GF2 SOFTWARE
Interim Report I

Name: Minh Nguyen (mhn28), Yi Lim (yhl48), Sujay Thakur (st670)
College: Christ's

1 Introduction

This project aims to develop a logic simulation program in Python. This first interim report will describe the general approach adopted, outline teamwork planning, specify the syntax for the language, outline error handling and provide examples of the language being used to describe circuits.

2 General Approach

Two possible approaches were considered to dictate the grammar of the definition files - a methodical and structured approach or a more flexible and relaxed approach. The methodical approach would entail separating the definition files into distinct sections specifying devices, connections and signals to be monitored separately. This would simplify the parsing of the definition file as the entire circuit is defined clearly in a specific section and is hence easier to interpret. The alternative approach would be to completely define each component and their connections one at a time. This approach is a more intuitive way to describe the logic circuit for the end user as it is easier to see where each component is connected to and it also emulates how a circuit is built in reality. Moreover, it is more flexible as it allows easy addition of further components. However, this approach would require a more complex parser as some components would be defined after they are connected and thus would require at least two iterations over the definition file in order to check that all connections are valid.
It was decided that the first approach would be preferred as it allows for easier error checking: the identification of errors is made easier due to the sectional approach in defining the components, connections and signals to be monitored. It is also less likely for the users to miss something out as the sequential nature of the approach ensures that the definition files will be written in a more structured way.

3 Teamwork Planning

It was decided that the overall task would be split into sub-tasks involving the `names`, `scanner`, `parser` and `GUI` modules. Minh would be in charge of the `names` and `scanner` modules, Sujay would be in charge of the `parser` module and Yi Heng would be in charge of the `GUI` module. Independent work would be carried out on assigned modules, with scheduled meetings and collaborative work twice a week. Testing on the modules will be conducted in collaboration with other members not directly responsible for the modules to ensure all scenarios are considered and that everyone completely understands the entire program. Given that testing for the `parser` module requires the `names` and `scanner` modules, Minh and Sujay would collaborate on ensuring these are finished earlier. Individual implementations will be done by 25 May, with 26-30 May left for more rigorous testing and integration.

4 EBNF for syntax

```
network = "START DEVICES", devicelist, "END DEVICES", "START
CONNECTIONS", connectionlist, "END CONNECTIONS", "START MONITOR",
monitorlist, "END MONITOR";

devicelist = device, {device};
device = name, "=", object, [",", "ip=", number], [",", "init=",
number], [",", "cycles=", number], ";";
name = word, {number};
word = letter, {letter};
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";
number = digit, {digit};
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
object = "CLOCK"|"SWITCH"|"AND"|"NAND"|"OR"|"NOR"|"DTYPE"|"XOR";

connectionlist = connection, {connection};
connection = output, "->", input, {",", input}, ";";
output = name, [".", outputname];
outputname = "Q"|"QBAR";
input = name, ".", inputname;
inputname = inputsig|"DATA"|"CLK"|"SET"|"CLEAR";
inputsig = "I", number;

monitorlist = output, {",", output}, ";";
```

It should be noted that `ip` is the number of inputs of the device, `init` is the initial value of the device and `cycles` is the number of simulation cycles after which the output changes state.

5 General Error Identification Approach

We have chosen to generally specify more rigid description file guidelines for the user, which reduces flexibility but simplifies error handling. Exact examples will be given in later sections. We want to provide useful error messages, with indicators to specify exactly where an error was made. We also want to report all errors at once, rather than sequentially report the next error every time the file is run. This would make debugging an easier task for the user.

6 Syntax Error Identification

The identification of syntax errors will be handled by the parser, based on the rules specified in Section 4. One notable idea is that all device parameters are specified as optional, and hence even if devices requiring such parameters lack them, these will not be flagged as syntax errors, but rather as semantic errors. This is to simplify the general syntax checks made by the parser. We have also specified that each device and connection expression ends with a semicolon. This ensures that following a syntax error in an expression, the parser could easily skip text until the next semicolon and then resume normal operation. Another idea is that the user-defined name of each device must be alphabets followed by numbers. We believe that this rigid naming system will ensure that users select useful names that will prevent confusion with multiple devices of the same type. To prevent grammar clutter, we have also allowed any output to specify `"Q"|"QBAR"`, and the validity of this (ie making sure that the device is a DType) will be checked as semantics. A similar idea exists for inputs, as seen in the grammar.

7 Semantic Error Identification

This section attempts to identify all semantic errors that will be handled by the parser. When specifying devices, there was a choice on whether redundant parameters should be flagged as semantic errors. For example, an XOR gate has 2 inputs by default, but if the user chooses to specify `ip=2`, it was decided that this would not be an error. However, if any other number was specified, this would be an error. The specifics for the different devices are outlined in Table 1.

