

GF2: Final Report

Nicholas Capel (nrjc2), Queens', Team 6

June 2, 2016

Contents

1	Software Description	2
2	Software Structure	3
3	Commentary on Teamwork	3
4	Major Contributions	4
4.1	Names Class	4
4.2	Scanner Class	4
4.3	Parser Class	4
4.4	Maintenance	5
5	Testing	5
5.1	Names and Scanner	5
5.2	Parser	5
5.3	Signal Generator	5
6	Recommendations	6
7	Conclusions	6
A	Test Definition Files	7
A.1	Bistable Latch	7
A.2	NAND Gate Implementation	8
A.3	4-bit Adder	9
A.4	DTYPE With Signal Generator	11
A.5	DTYPE With Implemented Delay	12
B	EBNF Logic	14
C	Single-page user guide	15
C.1	Opening the Logic Simulator and Definition Files	15
C.2	Navigation	15

D Description of Final Team Directory	16
E Gantt Chart	16

1 Software Description

The Software enables the simulation of both combinatorial and clocked logic circuits on a computer. The simulated logic circuit can be composed of the following devices:

Table 1: Devices

Device Type	Description
Clocks	Can output a regular waveform that changes every N cycles, where $N > 0$.
Switches	User is able to set switch outputs to 1 or 0.
Gates	Can simulate NAND / OR / NOR / AND gates from 1 to 16 inputs. XOR gates with 2 inputs can be simulated as well.
D-Type Flip-flops	D-Type contains a clock, a data, and a set and clear input. Transfers data input into the output on the rising edge of the clock cycle.
Signal Generators	Can output any arbitrary, periodic binary waveform that changes at most once every cycle.

The simulator reads the circuit definition from a file (in plain text, with the `.ge` extension), where the devices, the connections between them, the generators needed for inputs, and the signals outputs to monitor are specified.

A detailed description of the circuit definition syntax can be found in Appendix B, with example circuits in Appendix A. A one-page user guide can be found in Appendix C.

The user is able to specify a maximum of 30 points in the circuit to monitor, and can simulate a maximum of 500 cycles. the device names have a maximum length of 20 characters.

2 Software Structure

Table 2: Major Software Components

Class	Brief Description
Names	Contains a bimap that matches <code>namestrings</code> with <code>nameids</code> .
Scanner	Reads the definition file and produces a stream of symbols, their corresponding name ids, and the value (if applicable).
Parser	Calls on the scanner. Evaluates the stream of symbols from the scanner according to the EBNF file rules. Proceeds to create devices, connections and monitors from the definition file. Identifies syntax and semantic errors, and passes them to the error class.
Errors	Called by the parser class, keeps tracks of the number of errors, the error types, and prints them to the console.
Devices	Contains the definitions of all devices, as well as methods to construct, execute, and debug these devices.
Network	Contains the internal representation of the circuit, as well as methods to construct device inputs, outputs, and connections.
Monitor	Allows the monitoring of outputs from devices in the circuit. Contains functions to add, remove, and display outputs.
GUI	Constructs a GUI using <code>wxWidgets</code> . Split into <code>guimonitor</code> , <code>guiCanvas</code> , and <code>gui</code> .

The main program file is located in `logsim.cc`. Prior to the implementation of the GUI, `userint.cc` was used, but has since been abandoned. The scripts used for testing will be discussed in Section 5.

3 Commentary on Teamwork

At the beginning of the project, a timeline of implementation was developed, summarized by the Gantt Chart in Appendix E.

`git` was used for version control, with more than 150 commits made over the whole project lifecycle. Using `git` eliminated the need to worry about version control. Originally, the `parser`, the `scanner` and the `GUI` existed on different branches of the repo. However, when the `scanner` was completed, it was merged onto the `parser` branch, and in the integration phase, these were further merged onto the `master` branch. This allowed the three members to work separately, and easily combine their work when necessary.

