UNIVERSITY OF
CAMBRIDGE

GF2 Software:
Final Report

Karthik Suresh ks800
Wolfson College
Team 12

June 5 2019

Development of Logic Simulator

Engineering Department
University of Cambridge

1 Introduction

This report aims to outline the path traversed to develop a user-friendly, functional and intuitive Logic Simulator application. The successful development of the Logic Simulator required: thoughtful planning of the software structure; a good understanding between team members; disciplined adherence to code styling and version control practices; and rigorous testing procedures. The sections below give an overview of how certain decisions enabled timely development, minimum rework and enhanced functionality of the final product.

2 The Logic Simulator

2.1 Functionality

In the requirements specified at the start of the project, the Logic Simulator application needed to provide the following functionality:

- Allow a user to load a custom circuit definition file using the command line.
- Provide a graphical user interface through which a user can simulate a specified circuit, set switches, and add/remove monitored outputs.
- Display helpful error messages to the user in response to an attempt to load an erroneous file.

In the maintenance stage of the project the following items were specified as additional requirements the application needed to provide for:

- Allow a user to view simulation outputs in 3D and provide a way for the user to toggle between 2D and 3D views.
- Allow the user to use the Logic Simulator application in another language.
- Allow the user to specify a new signal generator type of device in their circuits.

In the final version of the Logic Simulator application produced, all the functions specified above have been implemented, and additional functionality has been included to maximise the utility the system provides for a user. The graphical user interface is made up of four main sections that work together to enable the user to take full advantage of the application's functions: the control center, the trace canvas, the activity log, and the menu bar.

The control center is a vertical bar on the right side of the graphical user interface that contains most of the application's user interaction mechanisms. The user can perform actions such as run/-continue simulations for a specified number of cycles, configure devices, toggle views, add/remove monitors, and reload files by clicking buttons or typing within boxes situated in the control center. In the 'Configure Devices' section of the control center, in addition to configuring switches additional functionality has been provided to allow users to configure the half-period of clocks. Upon selecting a configurable device the default input of the text box below will change to reflect

the current value of the device's configuration variable. This behavior was included to provide the user with useful information that could help them validate their future actions. In the event a circuit is loaded with no configurable variable the 'Apply' button in the 'Configure Devices' section is disabled to highlight to the user that no configurable device exists. Similarly, all buttons and inputs in the control center are disabled when the user loads an erroneous file. At the bottom of the control center, the buttons 'Reset' and 'Reload' provide additional functions to enhance a user's work flow. The 'Reset' button returns the view of both the 2D and 3D display to its initial view. This is useful especially in the 3D case as it provides users with a way to quickly restart an interaction with a 3D trace. The 'Reload' button was included as a function after feedback during the second interim project milestone. Using this button, users can load new changes to the file currently loaded in the interface without having to navigate through the menu bar and a file explorer.

The trace canvas is the section that occupies the majority of the space in the graphical user interface. In 2D view the trace canvas is actually a group of three canvases that represent labels, traces and an axis. A user can pan the view of the three canvases using their mouse, however labels will always remain aligned to their corresponding traces, and will always remain in view, thus preventing the need for users to stress their fine motor skills. Conversely, in 3D users can use their mouse to pan, rotate and zoom freely in their view. This provides them with the flexibility to interpret 3D traces from a variety of perspectives.

The activity log is located at the bottom of the application window and its function is to provide feedback or an acknowledgement for almost every user action. The activity log also displays the errors that exist within an incorrectly specified circuit definition file. This provides users with a way to see and act on error messages without having to leave the application window.

Lastly, the menu bar contains options for a user to open a new circuit definition file, close the application, toggle between 2D and 3D views, and toggle the language displayed within the application.

2.2 Software Structure

A modular structure was adopted to write the code for the Logic Simulator application. This involved coding different functions of the software within different classes, and defining methods within classes to provide an interface for integration. The application's code base consists of eleven primary classes as shown in Fig.1 below.

