GF2 First Interim Report

Lakee Sivaraya
ls914@cam.ac.uk

Dhillon Mistry
dm899@cam.ac.uk

Yunfei Zhai
yz703@cam.ac.uk

May 20, 2023

Contents

1 Introduction **i**

2 Teamwork Planning **i**
2.1 Role Assignment . . . i
2.2 Version Control and Collaborative Coding . . . ii

3 ENBF **iii**
3.1 Example 1 . . . iv
3.2 Example 2 . . . v

4 Errors **vi**
4.1 Error Handling . . . vi
4.2 Semantic Errors . . . vi

5 Syntax Errors **vi**
5.0.1 Missing blocks . . . vi
5.0.2 Missing Semicolons . . . vi
5.0.3 Missing Commas . . . vii
5.0.4 Incorrect Device Parameters . . . vii
5.0.5 Unknown Symbol . . . vii
5.0.6 More Error . . . vii

6 Conclusion **vii**

1 Introduction

This report will discuss the initial approach for designing the logic simulation program in Python, specifically addressing how we will undertake the five phases of the software engineering life cycle. It will specifically discuss teamwork planning, the EBNF, error handling, and provide examples of logic circuit descriptions. With regards to the approach, our priority is to ensure the code is as simple as possible upon implementation. To do this, we have collectively decided on the grammar of the language and have set specific deadlines to ensure we have operational code before the final deliverable to allow for debugging. Moreover, this will give us time to refine the code and provide a user-friendly interface. It is also important that the user guide thoroughly explains the logic simulator program, so throughout the coding process we will be adding strong explanations to ensure errors are handled and mitigated.

2 Teamwork Planning

2.1 Role Assignment

Phases	Tasks		Team member	Time finished or estimated
Interim report 1	EBNF and Error handling		Lakee	16th May
	Intro, Logic circuits		Dhillon	16th May
	Team planning, Conclusion		Yunfei	18th May
Interim report 2	Name translation names.py		Yunfei	19th May
	Scanner scanner.py	get_symbol()	Yunfei	22nd May
		get_name()	Yunfei	
		get_number()	Dhillon	
		advance()	Dhillon	
		skip_spaces()	Dhillon	
		error_handling()	Lakee	
	Parser parser.py	devices_parser()	Yunfei and Dhillon	24th May
		connections_parser()	Yunfei and Dhillon	
		monitors_parser()	Yunfei and Dhillon	
	Module Integration		Yunfei and Dhillon	28th May
	GUI gui.py		Lakee	28th May
	pytest.py		Dhillon	30th May
	Example definition files		**Group**	30th May
	User guide		**Individual**	2nd June
Final report	Maintenance		**Group**	6th June

Figure 1: Role Assignment Table

We decided to divide the projects into tasks according to the requirements notes with the detailed planning illustrated in the table. The table shows the main phases and sections of this project. The sections are divided further into individual tasks involving the detailed function needed. The team member who is responsible for each task is also listed, as well as the time to complete the tasks.

2.2 Version Control and Collaborative Coding

For the duration of the project, we will be using Git as our form of version control. We plan to use GitHub as it includes issue trackers and automatic unit testing. The repository can be found at https://github.com/lakeesiv/gf2. We will use Github CI to run unit tests on every commit to ensure that the code is working as expected and adhear to the PEP8 style guide.

In terms of project management, we will be using Notion to assign tasks and track our progress throughout the duration of this project. This will ensure everyone knows what exactly to work on for their assigned deliverables, and ensure we finish developing with time for maintenance and debugging.

Timeline

Table Timeline +

Aa Task	Tags	Deadline	Person
EBNF & Report	Report	May 18, 2023	Lakee Sivaraya
Intro, Logic Circuit Examples and Code & Report	Report, Errors	May 16, 2023 → May 18, 2023	Dhillon
Team Planning, Conclusions & Report	Report	May 16, 2023 → May 18, 2023	FrankZhai
Name Module	class	May 19, 2023	FrankZhai
Scanner Module	function, class	May 22, 2023	FrankZhai, Dhillon
Parser Module	function, class	May 24, 2023	FrankZhai, Dhillon
GUI	userInterface	May 28, 2023	Lakee Sivaraya
Tests	pytest	May 30, 2023	Dhillon
User Guide and Example Files	userInterface, Report	June 2, 2023	FrankZhai, Lakee Sivaraya, Dhillon
Maintenance	maintenance, Software	June 6, 2023	FrankZhai, Lakee Sivaraya, Dhillon

