UNIVERSITY OF CAMBRIDGE
Department of Engineering

Part IIA Project: GF2(P2) Software

Ayoife Dada AOD24@CAM.AC.UK

P2 Team 05
Homerton College
Department of Engineering
University of Cambridge
Cambridge
CB2 1PZ

Date: June 12, 2025

Abstract

Contents

1. Introduction

Problem Outline

The goal of this project was to create a logic simulation program. Rules for the language were specified in an EBNF file and example file using the language rules was used to write a well-known logic circuit (the full-adder) which was used throughout the project to ensure correctness of the output of the simulation.

A GUI was designed to display the output of the logic circuit. The interface allowed changes to be made to the circuit by allowing devices to be added or removed from the monitor list. Settings such as language and light-mode/dark-mode are set by changing the computer's settings. The design of the user interface was carefully considered and discussed by the team to make its use intuitive and easy. Improvements were made to the error handling and user interface of the simulation after constructive feedback from supervisors.

Approach chosen

The process allowed our team to go through the key phases of the software engineering life cycle: specification, design, implementation, testing, and maintenance.

The project provided insight into the challenges of large-scale software development. It was important that comments were understandable and that the code was easily readable i.e. algorithms, function names and variable names were reasonable so it could be easily understood by other members of the team and/or be re-used if required later on in the process. Furthermore, modular programming was essential so that it could be ensured that particular sections of the code could be tested and to make the code as a whole was understandable. This was useful when identifying the source of errors and in debugging. This highlighted how effective software engineering practices are essential to the success of a project.

An LL(1) framework was used.

2. Description of test procedures adopted

Testing functions

The Pytest python testing framework was used to test certain functions throughout the code to ensure that their outputs were as expected. This was important as it was helpful in the debugging process to identify which function may be leading to an error. These tests were run regularly to ensure that changes to the code did not effect the output of certain functions.

The scanner was tested by hard-coding the symbols that the scanner should identify and comparing it with the symbols that the scanner identifies. By running this test throughout the project, the team was confident that any source of error was not due the scanner giving the wrong output.

Tests for monitor, devices and network included checking the length of the monitor list, device list and connection list to see whether it was empty or not.

Testing for error handling

For the parser, the capsys fixture in pytest was used to access the output in by the code in the terminal during test execution. The captured outputs in the terminal were compared to the expected print error messages as known errors were introduced into some example EBNF text files. This ensured that our code detected the errors and displayed informative errors.

3. Description of logic simulator

Software Structure

The compiler scans through and reads the characters in the definition file and converts the text file into symbols. The parser receives these symbols and uses a set of rules to determine how these symbols are arranged to form a valid structure. If there is an error, an error message is displayed.

The parser uses python files called 'Names', 'Devices', 'Scanner', 'Monitor' and 'Network' to read the symbols, add valid device names, connections and monitor names to create a device list, a connection list and a monitor list. Errors raised using these python files are parsed in the Parse python file which handles the error and produces a print error message.

The user can run the gui using the terminal and if there are no errors, the network is created and an interface is displayed for the user to observe the monitored signals. The user can add or remove which signals are monitored using the gui.

Names, Devices and Scanner

The Names class maps variable names and string names to unique integers allowing the scanner to read and pass the information in the example text file to the parser. The Device class stores device properties (or the parameters). The Devices class makes and stores all the devices in the logic circuit but raises errors if there is a syntax or semantic error. These errors are handled in parser.

The Scanner class reads the definition file and converts the characters into symbols. The Symbol class stores the properties of a symbol in addition to keeping track of the line number and position being scanner so that print error messages can indicate where the error occurred with a pointer.

Parser, Monitors and Network

The Network class builds by connecting devices, allowing parser to get information about connections and make connections, and executing the network in logsim and updating the network in gui.

The errors raised by Devices, Monitors and Network are passed into the parser, which based on the type of error, displays a print error message suggesting how to fix the error and where the error is located.

Userint, logsim and gui

Userint allows the user to run the simulation from the command line, enter commands and adjust the network properties. Logsim and gui python files containing MyGLCanvas class and GUI class which creates the visual display for the user i.e. the canvas drawings, the main window and the widgets. Settings like the choice of language and dark mode are set by changing the language or light-mode/dark-mode computer settings. Alternatively, both settings can be chosen using commands in the terminal.