Initially, Nicholas was underloaded, due to the fact that the `scanner` and `names` classes are slightly easier to implement than the `parser` and `GUI` class. However, once he was finished with the implementation of the `scanner` and `names` class, he then proceeded to help with the `parser` class.

There was good separation of the workload, with close cooperation between the `scanner` and `parser`. The `GUI` class could and was handled nearly independently from the other classes due to the modularity of the design. We held daily scrum sessions in the DPO, with all team members sitting close together for easy communication and consultation.

In the final maintenance phase, the work was split into three separate tasks.

Table 3: Table of Maintenance Responsibilities

Task	Individual Responsible
Finite D-Type Hold Times	Lou Yuxiang
Implementation of Signal Generator	Nicholas Capel
Support for multiple languages	Kamile Rastene

Overall, planning and execution went well, with all features being delivered on-time and on-task.

4 Major Contributions

4.1 Names Class

The names class stores a list of all the words used within a definition file. It is initialised with both reserved words and punctuation, and is populated by the scanner as the definition file is read. The initial implementation as suggested in the handout suggested the usage of a vector-type data structure. However, the main problem with this data structure is the fact that lookup operations require searching through the whole structure, which is a $O(n)$ operation. Instead, the final implementation uses a variation of an unordered map (the bimap), which possesses a constant lookup time. The bimap is chosen over its cousin, the unordered map, because it allows for indexing of the structure by both the namestring and the nameid simultaneously. When the names table is short, using a bimap might lead to longer computational times, as the time spent computing the hash function might be longer than the time taken to search the entire vector. However, the decision was still made to use the bimap, as in larger circuits, the savings from the $O(1)$ asymptotic bound will become significant.

4.2 Scanner Class

The scanner reads through the definition files, and outputs a stream of symbols, their corresponding ids, and the value (if applicable). The scanner works by skipping all blank spaces, then looking at the next character. If it is a number, the scanner parses till the next non-number, and returns the entire number as an integer. Otherwise, if it is an alphabet, the scanner gets characters until a non-alphanumeric character is reached, adds that namestring to the names table, and returns the nameid to the parser. If it is neither, the character must be a punctuation character, and the scanner gets all subsequent punctuation characters and returns that to the parser class. Comments are defined in a C-like fashion, with `//` used to comment the entire line, and `/* */` used to comment entire blocks.

4.3 Parser Class

The parser calls on the scanner, and evaluates the stream of symbols from the scanner according to the rules of the EBNF file. It then creates the `DEVICES`, the `CONNECTIONS` and the `MONITORS`. In the parser class, I was mainly responsible for writing the code that created the connections and devices. In addition, I separated the error handling into a separate class called `errors`. This decision was made for the sake of modularity. We want the implementation of the parser to be separate from the handling of the errors.

4.4 Maintenance

During the maintenance phase, I was responsible for implementing the signal generator. This required modifications to the `network`, `device`, `scanner`, `names` and the `parser` class, as well as the EBNF file.

The function `makesiggen()` was added to the `device` class to allow for the direct adding of the siggen device from the parser. This is in contrast to all other device types, which are added via `makedevices()`. This is because the `makedevices()` method does not allow for the passing in of the desired clock signal into the signal generator. Additionally, as the signal generator is fundamentally very similar to a clock, the `executedevices()` and `updateclocks()` methods were also re-purposed to allow for the correct functioning of the signal generator.

When that was done, I helped in the implementation of the non-zero hold time for the D-types.

5 Testing

5.1 Names and Scanner

The names class was rather small and simple, with most of the functions being a rearrangement of functions already written and tested earlier in the GF2 handout. Hence, a simple test script was created, `testnames.cc`, that tests only the major cases.

