GF2 Software First Interim Report

Group20 Harrison(jz503) David(dxh21) Scofield(yl732)
College: Fitzwilliam, Pembroke, Magdalene

1. Introduction

The purpose of this project is to develop a logic simulation program in order to meet client's requirements. This first report mainly focuses on the specification and design, including the general approach, the EBNF syntax, error detection and handling and so on. Also, a team planning table is included to achieve more efficient cooperation and deliver higher quality product.

2. General approach & Team Planning

The project will be broken into individual tasks (code modules) within each phase of the software development life cycle. This modular approach has been widely used today in large software engineering teams (known as CI/CD) and proven to be efficient practice for both code development and maintenance. For the current stage, before the actual code implementation, the software skeleton will be based on: the EBNF syntax, the error detection and handling methods and the definition file which can be used as part of the user guide for future deployment. Therefore, after the collective decision on syntax style based on the client's requirements, a detail implementation schedule is given below:

Task	**Member**	**Due Date**
EBNF Syntax	Harrison	05/18
Definition file with examples	Scofield	05/20
Error detection and handling	David	05/20
First Interim Report	**Group**	**05/22 4:00pm**
Scanner Design	Harrison	05/28
Names Design	Scofield	05/28
Parser Design	David	05/28
Implementation	Group	05/29
Gui Design/Implementation	TBC	05/30
User guide	TBC	06/01
Second Interim Report	**Individual**	**06/03 11:00am**
Maintenance	TBC	TBC
Final Integration and test	TBC	06/08
Final Report	**Individual**	**06/10 4:00pm**

The schedule after the first report is for reference only and a detailed Gantt-Chart will be used when we explore deeper into the project. The code will continue to be revised and updated in future development.

Code can be found at: https://github.com/XiaoLuoLYG/GF2_Software As git is used for version control and to track the work of each team member in this project.