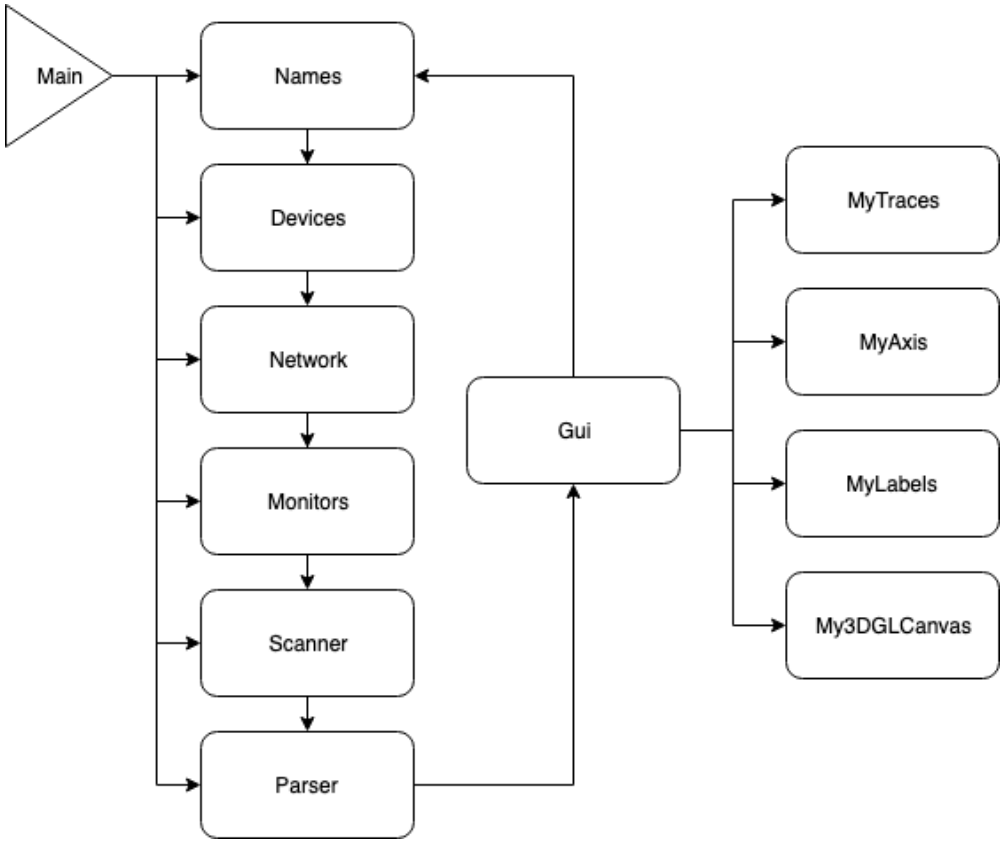

Figure 1: Interaction between application classes.

Within the application's `main` function the `Names` class is instantiated first after command line arguments are parsed successfully. The `Names` class stores all the unique name strings used in any part of the application in a names list, and contains methods to reference and return the indices or strings stored in the names list. Name strings are only added to the `Names` class by either the `Devices` or `Scanner` classes. User defined device names and the names of symbols are stored in the names list by the `Scanner` class while all device type, device input type, and device output type names are stored in the names list by the `Devices` class. The `Parser` class is the only place where the names list is queried.

The `Devices`, `Network`, and `Monitors` classes are the next three class to be instantiated as the application prepares for the creation of new devices, a new network and a corresponding set of monitored device outputs for which signals have to be recorded. Following this, the `Scanner`, and `Parser` classes are initialised. The loaded circuit definition file is first scanned and converted into a list of symbols by `Scanner` before `Parser` then moves through this list to ascertain whether any syntactic errors exist in the loaded circuit definition file. Within the `Parser` class, as device, connection, and monitor definitions are recognised, the device list within the `Devices` class and the monitors list within the `Monitors` class are populated accordingly. Upon finishing the parsing of a circuit definition file without identifying syntactic errors, the `Parser` class then invokes a method within the `Network` class to attempt to create the specified digital circuit. It is at this point that a semantic error check is conducted for the specified circuit. Upon the existence of either type of error, the `Parser` class invokes a method within the `Scanner` class to create and output an

informative error message to the user. Finally, regardless of whether the `Network` class manages to successfully create a network, the `parse_network` method within the `Parser` class will return and an instance of the `Gui` class will be initialised.

Upon an instance of the `Gui` class being initialised, the first action that is performed is the creation of instances of all `wxPython` widgets that make up the graphical user interface. This is the point at which instances of the `MyTraces`, `MyAxis`, `MyLabels` and `My3DGlCanvas` classes are also initialised. The application window is then displayed to the user and the user is free to interact with the application. Methods within the `MyTraces`, `MyAxis`, `MyLabels` and `My3DGlCanvas` classes are invoked in response to user actions to render graphics. When the user loads a new file, the `Gui` class initialises new instances of the `Names`, `Devices`, `Network`, `Monitors`, `Scanner`, and `Parser` classes to repeat the same process initially carried out in `main`.

3 Team Dynamics

From the outset a proactive approach was taken to planning, designing and developing the Logic Simulator application. The team was comprised of members who did not previously know each other, however each member was confident and enthusiastic about the task at hand, which made it easier to have productive discussions. In the first week of the project, the entire team worked together to develop a functional, flexible and intuitive EBNF logic description language specification. The concept of specifying a language was initially unfamiliar to all members and within a few hours of discussion it was clear that a greater understanding of the requirements was needed to create an effective EBNF specification. While coding activities were not scheduled in the prospective timetable until week 2, the team decided to dive straight into the code to get a flavour of what challenges the project will present. Initially, the work was divided by assigning each member to work on the development of one of the names (K. Suresh), scanner (D. Almasan), and parser (V. Raina) modules.

Starting development of the code base while concurrently finalising the EBNF specification allowed the team to identify potential obstacles and make deliberate decisions that would minimise the impact of any future issues. One such decision was assigning an independent production rule for a device configuration variable within the EBNF and using prescriptive punctuation to delimit symbols. When the requirement to create a new `SIGGEN` type device was revealed in the maintenance phase of the project, amending the EBNF to support this was simple because of its modular structure.

By the end of week 2, it was evident that the `Parser` and `Gui` modules would be the most intensive aspects of the project. The development of the `Names` class required the least amount of effort. Consequently, the responsibility for developing the features within the `Gui` model relating to layout, graphics and user actions was undertaken by Suresh. Almasan undertook the responsibility to add file loading, error display and foreign language features within the `Gui` module upon completing the development of `Scanner`. Having two individuals aware of how the code within the `Gui` module was structured proved to be useful as in the maintenance stage of the project the majority of the

changes outlined needed to be made in `Gui`.

Throughout the project there was a consensus within the team that individuals should aim to complete development tasks ahead of schedule to allow for enough time to conduct rigorous testing. In retrospect, this was perhaps the most crucial decision made by the team as only hours before both the second interim milestone and final report deadline, testing of obscure edge cases revealed flaws in the software that wouldn't have otherwise been discovered. For example, before the maintenance stage circuits had to contain at least one configurable device, however with a `SIGGEN` device this was no longer true. Thus, the application crashed upon loading such a circuit and additional validation steps had to be implemented in the `Gui` class.

