Team 3 - GF2 First Interim Report

Sherry Fu - jf731
James Lecomte - jccl4
Alex Man - ajkm3

May 18, 2024

1 Introduction

The purpose of this project is to develop a logic simulator in Python, capable of simulating both combinatorial and clocked logic circuits. The simulator will read a text file defining the logic elements, their connections, and monitor points, then execute the defined logic network under user control. After the network has been instantiated, the user will be able to execute commands in real time within the GUI to set inputs, run the simulation and modify monitor points. This report outlines our general approach, teamwork planning, syntax specification using EBNF, semantic error identification, error handling strategy, and provides example definition files and diagrams representing the logic circuits.

2 General Approach

Our approach follows the structured phases of software engineering: specification, design, implementation, testing, and maintenance. We began by forming a development team, then proceeded to specify the logic description language, identifying syntax and semantic rules, and error handling mechanisms. We divided the implementation into distinct modules, namely "names", "scanner", "parser", and "gui". Each team member took responsibility for different modules to ensure parallel progress and efficiency. The following section describes our teamwork allocation plan, see Table 1.

2.1 Teamwork Planning

Tasks	**Team member**	**Deadline**
EBNF language	James	13/05
Error handling	Alex	16/05
Example Circuit Generation	Sherry	16/05
Interim report 1	All	18/05
`name.py` complete	Sherry	19/05
`scanner.py` complete	Alex	20/05
`parser.py` complete	James	23/05
`gui.py` complete	Sherry	27/05
Interim report 2	Individual	30/05
Maintenance	All	06/06
Final report	Individual	06/06

Table 1: Teamwork Planning Table

3 Scanning and parsing definition files

For scanning and parsing we will use a LL(1) EBNF grammar.

3.1 EBNF Grammar

The following defines the alphabet for our EBNF grammar:

```
definition_file = { codeline } ;

codeline = (device_instantiation | connection | monitor), ";" ;

device_instantiation = device_type , device_identifier_group ;

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

monitor = "MONITOR" , output_identifier_group ;

connection = output_identifier_group, ">", input_identifier_group ;

output_identifier_group = ["("] , output_identifier , { "," , output_identifier } , [")"];
input_identifier_group =  ["("] , input_identifier , { "," , input_identifier } , [")"];
device_identifier_group = ["("] , device_identifier , { "," , device_identifier } , [")"];

output_identifier = device_identifier, [".", ("Q" | "QBAR")];
input_identifier = device_identifier, [".", ("CLK" | "DATA" | "SET" | "CLEAR" | number)];

device_identifier = letter, {letter | digit} ;
number = nonzero_digit, {digit} ;
```

3.2 Error handling

At the end of each line, there will be a semi-colon. When an error is detected, an error will be raised with a carat symbol ($\wedge$) pointing to the error with a description of what syntax error occurred. The parser will then skip the rest of the line until the next semi-colon is reached.

3.2.1 Syntax errors

During parsing, if any errors come up, the program will raise the error and print what the error was. For some errors such as putting a comma instead of the name of an input, the error message may also include what was expected or unexpected.

3.2.2 Semantic errors

Semantic errors will be processed after the program has been scanned, parsed and checked for syntax errors. When semantic errors are raised, the program will output a warning and attempt to skip the current instruction without fully abandoning the parse.

For each possible instruction, semantic errors will be raised in the following cases:

Device instantiation A device with the same name has already been instantiated.

Connections `e.g. "(Outputs) > (Inputs)"`
For inputs (right side of >):

- A connection is being made to an unassigned device name.
- A connection is being made to an input that is already connected to.
- A connection is attempted to a D-Type device without specifying which input to connect to. (Note for other devices, specification of which input to connect to is not required since they are all functionally identical.)
- A connection is attempted to a non-D-Type device that already has all of its inputs connected to.

For outputs (left side of >):

- A connection is being made from an unassigned device name.
- The output identifier of a D-Type device is not specified (either `.Q` or `.QBAR`), or inversely one of these has been specified for a non-D-Type device.

Monitoring The device to monitor has not yet been instantiated or is already being monitored.

3.2.3 Warnings

While not necessarily syntax or semantic errors, the following bad practices will be shown as warnings to the user after parsing the definition file, to catch mistakes in circuit definition.

