IIA GF2 Software: Final Report

Martin Jackson (mj380) - Churchill College
Team 8

Contents

1 Software function

This software project involved completing the development of a logic simulation program, and subsequently performing maintenance on it. The finished program is capable of simulating a circuit containing any number of connected:

- Clocks, which change state every `n` cycles, where `n` can be configured separately for each clock and can be any number greater than zero.
- Switches, which can be turned on or off by the user during simulation.
- AND, NAND, OR, NOR logic gates, with 1 to 16 inputs.
- XOR gates, with two inputs.
- D-type flip flops, with data, clock, set and reset inputs, and normal and inverted outputs.
- Signal generators, which produce an arbitrary periodic binary waveform of any length.

The simulator can read a circuit from a file (in plain text, extension `.gf2`, syntax as described in appendix B), simulate it, and display signal traces recorded during simulation. In addition, circuits can be edited from within the GUI. Instructions on how to use the program are given in appendix C. Example circuits are given in appendix A.

2 Teamwork

The team operated in a flexible manner. Though tasks were initially assigned to each team member, and each team member worked mainly on their assigned areas of the code, we assisted each other where necessary. I worked mainly on the GUI, but also wrote automated tests for the scanner and names classes, and helped out the other members of my team with any difficulties they encountered.

`git` and Github were used for version control, with over 250 commits made during the course of the project. Using `git` allowed us to work independently, with minimal time wasted on source code management and synchronising and merging changes. It was also extremely helpful in allowing more than one person to work on the same part of the program simultaneously, when this was necessary in order to complete tasks on time, or to allow other team members to help track down and fix complex bugs.

Github was used for bug tracking, though this was mainly for bugs that needed to be communicated to the rest of the team - any bugs we found in our own areas of the code we just fixed immediately.

Overall, the team worked well together, meeting all deadlines and producing a working program which we believe adequately meets the client's specifications.

2.1 Development phase

Areas of development were initially allocated as shown in the Gantt chart in the first report. I worked on the GUI, Jamie on the names and scanner classes, and Tim on the parser class.

Integration of modules was done as early as possible. The main `logsim` program was first compiled with a "File → Open" command that called the parser and scanner modules on 20th May (when these modules contained no code, just an outline of the interface). The GUI at this stage also had a textbox that

displayed all messages printed to `cout`. Integration of the GUI with the scanner and parser modules could then be tested using the main `logsim` program as soon as those modules were written. GUI development was performed almost entirely with the main `logsim` program, instead of using the supplied `guitest` program.

I completed the core functionality for the GUI on 27th May, as scheduled in the Gantt chart. The remaining time before the second interim report was scheduled for integration and final testing. However, since the modules had already been integrated and there were no problems visible in the GUI, and I was therefore ahead of schedule, I used this time to add the ability to edit device properties and connections to the GUI. Meanwhile, Tim and Jamie finished the parser and performed final testing.

2.2 Maintenance phase

We split the maintenance work between us, with one task per person.

I implemented the continuous simulation feature. However, since I had already implemented most of the functionality before the maintenance phase (by adding a scrollable signal traces display, and allowing an unlimited number of monitor samples), I was able to complete the task very quickly by just adding a timer and a button.

I was therefore able to make some enhancements to the GUI during the maintenance phase, and assist Jamie with the D-type bistable changes. Tim added the signal generator and made improvements to the parser.

3 Software

This section contains a description of the general software structure. Some more detailed notes on the parts of the software written or modified by me are also included. I was responsible for the GUI, and for most of the changes to the supplied source code (the devices, network, and monitors classes).

3.1 Circuit classes

The `network` class is used for storing and searching the list of devices, for managing connections between devices, and to handle allocation of memory for devices. The `devices` class handles creation of individual devices and simulation of the circuit. I added several functions to these classes to assist with device editing from the GUI, Tim added functions for the signal generator, Jamie made a function in the network class to sort the device list randomly, and Jamie and I jointly modified the D-type bistable code.

The `monitors` class is used to contain monitor data, and has functions for adding and removing monitors, and logging the current state of all monitored outputs. I modified this class to remove the limit on the number of samples and monitors, by using STL vectors.

The `names` class is used by all other classes to handle conversion between strings and the internal representation of strings. For efficiency, strings are represented internally in the logic simulator as integers.