For the scanner class, given that the class was only going to be written once and then never modified again, it was decided to not create a comprehensive testing suite for the scanner class either. Instead, a few sample files were loaded into the test scanning script, `testscanner.cc`. The testscanner would then output a list of the symbols received, the namestrings associated with the nameid in the names table, and the value (if applicable). These were then checked manually for correctness.

5.2 Parser

For the parser class, a more advanced testing suite was created, implemented in `testparser.cc`, in conjunction with `errors.cc`. When errors are detected in parsing, the error codes and lines are pushed onto the end of a vector in `errors.cc`. The automated testing suite contains a list of definition files to be tested and vectors of their expected error codes and lines. On running `testparser`, all the test cases are run sequentially, and if their codes and lines do not agree with what is expected, the system outputs a table displaying the difference between the errors expected and the errors obtained.

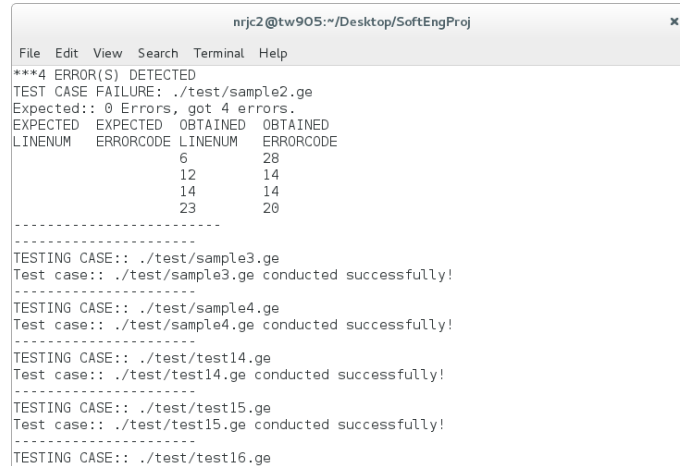
Thus, this formed the basis of our continuous integration test. Every time a change was made to the software, `testparser` was called to ensure that the change did not break the functioning of our program.

5.3 Signal Generator

The testing task was split into two subtasks. The first task is to ensure that the signal generator does, indeed, function as designed. The first stage of testing was done using `testsig.cc`, observing the output of the signal generator as the signal waveform sequence was varied.

Once the function of the signal generator was tested and found to be satisfactory, the `parser` class was modified to allow for the signal generator to be created using the definition test files. These were rigorously tested by creating extra definition files, some with errors and some without errors, and using the `testparser` to test the functionality.

Figure 1: Sample output from testparser



```
nrjc2@tw905:~/Desktop/SoftEngProj
File Edit View Search Terminal Help
****4 ERROR(S) DETECTED
TEST CASE FAILURE: ./test/sample2.ge
Expected:: 0 Errors, got 4 errors.
EXPECTED  EXPECTED  OBTAINED  OBTAINED
LINENUM   ERRORCODE LINENUM   ERRORCODE
          6         28
          12        14
          14        14
          23        20
-----
TESTING CASE:: ./test/sample3.ge
Test case:: ./test/sample3.ge conducted successfully!
-----
TESTING CASE:: ./test/sample4.ge
Test case:: ./test/sample4.ge conducted successfully!
-----
TESTING CASE:: ./test/test14.ge
Test case:: ./test/test14.ge conducted successfully!
-----
TESTING CASE:: ./test/test15.ge
Test case:: ./test/test15.ge conducted successfully!
-----
TESTING CASE:: ./test/test16.ge
```

6 Recommendations

I believe that all the requirements set out by the client have been fulfilled. However, I believe that several extra improvements could be made, and features added.