Finally, disciplined use of the git version control system was another enabling aspect of the team's dynamic. It ensured that at any point in time each team member could independently test and navigate the software with the most up to date version. Committing changes to the master branch frequently at appropriate stages allowed features of the software to be evaluated by each member in relation to their personal code development plans in a timely manner. An example of a case in which a git feature was particularly useful is when the foreign language functionality was being developed for the application. The unfamiliar and distributed nature of the code required to implement the foreign language functionality prompted Almasan to create a separate branch from master to continue development. This allowed Suresh to continue work on more straightforward aspects of the `Gui` on the master branch, concurrently. The two branches were then carefully and elegantly merged when both aspects of development were completed.

4 Code Review

This section reviews the code written and modified by K. Suresh.

4.1 load_gui.py

This file is run through the command line to start the logic simulator application. A function was added to this file to extract the name of the specified circuit definition file from its path. This was then used to implement modifications in the `load_gui.py` file to show the name of currently loaded file in the title of the application window.

4.2 names.py

This file contains the implementation of the `Names` class. The class is initialised with attributes to store the number of unique errors that have been declared across all other modules, and a list of all unique names that need to be referenced in all other modules. The module contains methods to: return a list of unique indices for a specified number of error codes; return a list of name list indices for new or existing names; query whether a name string exists; and return a name string when a valid index is passed. The module was straight forward to complete as most of the code from

the preparatory exercises could be recycled. The only additions included amending the `lookup` method to accept lists of names and creating a new method `query` to only check if a name exists.

4.3 gui.py

The precise authorship outline of `gui.py` can be seen in the file `division_of_labor_gui.txt` submitted along with the code, in the zip file.

This file contains the implementation of the `Gui` class. The `Gui` class is derived from the `wx.Frame` class. The class's `___init___` method contains all the code that handles the layout of the application window. First, the menu bar is created by initialising the relevant `wxPython` widgets. Menu items are appended to each menu widget and then each menu widget is appended to the menu bar widget. Using standard `wxPython` ID values for certain menu items allows certain labels to connect with specific standard features like keyboard shortcuts to open files and close the window.

Next, arguments passed to the class constructor that need to be accessed outside of the `___init___` method are defined as class attributes. Other values for which a constant record needs to be kept (e.g. number of cycles completed) or objects that need to be altered in other methods are also defined as class attributes. This includes most widgets, but notably those that contain labels and the canvas widgets, as these need to be dynamically changed in response to language toggle or view toggle user actions respectively. In almost all cases the size of widgets are not specified in an absolute manner, as all widgets are appended to `wxPython BoxSizers` or `StaticBoxSizers` which have constructor parameters that allow relative proportions, padding and behavior options (e.g. `wx.EXPAND`) to dynamically position widgets. This ensures that the display is robust to changes in window size. One case in which dimension which is specified in an absolute manner is the width of the labels canvas. As the length of labels is fixed for a particular file, fixing the 'x' dimension of the labels canvas prevents the labels canvas from expanding awkwardly when the user re-sizes the window.

Following widget definitions, widgets are then bound to methods so that events triggered by widgets can invoke the appropriate functions. After this, `BoxSizers`, `StaticBoxSizers`, and `StaticBoxes` are initialised to organise the layout of the application window. The order in which widgets are added in to sizers determines the order in which they appear in the application window, thus the addition of widgets to sizers has been conducted in a methodical manner. Setting window size properties and setting the main sizer of the window are the two last actions performed in the `___init___` method.

The `Gui` class contains 12 public methods that handle the response to any user action. Within most of these methods further calls are made to either: the canvas classes for drawing operations; devices class to configure devices; monitors class to configure monitors or fetch signals; or network class to run simulations. Certain methods such as `on_3D` and `on_2D` do not call methods in other classes but just update attributes of the `Gui` class.

The `Gui` class also contains 7 private methods which are called from within the class' public methods. One notable private method is the `_gen_lists` method which generates 3 multidimensional

lists to locally store the names and corresponding IDs of devices, monitors and configurable devices in an easily accessible format. In the case of configurable devices the current configuration variable of a device is also stored in a dimension of the corresponding array. In many cases such as editing the display of a list to reflect changes in monitors being added/removed or changing the default configuration input shown when a particular configurable device is selected having local access to the aforementioned information makes the code more readable as methods in other classes do not constantly need to be invoked.

4.4 canva.py

The file `canva.py` contains the implementation of the base class `MyOpenGLCanvas`, and its derived classes `MyTraces`, `MyLabels`, and `MyAxis`. The decision to create the three primary classes as derived classes of a single base class was taken to allow static class attributes to be shared between all three classes. This was necessary to constrain the 'y' pan variables of the traces and labels canvas to be the same and constrain the 'x' pan variables of the labels and axis class to be the same. For a change in the variable in one class to propagate to every instance of the same base class the variables needed to be defined as static attributes of a common base class.

Each of the three classes contain attributes to record last mouse positions, a method to control panning, a method to configure and initialise the `OpenGL` context, and a render method to handle drawing operations. The render method within the `MyTraces` class iterates over a list of traces generated for each monitor by calling the private method `__gen_trace`. Traces for each monitor are stored as a list of vertices and then appended to the traces class attribute. Initially the `GL.GL_LINESTRIPS` object was being used to draw traces however the results contained small discontinuities at vertex points. This was resolved by using the `GL.GL_LINES` object instead and manually adding offsets to trace lines so that they appear smooth. However, this solution isn't elegant as the results are optimal only for Linux. Back in the render function, after traces have been generated, for each set of vertices first the color of the trace is specified based on the device type of the monitor. Then an `OpenGL` method is called to sequentially draw lines between each pair of vertices. The `MyLabels` and `MyAxis` classes are structured in a similar way to the `MyTraces` class, except that instead of `__gen_trace` the main private function of each class is `__gen_text` and `__gen_axis` respectively. In the `__gen_text` method of the `MyLabels` class, the name of devices is truncated to prevent overflows in the canvas, the device type is included in the trace label, and the position of each label is also calculated and returned with the label.