4. Teamwork and Collaboration

Progress

Peer programming: Our team decided at the start of the project to meet regularly to ensure consistent progress and to make decisions about how to structure the code. For example, it was important to understand the logic behind the function of scanner to identify why the parser may not have produced the expected outputs, especially in debugging when error handling.

Individual Work: Each team member spent time working on implementing their assigned task before a team meeting. Our solutions would then be presented and discussed since there were often multiple ways complete the task.

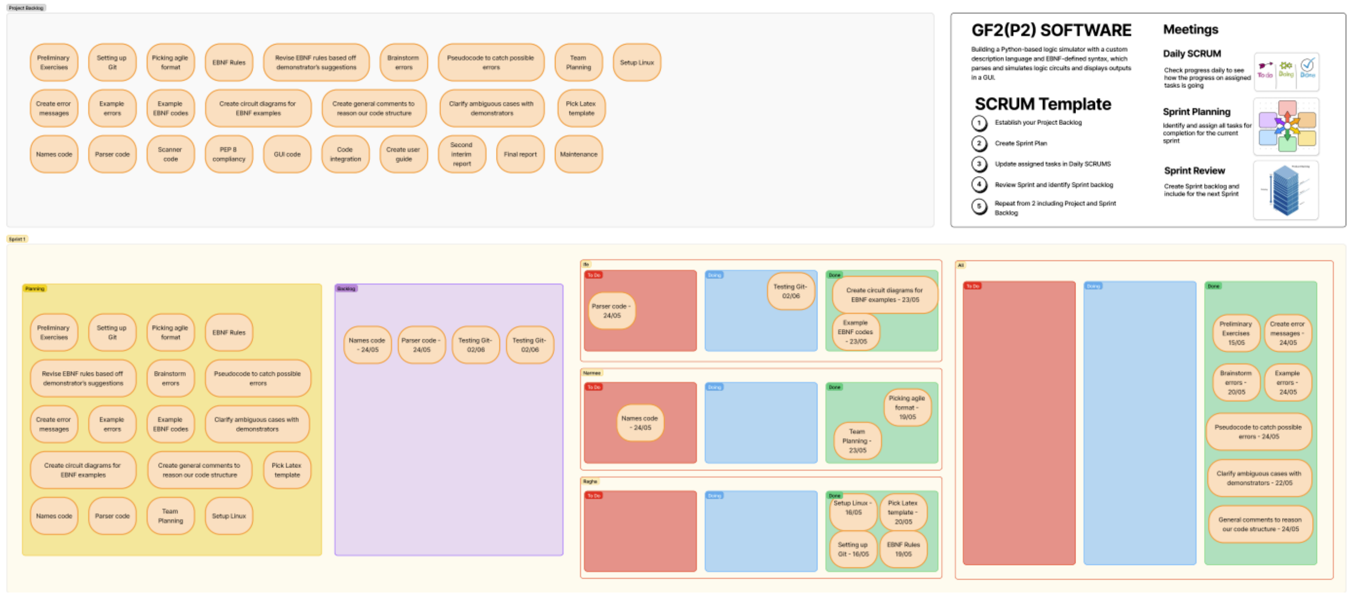
EBNF: After receiving feedback from the first interim report, the rules of our language were changed to make it simpler by removing unnecessary rules in the language. The language was then further modified during the maintenance phase to allow for the requirements of a new device.

Error handling: After implementing the logic used to scan and parse the example text files and including how the code should handle common errors, the team spent time try to break the simulator. When an error was find, it was often difficult to find a solution while maintaining the previous logical process that the code followed.

Online tools and platforms

Agile planning: Our team adopted the Agile Scrum framework (2) to manage the project lifecycle. Tasks were divided equally to suit each team member's individual strengths and were continuously reviewed and discussed (Figure 1).

Git branching: Git branches were implemented to efficiently manage code changes and maintain stable working versions. This allowed team members to try new solutions independently without affecting the main branch; revert back to previous versions without errors; and work simultaneously on separate branches before integration to the main branch.