3.2 File loading

Three classes are involved in loading a file. The `scanner` class transforms the characters in the input file into symbols for the parser, stripping out unnecessary things like comments and whitespace. The `parser` class calls the scanner class to obtain symbols, and checks that the symbol sequence conforms to the EBNF syntax and satisifes all semantic rules. It then calls appropriate functions in the circuit classes to create the devices, connections, and monitors specified in the definition file. The `error` class is called by the parser whenever it encounters an error, and is responsible for counting errors and warnings and printing out error and warning messages.

3.3 GUI - circuit interface

Since a lot of the GUI needs to use most of the circuit simulation classes, and these classes need to be recreated and pointers updated when loading a file, I created a new class named `circuit` to contain these pointers and pass them around between GUI classes. This class also has some functionality to help the GUI interact with the circuit modules, such as a function to run the simulation.

The observer pattern was used to notify all necessary GUI components of changes to the logic circuit, the monitors, and the number of monitor samples.

The circuit class is defined in `circuit.cc` and `circuit.h`, and classes to implement the observer pattern are in `observer.h`.

3.4 GUI - main frame

The main window is defined in `gui.cc`. This is the core of the application, creating all other controls and windows, and constructing the main instance of the `circuit` class.

3.5 GUI - canvas

The canvas for displaying monitor traces is defined in `gui-canvas.cc`.

Monitor traces are scaled to fit the window, with appropriate limits on how much they can be scaled by. Cycle numbers are drawn at appropriate intervals, with the interval changing according to the signal trace scale so that the cycle numbers do not overlap. The cycle number interval lies in the set $10^n \times \{1, 2, 5\}$ for $n \geq 0$.

The display of each monitor trace within the canvas is handled by a separate class, GLCanvasMonitorTrace. MyGLCanvas constructs a new instance of GLCanvasMonitorTrace for each monitor when the number of monitors changes.

Some things that I encountered that it may be useful to look into for future years:

- Calling the MyGLCanvas::Render() function directly can sometimes cause a crash. A safer alternative is to use Refresh() and Update() calls to queue a paint event instead.

- I needed to use the following sequence of calls to guarantee that the canvas was redrawn correctly: "`Refresh(); Update(); Refresh();`". This was most noticeable while testing a cross-compiled executable in Wine after implementing canvas scrolling. Before adding these three calls, the canvas

did not always update after scrolling. Another team consulted me after experiencing similar symptoms with their canvas, and using this sequence of calls seemed to fix it for them too. These calls may or may not all be needed if the canvas does not scroll, I have not tested it.

3.5.1 Scrolling

Since making a scrolling monitor traces canvas that works well took a little digging through the wxWidgets source code, and the solution may not be obvious, some explanation is merited.

The naive solution is to make MyGLCanvas large enough to fit all the monitors and samples, and place it as a child inside a wxScrolledWindow. This was the strategy that I tried first. However, this is quite laggy when the canvas size is large, and suffers from some artifacts during scrolling.

I therefore used a different strategy. I made the MyGLCanvas class inherit from both wxGLCanvas and wxScrollHelperNative. wxScrollHelperNative is the class that wxScrolledWindow inherits from - a helpful inheritance diagram and some comments can be found in the wxWidgets `wx/scrolwin.h` header file. wxScrollHelperNative is a mix-in class which manages scroll bar visibility, position and size for a control or window, based on the size of the scrolled area set with SetVirtualSize.

MyGLCanvas was then modified to implement its own scrolling. ScrollWindow, the function that wxScrollHelperNative calls in response to scrollbar movement in order to actually move the contents of the scrolled area, was overridden. The ScrollWindow function in MyGLCanvas modifies some member variables and requests a redraw of the canvas. The positions of the drawn monitor traces are then offset according to these member variables. The canvas and drawing area remain the same size and in the same position, it is only the virtual size and the positions of drawn points on the canvas which change.

When the canvas is resized or the number of monitors or recorded samples changes, the MyGLCanvas class updates its virtual size to be the minimum area needed to display them all, based on the minimum trace scaling factors.

In addition, to prevent lag when a large number of samples have been recorded, only those samples within the visible area will be fetched from the monitor class and drawn.