Within the `on_mouse` event of all three classes additional conditions have been included to ensure that the user cannot pan beyond the range of the objects drawn in the canvas.

4.5 canvas3D.py

The file `canvas3D.py` contains the implementation of the `My3DGLCanvas` class. The layout of the class is similar to the classes that handle 2D drawing, however instead of a helper function that generates and stores objects in a class attribute, the `My3DGLCanvas` class contains a pri-

vate method that just draws cuboids of a specified width, depth and height. Within the `render` method of the class, a two layer for loop iterates over each signal value for each monitor and the `_draw_cuboid` method is called to draw a cuboid in a specific position. Traces extend into the page (negative 'z' direction), have a constant 'y' value, and are separated in the 'x' direction by a constant step. Trace labels are generated by a private method and positioned at a fixed offset from the start of a trace.

Unlike in the 2D view classes, the `on_mouse` method in `My3DGLCanvas` enables users to pan, zoom, and rotate their view freely. The `on_reset` method within the class allows users to quickly restore pan, zoom, and rotate varibles to their original values.

4.6 devices.py

In the file `devices.py` a method `set_clock` was added to the `Devices` class to enable the half-period of a clock device within a circuit to be configured. The method was written in a similar way to `set_switch` to preserve a similar work flow for both configure device actions.

4.7 test_scanner.py

The file `test_scanner.py` contains functions to test each method in the `Scanner` class. For each test a specific text file has been written that is then used to confirm assertions of the results of method calls. The `pytest.fixture` technique is used to generate an instance of the `Names` class which is needed to create an instance of the `Scanner` class. In `test_get_symbol` 14 different assertions are made. It was suggested after the second interim milestone that `pytest.parameterize` should be used to implement this test in a neater way however, because the method being tested needs to be called sequentially before each assertion it was not possible to use the suggested technique. Instead, the test was rewritten to more clearly reflect the iteration over a series of cases.

5 Test Procedures

For the `Scanner`, `Parser`, and `Names` modules, `pytests` for each module were written by team members that did not write the code for the module. This was done to avoid bias in testing. The practice proved to be fruitful as tests written highlighted bugs that were not originally discovered by the module author. One example of this is that in Windows line endings are represented differently compared to Mac OS or Linux. Thus, an assertion error was flagged in a test written to verify that the pointer carrot is displayed correctly in a output error message, which led to the discovery of this fact. Rigorous physical testing was also conducted with a suite of over 30 example files to ensure that as the application was robust to even obscure edge cases. One example of this was the discovery that upon loading an erroneous file, the 2D view would in the application would clear itself however the 3D view maintained objects that were last rendered. It was discovered that this was the result of one misplaced indent.

6 Conclusion

Overall, the development of the Logic Simulator application went smoothly and from the creators perspective all requirements outlined should be satisfied. Efforts were made to improve functions where possible to provide users with the best experience possible. With more time, one possible improvement that could be made to the application would be to implement measures for cross platform adaptation, as currently there are few areas of the application that would not perform in the most optimal way on non-Linux platforms (e.g. smoothness of traces). Another area which was not tested was the computational efficiency of the code written. Tests were not conducted to evaluate how the user experience would be affected when the extreme magnitudes are employed for various application parameters. Especially because of the graphics based nature of the application, testing on variety of machines would help generate results that could then be presented to users to set expectations and guidelines for performance.

Appendix A: Example Circuit Definition Files

circuit1.txt

```
DEVICES: /* There are 3 clocks, 2 switches, 1 nand, 1 xor and 1 dtype */
CLOCK(8) = CLK1,
CLOCK(1) = CLK2,
CLOCK(1) = CLK3,
SWITCH(0) = SWC1,
SWITCH(1) = SWC2,
NAND(2)= NAND1,
XOR = XOR1,
DTYPE = DTYPE1;
# Start connections block
CONNECTIONS:
CLK1 = DTYPE1.SET,
CLK2 = DTYPE1.DATA,
CLK3 = DTYPE1.CLK,
SWC1 = DTYPE1.CLEAR,
DTYPE1.Q = NAND1.I1,
SWC2 = NAND1.I2,
NAND1 = XOR1.I1,
SWC2 = XOR1.I2;
/* For now,
only monitor 3 outputs*/
MONITORS:
NAND1,
XOR1,
DTYPE1.Q;

END
```