GF2(P2) SOFTWARE

Building a Python-based logic simulator with a custom description language and EBNF-defined syntax, which parses and simulates logic circuits and displays outputs in a GUI.

SCRUM Template

1. Establish your Project Backlog
2. Create Sprint Plan
3. Update assigned tasks in Daily SCRUMS
4. Review Sprint and identify Sprint backlog
5. Repeat from 2 including Project and Sprint Backlog

Meetings

Daily SCRUM

Check progress daily to see how the progress on assigned tasks is going

Sprint Planning

Identify and assign all tasks for completion for the current sprint

Sprint Review

Create Sprint backlog and include for the next Sprint

Figure 1: Project plan in the middle of sprint 1 (22/05)

5. Description of software written and modified by me

Names - code written by me

The 'get name string function' was written by me as well as the corresponding test in the 'test names' file.

Parser - code written by me

The 'make device parser' function in the Parse python file coded by me. It is a long function that makes devices (adding a device to the device list) if there are no errors or handles errors that are raised in other classes ensuring that suitable error messages are printed.

Tests - code written by me

I decided to use python's OS library for all file paths used in the project since it was clear that the code would need to be able to run on both Windows and Linux. Furthermore, I contributed to the test scanner file by determining the best way to check that the scanner line number and position pointed to the correct position in the text file that was being read.

The template for test parser was written by me. This included checks for the number of errors identified, the line number of the errors, and the print error message including where the error pointed to. Since the parser required several functions for the tests each with a large number of hard-coded expected outputs, these templates were then used by the other members in the group to complete the file.

Maintanence - code modified by me

Siggen

The majority of Siggen was implemented by me. This involved restructuring the language rules. A new variable was created called 'bitstring' during maintenance. 'Bitstring' had to have at least one 'bit' but any number of bits could be added after the initial 'bit'.

The new device Siggen was similar to clock and switch since these devices do not have pins and cannot be input into. Clock and Siggen were largely similar in their functionality so a large amount of code in the Devices, Network and Parse python files used to create Siggen was inspired by the clock device.

Siggen had to be included in various initilisations of classes such as in 'keywords list' and 'device id list' in the 'scanner' python file. In devices, 'cold setup' for Siggen had to be modified since it was important that the first state was the first bit written in the waveform specified. 'Update siggen' in the network python file had to be carefully considered since its operation was different from clock. The essential logic was to identify whether the waveform was on a rising or falling edge or staying the same. In order for Siggen's parameter to be valid, the waveform had to be a sequence of bits i.e. only 1 or 0. It was difficult to get the simulator to register leading zeros and required collaboration from the whole team to think of the best way to implement this while considering the way that our code had already been logically structured.

Improved error handling

A few errors were spotted during the first demonstration so the team collectively spent time identifying errors such as flagging an error when the clock period was set to zero and ensuring that the error message was informative.

Additional time was spent attempting to break the code, and further implementations such as improving the identification and hence error message for repeated devices was added to the code.

Throughout the project I updated several comments to improve their readability since the purpose is for comments to be easily understandable to people who did not write the code.

6. Improvements

Conclusions

1. Scanning and parsing has many advantages when creating a compiler:

   Early error detection prevents later issues and makes debugging more straightforward. Informative error messages can be output with good a understanding of the structure of the input, scanners and parsers. This helps users identify and fix problems more quickly.

   Parsers can be adapted to specific languages or grammar rules making them versatile. This was helpful during maintenance as changes to the code were simply additive even though the rules of the language were altered slightly.

2. GUI are an important part of many products because it is the only part of the product that the user sees and uses. Even though it is essential that the scanning and parsing handles errors well and the simulation does not crash, the user interface must be designed carefully so it can be used intuitively by a user. As learned from the team's first demonstration, this is an iterative process of improvement. In order to make the GUI more professional, it is important to sync the GUI settings with the computer's settings.

3. Version control is extremely useful because sometimes adding a new feature would unexpectedly change the output for a previous part of the code. It was important that the team could refer back to previous code that worked as expected to identify and understand sources of error.