Figure 2: Notion Table

3 ENBF

The aim for the EBNF design was to keep the language as simple as possible. Due to the simplicity of the design, a user will be able to design and monitor circuits very quickly. Below is the EBNF schema:

```
number            =  digit, [digit]
COMMENT_SYMBOL    =  "#"
comments          =  COMMENT_SYMBOL, {[letter | number]}, NEW_LINE

device_name  =  letter, {letter | digit}
devices_type =  "DTYPE" | "NAND" | "NOR" | "XOR" | "AND" | "OR" | "CLOCK" | "SWITCH"
dtype_ports  =  "DATA" | "CLK" | "SET" | "CLEAR" | "Q" | "QBAR"
port_name    =  ("I", number) | dtype_ports
port         =  ".", port_name

definition = "="
connection = "="

gate_params   =  "(" , number , ")"
switch_params =  "(" , (0 | 1) , ")"
clk_params    =  "(" , number , ")"


gate_def   =  device_name, {"," , device_name} , definition, devices_type, [gate_params], ";", [comments]
switch_def =  device_name, {"," , device_name} , definition, devices_type, [switch_params], ";", [comments]
clock_def  =  device_name, {"," , device_name} , definition, devices_type, clk_params, ";", [comments]

device_line =  (gate_init | switch_init | clock_init) | comments

device_io  =  device_name, [port]

conn_def   =  device_io, connection, device_io, {device_io}, ";", [comments]
conn_line  =  conn_def | comments
moint_def  =  device_io, {device_io}, ";", [comments]
moint_line =  moint_def | comments


devices_block =  "[devices]", device_line, {device_line}
conns_block   =  "[conns]", conn_line, {conn_line}
monit_block   =  "[monit]", moint_line, {moint_line}

ebnf =  devices_block, conns_block, [monit_block]
```

Listing 1: ENBF Schema

The EBNF is split into 3 "blocks" where two of them are required and the last `monit` block is optional. The blocks are defined using the syntax `[block_name]`, this is commonly used syntax in many config files, eg *PlatformIO's* `platformio.ini` files. Using this notation it is easy to see where a block starts and it allows for a block to be parsed very simply without having to worry about start and end of block phases. The 3 blocks that we use are:

- `devices` block is where all the devices in the circuit are defined, they are defined using traditional variable assigning syntax. The gates themselves are just the capitalised names and they take parameters which are used to adjust the properties of the device. Some of the devices also have default parameters which are commonly used, which enables for rapid development.
- `conns` block is used to define connections between devices, on the LHS you select which device output connects to inputs (can have multiple). The "=" sign is used again, since when you connect two IO they will have the same logic state, hence the "=" sign has logical meaning. For devices with only a single output, you do not need to specify it. Inputs are defined using "." followed by numbers (apart from DTYPE which has port names)
- `monit` block is where the user indicates which IO states should be monitored. This is an optional block.

We require semicolons at the end to represent the end of a definition, instead of using the newline character to represent the end of a definition. The reason why semicolons are used in languages, such as JavaScript, is for the purpose of obfuscation, and it gives users the freedom of arranging the code in any format, thus whitespaces and tab breaks are ignored in our parsing.

Comments are defined using `#` much like python, anything following the `#` will be treated as a comment until a newline character is found. A newline character is a requirement for the ending of a comment.