3.6 GUI - device editing

The device editing GUI is composed of a listbox for selecting a device, and several panels to display information about the selected device. A class is derived from wxPanel for each type of information panel - device details, or information about all inputs to a device, or details of a single output.

The observer pattern is used to notify relevant components when the selected device changes.

The main device editing window and associated dialogs are in `gui-devices.cc`, and the information panels in `gui-devices-infopanels.cc`. The device listbox is defined in `gui-misc.cc`.

3.7 GUI - options

The options dialog is defined in `gui-options.cc`. The wxConfig class is used to save option values. The observer pattern is used to notify relevant GUI components when options are changed.

3.8 Circuit simulation modifications

For the D-type flip flop maintenance modifications, our initial strategy to prevent the result of simulation depending on the order in which devices were declared in the definition file was to sort the list of devices in the network class randomly. However, this made testing difficult, as the randomised device order randomly changed logic gate propagation delays (used to delay the clock and data inputs by different amounts when testing the D-type changes), which made the offset between clock and data inputs randomly vary.

A better solution to the device declaration order problem is to double buffer changes to output signals. This means that changes to output signals are visible to other devices only during the next machine cycle, and never during the same machine cycle that the changes are made in. The `signalupdate()` function stores the new output signal in a new variable `nextsig`, and this is copied into `sig` at the end of every machine cycle. All devices look at `sig` to find the current state of an output, as they did in the original code. This completely eliminates all effects of device declaration order on circuit behaviour.

D-type flip flops were modified to check for rising/falling edges in the input immediately before, during, and immediately after the rising clock edge, and the output will be random if a rising or falling edge occurs in any of these three machine cycles. An option was added to the GUI to print warnings when a D-type output is indeterminate.

The randomised device order was left in the simulator after the double buffering was added, as a demonstration that behaviour no longer depends on device order.

Although it wasn't specifically requested in the maintenance modifications, a clock-to-output delay was also added to the D-type, to ensure that one of our example circuits (`sipo.gf2`) worked correctly. The output of the D-type bistable will only change after the hold time ends.

4 Test procedures

For the names and scanner classes, I wrote automated unit tests which compared the actual results from various functions to the expected results. The source code for these was included in my second interim report (appendix E). Some of the tests were written before the code to be tested was written, and some tests were modified or added afterwards to target potential problems that I spotted (essentially, peer review of the code). The process of fixing problems, and adding tests to look for new problems was iterated several times.

For the parser, continuous integration testing was used during implementation. The thorough unit tests for the scanner and names classes meant that the parser could be tested using a set of example circuit definition files, with reasonable confidence that any problems originated from the parser rather than the scanner or names classes. The working example definition files were used, along with a set of files with errors deliberately introduced to test the various error messages and error handling. A shell script was later written by Jamie to automatically load all these definition files into the logic simulator and log the output, to make it easier to check that all the error messages worked correctly.

Since this was the first time that I have used wxWidgets, the GUI software structure and functionality was mostly designed on the fly whilst I explored the capabilities of wxWidgets (though sketches of the desired GUI appearance were made before development was started). This, combined with the non-trivial work involved in making automated tests for a GUI, meant that no unit tests were produced for the GUI, and only manual system testing was used.

Usability testing was performed by observing the actions of the demonstrator during the software testing for the second interim report. Usability enhancements based on these observations and feedback from

the demonstrator were implemented during the maintenance phase (for example, a menu item to reload the current file, and remembering the selected directory between invocations of the open file dialog).

For most tests, best practice was followed by having different people write the tests and the code to be tested. Jamie implemented the scanner and names classes, and I wrote tests for them. Tim was responsible for most of the parser class, and Jamie created most of the circuit definition files to test individual error messages in the parser.

4.1 Testing D-type maintenance changes

The changes to D-type flip-flops were testing during implementation by printing various debugging messages to the log textbox when the output of the D-type was indeterminate. Circuits where the input signal changed before, during, and after the rising clock edge were created by adding logic gates before the D-type clock or data inputs to introduce a delay in signal propagation. Some of these circuits are shown in appendix A.

While Jamie was implementing changes in the devices and network classes, I added a feature to the GUI which allows the internal function of the logic simulator to be visualised, showing the times at which different signals rise or fall relative to each other. Some of the debugging messages used when testing the changes were left in the code, and an option added to the GUI to enable these warnings, notifying the user when the output of a D-type is indeterminate.