4. Team communication is important especially for large-scale projects because understanding how previous parts of worked was important for decision-making when deciding the best way to implement a new function as there were often multiple approaches that could be used.

Recommendations

1. Accessibility: Designing the simulator while considering accessibility would help make the product usable by individuals with certain disabilities.

2. Automatic Circuit Generation: The logic simulator can further be improved by adding a function to generate circuits from truth tables which can simplify the process of designing logic circuits.

3. Subcircuits: Allowing users to use well-known or create modular sub-circuits making complex designs easier to manage and produce.

4. Adding tests for GUI: Some form of testing could be included as it may provide some guidance about important features to include for a good interface as well as ensuring certain processes are working as expected rather than unexpectedly finding an error when using the GUI.

References

[1] *Logic Circuit Diagram Design*, Accessed on 21 May 2025, Available at https://https://online.visual-paradigm.com

[2] *Agile Team Planner*, Accessed on 23 May 2025, Available at https://www.figma.com

Appendices

Appendix A - EBNF Example Code with diagrams

1.
```
/*
    This configuration is for a DTYPE flip-flop
    The DTYPE is synchronised by a clock
*/
DEVICES D1:DTYPE,
        D2:DTYPE,
        N1:NAND 2,
        C1:CLOCK 8, # Period is a power of 2
        S1:SWITCH 0,
        S2:SWITCH 1,
        S3:SWITCH 0 ;

/* connect inputs and outputs */
CONNECT S1 > D1.SET,
        S1 > D2.SET,
        S2 > D1.DATA,
        S3 > D1.CLEAR,
        S3 > D2.CLEAR,

        C1 > D1.CLK,
        C1 > D2.CLK,

        D1.Q > D2.DATA,
        D2.Q > N1.I1,
        D2.QBAR > N1.I2 ;

MONITOR D1.QBAR,
        N1 ;

END
```

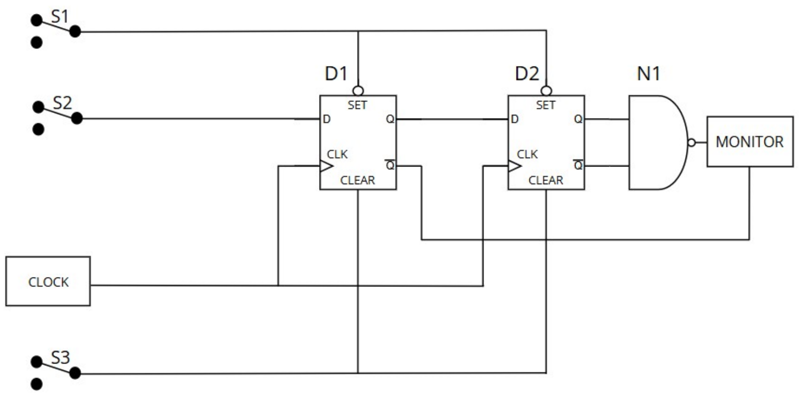


Figure 2: Visual logic circuit design for EBNF (example one) (1)

2.
```
/* This configuration is a full-adder */

/* name all devices */
DEVICES X1:XOR,
        X2:XOR,
        A1:AND 2,
        A2:AND 2,
        NO1:NOR 2,
        O1:OR 2,
        S1:SWITCH 1,
        S2:SWITCH 1,
        S3:SWITCH 0 ;

/* connect inputs and outputs */
CONNECT S1 > X1.I1,
        S1 > A1.I1,
        S2 > X1.I2,
        S2 > A1.I2,
        S3 > X2.I2,
        S3 > A2.I2,
        X1 > X2.I1,
        X1 > A2.I1,
        X2 > NO1.I1,
        A1 > O1.I1,
        A2 > O1.I2,
        O1 > NO1.I2 ;

/* monitor particular signals */
MONITOR X2, # this is the summer bit
        O1, # this is the carry bit
        NO1 ;

END
```

