gate.evaluate(1,1), 1)

    def test\_xor(self):

        xor\_gate = Xor\_Gate()

        self.assertEqual(xor\_gate.evaluate(0,0), 0)

        self.assertEqual(xor\_gate.evaluate(0,1), 1)

        self.assertEqual(xor\_gate.evaluate(1,0), 1)

        self.assertEqual(xor\_gate.evaluate(1,1), 0)

    def test\_not(self):

        not\_gate = Not\_Gate()

        self.assertEqual(not\_gate.evaluate(0), 1)

        self.assertEqual(not\_gate.evaluate(1), 0)

class TestLogicBoard(unittest.TestCase):

    '''Test: The logic board correctly initiallises, adds, and removes components from its gates array'''

    def setUp(self):

        self.board = Board()

        self.and\_gate = And\_Gate()

        self.or\_gate = Or\_Gate()

        self.xor\_gate = Xor\_Gate()

        self.not\_gate = Not\_Gate()

        self.switch1 = Switch()

        self.switch2 = Switch()

        self.output = Output()

    def add\_components\_to\_board(self):

        self.board.add\_gate(self.and\_gate)

        self.board.add\_gate(self.or\_gate)

        self.board.add\_gate(self.xor\_gate)

        self.board.add\_gate(self.not\_gate)

        self.board.add\_gate(self.switch1)

        self.board.add\_gate(self.switch2)

        self.board.add\_gate(self.output)

    def test\_board\_initially\_empty(self):

        self.assertListEqual(self.board.gates, [])

    def test\_adding\_components(self):

        self.add\_components\_to\_board()

        self.assertListEqual(self.board.gates, [self.and\_gate, self.or\_gate, self.xor\_gate, self.not\_gate, self.switch1, self.switch2, self.output])

    def test\_removing\_components(self):

        self.board.add\_gate(self.switch1)

        self.board.add\_gate(self.switch2)

        self.board.add\_gate(self.output)

        self.assertListEqual(self.board.gates, [self.switch1, self.switch2, self.output])

        self.board.remove\_gate(self.switch2)

        self.assertListEqual(self.board.gates, [self.switch1, self.output])

class TestConnectingComponents(unittest.TestCase):

    '''Test: The logic board correctly connects components.'''

    def setUp(self):

        self.board = Board()

        self.and\_gate = And\_Gate()

        self.switch1 = Switch()

        self.switch2 = Switch()

    def add\_components\_to\_board(self):

        self.board.add\_gate(self.and\_gate)

        self.board.add\_gate(self.switch1)

        self.board.add\_gate(self.switch2)

    def test\_connecting\_components(self):

        self.add\_components\_to\_board()

        self.board.connect\_gate(self.and\_gate, self.switch1, 1)

        self.board.connect\_gate(self.and\_gate, self.switch2, 2)

        self.assertListEqual(self.and\_gate.\_input\_nodes, [self.switch1, self.switch2])

        self.assertIn(self.and\_gate, self.switch1.\_output\_nodes)

class TestDisconnectingComponents(unittest.TestCase):

    '''Test: The logic board correctly disconnects components.'''

    def setUp(self):

        self.board = Board()

        self.and\_gate = And\_Gate()

        self.switch1 = Switch()

        self.switch2 = Switch()

    def add\_components\_to\_board(self):

        self.board.add\_gate(self.and\_gate)

        self.board.add\_gate(self.switch1)

        self.board.add\_gate(self.switch2)

    def test\_disconnecting\_two\_input\_components(self):

        self.add\_components\_to\_board()

        self.board.connect\_gate(self.and\_gate, self.switch1, 1)

        self.board.connect\_gate(self.and\_gate, self.switch2, 2)

        self.board.disconnect\_gate(self.and\_gate, self.switch1)

        self.assertListEqual(self.and\_gate.\_input\_nodes, [None, self.switch2])

class TestRemovingComponentsWithConnections(unittest.TestCase):

    '''Test: The logic board disconnects everything from a component it is removing.'''

