GF2: Software - Logic simulator

Tung X. Le

June 2023

1 Introduction

This report outlines a project to develop a Python-based logic simulation program (Python 3) that covers the five major phases of the software engineering life cycle: specification, design, implementation, testing, and maintenance. The objective is to create an intuitive and versatile logic simulator for accurately modeling digital circuits, enabling engineers and designers to validate designs, identify issues, and optimize performance. The project aims to deliver a comprehensive and user-friendly program, empowering users to enhance their digital circuit designs effectively.

2 Software structure

Our logic simulator is able to simulate any number of circuits which in include the following devices:

- Clocks
- Switches
- AND gates (Up to 16 inputs)
- NAND gates (Up to 16 inputs)
- OR gates (Up to 16 inputs)
- NOR gates (Up to 16 inputs)
- XOR gates
- D-Type flip flops
- Signal generators
- RC devices

To define circuits, we utilize text files written in a specific language described in our EBNF (Extended Backus-Naur Form), which can be found in Appendix B. When reading a definition file, any encountered errors are reported to the user with detailed information about the line and character where the error occurs. The number of devices in a network is only limited by the available memory of the computer, and there are no restrictions on the number of monitors that can be used. To effectively operate our logic simulator, please refer to the user guide provided in Appendix C.

The structure of our logic simulator follows modularization principles through object-oriented programming. The `Network` class represents the internal representation of the logic circuit, while the `Devices` class stores information about each device's properties. Device names are internally represented using unique integers. The `Names` class handles the storage of names as they are read and maintains a list of reserved names. When a file is opened, the parser class reads it character by character using an instance of the scanner class and the `get_symbol()` method. Based on the order of the symbols, the parser creates devices, connections, monitors, or raises errors as necessary. Once the entire definition file has been processed, the `UserInterface` class interprets user commands and interacts with the network, monitor, and devices classes accordingly.

3 Development

Our development process for the logic simulator consisted of four distinct phases: familiarization, design, implementation, and testing. Each phase had a specific timeframe and involved assigning tasks to individual team members or the entire team based on their expertise. For a visual representation of our project timeline, please refer to the accompanying Gantt chart.

To enhance collaboration and version control, we extensively utilized Git throughout the project. GitHub served as a valuable platform for keeping track of our plans and managing client-required features. Leveraging Git's capabilities, we were able to work on the same files independently and seamlessly merge our changes using the "git pull" command, which internally invokes the "git merge" command.

During the maintenance phase of the project, we divided the tasks and worked on them independently. The task assignments were as follows:

RC & **SIGGEN**: Tung

New languages: Thomas

Although the tasks were divided among team members, with the assigned person taking the lead on each task and contributing most of the code, the utilisation of Git empowered other team members to contribute seamlessly. Additionally, GitHub's user-friendly graphical interface facilitated effective issue tracking and management of feature requests, enhancing our overall repository management process.

4 My contribution

4.1 The legacy - old skeleton code

I was tasked with developing the `Names`, `Scanner`, and `Parser` classes. To streamline the development process, I initially created a logic-focused version of the classes and later added additional features to handle errors effectively.

The `Names` class acts as a repository for all the words used within a definition file, offering methods for manipulating them. It utilizes two dictionaries to establish a 1-to-1 relationship between word strings and their corresponding IDs. This design choice allows for queries to be executed in $\mathcal{O}(1)$ amortized time, significantly enhancing the software's performance.

The `Scanner` class makes use of Python's context manager and generator features to create a pipeline that yields symbols to the `Parser` class. Leveraging the context manager ensures that the program can perform cleanup operations and terminate gracefully (e.g., closing definition files) in the event of interruptions. The generator approach eliminates the need to read the entire definition file at once.

Both the `Scanner` and `Parser` classes employ regular expressions to check for syntax errors. This approach greatly enhances code readability compared to using extensive if-else blocks, making maintenance tasks more manageable.

During the project's maintenance phase, my responsibilities included implementing two new devices: the signal generator and the RC. The signal generator operates similarly to a clock but generates a periodic arbitrary waveform instead of a square wave. The waveform is user-defined and can have variable length and shape. In the definition file, users specify the waveform as an argument for the signal generator device. On the other hand, the RC device functions similarly to a clock but generates a HIGH signal for a specified number of cycles, which is specified in the definition file.

4.2 New code

The previous skeleton code appears to resemble C++ code that has been translated into Python. It lacks the utilization of Python features that could simplify the development process, particularly the use of `raise` and `catch` statements. The current implementation relies on integer error codes to communicate errors back to the user, which can be cumbersome to debug. Additionally, the program's structure, including the classes `Name`, `Scanner`, `Parser`, `Network`, and `Device`, while suitable for a logic simulator application, cannot be easily imported and used as a standalone package by other developers. Modifying or adding new custom devices requires modifying the source code of `Network` and `Device`.

