Logic Circuits Simulator Report

Ahmed Bamadhaf 900205060
John Ebrahim
Ziad Gaballah 900221960

Introduction

This report will discuss the design, development, testing and the challenges of the event-driven logic simulator. This project creates an event-driven logic circuit simulator that meets specific requirements outlined within the scope of digital circuit simulation, such as inputs, outputs and the time delays. By accepting inputs in the form of a Verilog file (.v), and stimuli file (.stim) which sets the values of the inputs accordingly. The simulator simulates what happens when the input changes from its initial state, which in this simulator is assumed to be 0. This includes the time delay changes and the output changes. This report will dive deeper into the implementation and functionality of this simulator.

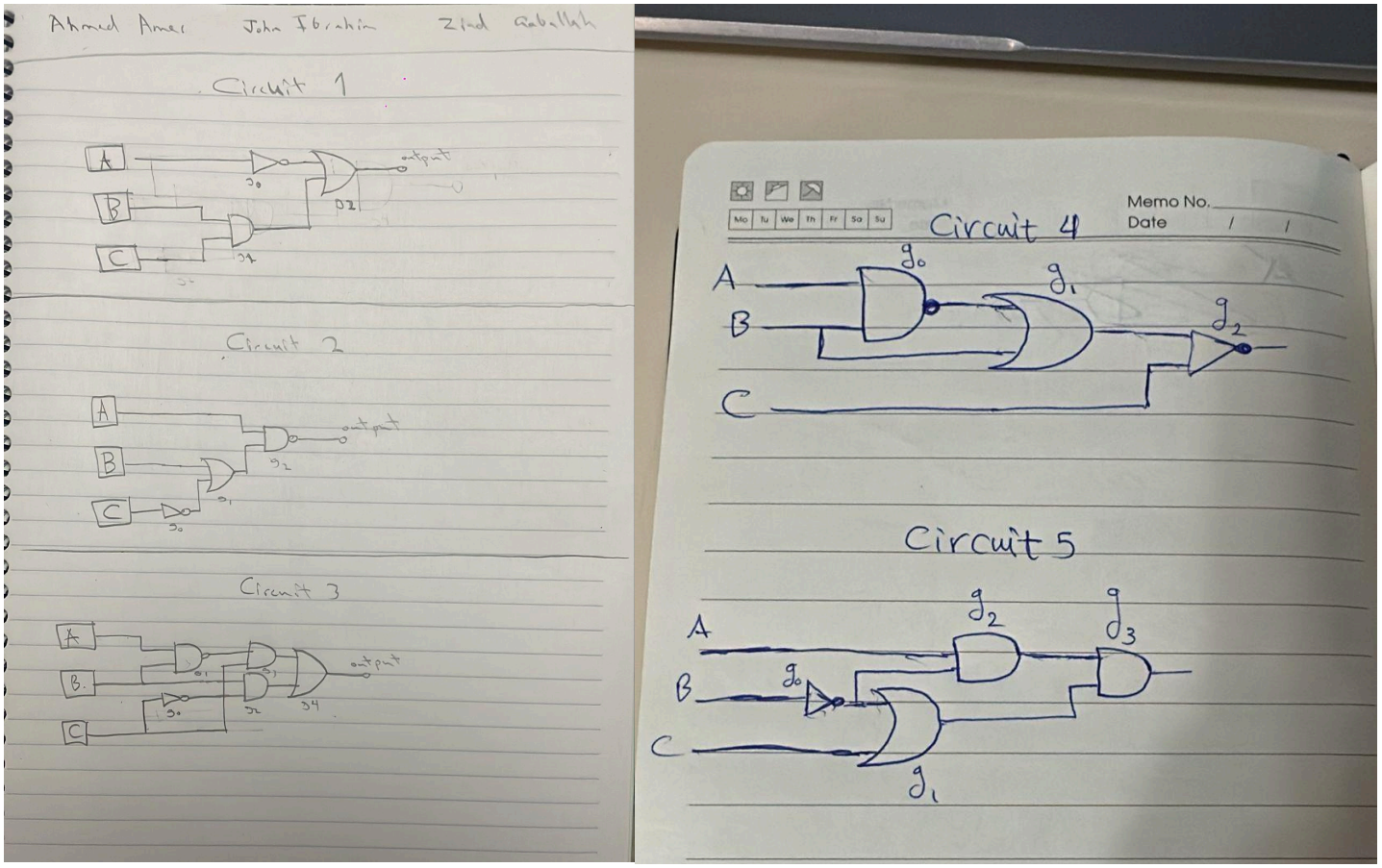
Used Data Structures and Algorithms

- File Parsing :
  - The code reads the Verilog file of the circuits so it can identify the inputs, outputs, wires, and gates. Then it eventually a nested Python dictionary that has all of that. What happens is that each line is split into components of Python lists and then they are split again by spaces. Searching for keywords such as inputs to store the relevant data and excluding data like comments.
  - The code then reads the stimuli file with inputs such as the ones in the .v file and their corresponding time stamp values and stores its data in a very similar way.

- Logic :
  - After parsing, all the data structures are passed to the “simulation” function which holds the main logic for the project. Since the project is an event-driven simulator, the process of listening for events here happens when we move to a different timestamp in the dictionary holding the stimuli file parsed data. At each different timestamp, it reads the new inputs, reflects the corresponding changes in the wires and output as indicated by each logic gate, and then adds it to the result dictionary which holds a value for each wire.
- Writing into the output file
  - The final “write” function writes the dictionary that we got from the simulation function and into a simuli file.

Testing

We designed a comprehensive set of test cases that encompass various logic gates and input combinations to ensure thorough coverage of the simulation capabilities. We made 5 test cases with different components. All of which are 3 input-based circuits that use a variety of gates namely AND, OR, NOR, NAND, and NOT gates to produce a single output. These circuits cover all the gates that can be done in the simulation except for the XOR, XNOR, and BUF gates.



Each test cases was executed using a simulation code with its output being stored in a .sim file. Each test case has a Verilog file along with a stimuli file (.sim) that modifies the inputs at certain times.

A note to take into consideration is that we assumed that each gate has no delay. Therefore, the event happens each time we jump into a different time stamp in the stimuli file, meaning all wires are updated and output at that particular moment

