UNIVERSITY OF CAMBRIDGE
Department of Engineering

Part IIA Project: GF2(P2) Software

Narmeephan Arunthavarajah na596@cam.ac.uk
P2 Team 05
Homerton College
Department of Engineering

Date: November 4, 2025

Contents

1. Introduction

This project involved the design and implementation of a logic simulator capable of modelling and simulating both logic circuits. The simulator processes a custom logic description language, defined using EBNF, which allows users to specify devices, their interconnections, and monitored signals in a structured text file. The system validates the syntax and semantics of the input, ensuring correctness before execution.

Key components of the project included:

1. Lexical analysis and parsing: A scanner converts the input file into symbols, while a parser verifies syntactic correctness and constructs the corresponding logic network.
2. Error handling: Comprehensive checks for semantic errors (e.g., invalid connections, undefined devices) with informative feedback.
3. Simulation engine: Integration with supplied modules (devices, network, monitors) to execute the circuit and track signal propagation.
4. Graphical User Interface (GUI): A wxPython-based interface with OpenGL rendering for visualising signal traces, enabling interactive control (e.g., running simulations, toggling switches).

The project emphasised software engineering best practices, including modular design, collaborative development (via Git), and rigorous testing (using pytest). The final deliverable not only met the client's requirements but also demonstrated adaptability through a maintenance phase, where modifications were efficiently implemented to address new specifications.

2. Description of the Logic Simulator and Software Structure

2.1 Function

The logic simulator is a Python-based tool for modelling and simulating digital circuits defined via a custom text-based description language. Users specify circuits by writing a definition file adhering to an EBNF syntax, we defined. The language supports three primary constructs:

- **Devices** are declared in the format `name:device_type parameter`, where the parameter is device-specific. For example, a clock device might be declared as `CLK1:CLOCK 4` (denoting a 4-cycle period), while a switch requires an initial state (`0` or `1`). Notably, devices like XOR and DTYPE require no parameters.
- **Connections** are defined using the arrow syntax `input > output`, such as `G1 > D1.DATA`.
- **Monitors** are specified by listing signal names (e.g., `D1.Q`), with individual pins excluded from monitoring.

The simulator performs rigorous validation, checking for both syntax errors (e.g., missing colons or invalid symbols) and **semantic errors** (e.g., undefined devices, floating pins, or non-binary waveforms). Errors are reported with precise line and position markers, along with a summary count. For example:

```
Expected a bit (0 or 1)
LINE 10:
```

```
        S1:SWITCH ,
                  ^

Device not found
LINE 15:
CONNECT S1 > D1.SET,
        ^

Device not found
LINE 16:
        S1 > D2.SET,
        ^

Summary: 3 error/s found
```

2.2 Software Architecture

The system follows a modular pipeline, depicted in Figure 1 and described below:

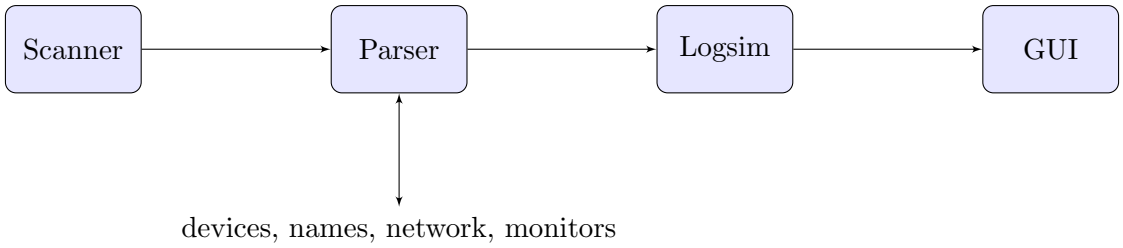

Figure 1: Workflow of the logic simulator

2.2.1 Scanner

The scanner performs lexical analysis, reading the input file symbol-by-symbol (e.g., names, numbers, punctuation) via the `get_symbol()` function. It skips comments (delimited by `/* ... */` or `# ... \n`) and tracks line numbers and positions for error reporting. However, it does not validate syntax; its primary role is to generate symbols with types (e.g., `KEYWORD`, `NAME`) and IDs (e.g., `DEVICES.ID`) for the parser.