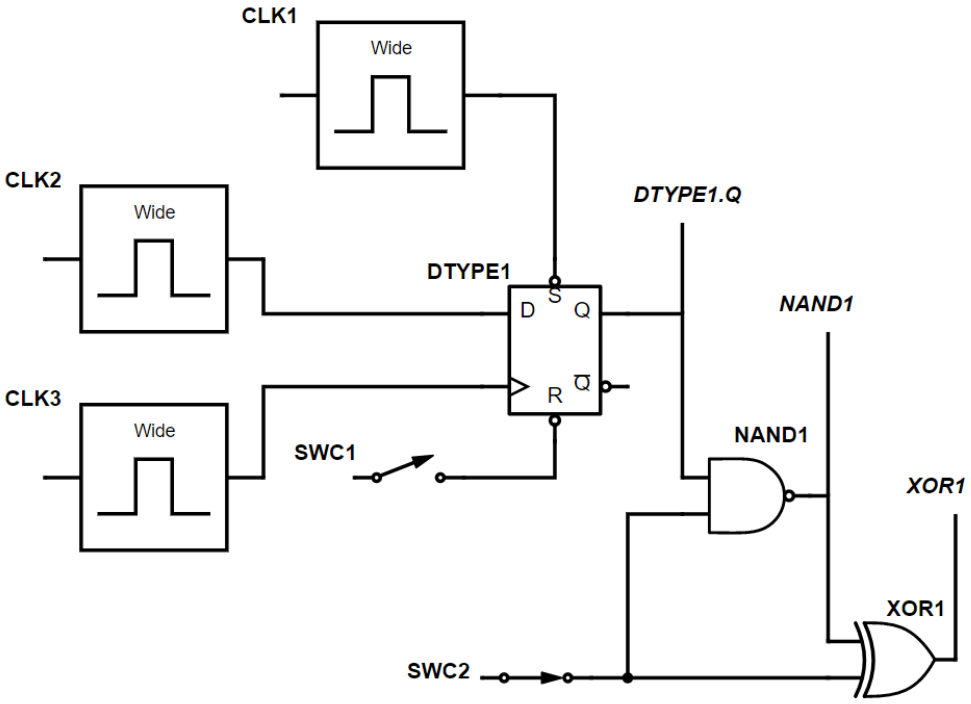

Figure 2: circuit1.txt

circuit_divide_by_6_counter.txt

```
/* This circuit counts in binary to a value of 5 after
which it restarts. d1.Q represents the least significant bit
and d3.Q represents the most significant bit. */
DEVICES:
CLOCK(1) = clk,
AND(2) = and,
SWITCH(0) = sw1,
DTYPE = d1,
DTYPE = d2,
DTYPE = d3;

# Asynchronous circuit may cause real circuit
# to have hazards.

CONNECTIONS:
clk = d1.CLK,
d1.QBAR = d1.DATA, d1.QBAR = d2.CLK,
d2.Q = and.I1,
d2.QBAR = d2.DATA, d2.QBAR = d3.CLK,
d3.Q = and.I2,
d3.QBAR = d3.DATA,
and = d1.CLEAR, and = d2.CLEAR, and = d3.CLEAR,
sw1 = d1.SET, sw1 = d2.SET, sw1 = d3.SET;

MONITORS:
clk,
d1.Q,
```

d2.Q,
d3.Q;

END

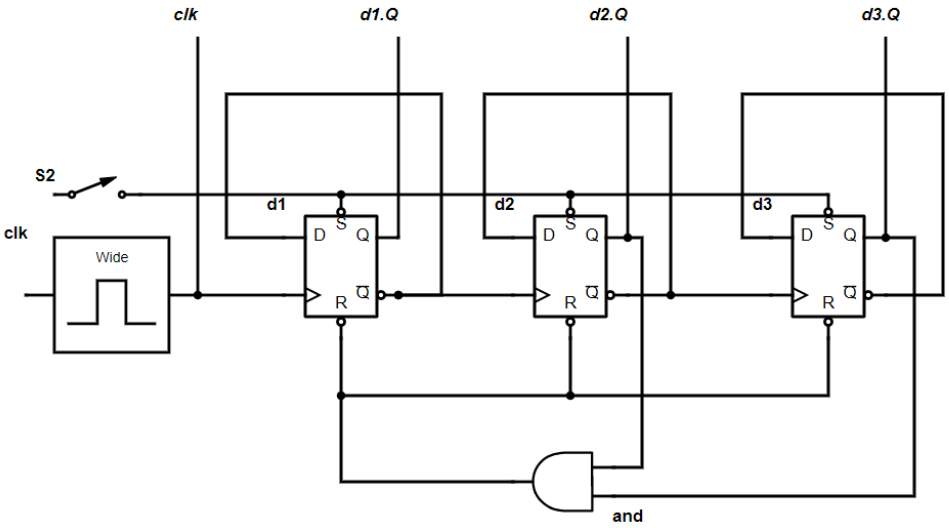

Figure 3: circuit_divide_by_6_counter.txt

circuit_pipo.txt

Parallel In Parallel Out Shift Register

/* 2 inputs NOR gates are used
in place of NOT gates */

DEVICES:
CLOCK(1) = clk,
DTYPE = d1,
DTYPE = d2,
DTYPE = d3,
SWITCH(0) = serial_in,
SWITCH(1) = load,
SWITCH(1) = inp1,
SWITCH(1) = inp2,
SWITCH(0) = inp3,
NAND(2) = nan1a,
NAND(2) = nan1b,
NAND(2) = nan2a,
NAND(2) = nan2b,
NAND(2) = nan3a,
NAND(2) = nan3b,
NOR(2) = nor1,
NOR(2) = nor2,
NOR(2) = nor3;

CONNECTIONS:
serial_in = d1.DATA,

```
d1.Q = d2.DATA,
d2.Q = d3.DATA,
clk = d1.CLK,
clk = d2.CLK,
clk = d3.CLK,
load = nan1a.I1,
load = nan1b.I1,
load = nan2a.I1,
load = nan2b.I1,
load = nan3a.I1,
load = nan3b.I1,
inp1 = nan1a.I2,
inp1 = nor1.I1,
inp1 = nor1.I2,
nor1 = nan1b.I2,
inp2 = nan2a.I2,
inp2 = nor2.I1,
inp2 = nor2.I2,
nor2 = nan2b.I2,
inp3 = nan3a.I2,
inp3 = nor3.I1,
inp3 = nor3.I2,
nor3 = nan3b.I2,
nan1a = d1.SET,
nan1b = d1.CLEAR,
nan2a = d2.SET,
nan2b = d2.CLEAR,
nan3a = d3.SET,
nan3b = d3.CLEAR;

MONITORS:
d1.Q, d2.Q, d3.Q;

END
```