Challenges

Throughout the project, we grappled with several challenges that influenced our simulation's design and implementation.

The event-driven project was an exposure to a new idea where we needed to learn new technics as parsing in the code. This took a long time as we needed time to understand the project accurately.

Meeting over Zoom was not a very comfortable experience since some of us had connection issues and faced so many time conflicts.

Initially, we wrote our code using C++ which all of us were familiar with. The algorithm used in the C++ code was to :

1- File Input/Output (I/O):

- The code utilizes file input/output operations to read data from external files

2- String Tokenization:

- The code tokenizes strings using the stringstream and getline() functions. It splits lines read from files into tokens based on the comma separator (,) and stores them in vectors.

3- Searching and Updating:

- Various algorithms are used for searching and updating wire values and gate outputs:
  - While(Time<10000) function was there to update the values every second orif it find a gate or reads an update from the stimuli file.
  - Conditional statements are used to determine gate types and perform corresponding operations.
  - The clear() function is used to clear the updated_wires vector at the end of each iteration.

4- Conversion: ‘ convert_to_int()’

- The function extracts integers from a string and returns them as integer values. In our code, this is used to extract integers from strings to get the timestamps and wire the values in file-read events.

The reason we changed to Python is that we could not debug the main issue in the code as it was not able to read the the updates in the wires in the circuit files. Also, there were problems reading the files sometimes that we did not how to solve.

One of the biggest challenges we faced was some members' unfamiliarity with Python. Eventually, we resolved to Python since it seemed easier to use for some members; in turn, this cost us time and it was very stressful.

Time management problems caused us problems. This was affecting us not knowing when or where to meet. However, we managed communication through WhatsApp and phone calls.

Contributions

Our team worked together closely to terminate this project. Although each member focused more on certain parts of the project, everyone pitched in ideas and helped other members when stuck on a particular issue. All 3 team members worked together to create the test circuits. Ziad made the Verilog files with its logic. Additionally, Ahmed worked on the stimuli files later fixing some problems in the format and inputs. Then John and Ziad worked on placing the stimuli files. Ahmed worked on the initial C++ code. However, due to problems that we have faced, we changed to Python where John worked on the code. Some errors appeared when the python code was complete but then we all debugged it as a team and solved the issues. We also like to thank ChatGPT for assisting us throughout the code. We used it to debug some of the code during the logic and parsing phase.