2.2.2 Parser

The parser validates syntax and semantics while constructing the circuit. It processes symbols from the scanner and enforces the EBNF rules. For example, the `device()` function checks for the `name:device_type` pattern, returning errors like `NO_COLON` if the colon is missing. Valid constructs are passed to backend functions:

- `make_device()`, `make_connection()`, and `make_monitor()` from `devices`, `network`, and `monitors` modules, respectively.
- Errors (e.g., `INVALID_PARAMETER` or `NONBINARY_WAVEFORM`) enforce line skipping (to next comma or semicolon) before further processing, while `NO_ERROR` allows continuation.

After parsing, `check_network()` ensures the circuit is valid (e.g., no floating pins).

2.2.3 User Interface

The simulator offers two interfaces:

- **Text-based (`userint`)**: Launched via `python logsim.py -c [file].txt`, it supports basic commands like `r N` (simulate `N` cycles).
- **GUI (custom)**: Activated by `python logsim.py [file].txt`, it supports multiple languages (e.g., English, Spanish, Chinese) and provides visual signal traces and interactive controls (user guide with functionality shown in the appendix). The GUI uses the system's default language settings unless overridden (e.g., `LANG=zh_CN.utf8`).

2.3 Key Strengths

The simulator's modular design enables robust error handling, clear user feedback, and maintainability. The separation of scanner, parser, and logsim modules facilitated parallel development. The multilingual GUI increases accessibility.

3. Commentary on the approach taken to teamwork and collaboration. Did things progress as anticipated? Which online tools and platforms did you use?

We continued using the Agile Scrum framework to manage teamwork using Figma. Tasks were tracked in a project backlog and progressed through weekly sprints with daily stand-ups to maintain alignment. The flexibility of Agile proved essential when we discovered, after submitting our first report, that our initial EBNF rules were overly complex and allowed unnecessary parameter definitions. This required significant rework and set us back temporarily. However, using Agile, we could quickly update the backlog by moving completed tasks back from the "Done" section and easily adapt our sprint plans.

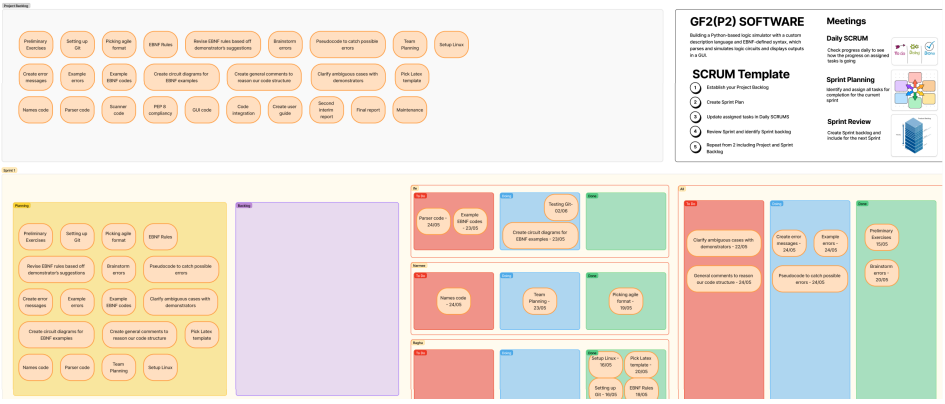

Figure 2: Screenshot of Agile sprint plan on Figma

We also used Git branches to manage collaborative coding efficiently. Branching allowed multiple team members to work in parallel, test independently, and integrate safely. This approach helped us maintain code stability despite major changes.

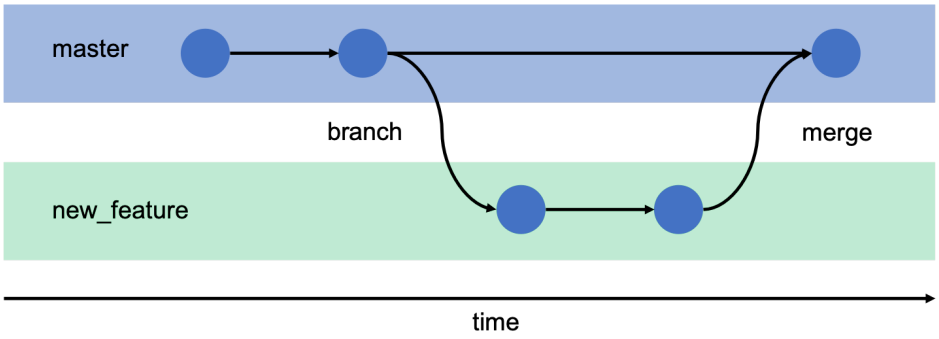

Figure 3: Visual representation of Git branching workflow

Our use of these tools meant that even unexpected setbacks could be managed effectively without derailing the overall project timeline.