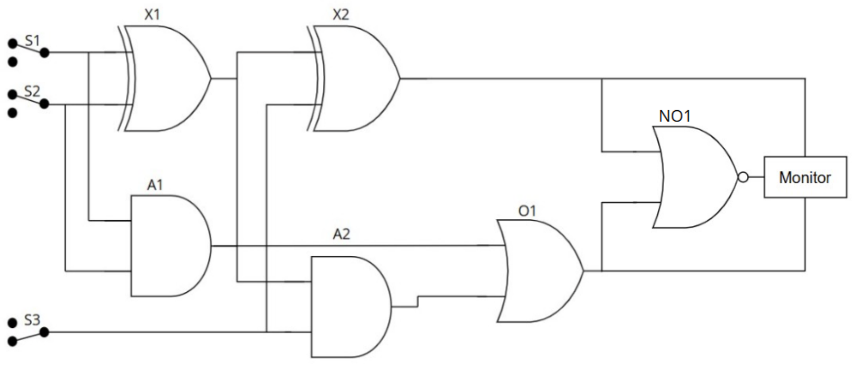


Figure 3: Visual logic circuit design for EBNF (example two) (1)

3.
```
/*
    This configuration is for a DTYPE flip-flop
    The DTYPE is synchronised by a signal generator
*/
DEVICES D1:DTYPE,
        D2:DTYPE,
        N1:NAND 2,
        SI1:SIGGEN 10100, # periodic signal
        S1:SWITCH 0,
        S2:SWITCH 1,
        S3:SWITCH 0 ;

/* connect inputs and outputs */
CONNECT S1 > D1.SET,
        S1 > D2.SET,
        S2 > D1.DATA,
        S3 > D1.CLEAR,
        S3 > D2.CLEAR,

        SI1 > D1.CLK,
        SI1 > D2.CLK,

        D1.Q > D2.DATA,
        D2.Q > N1.I1,
        D2.QBAR > N1.I2 ;

/* monitor certain signals */
MONITOR D1.QBAR,
        N1;

END
```

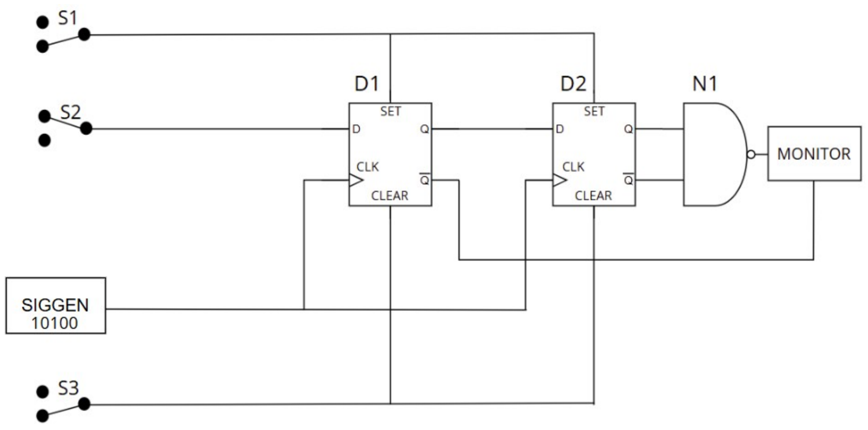


Figure 4: Visual logic circuit design for EBNF (example Three) (1)

