UNIVERSITY OF CAMBRIDGE
Department of Engineering

Part IIA Project: GF2(P2) Software

Raghavendra Narayan Rao RN436@CAM.AC.UK
RN436
P2 Team 05
Homerton College

Date: June 12, 2025

Contents

1. Introduction

This report details the development of a logic simulator and the outcomes. The objective of the logic simulator is to provide users with visualisation of logic circuits and to configure inputs into the circuit. To achieve this, we built a modular software. The logic simulator is broken down into two phases, the compilation phase and the graphical user interface (GUI). During compilation, the scanner and parser modules work together to identify symbols in the user's program file and interpret the symbols as instructions. These instructions are then carried out by the devices, network and monitor modules to build a circuit. Errors are displayed at the end of the compilation phase. If compilation is scuccessful, the GUI displays signals that the user intends to monitor.

This report also outlines how the project was managed. Specifically, an agile framework was used, along with pair programming. This allowed for a steamlined development process.

Lastly, the logic simulator underwent a maintenance phase where internationalisation and a signal generator were added. The GUI was also updated to mimic an oscilloscope display. To run the example files, run the following command with your preferred file:

```
python logsim.py example_definition_files/siggen_flip_flop.txt
```

2. Test Procedures

1. **Modular Tests**
   We have extensively used pytest fixtures in our pytests. This allowed us to easily scale the number of pytests to 70. Every test invokes a single class object to be tested. The tests are also given descriptive names. These practices have allowed for smooth debugging.

2. **Sessions for breaking code**
   Together with our friends, we regularly held code breaking sessions to capture errors that we may have missed out. This included repeated device specifications and faulty error recovery.

3. **capsys**
   Capsys allows us to log whatever has been printed in stdout (By default, python prints to stdout). Using capsys, our tests are very thorough.

   - The number of errors is printed onto the terminal so this is checked against `parser.error_count`.
   - The error messages (e.g. `Expected a comma`) and the order they are caught is tested.
   - A cursor indicating where the error occured is printed onto terminal. This is also tested with an exact string match for all errors.

4. **Style**
   PEP8 can be checked using pycodestyle but a linter assists in programmers conforming to PEP8 while coding. We used Pylint and mypy VS Code extensions to lint our code.

3. Logic Simulator

3.1 Function

Our Logic Simulator allows engineers to study and visualise signals in logic circuits. This is particularly useful when checking for validity of circuits and catching hazards (static/dynamic) prior to hardware implementation. This logic simulator supports the design of both combinational and synchronous circuits, including those driven by clocks or custom signal generators. The steps to use our logic simulator are as follows:

1. **Define a logic circuit** using our custom language. Logic gates and input sources (switches, clocks and custom signal generators) can be created and connected. It also allows for specification of signals to be monitored.
2. **Run the GUI**. The GUI allows users to monitor signals of their choice as well as toggle switches in real time. The user can run the simulation for either a given number of cycles or indefinitely. The signals will be displayed in a familiar osscilloscope setting. Check the user guide for details.

3.2 Software structure

1. **names**. The names module assigns keywords and user defined names with a unique ID.
2. **scanner**
   The scanner module picks up semantic symbols (keywords and punctuations) in the logic definition file. It ignores comments and delimiters. These symbols are passed to the parser simultaneously.
3. **parser**
   The parser receives symbols from the scanner, interprets them as devices (using the devices module) and builds a circuit from defined connections (using the network module). It also flags errors in the program file if compilation fails.
4. **devices**
   The devices module creates devices. Every device parameter is verified for correctness (e.g. clock's period has to be positive). The device's state and clock counter are tracked. Any errors in creating a device are flagged by this module, which the parser collates.
5. **network**
   The network module verifies that defined connections are valid and then builds a circuit. Similar to the devices module, it flags errors (for invalid connections).
6. **monitor**
   This module keeps track of signals that are being monitored. It also offers methods to add and remove monitored signals.
7. **User interface** allows users to interact with the logic simulator from the command line. It has all functionalities of the GUI minus the graphics.
8. **GUI**
   If no errors are caught by the parser, the program file is compiled successfully and the GUI is called. Along with the signals specified in the program file, the user can also add or remove signals to monitor.