3. EBNF Syntax

Our EBNF Syntax mainly contains three main sessions, namely, device, connection and monitor sessions. In the device section, we define the basic information for devices, consisting names and initial states, e.g. the type of the gates and how many inputs the gates have. The connection section provides the information about how devices are connected in the circuit. And in the last section, monitor session, we show the monitor points for the circuit. If this section does not appear, the output will be shown later. To summarise, the EBNF grammar for our syntax is :

```
Terminal = "Device",":","{",Device_info,Device_info_input, {Clock_initial_state},"}",
"Connection",":","{", Connection_info,"}",
"Monitor",":","{", Monitor_info,"}";

Device_info = Device_name, {",", Device_name}, Copula, Devices, "with", digit, "inputs",";" ;

Device_info_input = Device_name, {",", Device_name}, Copula, Devices, "initially", "with", digit,
"input",["s"],"}",";";

Connection_info = Device_name, {".",Device_name}, ->, Device_name,".",Gate_input,";";

Clock_initial_state = Device_name, {",", Device_name}, Copula, Devices, "initially", "with", digit,
"simulation", "cycles", ";" ;

Monitor_info =  {Device_name, {",", Device_name}},";";

Devices = "CLOCK" | "SWITCH" | "AND" | "NAND" | "OR" | "NOR" | "DTYPE" | "XOR" ;

Device_name = letter , { letter | digit };

Gate_input = letter, ( digit );

Copula = "is" | "are";

-> =  "->";

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" | "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 = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Furthermore, the design of the EBNF syntax aims to make the description of the circuit as simple as possible. For example, instead of “is connected to”, we decide to use the arrow “->” to represent the connection between two devices.

4. Definition file with 2 examples

Notes before reading the definition files:

- Our EBNF syntax defines CLOCK and SWITCH as two special devices as they require initialization (simulation cycles for CLOCK and input level for SWITCH). Therefore, in the device definition section for SWITCH, the 0/1 digit indicates the initial low/high level of the SWITCH, not the number of input.

- The semicolon at the end of each line is used for error handling (see error handling section)

- The line break at the curly brackets for each section is for readability only and is not compulsory by the syntax

- The order of the sections (Device - Connection - Monitor) do matter especially for the parser code later in the project

- Both example circuits are drawn in Simulink.

4.1 Example 1 - SR FlipFlop

In the first simple example, an SR flip flop circuit similar to the example provided in the handout will be used as shown in Figure 1. CLOCK, SWITCH and NAND gate are included in this example. The code that defines the circuit is given by Figure 2.

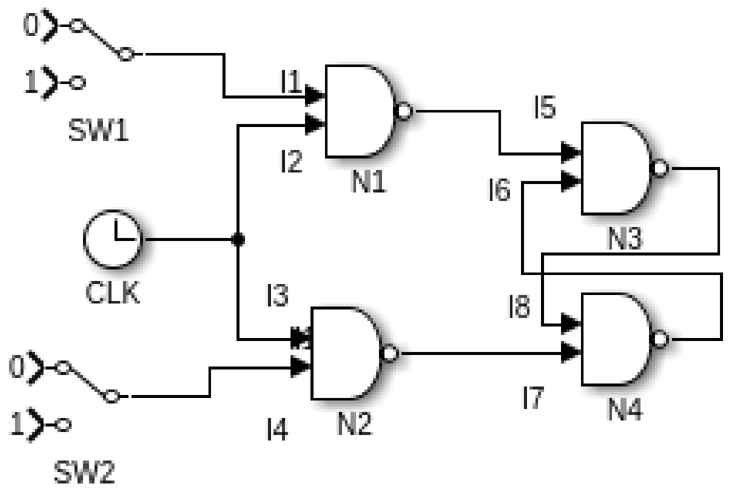
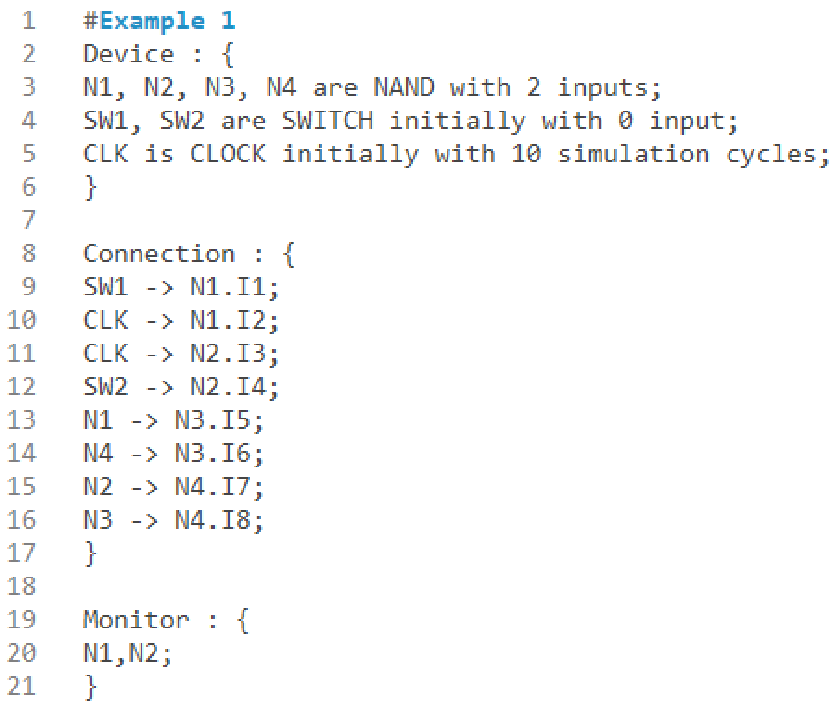


Figure 1: SR flip flop circuit with NAND gates



```
#Example 1
Device : {
N1, N2, N3, N4 are NAND with 2 inputs;
SW1, SW2 are SWITCH initially with 0 input;
CLK is CLOCK initially with 10 simulation cycles;
}

Connection : {
SW1 -> N1.I1;
CLK -> N1.I2;
CLK -> N2.I3;
SW2 -> N2.I4;
N1 -> N3.I5;
N4 -> N3.I6;
N2 -> N4.I7;
N3 -> N4.I8;
}