1. Better error recovery: Currently, error recovery uses only a single stopping symbol. In hindsight, we should have used a set of stopping symbols to allow greater flexibility. Each analysis function can then pass its stopping symbol to any function it calls.
2. More efficient scanner class: Currently, the getsymbol method uses multiple if statements to check for a symbol type, given the name id. Instead, an alternative implementation is to use the name id to index an array containing the symbol types. This leads to neater code and shorter run-times.
3. Creation of devices via the GUI: Currently, the user needs to create circuits using the definition files. It would be much better to allow the user to create the circuit definitions within the GUI itself.
4. Improved handling of warnings: Currently, warnings are thrown natively within the parser, while errors are displayed by calling on the **errors** class. Ideally, the code should be modified such that all warnings are handled by the **errors** class as well.
5. Saving plots as images: The GUI should be improved to include functionality that allows it to save plot images.
6. Improved simulation of gate propagation: The software can be improved to include a proper gate-by-gate simulation of the propagation of signals on each machine execution cycle.

7 Conclusions

In conclusion, although there are some areas of improvement, team 6 has met all the client's requirements. The software was delivered on time and on target. Our lack of hiccups can be attributed to good prior planning, proper allocation of responsibilities, and a reliance on **git** for version control.

A Test Definition Files

A.1 Bistable Latch

Figure 2: Bistable Latch

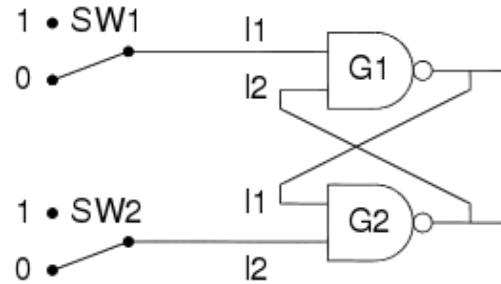
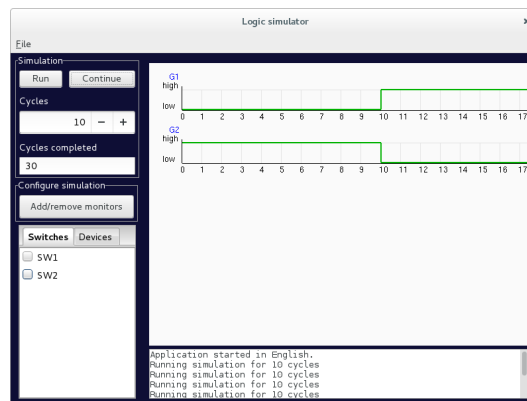


Figure 3: Test Results for Bistable Latch



```
DEVICES{  
  SWITCH SW1:0;  
  SWITCH SW2:0;  
  NAND G1:2;  
  NAND G2:2;  
}
```

```
CONNECTIONS{  
  SW1->G1.I1;  
  SW2->G2.I2;  
  G1->G2.I1;  
  G2->G1.I2;  
}
```

```
MONITORS{  
  G1;  
  G2;  
}
```

A.2 NAND Gate Implementation

Figure 4: NAND gate implementation

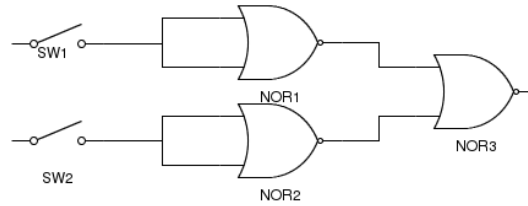
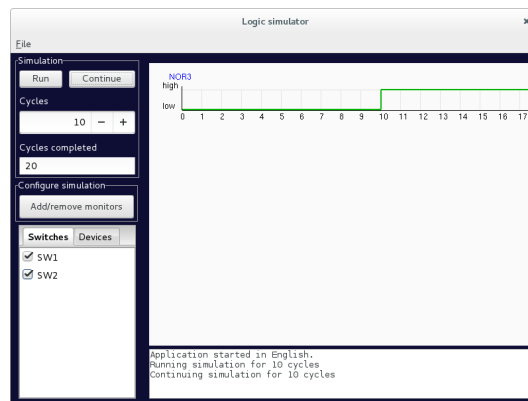