    def setUp(self):

        self.board = Board()

        self.xor\_gate = Xor\_Gate()

        self.switch1 = Switch()

        self.switch2 = Switch()

        self.output = Output()

    def add\_components\_to\_board(self):

        self.board.add\_gate(self.xor\_gate)

        self.board.add\_gate(self.switch1)

        self.board.add\_gate(self.switch2)

        self.board.add\_gate(self.output)

    def test\_removing\_component\_with\_connections(self):

        self.add\_components\_to\_board()

        self.board.connect\_gate(self.xor\_gate, self.switch1, 1)

        self.board.connect\_gate(self.xor\_gate, self.switch2, 2)

        self.board.connect\_gate(self.output, self.xor\_gate, 1)

        self.assertListEqual(self.xor\_gate.\_input\_nodes, [self.switch1, self.switch2])

        self.assertListEqual(self.switch1.\_output\_nodes, [self.xor\_gate])

        self.assertListEqual(self.switch2.\_output\_nodes, [self.xor\_gate])

        self.assertListEqual(self.xor\_gate.\_output\_nodes, [self.output])

        self.assertListEqual(self.output.\_input\_nodes, [self.xor\_gate])

        self.assertIn(self.xor\_gate, self.board.gates)

        self.board.remove\_gate(self.xor\_gate)

        self.assertListEqual(self.xor\_gate.\_input\_nodes, [None, None])

        self.assertListEqual(self.switch1.\_output\_nodes, [])

        self.assertListEqual(self.switch2.\_output\_nodes, [])

        self.assertListEqual(self.xor\_gate.\_output\_nodes, [])

        self.assertListEqual(self.output.\_input\_nodes, [None])

class TestComponentExpressions(unittest.TestCase):

    '''Test: The expression update correctly'''

    def setUp(self):

        self.board = Board()

        self.and\_gate = And\_Gate()

        self.or\_gate = Or\_Gate()

        self.not\_gate = Not\_Gate()

        self.switch1 = Switch()

        self.switch2 = Switch()

        self.output = Output()

    def test\_expression\_none\_when\_no\_input(self):

        expression = self.and\_gate.get\_expression()

        self.assertEqual(expression, None)

    def test\_with\_not\_enough\_inputs(self):

        self.board.connect\_gate(self.and\_gate, self.switch1, 1)

        expression = self.and\_gate.get\_expression()

        self.assertEqual(expression, None)

    def test\_with\_inputs\_that\_have\_no\_expression(self):

        self.board.connect\_gate(self.and\_gate, self.or\_gate, 1)

        self.board.connect\_gate(self.and\_gate, self.not\_gate, 2)

        expression = self.and\_gate.get\_expression()

        self.assertEqual(expression, None)

    def test\_with\_valid\_inputs(self):

        self.board.connect\_gate(self.and\_gate, self.switch1, 1)

        self.board.connect\_gate(self.and\_gate, self.switch2, 2)

        switch\_variable1 = self.switch1.get\_expression()

        switch\_variable2 = self.switch2.get\_expression()

        expression = self.and\_gate.get\_expression()

        valid\_expression = f"{switch\_variable1} and {switch\_variable2}"

        self.assertEqual(expression, valid\_expression)

    def test\_expression\_set\_to\_none\_when\_inputs\_removed(self):

        self.board.connect\_gate(self.and\_gate, self.switch1, 1)

        self.board.connect\_gate(self.and\_gate, self.switch2, 2)

        switch\_variable1 = self.switch1.get\_expression()

        switch\_variable2 = self.switch2.get\_expression()

        expression1 = self.and\_gate.get\_expression()

        valid\_expression = f"{switch\_variable1} and {switch\_variable2}"

        self.assertEqual(expression1, valid\_expression)

        self.board.disconnect\_gate(self.and\_gate, self.switch1)

        self.board.disconnect\_gate(self.and\_gate, self.switch2)

        expression2 = self.and\_gate.get\_expression()

        self.assertNotEqual(expression2, valid\_expression)

        self.assertEqual(expression2, None)

class TestStatesCorrectlyChange(unittest.TestCase):

    '''Test: The states of components update correctly when an input is altered.'''