Monitor : {
N1,N2;
}
```

Figure 2: SR flip flop circuit definition file

4.2 Example 2 - Complex Circuit

A more complex circuit drawn in simulink is shown in Figure 3. The function of the circuit is trivial as the purpose is to test the syntax and help users to better implement the circuit simulation. Only 3 devices' inputs are included as examples for clarity. Note that in Simulink, the D-type bistable module only has 3 inputs, the 4th CLEAR input is not shown in the diagram but included in the definition file in Figure 4. All functions ("CLOCK" | "SWITCH" | "AND" | "NAND" | "OR" | "NOR" | "DTYPE" | "XOR") are tested in this circuit.

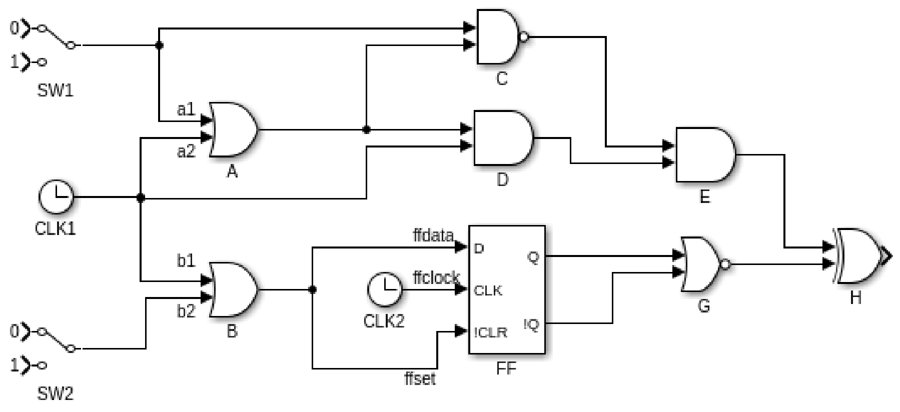
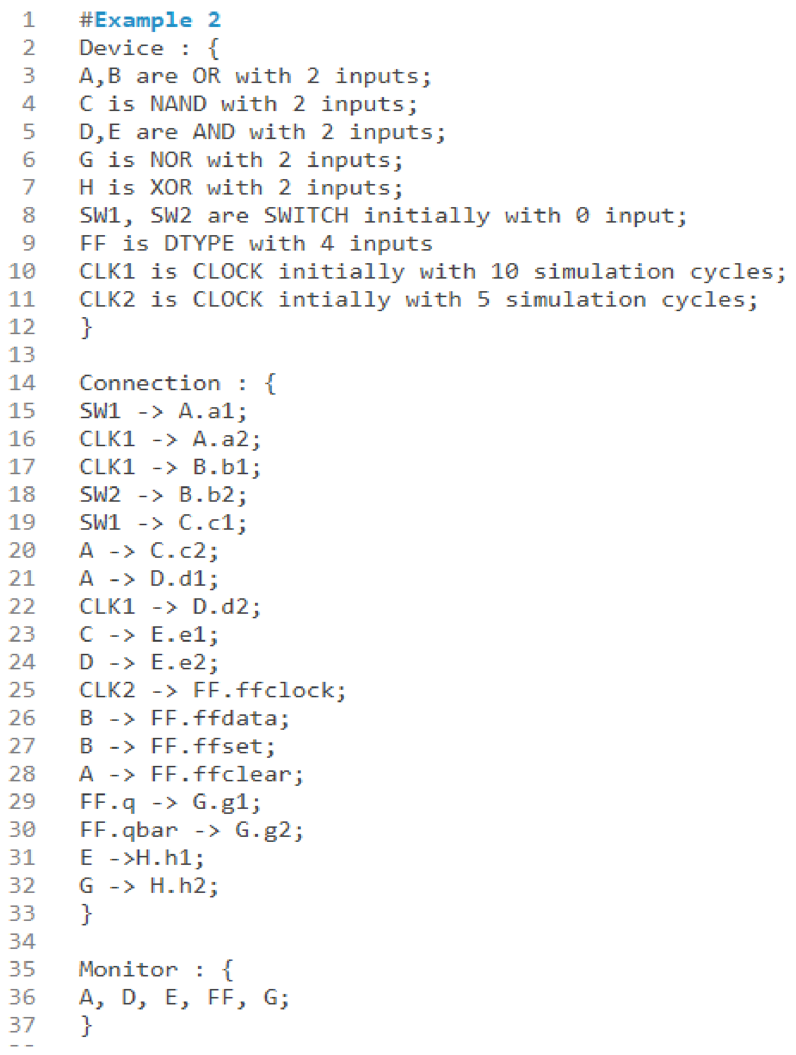


Figure 3: Complex circuit with all functions included



```
#Example 2
Device : {
A,B are OR with 2 inputs;
C is NAND with 2 inputs;
D,E are AND with 2 inputs;
G is NOR with 2 inputs;
H is XOR with 2 inputs;
SW1, SW2 are SWITCH initially with 0 input;
FF is DTYPE with 4 inputs
CLK1 is CLOCK initially with 10 simulation cycles;
CLK2 is CLOCK intially with 5 simulation cycles;
}