Figure 5: Test Results for NAND gate implementation



```
// Implementing an AND gate from NOR gates
DEVICES{
  SWITCH SW1:0;
  SWITCH SW2:0;
  NOR NOR1:2;
  NOR NOR2:2;
  NOR NOR3:2;
}

CONNECTIONS{
  SW1->NOR1.I1,NOR1.I2;
  SW2->NOR2.I1,NOR2.I2;
  NOR1->NOR3.I1;
  NOR2->NOR3.I2;
}

MONITORS{
  NOR3;
}
```


A.3 4-bit Adder

Figure 6: 4-bit Adder

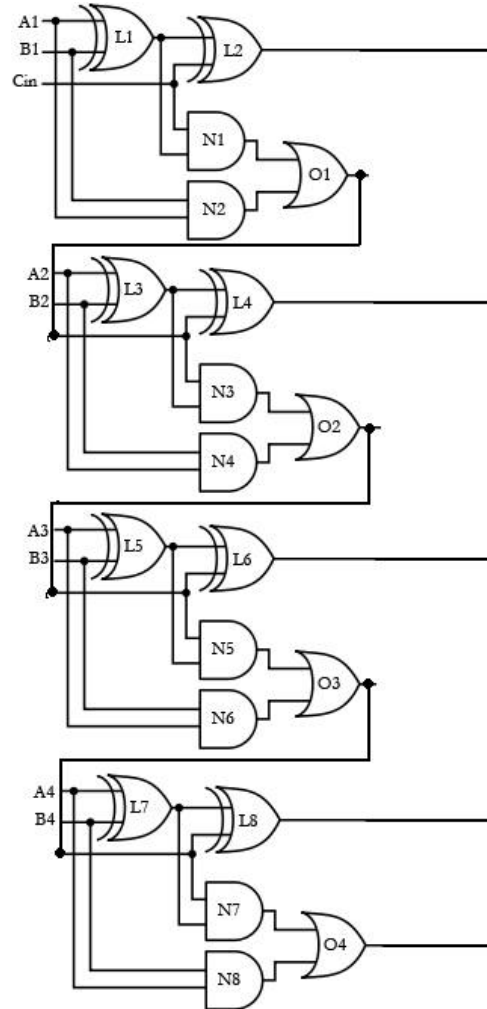
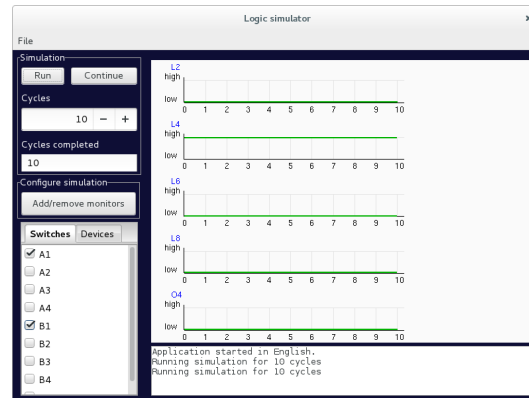


Figure 7: Test Results for 4-bit adder



```
// This defines the logic of a 4-bit adder.
```

```
DEVICES{
SWITCH A1:0;
SWITCH A2:0;
SWITCH A3:0;
SWITCH A4:0;
SWITCH B1:0;
SWITCH B2:0;
SWITCH B3:0;
SWITCH B4:0;
SWITCH CIN:0;
XOR L1;
XOR L2;
XOR L3;
XOR L4;
XOR L5;
XOR L6;
XOR L7;
XOR L8;
AND N1:2;
AND N2:2;
AND N3:2;
AND N4:2;
AND N5:2;
AND N6:2;
AND N7:2;
AND N8:2;
OR O1:2;
OR O2:2;
OR O3:2;
OR O4:2;
}

CONNECTIONS{
A1->L1.I1,N2.I1;
B1->L1.I2,N2.I2;
```

```

CIN->L2.I2,N1.I1;
L1->L2.I1,N1.I2;
N1->O1.I1;
N2->O1.I2;
O1->L4.I2,N3.I1;
A2->L3.I1,N4.I1;
B2->L3.I2,N4.I2;
L3->L4.I1,N3.I2;
N3->O2.I1;
N4->O2.I2;
O2->L6.I2,N5.I1;
A3->L5.I1,N6.I1;
B3->L5.I2,N6.I2;
L5->L6.I1,N5.I2;
N5->O3.I1;
N6->O3.I2;
O3->L8.I2,N7.I1;
A4->L7.I1,N8.I1;
B4->L7.I2,N8.I2;
L7->L8.I1,N7.I2;
N7->O4.I1;
N8->O4.I2;
}

MONITORS{
L2;
L4;
L6;
L8;
O4;
}

