GF2: Software

First Interim Report

James Crossley, Jonty Page, Vyas Raina

May 18, 2019

1 Introduction

The purpose of the project is to design a logic simulation program as per client specifications. There are existing modules provided by the client and the task is to write the remaining modules that are necessary for the implementation of the simulator. The project is split into five stages: specification, design, implementation, testing and maintenance. This report intends to outline the details of the general approach that will be adopted by the software team to permit successful design of a robust logic simulator.

2 General Approach

2.1 Top-down Modularisation

2.1.1 Modularisation

The general approach is to employ a traditional top-down modularisation design, where the overall problem is divided into smaller manageable tasks. This provides the following advantages:

- Team members can specialise in development of smaller sub-tasks
- Simple unit testing of modules (independently of other modules)
- Greater readability of programs and therefore easier to maintain the system - system can be easily altered to react to changes in client requirements

Figure 1 shows that the tasks have been split into two main modules: front-end and graphical user interface. Each module is then further divided into smaller sub-modules that can be easily implemented.

2.1.2 Integration

Before the different modules are integrated, it important to have performed thorough unit testing. To ensure there is no programmer's bias, the unit tests for a particular module will be written by a different team member to the one who developed the module.
With complete modules, the final stage would be to perform integration testing. The general method of testing would be to initially perform a variant of black box testing, where a set of definition files (investigating different aspects of the program) would be created. The actual output would be then compared to the expected outputs. If there is an error then the tester would proceed to determine the source of the error, fix it and create a `pytest` for it.