4. Description of the software written and/or modified by you.

Throughout the project, I worked closely with my group of three in a collaborative environment. We typically sat and coded together, constantly discussing ideas and reviewing each other's contributions. As a result, it is difficult to isolate entire sections of the project that were completed individually, since much of the development was a joint effort. Nevertheless, I will outline the key functions and components where I made significant contributions.

4.1 Name Lookup Function

I developed the `name_lookup` function, which returns a list of name IDs corresponding to each string in the provided `name_string_list`. If a name string is not already present in the names list, the function automatically adds it. This ensured that all names were correctly tracked and managed throughout the system.

4.2 Parser Functions

After jointly developing the scanner module with my group, I individually contributed to key parts of the parser.

I wrote the `device` function, which implements the logic for correctly identifying and processing device definitions. It ensures each device line matches the required syntax: `name:device_type`.

I also implemented the logic for validating monitor definitions. This included verifying the naming conventions, creating the corresponding monitor objects, and handling errors accurately according to our EBNF rules. This part of the code was carefully iterated and tested, as the group collaboratively attempted to break the system to ensure its robustness.

4.3 Graphical User Interface (GUI)

I was responsible for designing and coding the entire Graphical User Interface (GUI). This involved creating functional buttons linked to system operations and ensuring the GUI fulfilled all required features. Details and images of the GUI features are provided in the User Guide in the Appendix.

4.4 Maintenance and Additional Features

For system maintenance and usability improvements, I implemented the continuous run function that allows the logic simulation to cycle until manually stopped. This involved redesigning the GUI to support continuous running, making the signal graph scrollable as well as draggable and forcing the GUI to follow the signal progression in real time.

5. Description of the test procedures adopted.

To ensure the reliability and correctness of our software, we implemented systematic test procedures for each module. Specifically, we wrote unit tests for the `names`, `scanner`, `parser`, and `maintenance` components. For each module, we created a dedicated Python test file using the `pytest` framework.

Each test was associated with a corresponding input file or error case stored in clearly organised folders. We maintained a structured file hierarchy where test input files and their expected outputs were grouped within individual folders for each module. This structure significantly improved efficiency, allowing us to quickly identify relevant test cases and rerun the entire suite whenever code changes were made.

Our modular coding approach further supported this testing strategy, as changes in one part of the code rarely caused unintended side effects in other parts. Regular use of the test suite gave us confidence that the overall system integrity remained intact throughout development.

This process of modular design combined with structured and automated testing enabled fast, reliable verification during both the initial implementation and later maintenance stages of the project.

6. Conclusions and recommendations for improvements.

This project provided valuable experience in the full software engineering lifecycle, from specification and design to testing and maintenance. Working collaboratively within a small team under real project constraints highlighted the importance of clear communication, modular design, and efficient version control.

The project was successful, as we were able to creates a fully functional logic simulator that met the client's requirements, with both text-based and graphical user interfaces. The maintenance phase further extended the functionality by introducing a continuous run mode, a new device type and internationalisation.

Recommendations for improvements would be:

- **Cross-Platform GUI Compatibility:** One major limitation was that the final GUI was required to run on the DPO Linux workstations. While our GUI was fully functional in this environment, it would have been good to develop a solution that worked across both Linux and Windows platforms. We did, in fact, create a Windows version of the GUI, with more visually appealing switches and additional features, which is fully operational and remains available in our team's Git repository.

- **Automated GUI Testing:** While our back-end modules were extensively tested using `pytest`, formal unit testing of the GUI was not introduced. Introducing automated GUI testing tools in future projects could improve test coverage and reduce interface bugs.

Overall, the project provided an excellent platform to apply and develop both technical and collaborative skills in a realistic software development environment. The lessons learned regarding cross-platform design, modularity, and user-centred interface improvements will be highly valuable for future projects.