To address these issues, I have introduced a new program structure. For the core simulation engine, only two base classes are now required: `Device` and `Connection`. The `Circuit` class serves as the base

class for all devices in the program, including clocks, switches, gates, and more, while the `Connection` class facilitates the wiring between these devices.

All `Device` objects have an `update` method that is automatically called when the object's internal properties change. This change propagates throughout the network. `Connection` objects have a `connect` method that can be used to establish connections between two devices.

For detailed implementation information, please refer to the Git repository.

5 Test procedures

During our development process, we followed industry-standard practices by incorporating two main types of testing: unit testing and system testing.

For unit testing, we made use of the popular testing framework Pytest. We created test cases that invoked the classes and methods, comparing the actual outputs with the expected results. This approach allowed us to validate the functionality of individual code units.

For system testing, we executed the terminal version of the application with the "-c" option. We conducted tests using both functional and non-functional files, encompassing all devices and their various configurations. This comprehensive testing approach ensured that the application behaved as intended and met the desired specifications.

To adhere to best practices, we adopted a collaborative approach where each team member was responsible for writing tests for specific sections of the code. This practice helped ensure a thorough and unbiased testing process. During the usability testing phase, we observed the demonstrator's interaction with the software, particularly during the software testing for the second interim report. Based on the feedback received, we made necessary adjustments and implemented suggested changes. These included improving error messages and incorporating an escape mechanism for commenting the entire file, ultimately enhancing the overall user experience.

Furthermore, during the maintenance phase, I conducted tests to verify the functionality of the signal generator. This involved connecting it to the D-Type clock port and monitoring the output changes at the expected intervals. Through this test, we assessed whether the device's output met the client's requirements in terms of accurately triggering the rising edge signal, thereby providing an improved version of the clock.

6 Conclusions

Our utilisation of Git for collaboration, version control, bug tracking, and feature requests proved highly effective. This enabled us to dedicate more time to coding and minimised the effort spent on code management especially when we work at different time. Thanks to our comprehensive testing approach, encompassing both unit and system tests, we swiftly detected and resolved any bugs when developing new features and it enhances the ability to scale the project and expand the team. This streamlined our development process, allowing us to focus on writing code rather than troubleshooting.

If given additional time, there are several areas where I would strive to enhance our logic simulator:

- Expand the device library to include more devices
- Allow creation of custom devices by users
- Display the circuit in the GUI
- Save changes made to a circuit to the definition file using the GUI

7 Appendix A: Example Files

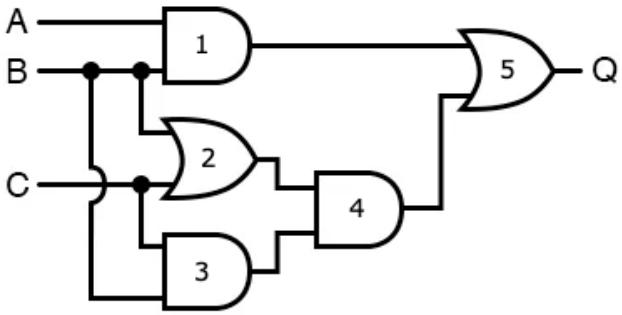

Figure 1: Example Logic Circuit 1

```
# Hardware
SW A(0), B(0), C(0)
AND g1(2), g3(2), g4(2)
OR g2(2) g5(2)

# Connections
A = g1.I1
B = g1.I2
B = g2.I1
B = g3.I2
C = g2.I2
C = g3.I1
g1 = g5.I1
g2 = g4.I1
g3 = g4.I2
g4 = g5.I2

# Monitoring
MONITOR g5
```