As an example, the visualisation of internal circuit function for a circuit where the input changes during clock rise, setup time, or hold time for each of three D-type bistables is shown in figure 1. The circuit definition file is shown in appendix A.4. The warnings generated are:

```
Warning: indeterminate output for D-type D3, input changed during clock rise
Warning: indeterminate output for D-type D1, input changed during setup time
Warning: indeterminate output for D-type D2, input changed during hold time
```

The resulting outputs are random. Circuits without any input change inside the three machine cycles centred on the rising clock edge were also tested, to confirm that the D-type modifications were operating correctly - in those cases, the output was not random.

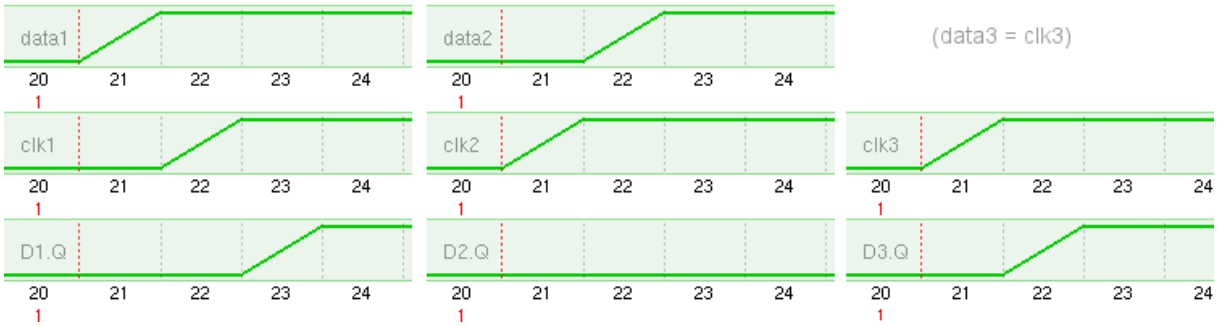


Figure 1: Visualisation of internal logic simulator function, for D-types where the data signal changes around the time of the rising clock edge. The dashed line just to the right of the monitor name signifies the start of the simulation cycle. Each cycle shown after that dashed line is one `machinecycle`, corresponding to one internal step of the simulation.

5 Recommendations for improvement

I believe that the client's requirements have been fulfilled. Some extra features not requested by the client have also been added. However, there are some additional ideas which could have been implemented if there was time. These include:

1. Exporting monitor traces as a CSV file
2. Printing monitor traces, or saving them as an image.
3. Adding more device types to the logic simulator.
4. Improved simulation of circuit timing, such as gate propagation delays. As an extension to this, perhaps some way of alerting the user to logic hazards.
5. Generation of a circuit from a boolean algebra expression.
6. A way of resetting all switch states to the initial values in the circuit definition file, without reloading the whole circuit and resetting all other component states in the process.
7. Ability to reorder monitors, perhaps by dragging and dropping them in the traces display.
8. Saving the circuit to a file. This would allow new circuits to be created from within the GUI, or modifications to existing circuits to be saved.
9. Compatibility with files from other logic simulation programs (limited to devices that exist within this logic simulator).

6 Conclusions

Though there are areas where the logic simulator could be improved, we believe that the client's requirements have been met. The thorough testing ensured that almost all parts of the logic simulator worked well. Almost all required functionality was complete in time for each deadline. Use of `git` for version control was extremely useful for tracking and merging changes, and played a key part in allowing the team to collaborate efficiently.

A Example Definition Files

Example definition files for functionality that existed prior to the maintenance phase can be found in the second interim report in appendix E. Only the additional files for functionality added in the maintenance phase of the project are shown below.

Test circuits for the D-type maintenance modifications comprise a D-type bistable with CLK and DATA inputs connected to a clock device. Various logic gates are placed in the CLK and DATA signal paths to provide differing propagation delays, so that the rising CLK edge arrives at a different time to the rising DATA edge.