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

7. Appendix

7.1 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
```

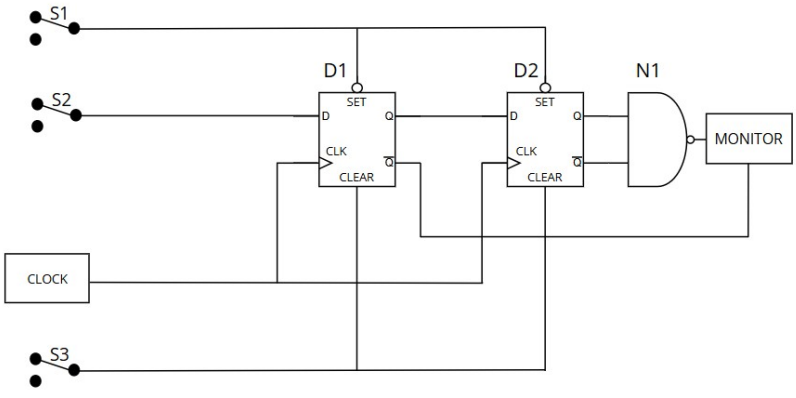

Figure 4: Visual logic circuit design for EBNF (example one) (1)

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
```

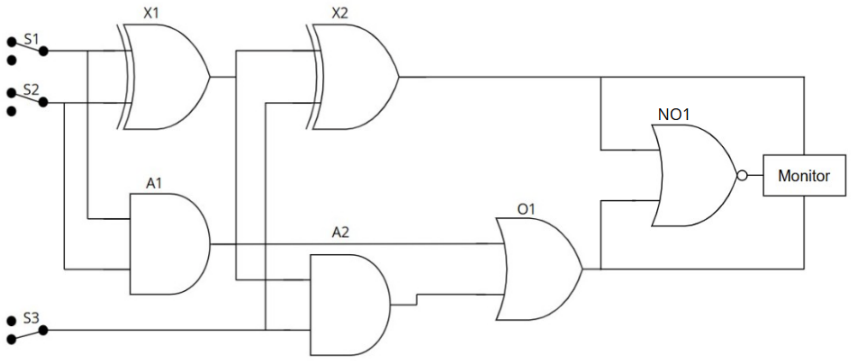

Figure 5: Visual logic circuit design for EBNF (example two) (1)