4. Team Planning

4.1 Agile Scrum Framework

Our team adopted the Agile Scrum framework (2) to manage the project lifecycle. Each weekly sprint began with planning and task assignment, followed by daily Scrum meetings to track progress. Incomplete tasks were reviewed and moved to the next sprint backlog.

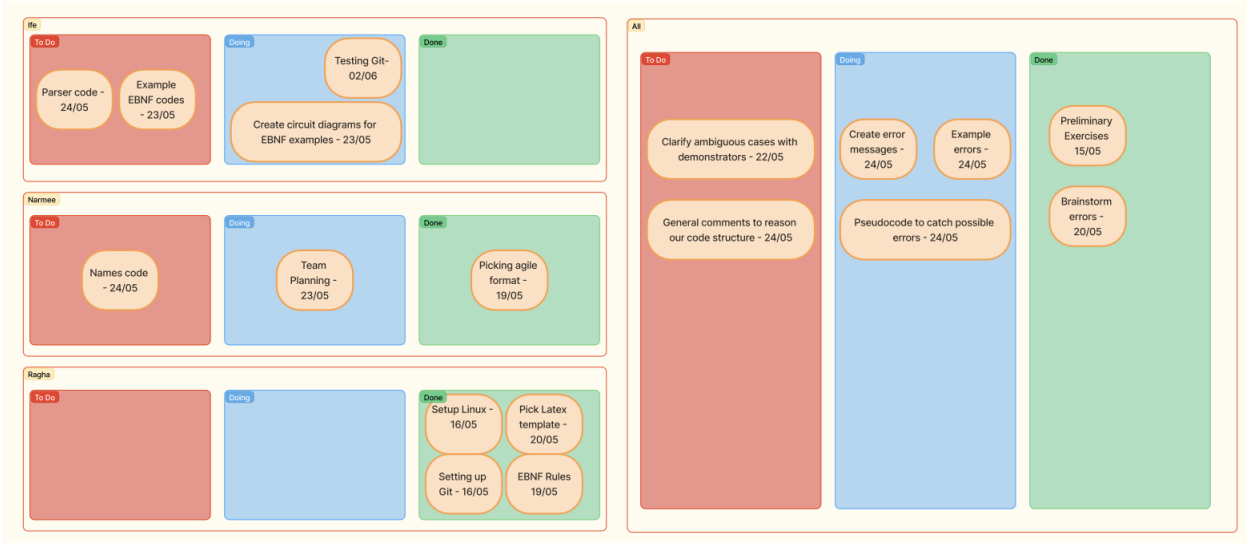


Figure 1: Project plan in the middle of sprint 1 (22/05)

4.2 Git Branches

For this project, we implemented Git branching to efficiently manage code changes and maintain stable working versions.

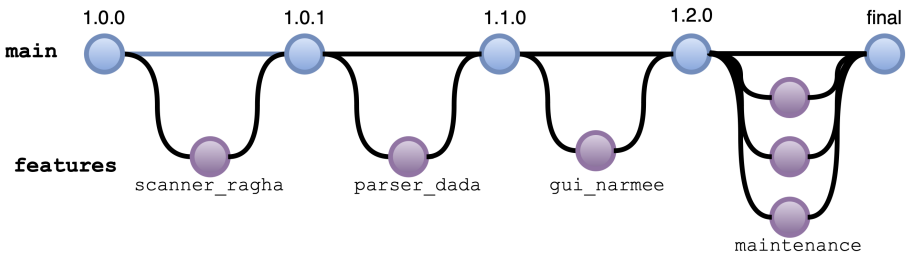


Figure 2: Visual representation of the git branch process

4.3 Setbacks

Under the guise of easing maintenance, our group initially created an EBNF with excessive freedom, resulting in numerous semantic errors. It became apparent after our first interim report that we had to majorly revise our approach. As advised by our supervisors, we made the right changes and began with a strong foundation, allowing us to ace our second report.

5. My Contributions

5.1 Pair Programming

We adopted the pair programming agile technique for the whole of this project so each one of us contributed to every part of the repository. Therefore, it will be difficult to single out an individual contribution. Here is my best attempt at highlighting areas that I contributed significantly.