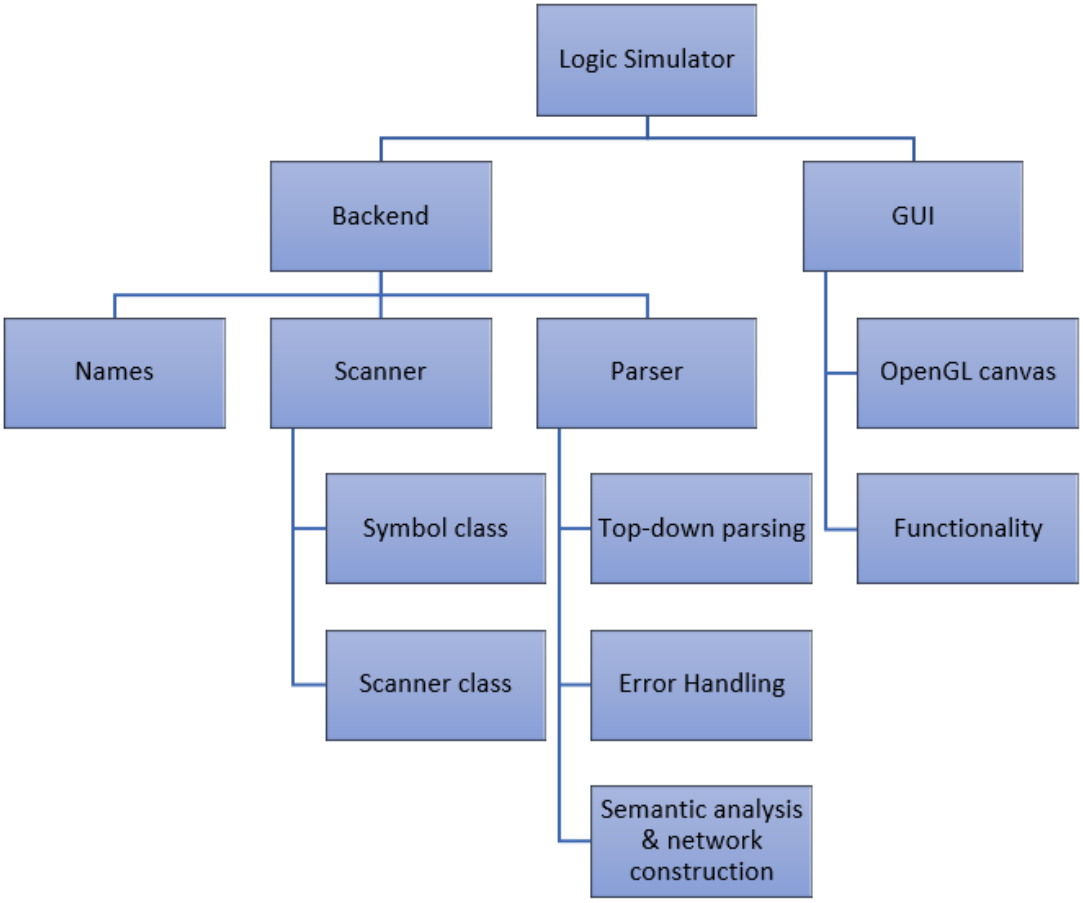


Figure 1: Top-down modularisation design of Logic Simulator

2.2 Team Planning

2.2.1 Task allocation and time-scale

Different team members have been allocated tasks based upon their expertise. However, it is important to account for the time constraints of the project and the dependencies between the different modules. Therefore, a Gantt chart has been used to display the task allocations and the dates by which they have to be completed. Figure 2 shows the Gantt Chart.

2.2.2 Collaborative Coding

To allow developers to independently program with the same set of files it is useful to use a version control system. GitHub has been selected as the choice of version control system, where a private repository has been created for the entire team. The *git* commands allow simple documenting of the commits made by different team members, which allows team members to clearly communicate changes they have made. Furthermore, all team members have agreed to comply with the PEP 8 Python code style to ensure all code produced is consistent and therefore easy to read and update. Consistent documentation of the code will be ensured and then tested using the command `pydoc`.

3 Logic Description Language

It is essential to design a strictly defined logic description language that a user abides by when specifying the composition of the logic circuits.

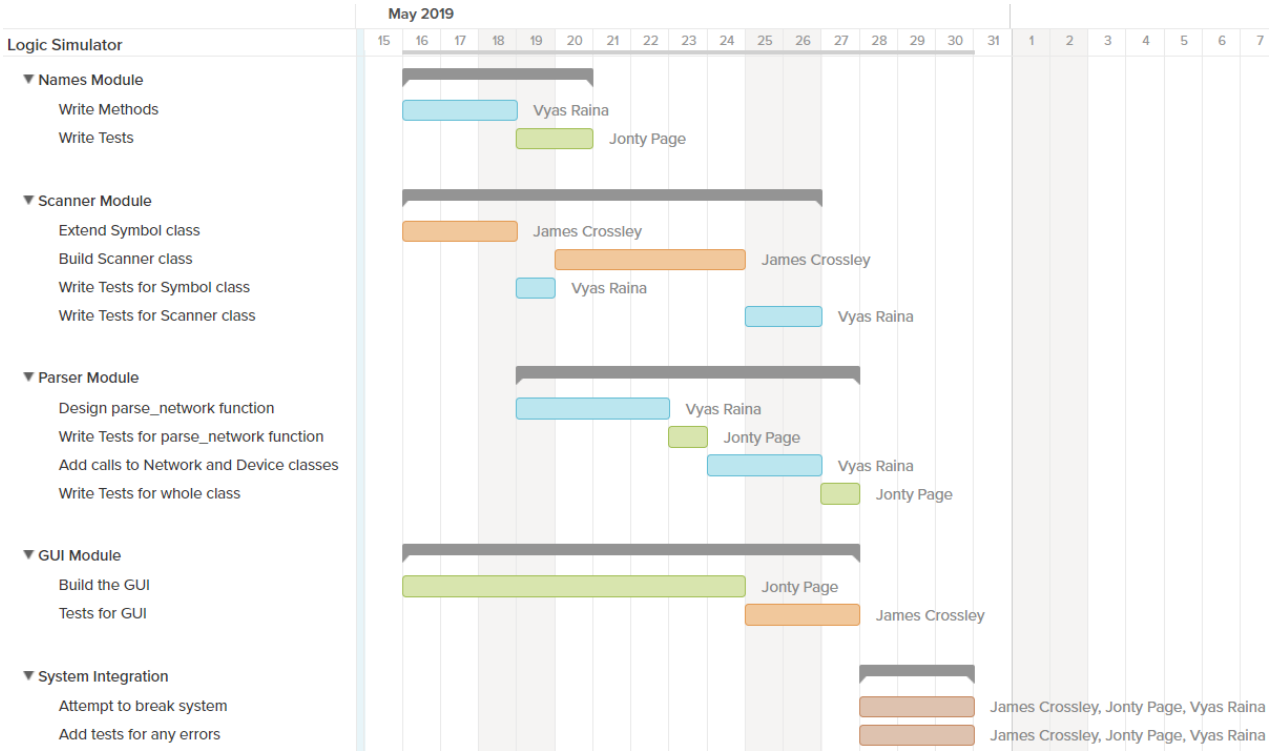


Figure 2: Gantt Chart

3.1 Syntax of the language

The syntax of the language has been specified through Extended Backus Naur Form (EBNF) notation. It is important to note that the EBNF definition of the language doesn't account for comments by the user. However, comments will be dealt with in the following manner:

- Comments must begin with "#"
- Only single-line comments are permitted
- End of a comment will be indicated by end of line
- The Scanner module will strip out all of the comments in the same way all the white space will be removed (comments removed first so EOL is known)

The EBNF definition of the logic description language is given below in section 3.1.1. Justification for the syntax of the language:

- The user has to define all their devices with its parameters first. The user then defines their connections between the devices and then finally defines which pins to monitor. This simple setup ensures that the user can methodically communicate all components of the logic network.
- Input pins and Output pins have been defined as separate syntactic categories. This ensures that if a user attempts to connect an output pin to another output pin this will create a syntax error.
- For all single output components, the output pin is accessed using `name.O`. Although the 'O' could have been omitted, its addition is useful for clarity and readability of logic description.
- Use of semi-colons to terminate each phrase. This makes error recovery simpler, as the program knows where to continue from when an error is detected in a particular phrase.
- 'END' termination to the file is useful for telling the Parser that there are no more statements.
- The EBNF grammar has been written to conform to `LL(1)` form so that the parser only has to look at the next symbol.

3.1.1 EBNF description

```
specfile = device, {device}, {connection}, monitor_point, {monitor_point}, 'END;';


device = name, 'is', type, ['with', 'initial', '1'|'0'], ['with', (number, 'inputs' | '1', 'input')],
         ['with, 'period', number], ';';

connection = name, '.', output_pin, 'connects', name, '.', input_pin, ';';

monitor_point = 'monitor', name, '.', output_pin | input_pin, ';';


name = alpha, {alphanum};

type = 'CLOCK'| 'SWITCH'| 'AND'| 'NAND'| 'OR'| 'NOR'| 'DTYPE'| 'XOR' ;

output_pin = 'O'| 'Q'| 'QBAR';

input_pin = 'I1'| 'I2'| 'I3'| 'I4'| 'I5'| 'I6'| 'I7'| 'I8'| 'I9'| 'I10'| 'I11'| 'I12'| 'I13'| 'I14'| 'I15'
          | 'I16'| 'DATA'| 'CLK'| 'SET'| 'CLEAR';

number = digit, {with_zero_digit};

alpha  = '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' ;

digit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

alphanum = alpha|with_zero_digit;

with_zero_digit = digit|'0';
```

3.2 Semantic Constraints

Table 1 lists all the possible semantic errors that may occur and how these semantic errors will be handled.

3.3 Errors

There are two type of errors that have to be handled: syntax errors and semantic errors. The general approach will be to terminate the program when an error is detected and report a suitable error message. Specific consideration for each type of error:

- Syntax Errors - These errors violate the syntax defined by the EBNF notation. These errors will determined by the Parser. The program will then terminate, displaying a specific Syntax Error message. As well as the descriptive message, the program will indicate the position of the error in the input file (line number and position in the line).
- Semantic Errors - These errors comply to the rules defined by the EBNF notation, but however are not logically compatible in the context of the circuit. Table 1 has a list of semantic errors and how each error will be handled.