```

A.4 DTYPE With Signal Generator

Figure 8: DTYPE Clocked by Signal Generator

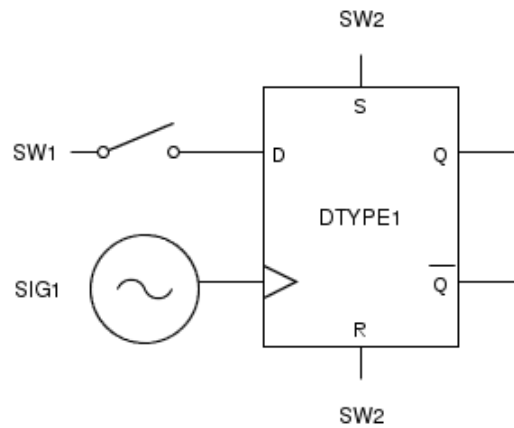
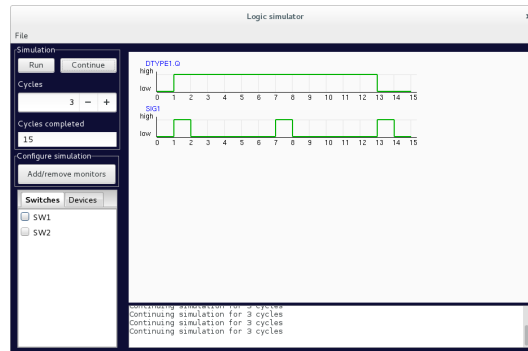


Figure 9: Test Results for DTYPE with signal generator



```
//This file describes a DTYPE clocked by a siggen
DEVICES{
SWITCH SW1:0;
SWITCH SW2:0;
SIGGEN SIG1:0,1,0,0,0,0;
DTYPE DTYPE1;
}
```

```
CONNECTIONS{
SIG1->DTYPE1.CLK;
SW1->DTYPE1.DATA;
SW2->DTYPE1.SET,DTYPE1.CLEAR;
}
```

```
MONITORS{
DTYPE1.Q;
}
```

A.5 DTYPE With Implemented Delay

Figure 10: DTYPE with delay implemented using AND gates

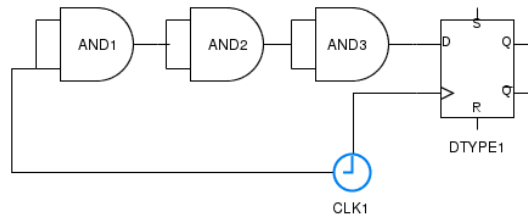
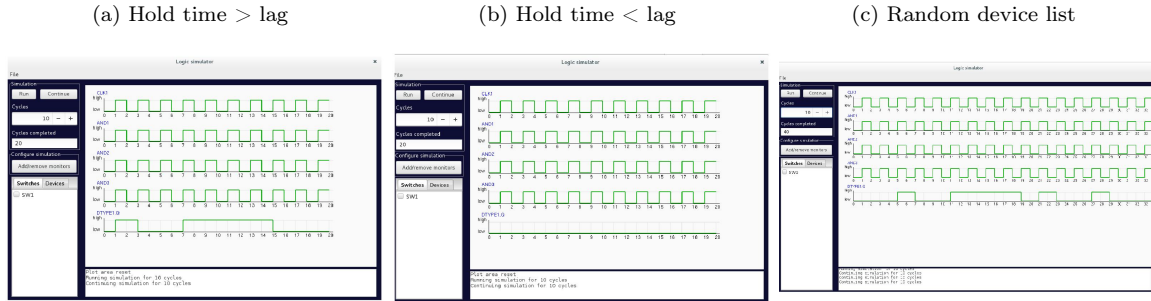