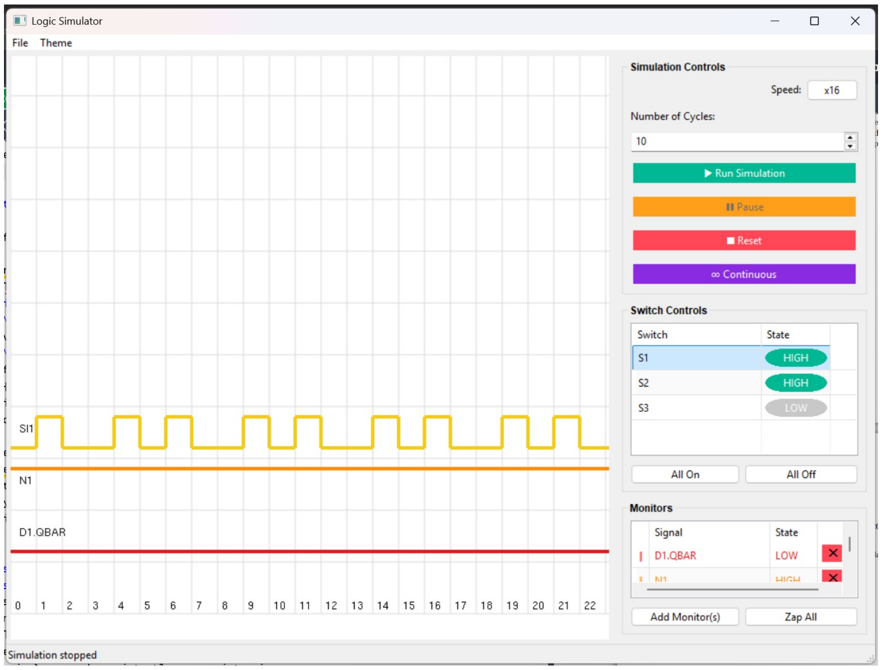


Figure 5: Results of running third example file ?? (1)

Appendix B - Logic language specification

```
specfile = devices, {devices | connection | monitor} ;

devices = "DEVICES ", device, {",", device}, eol ;

device = name, ":", ((("CLOCK" | "AND" | "NAND" | "OR" | "NOR"), " ", posnumber ) | ("SIGGEN", " ", bitstring) | ("SWITCH", " ", bit) | ("DTYPE" | "XOR")) ;


connection = "CONNECT ", con, { ",", con}, eol ;

con = signal, ">", signal ;

pinname = numberedpin | fixedpin ;

numberedpin = "I", ("1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" | "11" | "12" | "13" | "14" | "15" | "16") ;

fixedpin = "DATA"| "CLK" | "SET" | "CLEAR" | "Q" | "QBAR" ;


monitor = "MONITOR ", signal, { ",", signal}, eol ;


end = "END" ;


name = letter, {letter | digit} ;
```

```

eol = ";" ;

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N
    " | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
     | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p"
    | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;

posdigit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

posnumber = posdigit, {digit} ;

bit = "0" | "1" ;

bitstring = bit, {bit} ;
```