Error	Description	Error handling
Multiple parameters for a given type	Each input statement following 'with' in the 'devices' portion of EBNF is optional; this is required so that no inputs can be specified for the "CLOCK" type. Consequently, multiple types and numbers of input can be specified for one device.	If parser detects > 1 statement after 'with', break and report suitable error to user. For example the error message could be "Too many parameters for <name>".
Device arguments unsuitable for type	Arguments after 'with' may be unsuitable for the device type e.g. "AND1 is AND with period 3" (period reserved for type: "CLOCK").	If parser detects an unsuitable input argument for the given type, break and report this to the user. Example message: "<name> of <type> doesn't have attribute <attribute>".
Specifying one input	One input can be specified as "with 1 input" or "with 1 inputs"; both are legal.	The parser should allow for both forms.
Monitoring inputs/outputs	Monitoring points can be specified on both inputs and outputs. There is nothing inherently wrong with this but it should be kept in mind if allowed.	Decision made to allow monitoring of both points; build this function into the parser.
Floating pins	The EBNF syntax for "device" allows any number of connections to be declared. Multiple devices with floating inputs/outputs could be created. Monitoring points could then be connected to these.	The parser should check that all inputs are connected to something, or have initial values specified. Break and display a suitable error if not. Example message: "<name.pin> is floating".
Duplicate Device Names	Multiple device instances can be created with the same name, possibly with different properties. This may leave no way of identifying correct pins for connection/-monitoring purposes.	If the parser detects duplicate names, break and report a suitable error to the user. Example message: "<name> is duplicate".
Non-existent Pin Connects or Monitoring Points	'Connection' or 'monitoring_point' could reference non-existent pins. "CLOCK1.I1 connects AND1.CLEAR" is syntactically accurate but nether pin combination is possible.	If the parser detects invalid pin combinations, break and print a suitable error to the user. Example message: "<name.pin> doesn't exist".
Duplicate Monitoring Points	Duplicate monitoring points could be specified.	This doesn't cause any catastrophic issues but duplicates should be neglected to prevent using unnecessary resources.

Table 1: Semantic Errors

3.4 Example Inputs

Figure 3a shows the definition of the logic circuit in Figure 4a and Figure 3b shows the definition of the logic circuit in Figure 4b, where both definition files conform to EBNF rules defined.

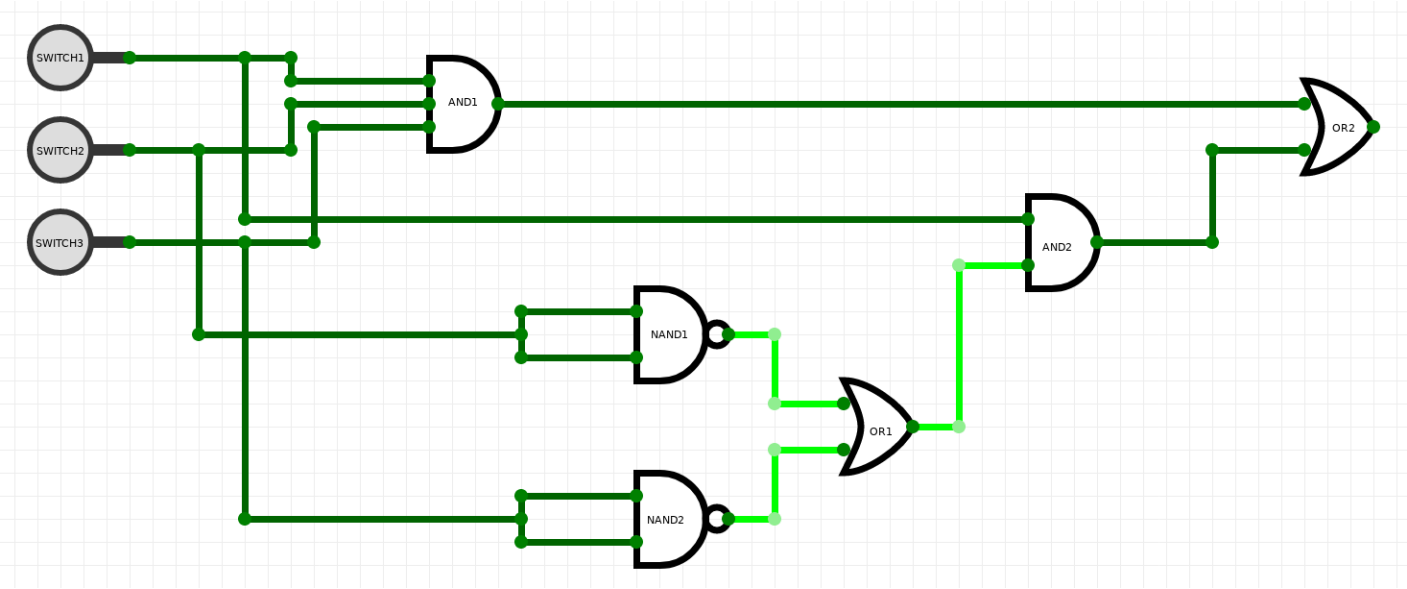
```
SWITCH1 is SWITCH with initial 0;
SWITCH2 is SWITCH with initial 0;
SWITCH3 is SWITCH with initial 0;
AND1 is AND with 3 inputs;
AND2 is AND with 2 inputs;
OR1 is OR with 2 inputs;
OR2 is OR with 2 inputs;
NAND1 is NAND with 2 inputs;
NAND2 is NAND with 2 inputs;
SWITCH1.O connects AND1.I1;
SWITCH2.O connects AND1.I2;
SWITCH3.O connects AND1.I3;
SWITCH2.O connects NAND1.I1;
SWITCH2.O connects NAND1.I2;
SWITCH3.O connects NAND2.I1;
SWITCH3.O connects NAND2.I2;
NAND1.O connects OR1.I1;
NAND2.O connects OR1.I2;
SWITCH1.O connects AND2.I1;
OR1.O connects AND2.I2;
AND1.O connects OR2.I1;
AND2.O connects OR2.I2;
monitor OR2.O;
END;
```