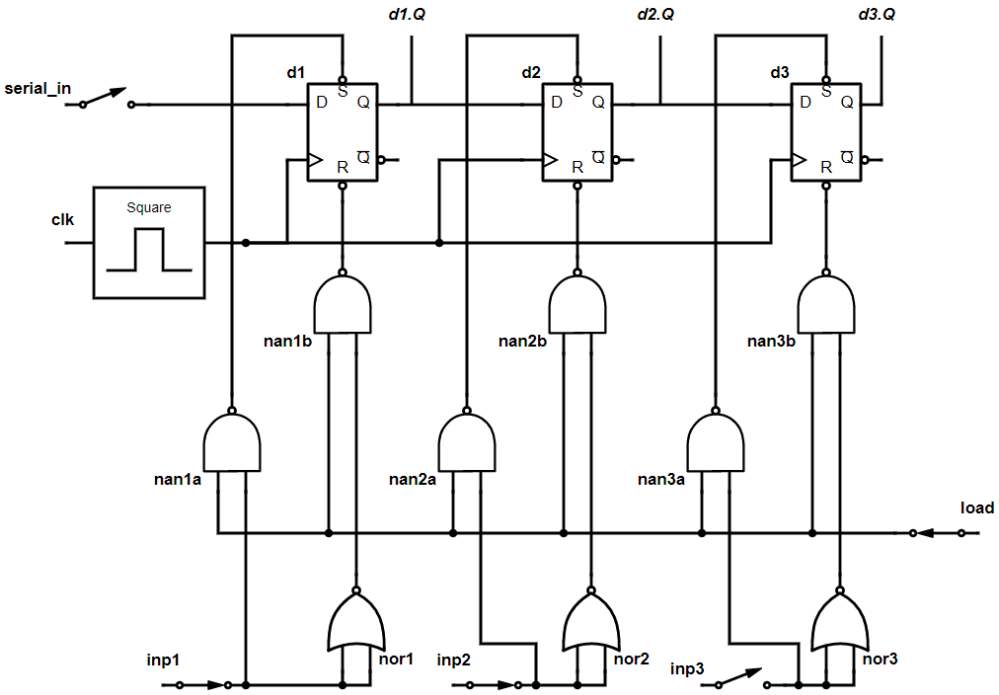

Figure 4: circuit_pipo.txt

circuit_vending_machine.txt

```
/* Vending machine circuit
Only allowed to input č1 and 50p coins
No change provided if č2 inserted
*/

# Only 1 coin can be inserted at a time
# so both inputs shouldn't be high at the same time

DEVICES:
    AND(2) = and1, AND(2) = and2, AND(2) = and3, AND(2) = and4, AND(2) = and5,
    AND(2) = anda, AND(2) =  andb, AND(2) = andc,
    NOR(1) = notb,
    OR(4) = or1, OR(3) = or2,
    AND(2) = deliver,
    DTYPE = d1, DTYPE = d2,
    CLOCK(3) = clk,
    SWITCH(0) = 1pound, SWITCH(0) = 50p,
    SWITCH(0) = swc1, SWITCH(0) = swc2;

# Inputs will be changed one at a time to simulate vending machine

CONNECTIONS:
    1pound = and1.I1, 1pound = and2.I1, 1pound = andc.I2,
    50p = and5.I1, 50p = anda.I1, 50p = notb.I1,
    notb = andb.I2,
    d1.QBAR = and1.I2, d1.QBAR = and4.I1,
    d1.Q = and3.I1, d1.Q = deliver.I1,
```

```
    d2.QBAR = and2.I2, d2.QBAR = and3.I2, d2.QBAR = and5.I2,
    d2.Q = and4.I2, d2.Q = deliver.I2,
    clk = d1.CLK, clk = d2.CLK,
    and1 = or1.I1,
    and2 = or1.I2,
    and3 = or1.I3, and3 = andc.I1,
    and4 = andb.I1, and4 = anda.I2,
    and5 = or2.I3,
    anda = or1.I4,
    andb = or2.I1,
    andc = or2.I2,
    or1 = d1.DATA,
    or2 = d2.DATA,
    swc1 = d1.SET, swc1 = d2.SET,
    swc2 = d1.CLEAR, swc2 = d2.CLEAR;

/* Only need to monitor overall output from circuit */

MONITORS:
    deliver;

END

# Additional interesting points to monitor include dtype inputs.
```