3.1 Example 1

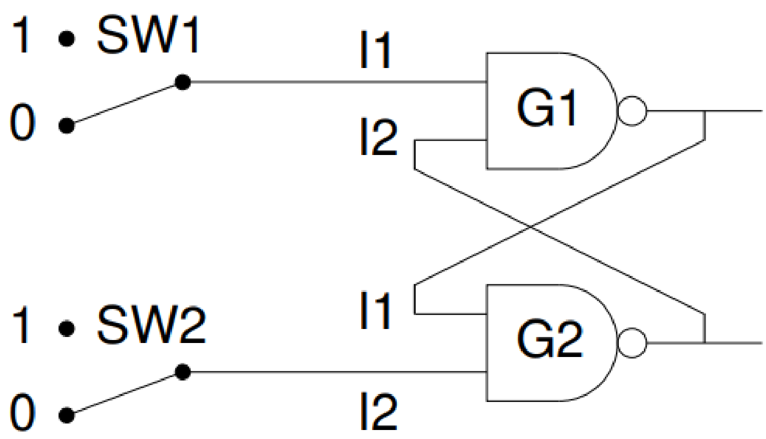

Figure 3: Example 1

The following is the EBNF for this circuit:

```
[devices]
  G1, G2 = NAND;      # Defaults to 2 inputs
  SW1, SW2 = SWITCH; # Defaults to 0 i.Ie "False"

[conns]
  SW1 = G1.I1;
  SW2 = G2.I2;
  G1=G2.I1;             # Whitespace not required
  G2 = G2.I2;

# new blank lines are ignored

[monit]                  # Need a newline after a block
  G1, G2;
```

Listing 2: Code for example 1

```
[devices]G1,G2=NAND;SW1,SW2=SWITCH;[conns]SW1=G1.I1;SW2=G2.I2;G1=G2.I1;G2=G2.I2;[monit]G1,G2;
```

Listing 3: Obfuscated Code

3.2 Example 2

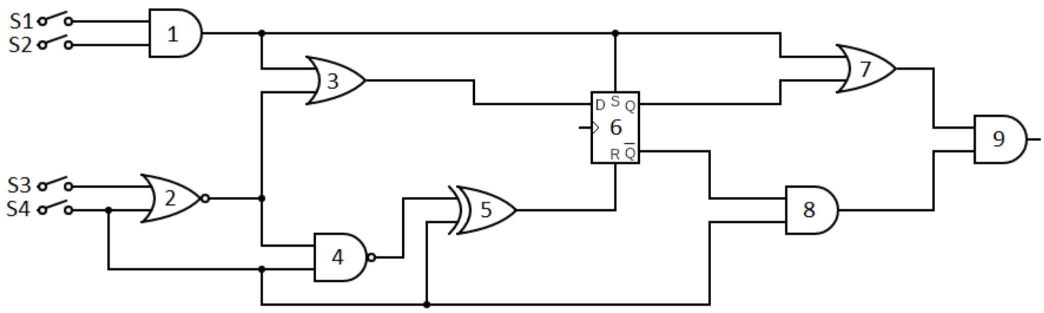

Figure 4: Example 2

The following is the EBNF for this circuit:

```
[devices]
    G1, G8, G9 = AND;          # Default 2 input AND gate
    G2 = NOR(2);               # Can specify number of inputs but defaults to 2 if not specified
    G3, G7 = OR;
    G4 = NAND;
    G5 = XOR;

    SW1, SW2 = SWITCH;         # Defaults to 0 which is "False"
    SW3, SW4 = SWITCH(0);      # Can be specified as 0 for "False" but defaults to 0 if not specified

    G6 = DTYPE;
    CLK1 = CLOCK(5);           # Clock of period 5 (Not shown in the diagram)


[conns]

    # Switch connections
    SW1 = G1.I1;
    SW2 = G1.I2;
    SW3 = G2.I1;
    SW4 = G2.I2;

    # Clock to DTYPE
    CLK1 = G6.CLK;

    G1 =  G3.I1, G6.SET, G7.I1; # Can connect output to multiple inputs, SET is "S" in the diagram
    G2 =  G3.I2, G4.I1;
    G3 =  G6.DATA;              # DATA is "D" in the diagram
    G4 =  G5.I1;
    G5 =  G6.CLEAR ;            # CLEAR is "R" in the diagram

    G6.Q = G7.I2;                # Need to specify which output of DTYPE since there are 2
    G6.QBAR: G8.I1;

    G7 = G9.I1;
    G8 = G9.I2;


[monit]
    G9, G8;
    G6.DATA;
```

Listing 4: Code for Example 2

```
[devices]G1,G8,G9=AND;G2=NOR(2);G3,G7=OR;G4=NAND;G5=XOR;SW1,SW2=SWITCH;SW3,SW4=SWITCH(0);G6=DTYPE;CLK1=CLOCK(5);[conns]SW1=G1.I1;SW2=G1.I2;SW3=G2.I1;SW4=G2.I2;CLK1=G6.CLK;G1=G3.I1,G6.SET,G7.I1;G2=G3.I2,G4.I1;G3=G6.DATA;G4=G5.I1;G5=G6.CLEAR;G6.Q=G7I.2;G6.QBAR:G8.I1;G7=G9.I1;G8=G9.I2;[monit]G9,G8;G6.DATA;
```