	Required Parameters	**Optional Parameters**	**Incorrect Parameters**
Clock	- “cycles” (only positive integers)	- “ip” (defaults to 0, so can only be set as 0)	- “init”
Switch	- “init” (only 0/1)	- “ip” (defaults to 0, so can only be set as 0)	- “cycles”
Gates (exc. XOR)	- “ip” (only integers from 1-16)		- “cycles” - “init”
DType		- “ip” (defaults to 4, so can only be set as 4)	- “cycles” - “init”
XOR		- “ip” (defaults to 2, so can only be set as 2)	- “cycles” - “init”

Table 1: Device Semantic Errors.

It should be noted that a semantic error will be flagged if an incorrect parameter is set or if an optional parameter is set incorrectly, with a pointer in the error message indicating the erroneous parameter.

When specifying connections, all inputs must be used exactly once. When devices are initialised, a dictionary will be created with all the inputs. Then, when the connections are being initialised, a counter will be set up for each input. At the end of the initialisation, each counter value will be checked to ensure it is `1`. This allows all inputs to be checked simultaneously, in case there are multiple of such errors (rather than specifying each error once encountered and not identifying the rest). When gate input n is used for a gate X, the input has to be specified as X.In, where $n \in \{1, ..., \texttt{ip}\}$ (specified in `devicelist`). This can simply be checked by the parser, since we are restricting the values of n (simply check if value is positive and less than `ip`). When DType outputs are used, ensure an output name is specified (the actual value is checked in the syntax directly). When DType inputs are used, ensure that only DATA, CLK, SET, CLEAR is allowed. These can easily be checked by the parser.

8 Definition File Examples

This is an example of a combinational circuit, shown in Figure 1, along with its description file.

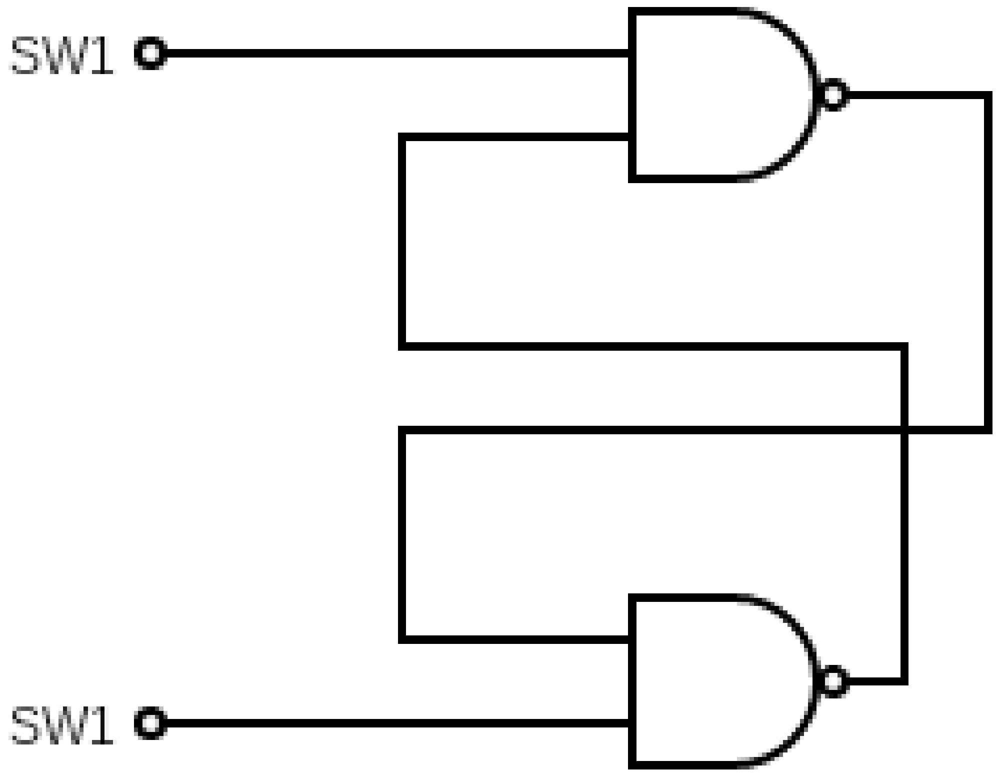

Figure 1: Example combinational circuit.

```
START DEVICES
SW1 = SWITCH, init=0;
SW2 = SWITCH, init=0;
G1 = NAND, ip=2;
G2 = NAND, ip=2;
END DEVICES

START CONNECTIONS
SW1 -> G1.I1;
G1 -> G2.I1;
G2 -> G1.I2;
SW2 -> G2.I2;
END CONNECTIONS

START MONITOR
G1, G2;
END MONITOR
```

This is an example of a sequential circuit, shown in Figure 2, along with its description file.

