Project Report 1: Event-Driven Circuit Simulator

Yahia Kilany, Mahmoud Refaie, Seifeldin Elwan

Department of Computer Science and Engineering, The American University in Cairo

CSCE 2301: Digital Design I

Dr. Dina Mahmoud

November 2, 2024

Introduction

Digital circuit simulation plays a crucial role in verifying and analyzing circuit designs before physical hardware implementation. By creating a software-based representation of a digital circuit, engineers can explore circuit behavior under various input conditions, identify potential issues, and validate functional correctness without incurring the costs and risks of hardware testing.

Key objectives of this project include:

- **Verilog Parsing:** Develop a parsing mechanism to interpret Verilog files, converting digital circuit designs into a structured format usable by the simulator. This includes identifying components, connections, and behavior based on Verilog syntax.
- **Stimuli Parsing:** Implement a parsing system for stimuli files (.stim) that define test inputs over time.
- **Event-Driven Circuit Simulation:** Build an event-driven logic simulator to process circuit designs, updating the state of the circuit based on signal changes, stated in the stimuli file, over time.
- **Waveform Generation:** Generate waveform outputs representing circuit behavior over time. These waveforms will visually display signal propagation, timing relationships, and responses to stimuli, helping identify issues in logic or timing within the circuit.

The project is built with a strong emphasis on modularity and clear organization. Each class and function is divided into header (.h) and implementation (.cpp) files to enhance readability and scalability. This modular design also aids in testing individual components, supporting more manageable and isolated debugging. This report details the general system design, data structures and algorithms used, test cases, and the challenges encountered throughout the development process.

Classes and Data Structures

To support an efficient and accurate simulation, several data structures are implemented to manage the components and dynamics of the circuit. Each structure plays a vital role in organizing data and managing connections.

The primary classes include the Wire struct and the Gates class, which represent fundamental components of a digital circuit. Additionally, more abstract concepts are encapsulated within the Event class, which models changes in circuit states, and the Circuit class, which serves as the central repository for the circuit's structure and simulation logic. Together, these data structures ensure that the simulator can effectively model the complexities of digital circuits while providing a framework for event-driven simulation.

1- Wire struct A struct that represents the basic block of the program. It simulates the functionality of a wire in a digital circuit.

- **Attributes**

  ***Name*:** which includes the name of the wire.

  ***Type*:** describes the function of the wire (input, output, wire).

  ***Value*:** The boolean value carried by the wire.

  endGates: A vector that contains pointers to the *Gates* class that the wire connects to.

2- Gates Class

A class that represents the primitive logic gates (And, Or, Nor, etc.).

- **Attributes**

  Inputs: A vector of pointers to the *Wire* struct that represents the fan-ins of the gate. It

  allows for a single to have more than 2 inputs.

  Output: A pointer to the wire struct that represents the fan out of the gate.

  Delay: an integer that represents the total delay of the gate in picoseconds.

  Type: A string describing the type of the gate (and, or, nand, nor, etc.).
- **Methods**

  evaluate(): A function that loops over the *inputs* vector of the gate applying the appropriate operation with respect to an event object according to the information saved in the *Type* variable and returns an event object representing the change of the output of the gate.

3- Event Class

Represents change in the circuit, encapsulating the time, affected wire, and new state.

- **Attributes**

  time: The time at which the event occurs.
  wireName: The name of the wire that changes state.
  value: The new boolean value of the wire.

4- Circuit Class

A central class that holds the entire circuit's structure, including collections of wires and gates, and handles the algorithm of the event simulation.

- **Attributes**

  Wires: An unordered_map of pointers to the *Wire* struct linking the names parsed from the verilog file to their respective values
  eventQueue: A priority queue of *Event* objects organizing ascendingly according to
  their timestamps.
  Gates: A vector of *Gate* pointers of the gates that exist in the circuit
  moduleName: Name of the module being simulated

- **Methods**

  Simulate(): A function that implements the event-driven simulation algorithm.

As we can see the simulator employs vector, priority_queue, and unordered_map data structures to effectively manage circuit elements and events. Vectors are utilized to store lists of gates and their inputs, allowing for efficient iteration and dynamic resizing as components are added during parsing. The priority queue organizes events by timestamp, ensuring accurate processing of signal changes in the correct temporal order, which is crucial for simulating digital circuits. Meanwhile, the unordered map links wire names to their corresponding Wire objects, enabling quick access and updates during event processing. Together, these data structures provide the necessary efficiency and organization to handle the complexities of digital circuit simulation.

Functions and Algorithms

In the development of the event-driven logic circuit simulator, a series of key functions and algorithms were implemented to facilitate the effective parsing of circuit definitions and stimulus files, as well as to drive the core simulation process. These functions serve to bridge the gap between the textual representations of circuit designs and their dynamic behavior during simulation.