Figure 11: Test Results for DTYPE with implemented delay



```
/* This circuits tests if the finite hold time is implemented
and if output is independent of order of definition
```

1. To test if a finite hold time is being implemented, we first suppress the function `randdevices()` which randomises the devicelist. Then use this exact circuit definition file. Since AND3 is defined last, it will be run first followed by AND2 and AND1. There will be 3 machine cycle delays which is 1 machine cycle longer than the hold time. This will then cause a determinate output as shown in figure 8. This proves that we have managed to set a finite hold time.

2. To test if a random output is indeed generated. All we have to do is the change the sequence in which the AND gates are defined. This will cause the change to propagate faster than the original sequence of definition such that data input will change during the hold time. This will cause an indeterminate output as shown in figure 9.

3. To test if output is independent of order of definition, we use the exact circuit definition file below but enable the `randdevices()` function. This should cause an indeterminate output as shown in figure 10.

```
*/
DEVICES{
SWITCH SW1:0;
DTYPE DTYPE1;
CLOCK CLK1:1;
AND AND1:1;
AND AND2:1;
AND AND3:1;
}
CONNECTIONS{
SW1->DTYPE1.SET,DTYPE1.CLEAR;
CLK1->DTYPE1.CLK,AND1.I1;
AND1->AND2.I1;
AND2->AND3.I1;
AND3->DTYPE1.DATA;
```

```

}
MONITORS{
CLK1;
AND1;
AND2;
AND3;
DTYPE1.Q;
}