A.1 Signal Generator Example

A.1.1 Definition File

```
DEVICES
SIGGEN S1
    :1101010101010101001010101010101010010101010101010101010101010100101000110101010101001;
SIGGEN S2: 1 1   1  0 0 1 0 1 0 1 0 1 0;
SIGGEN S3:1;
SIGGEN S4:0;
AND A1:4;
END

CONNECTIONS
A1.I1=S1;
A1.I2=S2;
A1.I3=S3;
A1.I4=S4;
END

MONITORS
S1;
S2;
S3;
S4;
A1;
END
```

Listing 1: siggen.gf2

A.1.2 Circuit Diagram

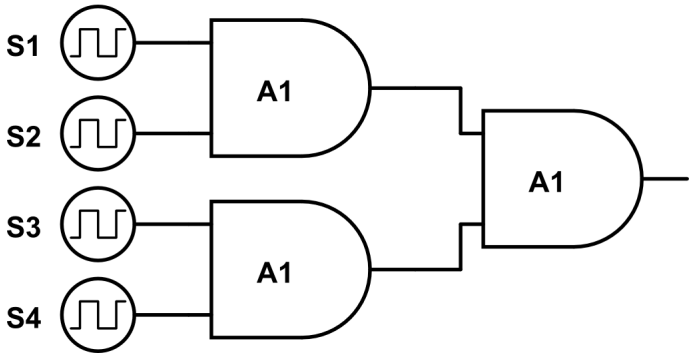


Figure 2: Circuit diagram using a signal generator

NB The software used to draw the circuit diagram does not support the AND gates with four inputs. Therefore three AND gates with two inputs each were substituted for the single AND gate, A1, with four inputs as can be seen in Fig. 2.

A.2 D-type example 1

```
/* Demonstration of the effect of device declaration order on D-type behaviour
   With the original code, D1 is always high, D2 is always low.
   Outcome is random for both D1 and D2 with both the randomised device order
   implementation and the final implementation of the D-type maintenance modifications. */

DEVICES
SWITCH R:0;
SWITCH S:0;

CLOCK clk:1;
OR clk1:1; /* rising clk edge delayed by 1 machine cycle */
OR clk2:1; /* rising clk edge delayed by 2 machine cycles */

DTYPE D1;
OR clk2o:1; /* rising clk edge delayed by 1 machine cycle for D1 or 2 for D2 */
DTYPE D2;

END

CONNECTIONS
clk2o.I1 = clk1;
clk2.I1 = clk1;
clk1.I1 = clk;

D1.CLK = clk2;
D2.CLK = clk2;
D1.DATA = clk2o;
D2.DATA = clk2o;

D1.SET = S;
D1.CLEAR = R;
D2.SET = S;
D2.CLEAR = R;
END

```

```
MONITORS
clk2;
clk2o;
D1.Q;
D2.Q;
END
```

Listing 2: dtype-order.gf2

A.3 D-type example 2

```
/*
This circuit demonstrates one case where the initial implementation using
randomised device order works, and two problems with it.

When run with randomised device order:
For D1, rising data edge is always one machinecycle after the rising clock edge.
For D2, rising data edge is always one machinecycle before the rising clock edge.
The input therefore changes within the setup/hold time, around the time
of the rising clock edge, yet the output is always determinate.
D1 is always high, D2 is always low. The device order only affects when the D1 rise occurs.

D3, where data and clk rising edges occur simultaneously, has a random output.
*/

DEVICES
SWITCH R:0;
SWITCH S:0;

CLOCK data1:1;
DTYPE D1;
OR clk1:1;

CLOCK clk2:1;
DTYPE D2;
OR data2:1;

CLOCK clk3:1;
DTYPE D3;

END

CONNECTIONS
clk1.I1 = data1;
D1.DATA = data1;
D1.CLK = clk1;
D1.SET = S;
D1.CLEAR = R;

data2.I1 = clk2;
D2.DATA = data2;
D2.CLK = clk2;
D2.SET = S;
D2.CLEAR = R;

D3.DATA = clk3;
D3.CLK = clk3;
D3.SET = S;
D3.CLEAR = R;
```

```
49
50 END
51
52 MONITORS
53 data1;
54 clk1;
55 D1.Q;
56 data2;
57 clk2;
58 D2.Q;
59 clk3;
60 D3.Q;
61 END
```