The program starts by prompting the user to enter a file path to the Verilog file and then asks for the file path of the stimulus file. It then creates an output directory where the output .sim file will be generated. If no input is provided, the program defaults to looping over the .v and .stim files in the Tests directory.

The primary parsing functions, ParseVerilog(filename) and ParseStim(filename), are designed to read and interpret the respective .v and .stim files, extracting crucial information that defines the circuit's structure and its operational stimuli. The core simulation function, Simulate(), orchestrates the event-driven simulation algorithm, managing the flow of events and ensuring that changes propagate accurately throughout the circuit. This section delves into the specifics of these functions, detailing their roles, mechanisms, and the overall architecture that underpins the simulator's functionality.

ParseVerilog(filename)

The function is responsible for reading a Verilog file (with a .v extension) and extracting relevant information to construct the corresponding circuit components in the simulator. It begins by opening the specified Verilog file, checking for successful access, and handling errors that may arise, such as file not found or permission issues. Using a string stream object, the function reads the file line by line, enabling efficient parsing and manipulation of text. For each line, it applies a series of parsing rules to identify and extract components such as module names, input/output definitions, gate instantiations, and connections. Based on this extracted information, it dynamically creates instances of the *Wire* and *Gates* classes, linking them according to the logic specified in the Verilog file simultaneously. Throughout the parsing process, the function includes error-checking mechanisms to ensure the input file adheres to expected formats and conventions, logging errors for any discrepancies detected. After successfully parsing the Verilog file, the function populates A *Circuit* object with the wires and gates extracted from the Verilog file. Finally, it returns the populated Circuit object

ParseStim(filename)

The parseStimFile function is responsible for reading a stimulus file (with a .stim extension) and extracting events that dictate how the circuit's inputs change over time during the simulation. It starts by opening the specified stimulus file, checking for successful access, and managing potential errors such as file not found or access permission issues. By utilizing a string stream object, the function processes the file line by line, which allows for efficient parsing and manipulation of the input text. For each line of the stimulus file, the function implements parsing rules to identify relevant information such as the time of the event, the name of the wire affected, and the new value to be assigned. It begins by removing unnecessary characters, such as comments or whitespace, to ensure a clean extraction of data. Each extracted event is encapsulated in an Event object, which includes the time, wire name, and new value. As the function processes each line, it constructs a vector of Event objects that represent the changes to the circuit's inputs over time returning the vector at the very end.

Simulate()

The core component of the event-driven simulation algorithm. It is a member function of the *Circuit* class responsible for processing the events stored in the event queue and updating the state of the circuit accordingly. It begins by initializing a queue specifically for gates that will be affected by the events being processed. The function opens an output file ,named after the *moduleName* with the extension ".sim", to log simulation results, ensuring that it captures the circuit's behavior over time. As long as there are events in the event queue, the function retrieves the next event, logs its details, and updates the corresponding wire's state based on the event's specifications. For each wire affected by the event, the function then identifies all gates connected to that wire, adding them to the gate queue for evaluation. The gates are processed in a first-in, first-out manner, with the evaluate method called on each gate to apply the appropriate logic operation based on its inputs. This evaluation results in the generation of new events, which are then pushed back into the event queue to ensure that any changes propagate through the circuit. The function continues this loop until the event queue is empty, signifying that all scheduled events have been processed. By effectively managing the flow of events and updating wire states in response to changes.

Testing and Challenges

To test the simulator, five example circuits were constructed to verify the program's functionality and correctness. Each circuit was designed to represent various configurations and scenarios typical in digital logic, providing a comprehensive assessment of the simulator's capabilities.

The main challenges encountered during development included the parsing of the complex syntax inherent in Verilog files. The inflexibility of C++ for parsing tasks made this particularly challenging. After research, we opted to utilize the string stream class, which significantly simplified the parsing process, allowing for efficient manipulation of the file's contents.

Another challenge that arose involved the repetition of the same event when two input variables changed simultaneously, resulting in the output of the gate remaining constant. This led to the gate being evaluated twice as it was pushed to the gate queue, causing redundant outputs. To address this bug, we implemented a check to compare the last two events before adding new ones to the output, ensuring that only unique events were processed and logged. This solution effectively eliminated the problem of duplicate outputs, enhancing the overall reliability of the simulator.

The final challenge was the creation of the *waveform.py* file which is a Python script that draws the waveform of the output. Since we did not have experience with Python, it was challenging to utilize the language. Thus we had to look online for solutions and use AI software for help in generating the file which required heavy edits until we reached a usable file.(See Appendix A for screenshots)

Members' Contribution

Yahia:

- Developed the *Wire* struct
- Developed the *Circuit* class
- Developed the Event class
- DevelopedThe waveform.py script

Mahmoud:

- Developed the *ParseStim()* function and the *ParseVerilog()* function with the help of Yahia.
- Helped with the debugging processes. Fixed some logic errors within the above-mentioned functions.

