Kozell YON-BROOKS

5727

51319

boolean logic circuit simulator

AQA Computer Science NEA

[Date]

Contents

[Analysis 4](#_Toc100176312)

[The Problem 4](#_Toc100176313)

[End Users 4](#_Toc100176314)

[Overview of Logic Gates 4](#_Toc100176315)

[Features Needed for GCSE Students 5](#_Toc100176316)

[Analysis of existing systems 7](#_Toc100176317)

[Acceptable Limitations 8](#_Toc100176318)

[Platform, Programming Language, and Modules 8](#_Toc100176319)

[General Objectives 9](#_Toc100176320)

[Specific Objectives 9](#_Toc100176321)

[Design 11](#_Toc100176322)

[Overview 11](#_Toc100176323)

[Input Output Process Storage 11](#_Toc100176324)

[User Interface 12](#_Toc100176325)

# 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.

## 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 (Example shown in Figure 4).

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 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 menu bar at top of 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 be placed manually; this is tedious and makes repositioning components difficult. |

## Acceptable Limitations

Hardware and software constraints – The program must run on school computers and, therefore, not resource-intensive. The school uses Windows 10 as its operating system, so the program does not need to run on any other OS.

The…

## 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 (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.

For the GUI, python has an array of frameworks to help build user interfaces. These include PyQt5, Tkinter, Kivy, wxPython, 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>

## General Objectives

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 components will be able to be linked and added/removed. 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 Objectives

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.
   2. There will be a board class containing the components.
      1. The board will create and destroy components.
      2. The board will store the created components in an array and tree structure.
      3. The board will tell the gates to connect and disconnect to and from each other.
      4. The board will traverse its tree of components, making them evaluate each of their states.
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. 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. 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. More…

# Design

## Overview

This design section will show how I will accomplish the objectives above and how the system will look and operate......

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 GUI is also constructed using object oriented design

## 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 string |
| Process | Storage |
| Truth Table Generator: Will process a Boolean expression into a truth table.  Boolean Evaluation: Each gate will use the values of their input with their respective Boolean operators to produce their output value.  More idk | I don’t think I will be able to implement the saving and loading in time ☹ |

Graphical user interface

Description automatically generated

Figure : Class diagram for circuit components

Figure 8 above show 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 the 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, and a method that gets a truth table using the boolean expression of a given component. *(MIGHT CHANGE THIS!!!!11!)*

## Data Structure of Logic Components

Because the components all have only one output and at most 2 inputs, I thought a binary tree would be a suitable data structure. A binary tree is a rooted The outputs or last components 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). To

Below is an abstraction of how the gate logic will use this structure.

CLASS Node:

instance.data = [None, None]

instance.left = None

instance.right = None

FUNCTION set\_data(data):

instance.data = data

ENDFUNCTION

FUNCTION set\_left(left\_node):

instance.left = left\_node

ENDFUNCTION

FUNCTION set\_right(right\_node):

instance.right = right\_node

ENDFUNCTION

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

ENDIF

ENDFUNCTION

## User 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 show gives a

Diagram

Description automatically generated

Figure : Main Graphical User Interface Design

A picture containing diagram

Description automatically generated

Figure : Truth Table popup UI

Gate Canvas: This is the canvas that will display Gates and Connection Lines, it will also handle the logic of creating, deleting, and

Gates: This is the GUI side of the components; they will have an attribute that links to a logic component. They will.

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 components together, when the connect tool is selected, a user can drag from one node to another, this will draw a connection line between them, link the logic side of gates through the Board class, and update the states of all gates and lines on the board to account for changes the added component has made. The nodes will only appear when either the connect or disconnect tool is selected.

Connection Lines: 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. They will also change colour depending on the state of the component whose out\_node it is connected to. This will allow the user to see how the various parts of their circuit affect the logic without having to add OUTPUTs to each component.

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

truth\_dictionary 🡨 dictionary

list\_of\_outputs 🡨 []

FOR i 🡨 1 TO num\_permutations:

expression\_with\_variables 🡨 expression

FOR variable in truth\_dictionary:

expression\_with\_variables 🡨 expression\_with\_variables.replace(variable, INT\_TO\_STRING(dictionary[variable][i]))

ENDFOR

output 🡨 EVALUATE(expression\_with\_variables)

list\_of\_outputs.append(output)

ENDFOR

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

RETURN truth\_dictionary

ENDFUNCTION

Algorithms:

Truth Table Generator:

SUBROUTINE generateTruthTable(expression):

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("(", "")

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

temp\_exp 🡨 temp\_exp.replace(" ")

FOR( i IN temp\_exp):

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

variables.append(i)

ENDIF

ENDFOR

num\_variables 🡨 len(variables)

temp 🡨 string()

FOR(i 🡨 0 TO num\_variables):

temp 🡨 temp + "0"

ENDFOR

FOR(i 🡨 0 TO num\_variables):

temp 🡨 temp + "1"

ENDFOR

p 🡨 Get\_Permutations(temp, num\_variables) #Function that returns permutations of a list

variable\_permutations 🡨 []

FOR(i IN p):

if i not in variable\_permutations:

variable\_permutations.append(i)

dictionary = dict()

for i,j in enumerate(variables):

dictionary[j] = [ii[i] for ii in variable\_permutations]

final\_dictionary = getTruthTableOutputs(expression, dictionary, len(variable\_permutations))

SUBROUTINE truth\_dict\_to\_string(final\_dicitonary, input\_expression)

final\_string = ''

final\_string = final\_string + str("Boolean Expression: " + input\_expression.upper() + "\n")

for i in final\_dictionary:

if i.lower() == 'out':

final\_string = final\_string + (f"|{i}")

else:

final\_string = final\_string + (f"| {i} ")

final\_string = final\_string + ('|\n')

for i in range(len(final\_dictionary[‘OUT’])):

for ii in final\_dictionary:

final\_string = final\_string + (f"| {final\_dictionary[ii][i]} ")

final\_string = final\_string + ('|\n')

RETURN final\_string

ENDSUBROUTINE

The algorithm above will take an expression, dictionary of inputs, and the number of permutations as parameters.

A subroutine “getTruthTableOutput” with parameters: expression; which is a string, dictionary; which is a dictionary of the inputs variables with a list of their values for each row e.g. {a:[0,1,0,1], b:[0,0,1,1]}, and num\_permutations; which is the number of rows in the table or the length of the lists in the dictionary.

Will make a copy of dictionary called truth\_dictionary and an empty list called list\_of\_outputs.

For loop from 0 to num\_permutations, each loop makes a copy of expression called expression with variables, then has a for loop which iterates through the dictionary, each loop