Appendix C - User guide

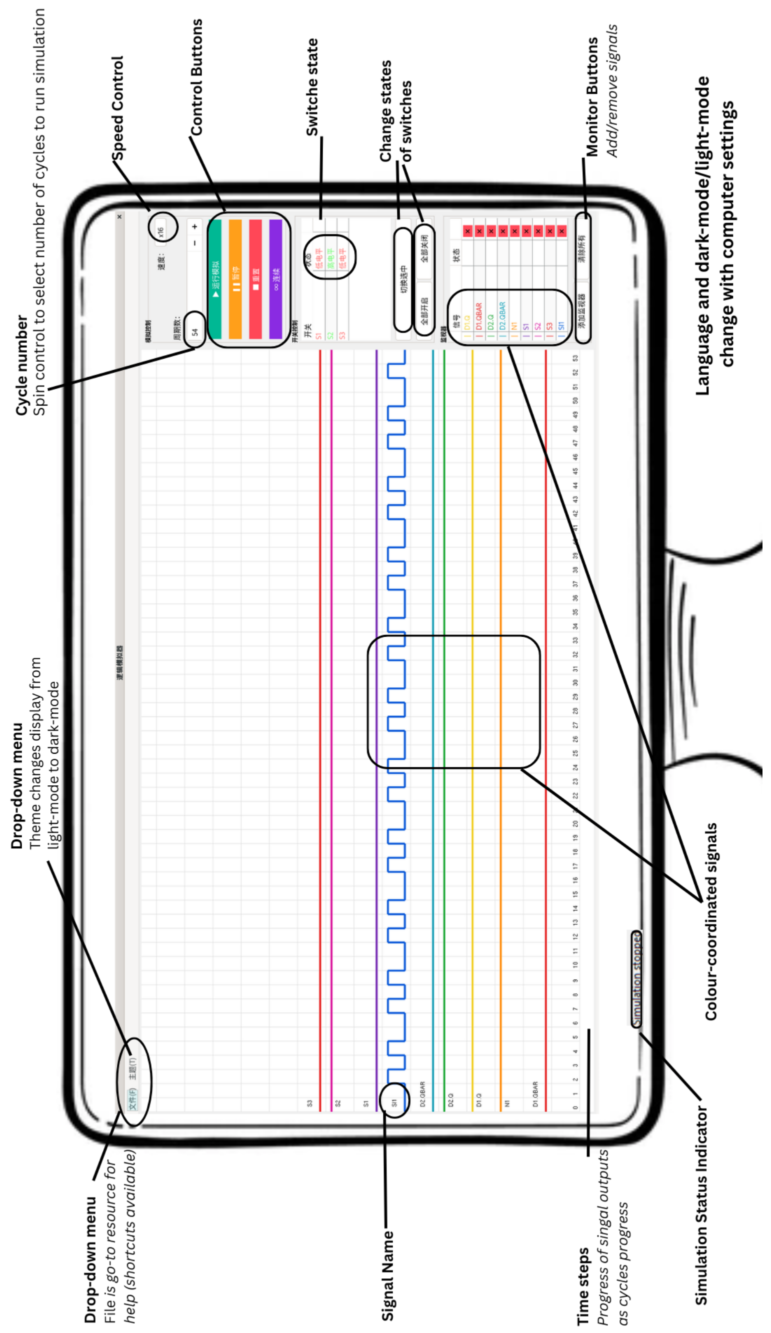

Cycle number
Spin control to select number of cycles to run simulation
Speed Control
Control Buttons
Switche state
Change states of switches
Monitor Buttons
Add/remove signals
Language and dark-mode/light-mode change with computer settings
Drop-down menu
Theme changes display from light-mode to dark-mode
Colour-coordinated signals
Drop-down menu
File is go-to resource for help (shortcuts available)
Signal Name
Time steps
Progress of singal outputs as cycles progress
Simulation Status Indicator
Simulation stopped

Figure 6: GUI user guide (1)

Appendix D - Final team folder

```
+- prelim/
+- README.md
+- .gitignore
+- logsim/
   +- test_parser.py
   +- test parser/ (e.g. test_print_error_devices.txt, ...)
   +- test_maintenance.py
   +- test maintenance/ (e.g. siggen_flip_flop_incorrect.txt, ...)
   +- test_scanner.py
   +- test scanner/ (e.g. test_print_error.txt, ...)
   +- test_names.py, test_network.py, test_monitors.py, test_devices.py
   +- locale/
      +- ta_IN/LC_MESSAGES/{messages.po, messages.mo}
      +- zh_CN, es_ES, kn_IN, yo_NG
   +- logsim.py
   +- EBNF/ (e.g. EBNFv3.txt, ...)
   +- scanner.py, names.py, parse.py, devices.py, network.py
   +- monitors.py, userint.py, gui.py
   +- clock_flip_flop.txt
```

Test files are organised into folders (test scanner, test parser, test maintenance) containing text files with perfect example code and code with errors. Description of provided python files (scanner, network, parser, devices, logism, gui) and key functions in each.

1. **prelim** Preliminary exercise builds up names functions such as lookup.
2. **test_parser.py, test_maintenance.py, etc.** Every module is tested, collectively amounting to 70 unit pytests.
3. **test_parser/, test_maintenance/, etc.** Together, they contain 21 text files with correct and broken code written in our custom programming language.
4. **messages.po**. This contains translations of every string displayed in GUI.
5. **messages.mo**. This is a byte file compiled from the messages.po file.
6. **EBNF/** Formal grammar specifications used by the parser. There are 3 versions, with the latest version including siggen.
7. **names.py** assigns keywords and user defined names with a unique ID.
8. **scanner.py** picks up symbols in the logic definition file and passes them to the parser.
9. **parse.py** interprets symbols, creates devices, builds a network and flags errors.
10. **devices.py** creates devices and tracks their states
11. **network** verifies connections and builds a circuit.

12. **monitor** tracks signals being monitored.
13. **User interface** allows users to interact with the logic simulator from the command line.
14. **GUI** allows users to interact with the logic simulator using a mouse.
15. **clock_flip_flop.txt, siggen_flip_flop.txt, full_adder.txt** Example definition files for testing GUI..