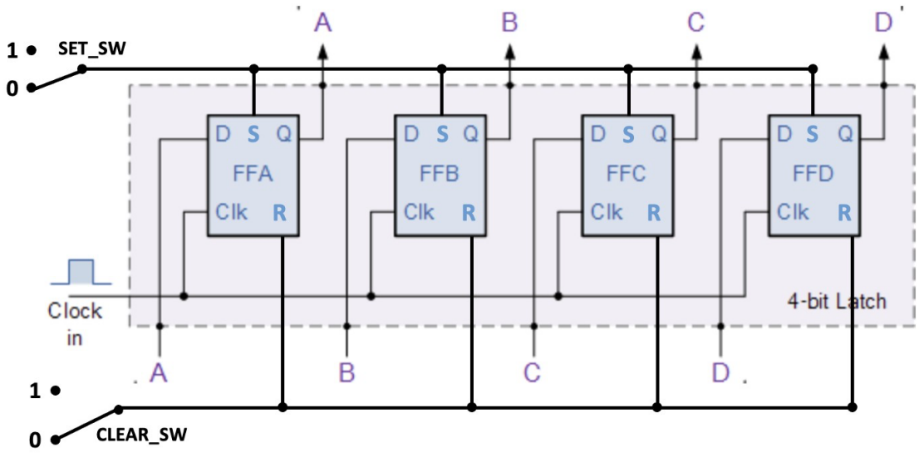

Figure 2: Example Logic Circuit 2

```
# Hardware
SW set_sw(0), clr_sw(0), A(0), B(1), C(0), D(1)
CLK clock(20)
DTYPE ffa, ffb, ffc, ffd

# Connections
clock = ffa.CLK
clock = ffb.CLK
clock = ffd.CLK
clock = ffd.CLK
set_sw = ffa.SET
set_sw = ffb.SET
set_sw = ffb.SET
set_sw = ffb.SET
clr_sw = ffa.CLEAR
clr_sw = ffb.CLEAR
clr_sw = ffb.CLEAR
clr_sw = ffb.CLEAR

A = ffa.DATA
B = ffb.DATA
C = ffc.DATA
D = ffd.DATA

# Monitoring
MONITOR ffa.Q, ffb.Q, ffc.Q, ffd.Q
```

8 Appendix B: EBNF

```
program = { block } , EOF ;

block = declarations , connections , [ monitors ] , { declartions , connections , [ monitors ] } ;

declarations = declaration , [ comment ] , EOL , { declaration , [ comment ] , EOL } ;
connections = connection , [ comment ] , EOL , { declaration , [ comment ] , EOL } ;
monitors = monitor, [ comment ] , EOL , { monitor , [ comment ] , EOL } ;

comment = "#" , { character } ;

connection =  port , " = " , port , { " = " , port };

port = identifier , "." , ( input | output ) ;

input = ( "i" | "I" ) , natural number
      | ( "d" | "D" ) , ( "a" | "A" ) , ( "t" | "T" ) , ( "a" | "A" )
      | ( "c" | "C" ) , ( "l" | "L" ) , ( "k" | "K" )
      | ( "s" | "S" ) , ( "e" | "E" ) , ( "t" | "T" )
      | ( "c" | "C" ) , ( "l" | "L" ) , ( "e" | "E" ) , ( "a" | "A" ) , ( "r" | "R" ) ;

output = ( "o" | "O" ) , natural number
       | ( "q" | "Q" )
       | ( "q" | "Q" ) , ( "b" | "B" ) , ( "a" | "A" ) , ( "r" | "R" ) ;

monitor = ( "m" | "M" ) , ( "o" | "O" ) , ( "n" | "N" ) , ( "i" | "I" ) , ( "t" | "T" ) ,
          ( "o" | "O" ) , ( "r" | "R" ) , port , { "," port } ;

declaration = clock declaration | switch declaration | gate declaration | dtype declaration
            | xor declaration ;

clock declaration = ( "c" | "C" ) , ( "l" | "L" ) , ( "k" | "K" ) , clock , { "," , clock } ;

clock = identifier , "(" , natural number , ")" ;

switch declaration = ( "s" | "S" ) , ( "w" | "W" ) , switch , { "," , switch } ;

switch = identifier , "(" , ( "1" | "0" ) , ")" ;

gate declaration = gate type , gate , { "," , gate } ;

gate = identifier , "(" , natural number , ")" ;

gate type = ( "a" | "A" ) , ( "n" | "N" ) , ( "d" | "D" )
          | ( "o" | "O" ) , ( "r" | "R" )
          | ( "n" | "N" ) , ( "a" | "A" ) , ( "n" | "N" ) , ( "d" | "D" )
          | ( "n" | "N" ) , ( "o" | "O" ) , ( "r" | "R" ) ;

dtype declaration = ( "d" | "D" ) , ( "t" | "T" ) , ( "y" | "Y" ) , ( "p" | "P" ) , ( "e" | "E" ) ,
     identifier ;

xor declaration = ( "x" | "X" ) , ( "o" | "O" ) , ( "r" | "R" ) , identifier ;

resevered word = ( "c" | "C" ) , ( "l" | "L" ) , ( "k" | "K" )
               | ( "s" | "S" ) , ( "w" | "W" )
               | gate type
               | ( "d" | "D" ) , ( "t" | "T" ) , ( "y" | "Y" ) , ( "p" | "P" ) , ( "e" | "E" )
               | ( "x" | "X" ) , ( "o" | "O" ) , ( "r" | "R" ) ;

identifier = ( ( letter | "_" ) , { letter | digit | "_" } ) - reserved word ;

character = digit | letter | punctuation | "_" | " " ;

punctuation = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">" | "'" | `"' | "=" | "|" | "." | ","
       | ";" | "-" | "+" | "*" | "?" ;

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" ;

natural number = ( digit - "0" ) , { digit } ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
EOL = "\n" | "\r" | "\r\n" | "\v" | "\f" | "\x1c" | "x1d" | "\x1e" | "\x85" | "\u2028" | "\u2029" ;
```

9 Appendix C: User Manual

Appendix C requires access to GUI which unfortunately isn't available at the moment as Thomas is currently ill.

10 Appendix D: File Listings

Thomas is currently ill so we haven't finalise our file listings. I'm not sure what to do in this situation.