Connection : {
SW1 -> A.a1;
CLK1 -> A.a2;
CLK1 -> B.b1;
SW2 -> B.b2;
SW1 -> C.c1;
A -> C.c2;
A -> D.d1;
CLK1 -> D.d2;
C -> E.e1;
D -> E.e2;
CLK2 -> FF.ffclock;
B -> FF.ffdata;
B -> FF.ffset;
A -> FF.ffclear;
FF.q -> G.g1;
FF.qbar -> G.g2;
E ->H.h1;
G -> H.h2;
}

Monitor : {
A, D, E, FF, G;
}
```

Figure 4: Complex circuit definition file

5. Error detection and handling

Error detection and handling is an important aspect of a program. There are two general types of errors, syntax errors and semantic errors. Each of these require their own ways of detection, reporting and handling.

5.1 Syntax Errors

Syntax errors occur when the definition file does not perfectly conform to the defined syntax. Common syntax errors could include:

1) For Connection_info:

   ```
   N1 -> N2.I3
   ```

   where a semicolon ";" is missing at the end of the line.

2) For Device_info:

   ```
   N1, N2, N3 N4 are NAND with 2 inputs;
   ```

   Where a comma is missing between N3 and N4.

Syntax errors are raised when the parser goes through the definition file, once a syntax error is encountered, the parser will skip the line containing the error by moving to the next semicolon. A marker should also identify the location of the syntax error, a generic message "Syntax error on line (.)" could be passed, however if the error can be identified (e.g. if the parser expects a comma but instead reads a letter), the message "Error: Expected comma" can be passed.

After identifying a syntax error and skipping to the next semicolon, the parser continues to read the rest of the definition file and keeps a count of the number of total errors.

5.2 Semantic Errors

Semantic errors occur when the definition file perfectly conforms to the defined syntax however parts of the definition file doesn't make logical sense. Below includes all possible semantic errors and how they will be detected and reported.

Type of error for `Device_info`	Detection	Reporting

`A, B are OR with 2 inputs;`
`B is NAND with 2 inputs;`
 	
Device B is defined twice in the definition file.
	Once a gate has been defined already, its name is stored, if it appears again in the definition file, an error is detected.	"Device B is defined multiple times"

`A is AND with 17 inputs;` The number of inputs and outputs for	Once the type of gate has been identified, a range of numbers is expected, if the	"17 exceeds maximum number of

devices is not defined in the syntax, therefore an error is raised when the number of inputs is specified wrongly.	number in the definition file does not belong in this set of expected numbers, an error is detected.	inputs for A”
`A are OR with 2 inputs;` In this case, since there is only one device being defined, we expect “is” instead of “are”. This indicates a typo of some sort so an error should be raised for the user to check the definition file for possible missing devices.	Once a singular gate is detected, the parser expects “is”, if “are” is found to come next, an error should be raised here instead of further down in the definition file to save computation.	“Typo, possible missing/extra device”

Type of error for `Connection_info`	**Detection**	**Reporting**
`A1 -> A6.a2` Where any of these devices might not exist.	Check the devices in the connection section against the devices defined previously, if a device is not defined in `device_info`, an error is raised.	“Device A6 not defined”
`A1 -> A6.a2` Where a2 does not belong to A6	Check the inputs against `device_info`	“a2 does not belong to A6”
`A1 -> A3.A6` Where A6 is an output instead of an input.	Check the inputs against `device_info`	“A6 is not an input”

`A1 -> A3.a2`
`A2 -> A3.a2`
 	
Where two outputs are connected to a single input.
	Once a connection has been defined, the nodes in question are stored, if they appear again later on in the definition file, an error is raised.	“A1 and A2 are both connected to A3.a2”

`A1 -> A6.a2` Where A6 requires 2 inputs but only receives 1.	At the end of the connection section of the definition file, check that every input is connected to another device.	“A6 requires 2 inputs but only has 1”

6. Conclusion

Using everything specified above, a user will be able to fully define a logic circuit by writing a definition file using the EBNF syntax. Any errors in the definition file will then be detected and both syntax and semantic errors will be reported in a way that makes identification of the error and correction as easy as possible.