Listing 3: dtype-random-determinate.gf2

A.4 D-type example 3

```
1  /*
2  This circuit shows the effect of a change in the input signal
3  around the rising clock edge for the final version of the logic
4  simulator. This is slightly different to dtype-random-determinate.gf2
5  due to the double buffering of output signal states, which changed the
6  gate propagation delay. Gate type has been changed from OR to AND.
7
8  With double-buffering, the following timings are always true:
9  AND gate: rising edge delayed by 1, falling edge delayed by 2
10 OR  gate: rising edge delayed by 2, falling edge delayed by 1
11 */
12
13 DEVICES
14 SWITCH R:0;
15 SWITCH S:0;
16
17 CLOCK data1:1;
18 DTYPE D1;
19 AND clk1:1;
20
21 CLOCK clk2:1;
22 DTYPE D2;
23 AND data2:1;
24
25 CLOCK clk3:1;
26 DTYPE D3;
27
28 END
29
30 CONNECTIONS
31 clk1.I1 = data1;
32 D1.DATA = data1;
33 D1.CLK = clk1;
34 D1.SET = S;
35 D1.CLEAR = R;
36
37 data2.I1 = clk2;
38 D2.DATA = data2;
39 D2.CLK = clk2;
40 D2.SET = S;
41 D2.CLEAR = R;
```

```
42
43 D3.DATA = clk3;
44 D3.CLK = clk3;
45 D3.SET = S;
46 D3.CLEAR = R;
47
48 END
49
50 MONITORS
51 data1;
52 clk1;
53 D1.Q;
54 data2;
55 clk2;
56 D2.Q;
57 clk3;
58 D3.Q;
59 END
```

Listing 4: dtype-final-test.gf2

B EBNF

```
1  specfile = devices connections monitors
2
3  devices = 'DEVICES' dev ';' {dev ';'} 'END'
4  connections = 'CONNECTIONS' {con ';'} 'END'
5  monitors = 'MONITORS' {mon ';'} 'END'
6
7  dev = clock|switch|gate|dtype|xor|siggen
8  con = devicename'.'input '=' devicename['.'output]
9  mon = devicename['.'output]
10
11 devicename = letter {'_'|letter|digit}
12 input = 'I'('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|
13   '10'|'11'|'12'|'13'|'14'|'15'|'16')|'DATA'|'CLK'|'SET'|'CLEAR'
14 output = 'Q'['BAR']
15
16 clock = 'CLOCK' devicename':'digit{digit}
17 switch = 'SWITCH' devicename':'('0'|'1')
18 gate = ('AND'|'NAND'|'OR'|'NOR') devicename':'('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|
19   '9'|'10'|'11'|'12'|'13'|'14'|'15'|'16')
20 dtype = 'DTYPE' devicename
21 xor = 'XOR' devicename
22 siggen = 'SIGGEN' devicename':'('0'|'1'){'0'|'1'}
```

Listing 5: EBNF

C User Guide

Opening files. There are two methods of opening a file. Either pass the filename as a command line argument when starting the logic simulator, or use the "File → Open" menu item and select the correct file. Logic circuit definition files are plain text files with extension `.gf2`. Examples are provided in section A. Any errors which occur while loading the file will be shown in the textbox at the bottom of the window.

Running the simulation. After opening a circuit, select the number of cycles to simulate using the textbox in the top right corner. Then click the "Run" button just below the textbox. Results will be displayed in the large space to the left. The run button clears the displayed results and simulates the circuit for the specified number of cycles. To simulate for additional cycles without clearing the displayed results, use the "Continue" button instead. The only limit on the number of displayed signals and simulated cycles is the available memory on the computer. If necessary, scrollbars will appear around the result display area.

Running continuously. The "Run continuously" button can be used for indefinite, continuous simulation, with signal traces scrolling (in the manner of an oscilloscope) until stopped. The rate of simulation is adjustable in the options dialog, accessible from the options menu, and can also be changed using keyboard shortcuts as displayed in the options menu.