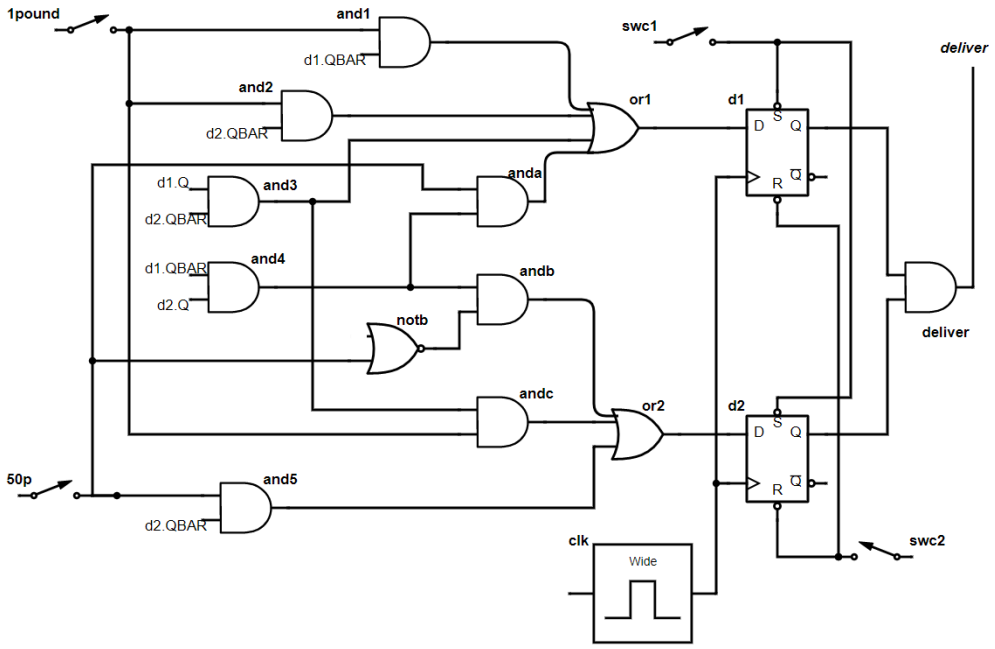

Figure 5: circuit_vending_machine.txt

circuit_siggen.txt

```
/* This is a half adder circuit as there is no carry in bit */

DEVICES:
# Note signal generators of different lengths are defined
```

```
SIGGEN(1-0-0-0-1-1-1-0-1) = siggen1,
SIGGEN(0-0-1-1-1) = siggen2,
AND(2) = and1,
XOR = xor1;

CONNECTIONS:
siggen1 = and1.I1,
siggen1 = xor1.I1,
siggen2 = and1.I2,
siggen2 = xor1.I2;

MONITORS:
# The XOR output is the sum
and1,
# The AND output is the carry bit
xor1; END
```

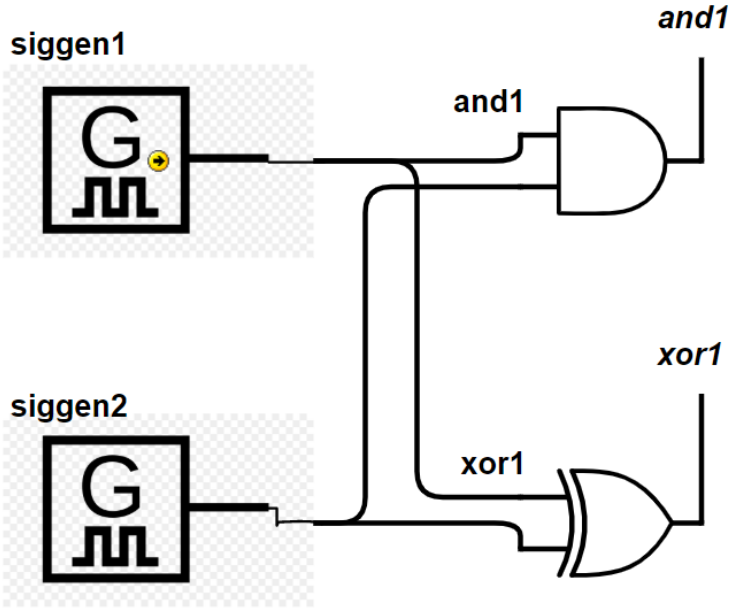

Figure 6: circuit_siggen.txt

Test Results

```
============================= test session starts ==============================
platform linux -- Python 3.6.3, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /groups/IIA/GF2/GF2team12/ks800/Logic simulator, inifile:
collected 88 items

test devices.py ..............
test monitors.py ........
test names.py .......
test network.py ........................
test parse.py ..........................
test scanner.py ......

========================== 88 passed in 2.69 seconds ===========================
```

Figure 7: Pytest results.

Apart from pytests, rigorous testing was conducted on the graphical user interface to ensure users have a smooth experience. Several hours were spent testing edge cases to ensure all types of errors

are reported in an informative manner, and that the interface does not crash following a sequence of reasonable actions.

Appendix B: Updated EBNF Specification

EBNF_Final.txt

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

specfile = dev_block, con_block, mon_block, "END";
dev_block = "DEVICES", ":", dev, {",", dev}, ";";
con_block = "CONNECTIONS", ":", con, {",", con}, ";";
mon_block = "MONITORS", ":", signame, {",",signame}, ";";

dev = logictype, ["(", config_list, ")"], "=", name ;
logictype = "CLOCK" | "SWITCH" | "AND" | "NAND" | "NOR" | "OR" | "DTYPE" | "XOR" | "SIGGEN";
config_list = configvar, {"-", configvar}
configvar = digit | (digitplus, {digit});
name = letter, {digit | letter};