(a) Definition File 1

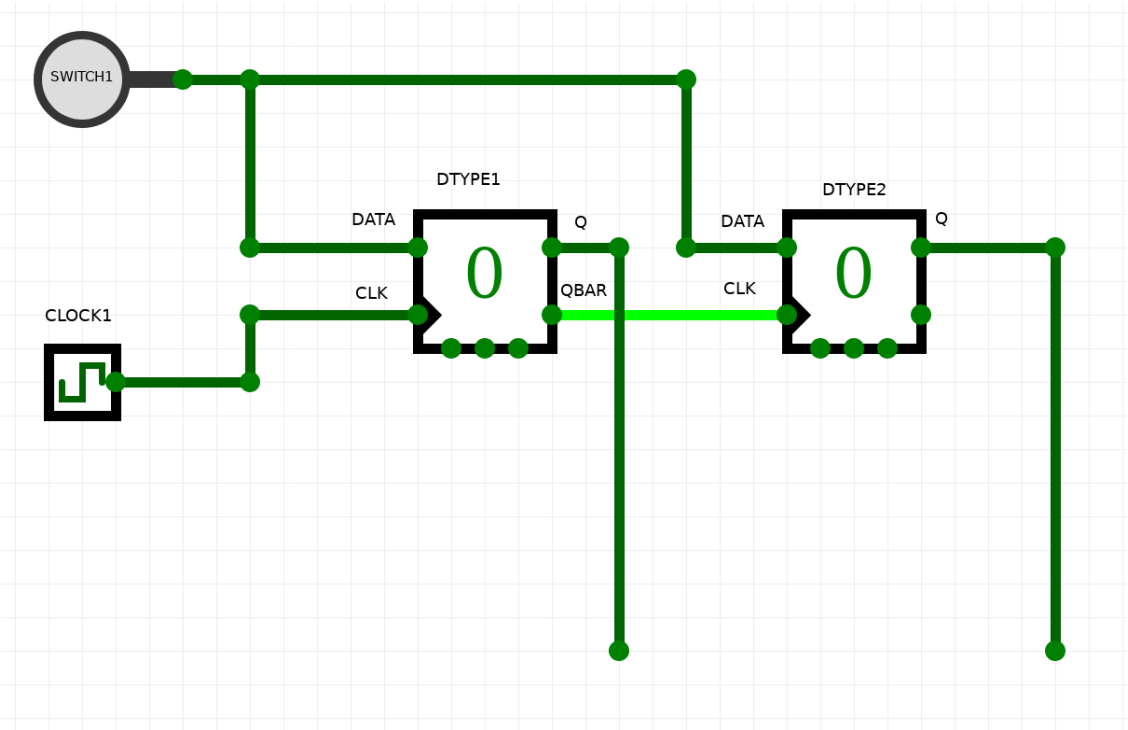
```
SWITCH1 is SWITCH with initial 0;
CLOCK1 is CLOCK with period 10;
DTYPE1 is DTYPE;
DTYPE2 is DTYPE;
SWITCH1.O connects DTYPE1.DATA;
CLOCK1.O connects DTYPE1.CLK;
DTYPE1.QBAR connects DTYPE2.CLK;
SWITCH1.O connects DTYPE2.DATA;
monitor DTYPE1.Q;
monitor DTYPE2.Q;
END;
```

(b) Definition File 2

Figure 3: Example Logic Definition Files



(a) Circuit 1



(b) Circuit 2

Figure 4: Circuits implemented by logic definition files in Figure 3