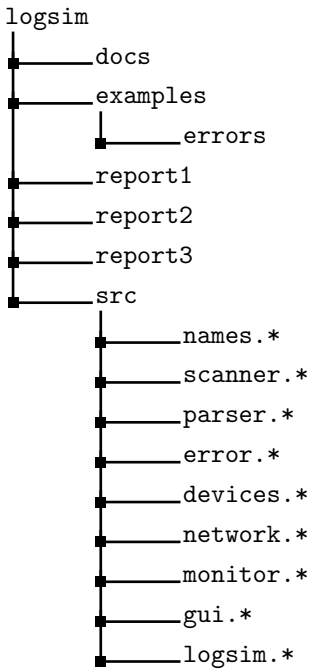
Adding or removing monitors. To change which signals are monitored and displayed, use the add or remove monitors buttons on the right. The add monitors button will show a list of unmonitored outputs that can be selected for monitoring, and the remove monitors button a list of current monitors that can be removed. In both cases, multiple items may be selected. After adding a monitor, you must use the run button before using the continue button, as no samples were recorded for the new monitor during the cycles displayed on screen, so the recorded signals cannot be continued, they must be reset first.

Changing switch states. Use the checkboxes located in the bottom right corner of the window. Changes take effect immediately. Run or continue the simulation to see the effect on the circuit.

Editing devices. Click the edit devices button on the right hand side. This will open a window allowing you to add, edit, or delete devices. Select a device to edit using the list on the left. The top panel allows device properties to be modified. The bottom panels show inputs and outputs for the selected device, and which devices are connected to it. They also allow connections to be added and deleted. All changes to devices and connections take effect immediately, except for changes to device properties, which must be confirmed using the "Apply changes" button.

D-type bistable model. The D-type bistable model incorporates non-zero setup and hold times, and a clock-to-output delay. In the options dialog, there is a debug option that will cause warnings to be printed when the input changes during setup or hold time, and also an option to display the state of monitored signals during each step used internally in a simulation cycle (called a `machinecycle`, with all gate propagation delays being at least one `machinecycle` long). These options will help you find and fix any changes in D-type input signals during setup and hold times. Setup and hold times are one `machinecycle` either side of the `machinecycle` containing the rising clock edge, and a random output will be produced if the input rises or falls during setup or hold times, or during the rising clock edge.

D File Listing



docs contains documentation relevant to our logic simulator such as: EBNF, reserved words, and our Gantt chart.

examples contains our example definition files as listed in Appendix A as well as the shell script written by Jamie to open each example definition file in the logic simulator and log the output for testing purposes.

errors contains definition files which contain deliberate errors and are used to test our error checking functionality. In addition this folder contains the error.sh shell script written by Jamie to open each erroneous definition file in the logic simulator and log the output for testing purposes.

report1, report2 and **report3** contain our first, second and final report respectively.

src contains the source code for our logic simulator. The functionality of the major classes is outlined beneath.

names stores a list of all the words used within a definition file, and methods to manipulate them. It is initialised with only the reserved words, but can be populated as a definition file is read.

scanner reads through the definition file, character by character, and is able to return complete symbols to the parser. It is able to return the internal representation of a symbol, the type of symbol and optionally the value.

parser analyses the definition file as it is read in according to the rules laid out in our EBNF. It is then able to create devices, connections and monitors that are laid out in the definition file.

error is used by the parser to count errors and warnings and print the error and warning messages.

devices is used to create and execute devices.

network stores information about the devices, including the connections between each device.

monitor implements signal monitors, which are used to display the points of interest, as determined

from the definition file, in the GUI.

gui files provide a graphical user interface for the user, and allow for advanced features such as circuit and monitor editing without having to edit the definition file. The GUI classes are divided into several files according to their function.

- **gui** contains the main window, and the dialogs for adding or removing monitors.
- **gui-canvas** is the canvas for displaying signal traces.
- **gui-devices** is the device editing dialog, and the dialogs for creating new devices or connections that are used by the device editing dialog.
- **gui-devices-infopanels** contains the panels used for displaying information in the device editing dialog. A panel contains either device details, or information about all inputs to a device, or details of a single output.
- **gui-id** defines unique identifiers for all controls that need a unique identifier.
- **gui-misc** contains miscellaneous widgets and GUI functions.
- **gui-options** contains the dialog for changing simulation options, and the class that holds the current option values.

logsim handles command line arguments, and initialises the user interface.

E Source code before maintenance modifications

A complete listing of all source code written by me, not including modifications made during the maintenance phase, can be found in my second interim report (attached).