```

B EBNF Logic

```

specfile = devicelist connectionlist monitorlist;

devicelist = 'DEVICES' , '{' device , ';' , {device , ';' } , '}' ;
connectionlist = 'CONNECTIONS' , '{' , {connection , ';' } , '}' ;
monitorlist = 'MONITORS' , '{' , {monitor , ';' } , '}' ;

device = clock|switch|gate|dtype|xor|siggen;
clock = 'CLOCK' , devicename , ':' , digit{digit};
switch = 'SWITCH' , devicename , ':' , ('0'|'1');
siggen = 'SIGGEN' , devicename , ':' , ('0'|'1') , {',' , ('0'|'1') } ;
gate = ('AND'|'NAND'|'OR'|'NOR') , devicename , ':' , ('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|
'9'|'10'|'11'|'12'|'13'|'14'|'15'|'16');
dtype = 'DTYPE' , devicename;
xor = 'XOR' , devicename;

connection = devicename , ['.' , output] , '->' ,
devicename , '.' , input , {',' , devicename , '.' , input};
monitor = devicename , ['.' , output];

input = 'I'('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|
'10'|'11'|'12'|'13'|'14'|'15'|'16')|'DATA'|'CLK'|'SET'|'CLEAR';
output = 'Q'['BAR'];

devicename = letter {'_'|letter|digit};

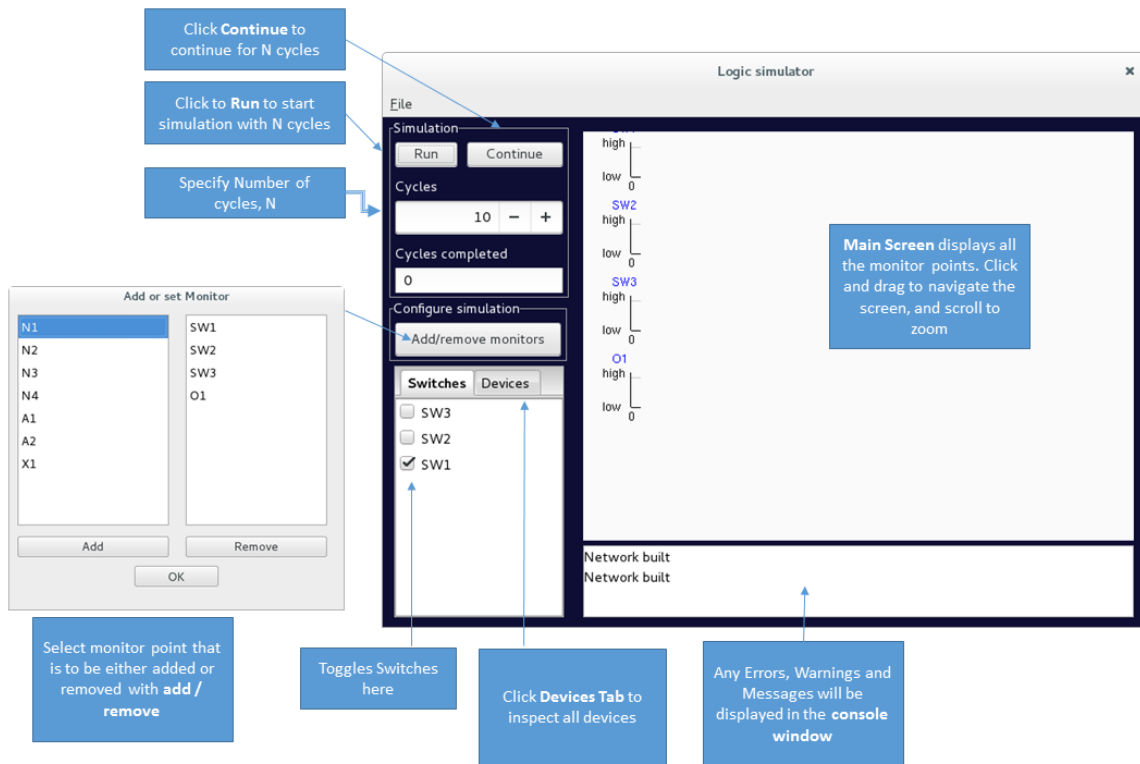
```

C Single-page user guide

C.1 Opening the Logic Simulator and Definition Files

The program can be run by browsing into the main directory and typing `./logsim` followed by the return key. To open and load a definition file, click **File**, then **Open**. Only `.ge` and `.txt` files can be loaded. If there are any errors, they will be displayed in the console display along with their corresponding error codes. In order to open the software in Lithuanian, type `'LC_ALL=lt_LT ./logsim'` into the terminal.

C.2 Navigation



D Description of Final Team Directory

File	Description
Logsim	Main Executable, double click to run
test	Contains various test files
lt	Contains language files
dummycircuit.ge	Description of default circuit that is loaded by logsim

E Gantt Chart

	May Week 2	Week 3	Week 4	June
Participants	12 13 14 15 16 17 18	19 20 21 22 23 24 25	26 27 28 29 30 31	1 2 3 4 5
All	Interim report 1			
Nicholas Capel	Specify interface - names			
Nicholas Capel	Specify interface -scanner			
Lou Yuxiang	Specify interface - parser			
Kamilè Rastené	Specify interface - GUI			
Nicholas Capel		Implement - names		
Nicholas Capel		Implement - scanner		
Lou Yuxiang		Implement - parser		
Kamilè Rastené		Implement - GUI		
All			Interim report 2	
All			Integration and final testing	
All			Maintenance	
All				Final report