- Device defined and not used (i.e. no inputs or outputs allocated)

4 Example definition file

4.1 Example 1: SR flip-flop Circuit

The code that defines the circuit in Figure 1 is given by:

```
SWITCH SW1, SW2;
NAND G1, G2;

### Showcasing different ways to express connections
Implicit: outputs will connect to the first available input
Explicit: outputs will connect to specific input (required for D-Type)
###
(SW1, G2) > G1;

SW2 > G2.2;
G1 > G2.1;

MONITOR G1, G2;
```

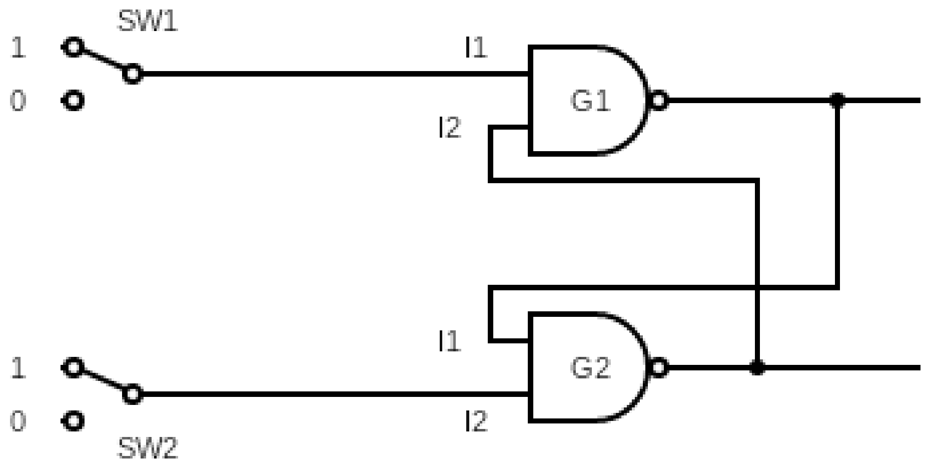

Figure 1: SR flip-flop circuit

4.2 Example 2: Comprehensive Circuit

The code that defines the circuit in Figure 2 is given by:

```
SWITCH SW1, SW2, SW3, SW4;
NAND A1, A3, A4;
AND A2;
XOR X1;
OR N1;
NOR N2;

# Pre-D-Type circuit
(SW2, SW3) > A1;
(SW3, SW4) > X1;
(SW1, SW2, A1) > N1;
(A1, X1, SW4) > N2;
(N1, N2) > A2;

# D-Type circuit
DTYPE D1;
CLOCK C;
SWITCH SW5, SW6;

SW5 > D1.SET;
SW6 > D1.CLEAR;
A2 > D1.DATA;
C > D1.CLK;

# Post-D-Type circuit
```

```
(SW1, D1.Q) > A3;
(D1.QBAR, SW4) > A4;

MONITOR A3, A4;
```

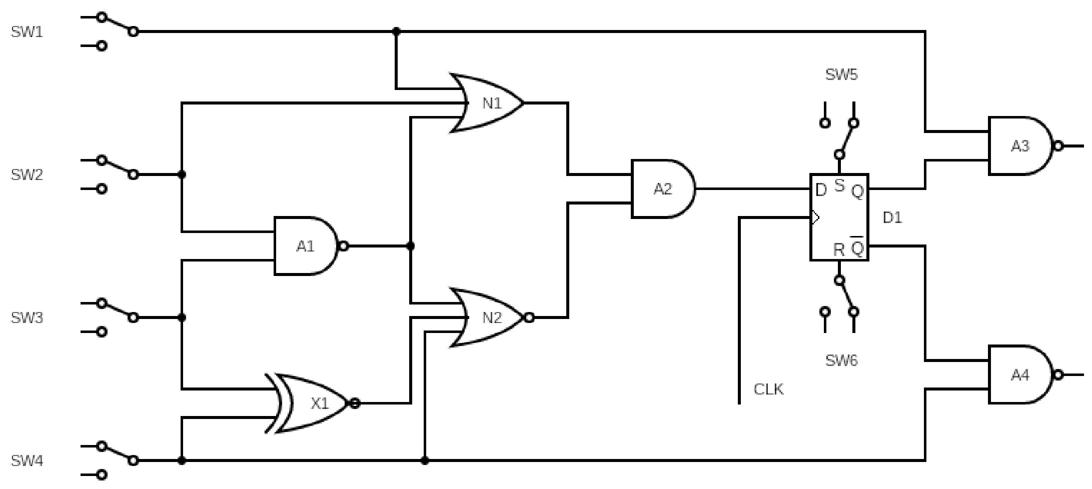

Figure 2: Comprehensive circuit