Listing 5: Obfuscated Code for Example 2

4 Errors

4.1 Error Handling

Eliminating errors will be crucial to ensuring our program runs smoothly. All errors when reported will alert the user what kind of error it is, what line the error is found, and what character in the line the error arises from. The process of error detectio requires the usage of the `scanner` and `parse` to detect errors. The `parser` will detect errors with the `scanner` reporting the location of such error. We will define a specfifc `error_handler` class that will handle the construction of the error message using the `error_type`, `line` and `char` as parameters. The `error_handler` will then print the error message to the user.

```
ERROR (error\_type) found in line {line} at character {char}
{descriptions}
Line: {line}
  {line};
```

We reprint the line in which the error occurs to allow the user to easily identify the error. In the following sections we will discuss the different types of errors that may occur and how we we will handle them.

4.2 Semantic Errors

Semantic errors are errors that occur when the code is syntactically correct but does not make sense. For example, connecting an output to an output. Here are some examples of semantic errors that may occur and how we will detect them:

Error Type	Description	Detection
INPUT_TO_INPUT	A device input is connected to another input	When parsing the `conns` section check to see if the LHS expression is an output and the RHS contains inputs only
OUTPUT_TO_OUTPUT	A device output is connected to another output	When parsing the `conns` section check to see if the LHS expression is an output and the RHS contains outputs only
UNDEFINED_DEVICE	A device is not defined	When parsing the `conns` section check to see if the device has been definied in the `devices` block
INVALID_PARAM	A device is defined with an invalid parameter	When parsing the `devices` block check to see if the parameter is valid for the device
INVALID_PORT	A device is defined with an invalid port	When parsing the `conns` block check to see if the port exists on the device
DUPLICATE_DEVICE	A device is defined more than once	When parsing the `devices` block check to see if the device has already been defined
FLOATING_INPUT	An input has no connections	When we reach the end of the `conns` block, we check to see if all inputs were used

There are more semantic errors that may occur, but these are the most common ones that we will be checking for, we will most likely add more as we continue to develop and test our program.

5 Syntax Errors

The following sections will describe some of the syntax errors that may occur and how we will detect them.

5.0.1 Missing blocks

Missing block errors are easy to detect, as we parse the entire text and check if the block names are present in the code. If not we alert the user that a block is missing.

5.0.2 Missing Semicolons

There are some trivial errors that will come up, such as missing semicolons, when a user forgets a semicolon we end up parsing a line with more than two “=” signs, thus we can alert the user that there maybe a missing “;” on the parsed line.
When a user forgets a semi colon, then we will end up parsing a line with more than 1 ”=” sign.

```
G1 = NAND G2 = OR # Missing semi colon
```

Then we can inform the user that there is a missing semicolon at the parsed line.

5.0.3 Missing Commas

A user may forget to add a comma between multiple definitions, as such

```
G1 = G2.I1 G2.I2  # Missing comma
```

When we parse this, we will end up reading a device name with a white space, which is not allowed within our EBNF spec, therefore an error will be raised stating that there is a missing comma in the specific line.

5.0.4 Incorrect Device Parameters

Errors in the device parameters can be easily checked as we can ensure that the parameters values fall within the acceptable range (eg 1-16 inputs for the gates). For devices that require parameters, like the clock, we can alert the user that "A CLOCK at line n is missing its period parameter". Errors will be immediately thrown as the parser finds them line by line, and the parse will stop parsing when it finds an error. This ensures that the user will fix any errors line by line instead of being overwhelmed by many error messages.

5.0.5 Unknown Symbol

If a user for some reason decides to use a symbol which isn't in our EBNF (in places other than the comments), then we will throw an unrecognized symbol error.

5.0.6 More Error

There are of course many more errors but they are not stated in this document. Most the errors will be found through our work on writing tests.

6 Conclusion

The tasks we have done so far:

- Task clarification and team planning
- EBNF definition and logic circuits examples
- Error handling
- Collaborative coding using Github

The tasks we are planning to do in the next phase:

- Name translation
- Scanner
- Parser
- GUI
- Tests on modules
- Modules integration