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
```

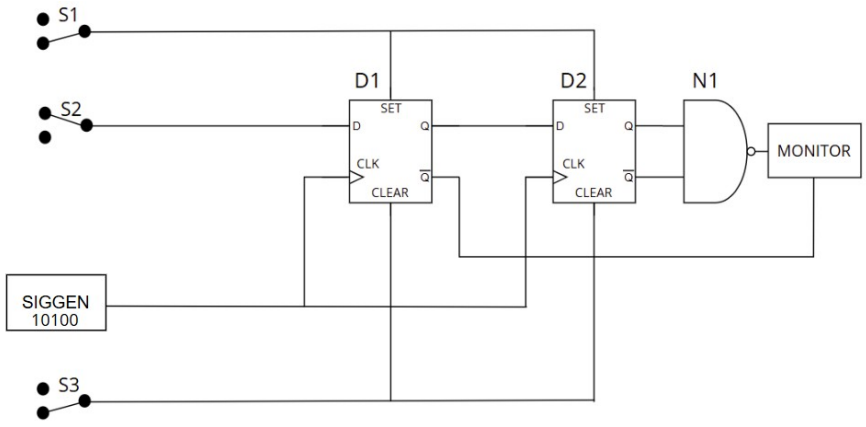

Figure 6: Visual logic circuit design for EBNF (example Three) (1)

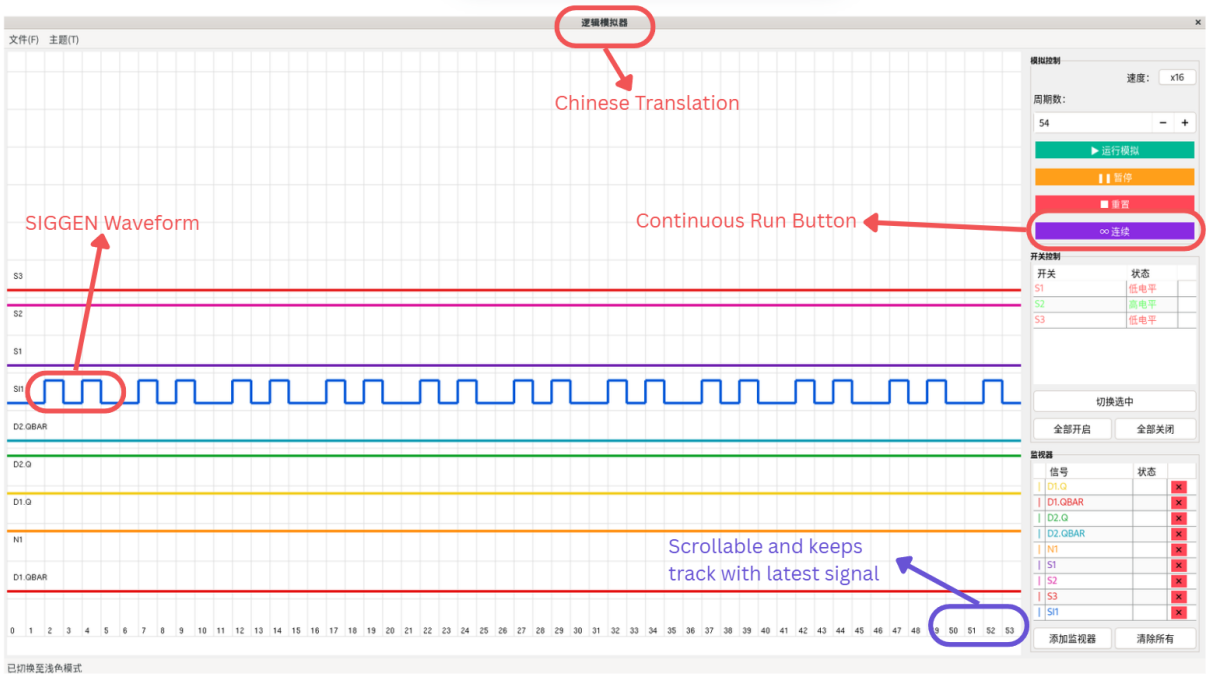

Figure 7: Results of running third example file in light mode

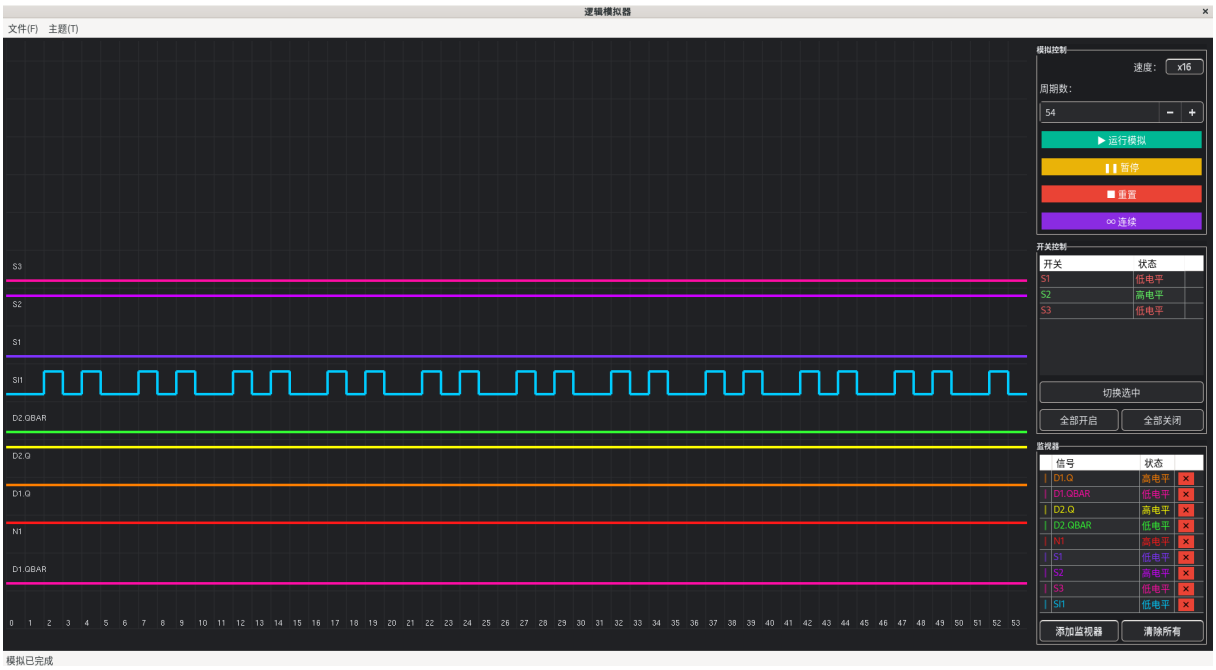

Figure 8: Results of running third example file in dark mode

7.2 Specification of logic description language