5.2 EBNF

I wrote EBNF version 1, which we used for the first interim report. The latest EBNF version we have is a tweak of the original version. Key ideas that I designed are ensuring the correct ordering of device definitions before connections, creating valid input keywords (I1 to I16) for gates and allowing for multiple devices and connections to be defined in one line.

5.3 Names

I developed a majority of the logic behind names and wrote a significant portion of the pytests for names.

1. I used 2 dictionaries as fields of the Names class to implement O(1) retrieval of names and ids.
2. I added pytest fixtures in all tests to reduce repeated code and make tests modular.

5.4 Scanner

1. I created the error recovery system for the scanner. This recovery system dynamically allocates a recovery symbol based on the line the scanner is on. For example, if the scanner is currently traversing device definitions, it will recover at the next comma or semicolon found. Suppose commas or semicolons are also missed, keywords are used as stopping symbols instead. This is achieved using a `self.parent` flag.
2. The scanner has to skip comments and delimiters like spaces, tabs and new lines. I implemented this feature in scanner by using while loops to skip these characters before detecting meaningful symbols. I also wrote multiple pytests to test for closed comments without closing symbols and open comments placed after file ends.
3. I edited a function written by a team member to include initial zeroes in siggen's waveform.

5.5 Internationalization

1. As the only team member with a unix based system (Mac), I worked on adding language support. First, I used xgettext to extract all strings displayed in `gui.py`. Next, with google translate's library, I created a python script, `translate.py`, that creates a .po file with the translations I need. I also ran the translations through LLMs for verification. Next, I used msgfmt to create a byte file (.mo) of the .po file. These define a Locale, allowing for translation within wxPython widgets.
2. I have added the ethnic languages of all our team members (tamil, kannada, yoruba). Chinese and spanish were also added as these languages are well supported by wxPython.

GUI

I added the feature where the GUI picks up current system settings for dark mode and displays accordingly.

6. Conclusion

6.1 Conclusion

Our logic simulator provides users with the ability to visualise the output of logic circuits before implementing them in hardware. The user will first define their logic circuit using our custom language and then run the GUI after compilation. Our logic simulator is built with modularity at its core. The various modules enable addition of new devices easily. Adding a custom signal generator only required adding an additional if block in necessary modules. We faced a major setback after the first interim report as our EBNF was not appropriate for the provided modules. Regardless, we bounced back and have developed a working logic simulator.

6.2 Improvements

1. **Switch toggle**
   We initially used a graphical switch button that could be switched on and off with a click on our laptops. However, with the DPO mouse, a double click was required. We could not resolve this bug so we had to remove the aesthetic switch button. With more time, we would like to replace most text with interactive graphics.

2. **System settings**
   We are using system settings to pick up language and theme (light and dark mode). However, these are only picked up during app initialization. Hence, if the user changes their system settings while the app is running, these changes are not picked up. We can account for this by regularly listening for sustem settings.

3. **Type checks**
   With typing (Python 3.5 and above), we can ensure our app runs securely. This is the industry standard but due to time constrains, we havwe not been able to implement type checks.

4. **GUI that has an IDE**
   Currently, the user has to run the logic simulator in 2 stages. The first is the compilation stage and if that passes, then the GUI can be run. Instead, a GUI that directly allows for the logic definition file to be attached or the code to be typed in is preferred.

5. **OS Support**
   At the moment, we have seperate GUIs for unix based systems and windows. This is due to differences in how both systems display widgets and handle interrupts. A single GUI that adopts to any operating system is something we hope to add in the future.

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 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
```

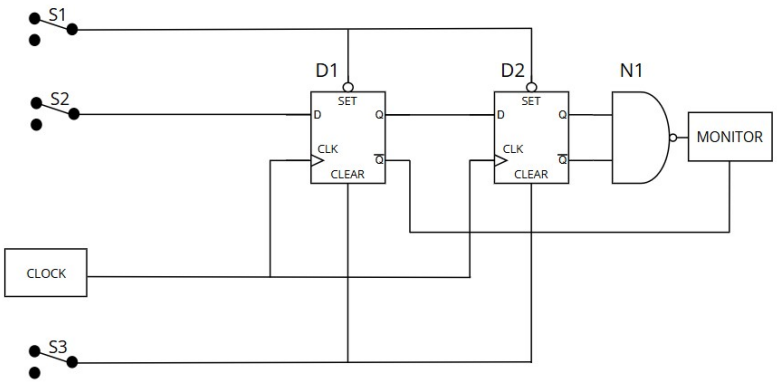


Figure 3: Visual logic circuit design for EBNF (example one) (1)

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
```