    def setUp(self):

        self.board = Board()

        self.and\_gate = And\_Gate()

        self.or\_gate = Or\_Gate()

        self.xor\_gate = Xor\_Gate()

        self.not\_gate = Not\_Gate()

        self.switch1 = Switch()

        self.switch2 = Switch()

        self.output = Output()

        self.board.add\_gate(self.and\_gate)

        self.board.add\_gate(self.or\_gate)

        self.board.add\_gate(self.xor\_gate)

        self.board.add\_gate(self.not\_gate)

        self.board.add\_gate(self.switch1)

        self.board.add\_gate(self.switch2)

        self.board.add\_gate(self.output)

    def test\_and(self):

        self.board.connect\_gate(self.and\_gate, self.switch1, 1)

        self.board.connect\_gate(self.and\_gate, self.switch2, 2)

        self.assertEqual(self.and\_gate.get\_output(), 0) #[0,0] -> 0

        self.switch1.flip()

        self.assertEqual(self.and\_gate.get\_output(), 0) #[1,0] -> 0

        self.switch1.flip()

        self.switch2.flip()

        self.assertEqual(self.and\_gate.get\_output(), 0) #[0,1] -> 0

        self.switch1.flip()

        self.assertEqual(self.and\_gate.get\_output(), 1) #[1,1] -> 1

    def test\_or(self):

        self.board.connect\_gate(self.or\_gate, self.switch1, 1)

        self.board.connect\_gate(self.or\_gate, self.switch2, 2)

        self.assertEqual(self.or\_gate.get\_output(), 0) #[0,0] -> 0

        self.switch1.flip()

        self.assertEqual(self.or\_gate.get\_output(), 1) #[1,0] -> 1

        self.switch1.flip()

        self.switch2.flip()

        self.assertEqual(self.or\_gate.get\_output(), 1) #[0,1] -> 1

        self.switch1.flip()

        self.assertEqual(self.or\_gate.get\_output(), 1) #[1,1] -> 1

    def test\_xor(self):

        self.board.connect\_gate(self.xor\_gate, self.switch1, 1)

        self.board.connect\_gate(self.xor\_gate, self.switch2, 2)

        self.assertEqual(self.xor\_gate.get\_output(), 0) #[0,0] -> 0

        self.switch1.flip()

        self.assertEqual(self.xor\_gate.get\_output(), 1) #[1,0] -> 1

        self.switch1.flip()

        self.switch2.flip()

        self.assertEqual(self.xor\_gate.get\_output(), 1) #[0,1] -> 1

        self.switch1.flip()

        self.assertEqual(self.xor\_gate.get\_output(), 0) #[1,1] -> 0

    def test\_not(self):

        self.board.connect\_gate(self.not\_gate, self.switch1, 1)

        self.assertEqual(self.not\_gate.get\_output(), 1) #[0] -> 1

        self.switch1.flip()

        self.assertEqual(self.not\_gate.get\_output(), 0) #[1] -> 0

    def test\_output(self):

        self.board.connect\_gate(self.output, self.switch1, 1)

        self.assertEqual(self.output.get\_output(), 0) #[0] -> 0

        self.switch1.flip()

        self.assertEqual(self.output.get\_output(), 1) #[1] -> 1

class TestTruthTable(unittest.TestCase):

    '''Test: Correct truth tables are produced for valid expressions'''

    def setUp(self):

        self.board = Board()

        self.and\_gate = And\_Gate()

        self.switch1 = Switch()

        self.switch2 = Switch()

        self.output = Output()

        self.board.add\_gate(self.and\_gate)

        self.board.add\_gate(self.switch1)

        self.board.add\_gate(self.switch2)

        self.board.add\_gate(self.output)

    def test\_with\_valid\_expression(self):

        '''generate\_truth\_table returns a tuple with the truth table dictionary and the expression string.

        Tests that the truth dictionary is correct as expression will not change.'''

        expression = "A AND B"

        tt = generate\_truth\_table(expression)

        self.assertDictEqual(

            tt[0],

            {

                'a':[0,0,1,1],

                'b':[0,1,0,1],

                'OUT':[0,0,0,1]

            }

        )

        self.assertEqual(tt[1], expression)

    def test\_with\_invalid\_expression(self):

        expression = "AND"

        tt = generate\_truth\_table(expression)

        self.assertEqual(tt, "Invalid Input")

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

## Video Testing Link

<https://youtu.be/HILKC_kIy70>

# Evaluation

Stuff