```
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} ;
```

7.3 User Guide

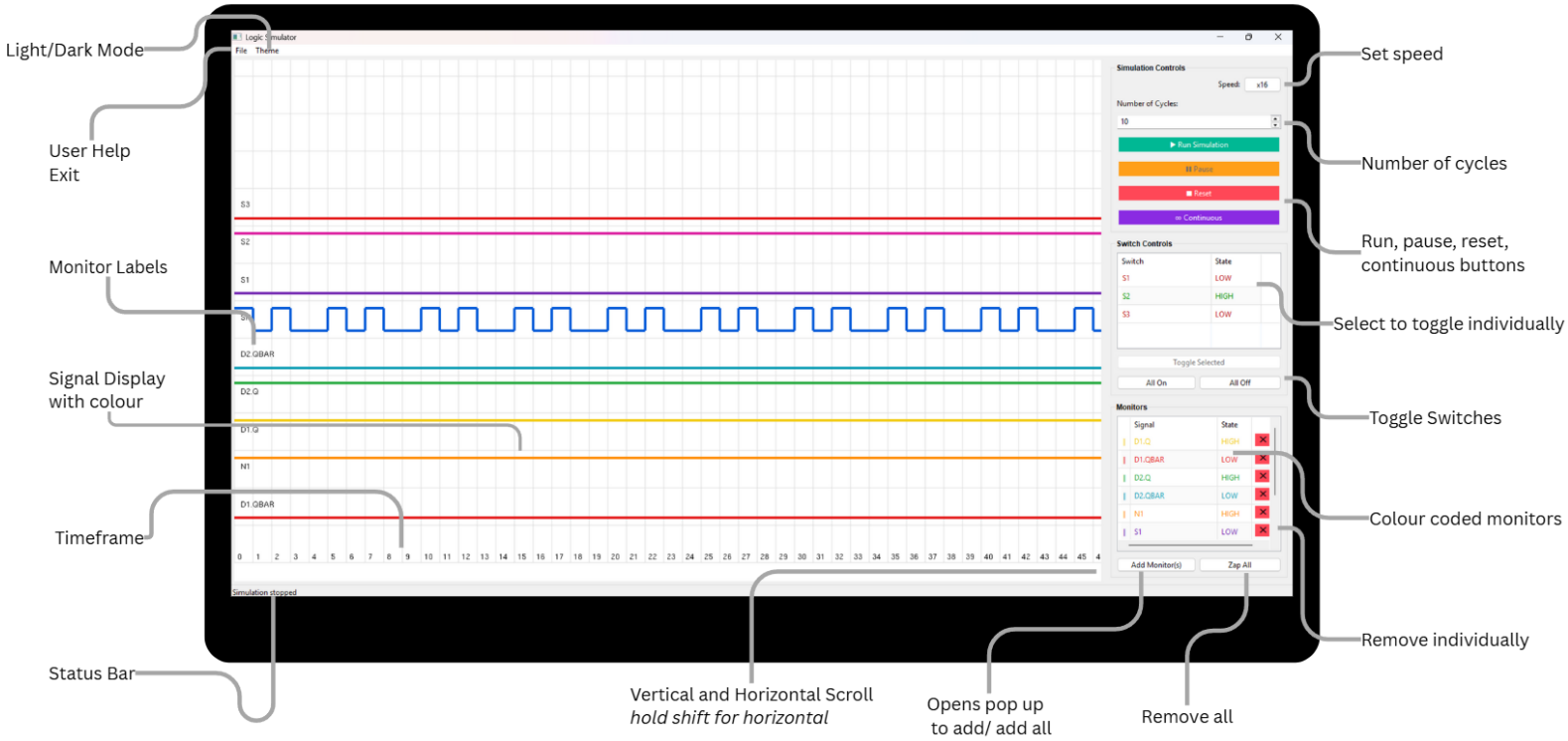

Figure 9: User Guide for GUI after maintenance

7.4 Repository Structure Overview

```
+- 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
```

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..