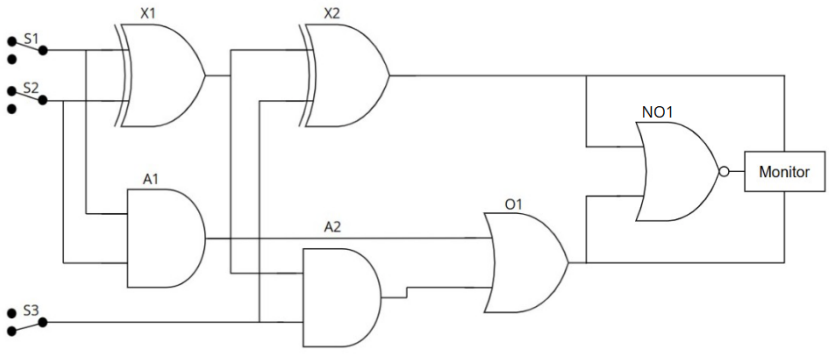


Figure 4: Visual logic circuit design for EBNF (example two) (1)

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
```

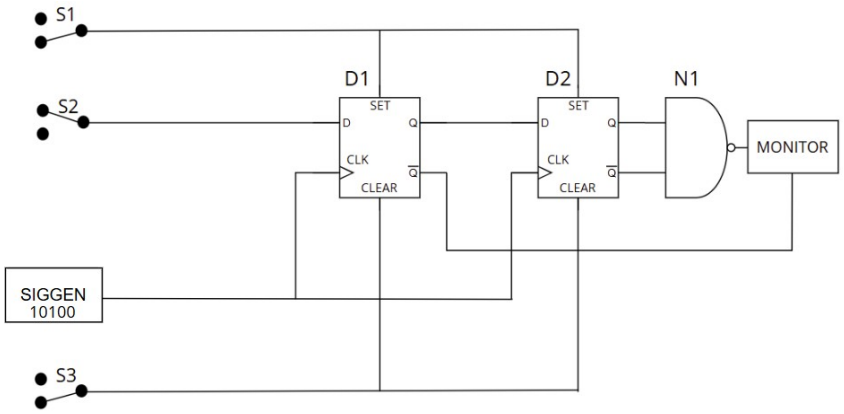


Figure 5: Visual logic circuit design for EBNF (example Three) (1)

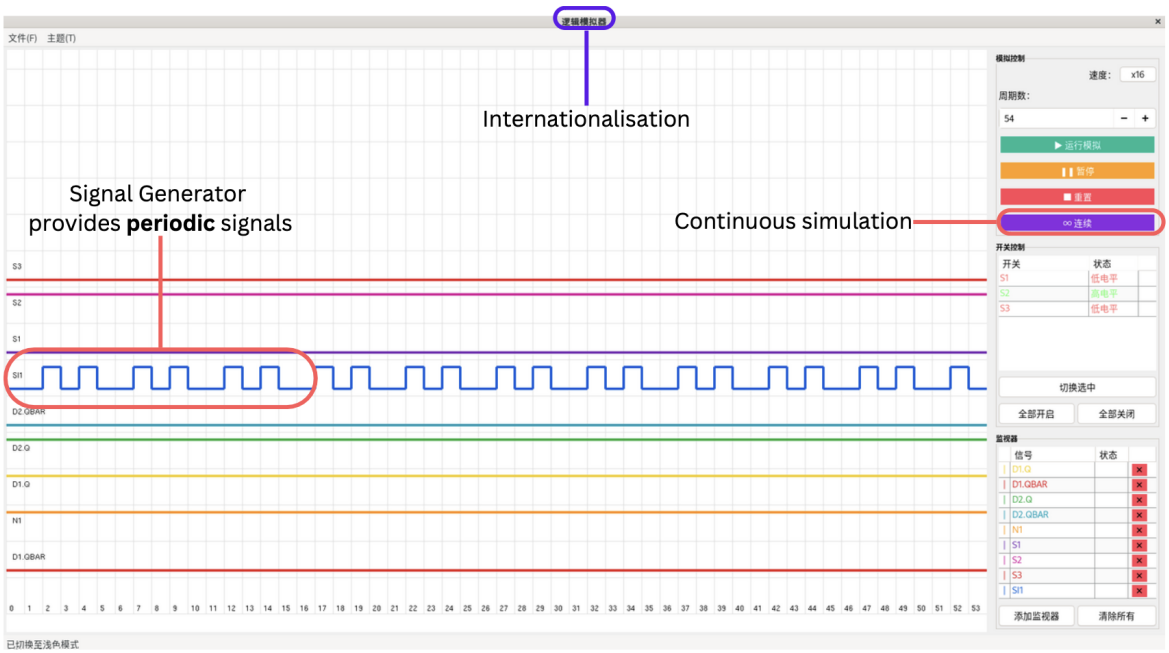


Figure 6: Results of running third example file (5)

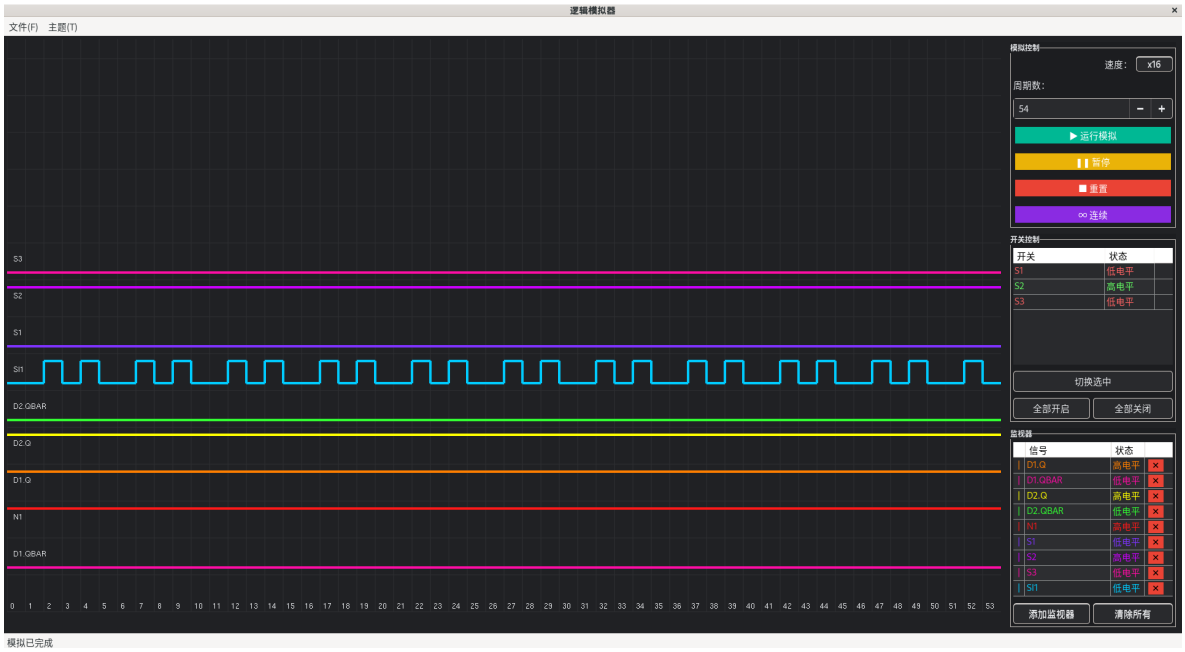


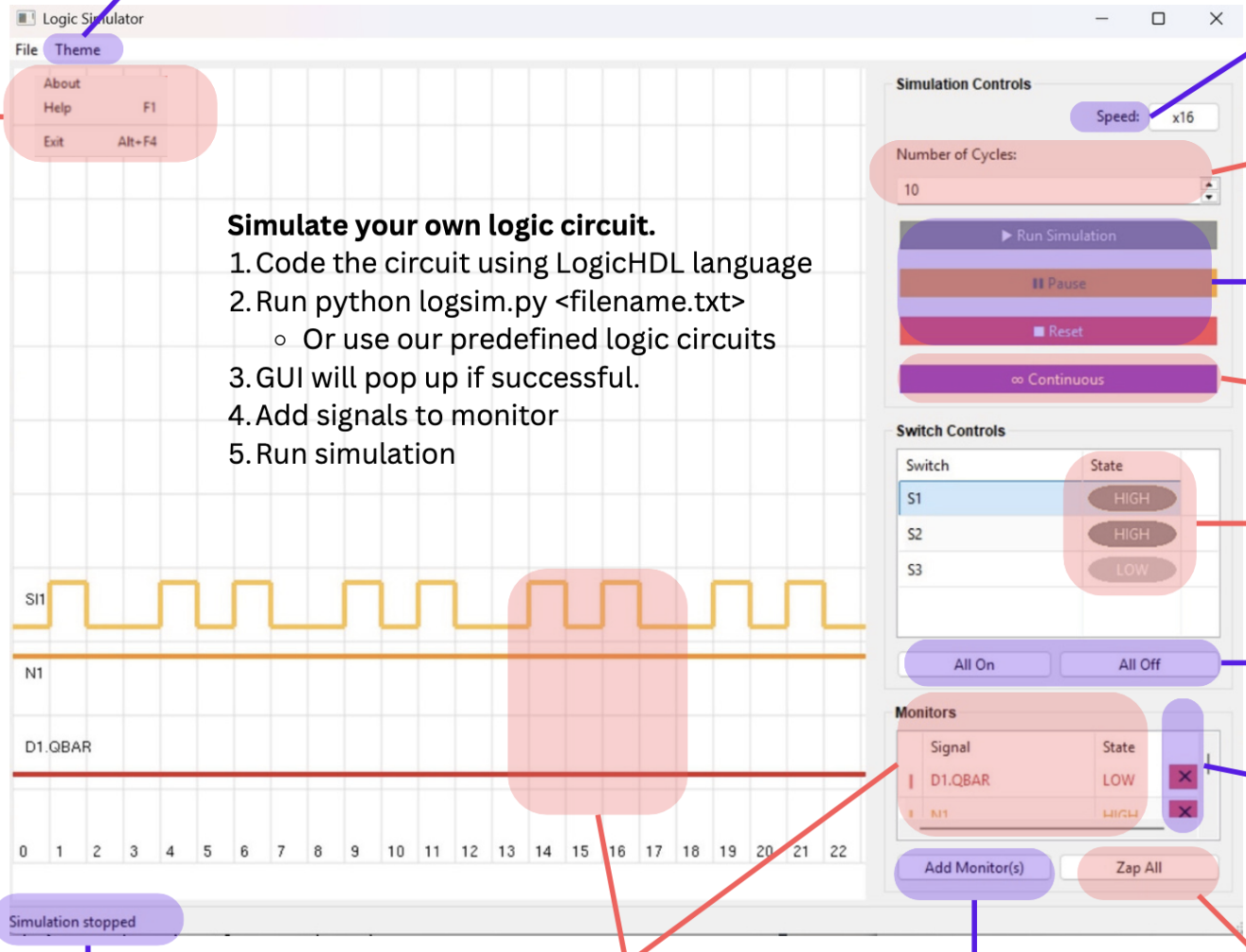
Figure 7: Dark mode set with local system settings

7.2 Appendix B: 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} ;
```

Switch between **Light** and
Dark mode for power saving
(Can also be set with computer settings)

Go-to **resource** for
understanding how to use
the simulator effectively



Control speed
of simulation

Set number of cycles
by **typing** or **spinning**

Start, **pause** or **reset**
simulation plot

Monitor signals **Indefinitely**

Toggle switches

Conveniently switch
on/off **all switches**

Remove individual signals

Remove all signals

Check **status** of simulation

Colour coded signals
for easier reference

Choose **multiple**
monitors to add

7.3 Appendix D: 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
   +- example definition files/ (e.g. siggen_flip_flop.txt, full_adder.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. run **example_definition_files** with `python logsim.py example_definition_files/full_adder.txt`