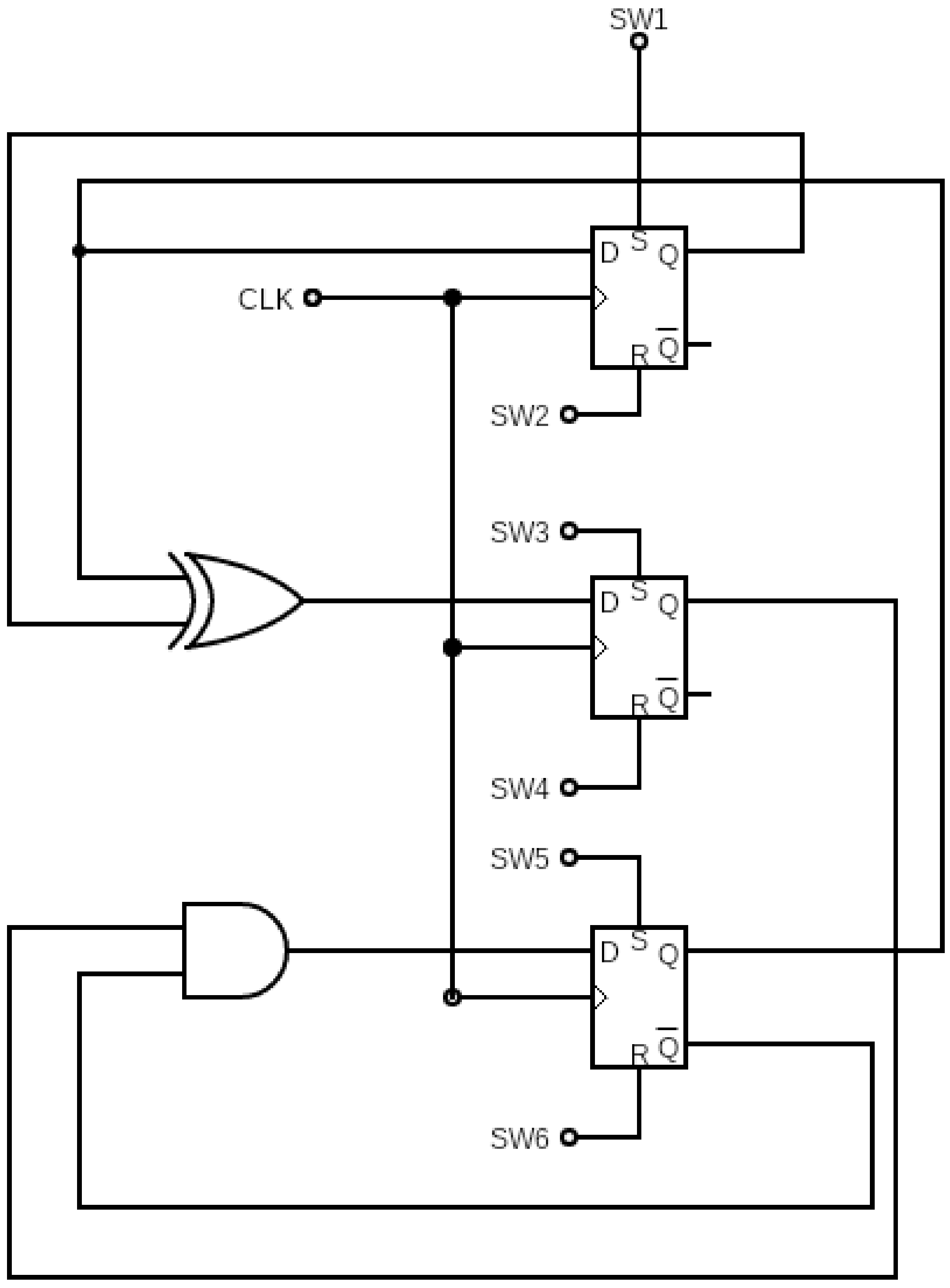

Figure 2: Example sequential circuit.

```
START DEVICES
CLK = CLOCK, cycles=10;
D1 = DTYPE;
D2 = DTYPE;
D3 = DTYPE;
XOR = XOR;
AND = AND, ip=2;
SW1 = SWITCH, init=1;
SW2 = SWITCH, init=0;
SW3 = SWITCH, init=1;
SW4 = SWITCH, init=0;
SW5 = SWITCH, init=1;
SW6 = SWITCH, init=0;
END DEVICES

START CONNECTIONS
SW1 -> D1.SET;
SW2 -> D1.CLEAR;
CLK -> D1.CLK, D2.CLK, D3.CLK;
SW3 -> D2.SET;
SW4 -> D2.CLEAR;
SW5 -> D3.SET;
SW6 -> D3.CLEAR;
D1.Q -> XOR.I2;
D2.Q -> AND.I1;
D3.Q -> XOR.I1, D1.DATA;
D3.QBAR -> AND.I2;
XOR -> D2.DATA;
AND -> D3.DATA;
END CONNECTIONS

START MONITOR
D1.Q, D2.Q, D3.Q;
END MONITOR
```