con = signame, "=", signame;
signame = name, port;
port = [".", name];
```

The EBNF specification required no changes following the first interim report, apart from the introduction of a new production rule during the maintenance stage. This change was necessary to allow users to specify a device of type SIGGEN in their circuits. In the original requirements, no device had more than one configurable state thus the EBNF was restricted to prevent users from specifying more than one configuration variable. However, for a device of type SIGGEN a user needs to be able to specify a signal using a sequence of binary variables. Thus, the config_list production rule was created to allow users to specify a string of configuration variables for a device. Errors due to the specification of more than one configuration variable for non SIGGEN devices, and the specification of non binary configuration variables for SIGGEN devices are categorised as semantic and checked accordingly.

Appendix D

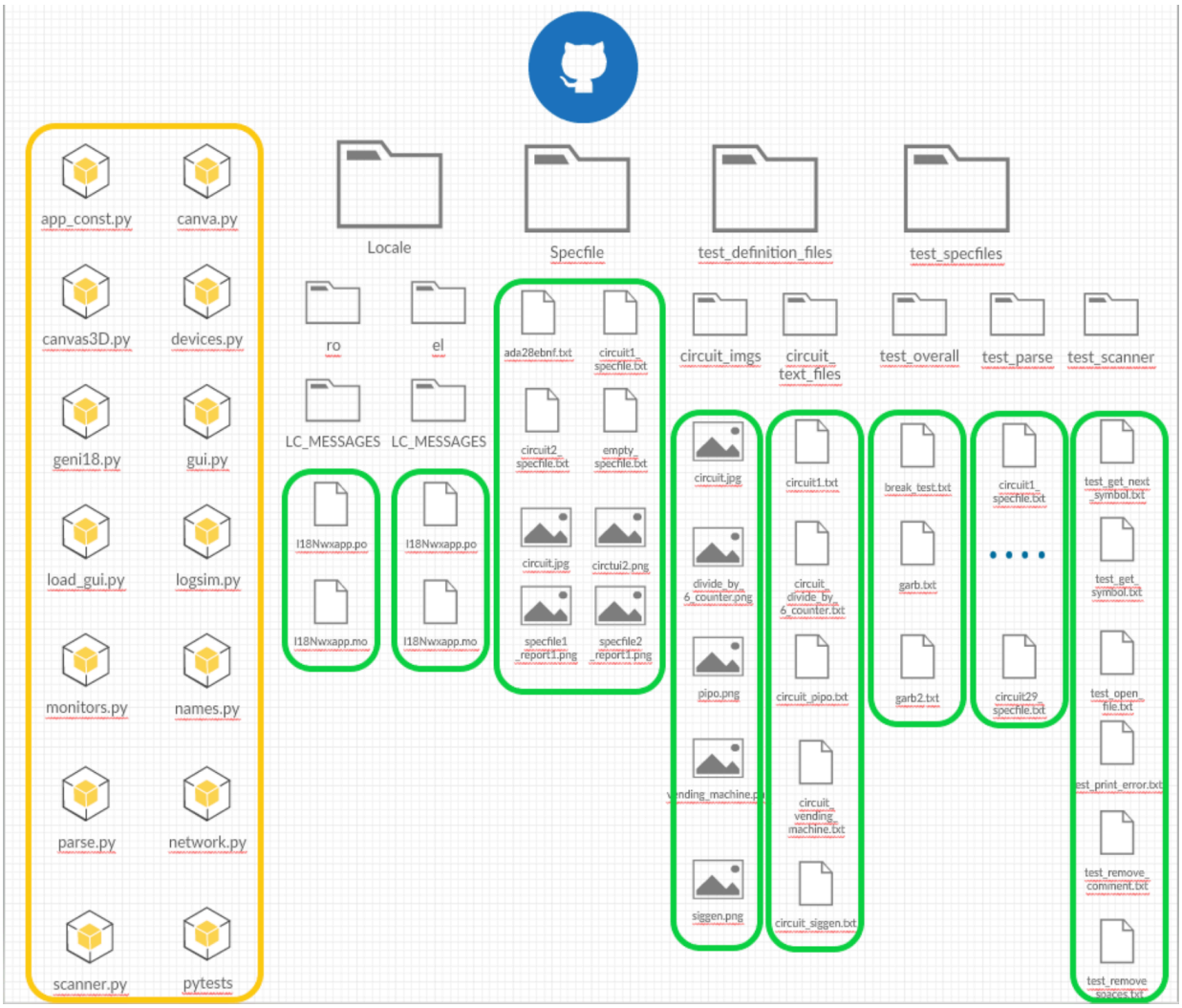

Figure 8: File Layout

The root folder of the repository is named `Logic_simulator` and all code files are situated in this folder. The application is opened by running `load_gui.py` using the following command in a terminal after navigating to the repository: `python load_gui.py`

The files `gui.py`, `canva.py`, and `canvas3D.py` contain the code that implements the graphical user interface. The files `scanner.py`, and `parse.py` implement the scanning and parsing engines respectively. The test files that can be invoked using `pytest` are also situated in the root folder. The folder `Locale` contains files required to support the Romanian language functionality. The sub-folders of the folder `test_definition_files` contain 5 example circuit definition files and their corresponding diagrams. Lastly, the `test_specfiles` folder contains all the files used to test the scanning, parsing and overall function of the application.

Appendix C - User Guide: Digital Circuits Logic Simulator

Start: Navigate to the folder which contains the file `load_gui.py` using Terminal. Type the command `python load_gui.py` into the command line and press enter.

1. Menus: To load a new circuit definition file in the GUI click on `File` > `Open` to open a file explorer (Keyboard Shortcut: `Ctrl` + `O`). Toggle between 2D and 3D views by selecting options in `View` tab. Toggle between English and Romanian by selecting options in `Language` tab.

2. Running Simulations: Specify a number of simulation cycles. Click the `Run` button to run a fresh simulation, or click `Continue` to extend a prior simulation.

3. Configuring Devices: Select a configurable device (CLOCK or SWITCH) from the dropdown list. Clicking `Apply` will update the configuration variable of the selected device.

4. Selecting Monitored Outputs: Select any output(s) and click `Show` to make them monitors.

5. Removing Monitored Output: Select any output(s) and click `Remove` to zap them as monitors.

6. Activity Log: The activity log will show the following output: all error messages generated when an erroneous circuit definition file is loaded; messages indicating the result of, trying to open a file, running/continuing a simulation, configuring a device, creating a monitor, and zapping a monitor.

7. Toggle, Reset, Reload: Click these buttons to change between 2D and 3D views, reset a view's orientations to default, and to reload the currently loaded file.

2D View: Pan view by holding `Left Mouse Button` and dragging.

3D View: Pan view by holding `Right Mouse Button` and dragging. Rotate view by holding `Left Mouse Button` or `Middle Mouse Button` and dragging.