- Contributed in writing the project report.
- Implemented three test circuits, Including their .stim files

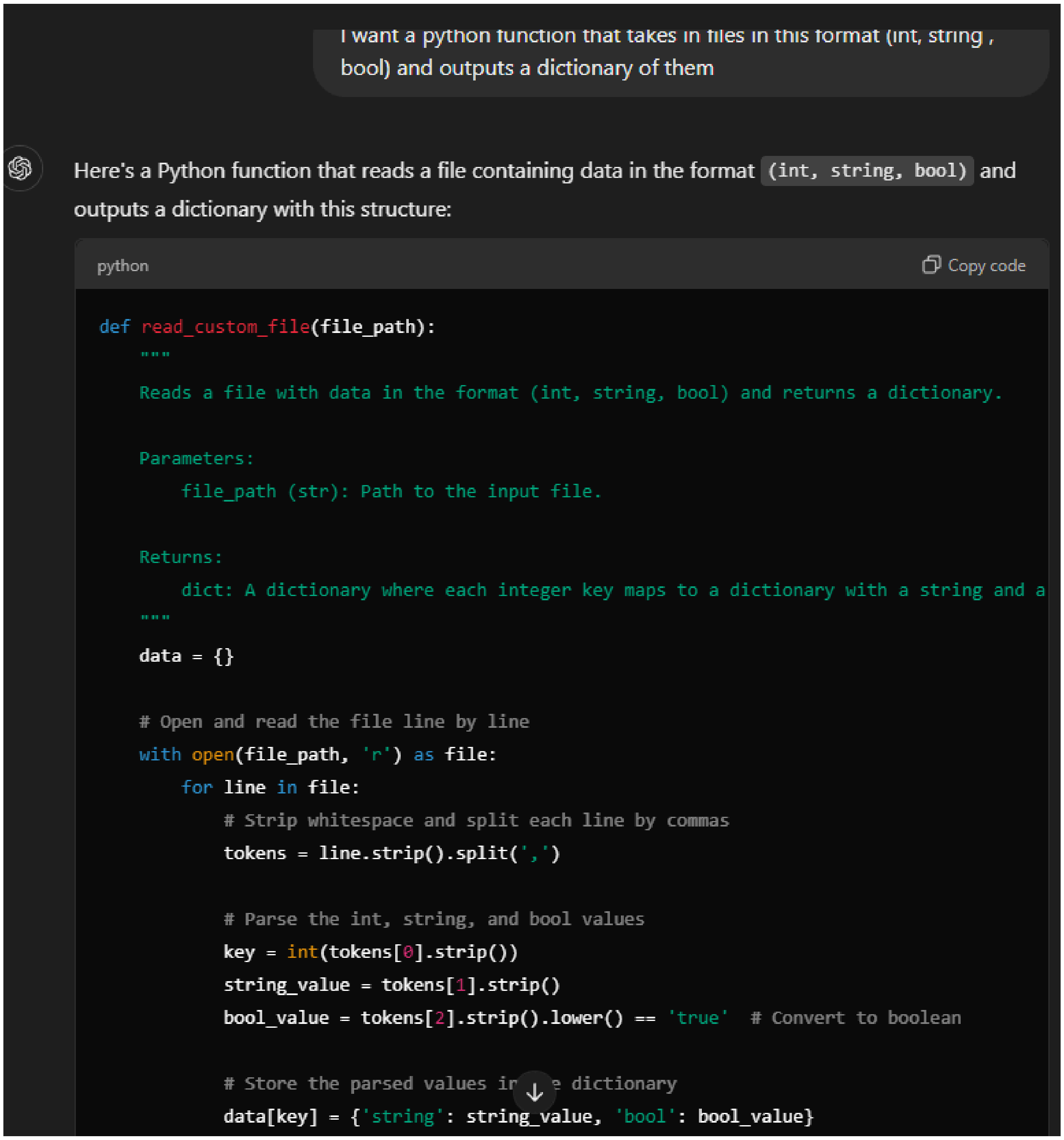
Seifeldin :

- Developed *Gates* Class with the help of Yehia
- Helped with the Debugging Process
- Wrote the Project report with the help of Mahmoud and Yehia
- Helped develop waveform.py

Summary

This project integrates multiple components to create a functional digital circuit simulator, with features for Verilog parsing, event-driven simulation, and testing. Through structured data representations and event-driven logic simulation, the system simulates and visualizes complex digital circuits. Automation of testing and waveform generation ensures circuits behave as expected under various stimuli, though challenges around parsing and timing precision demand a rigorous design and testing approach.

Appendix A

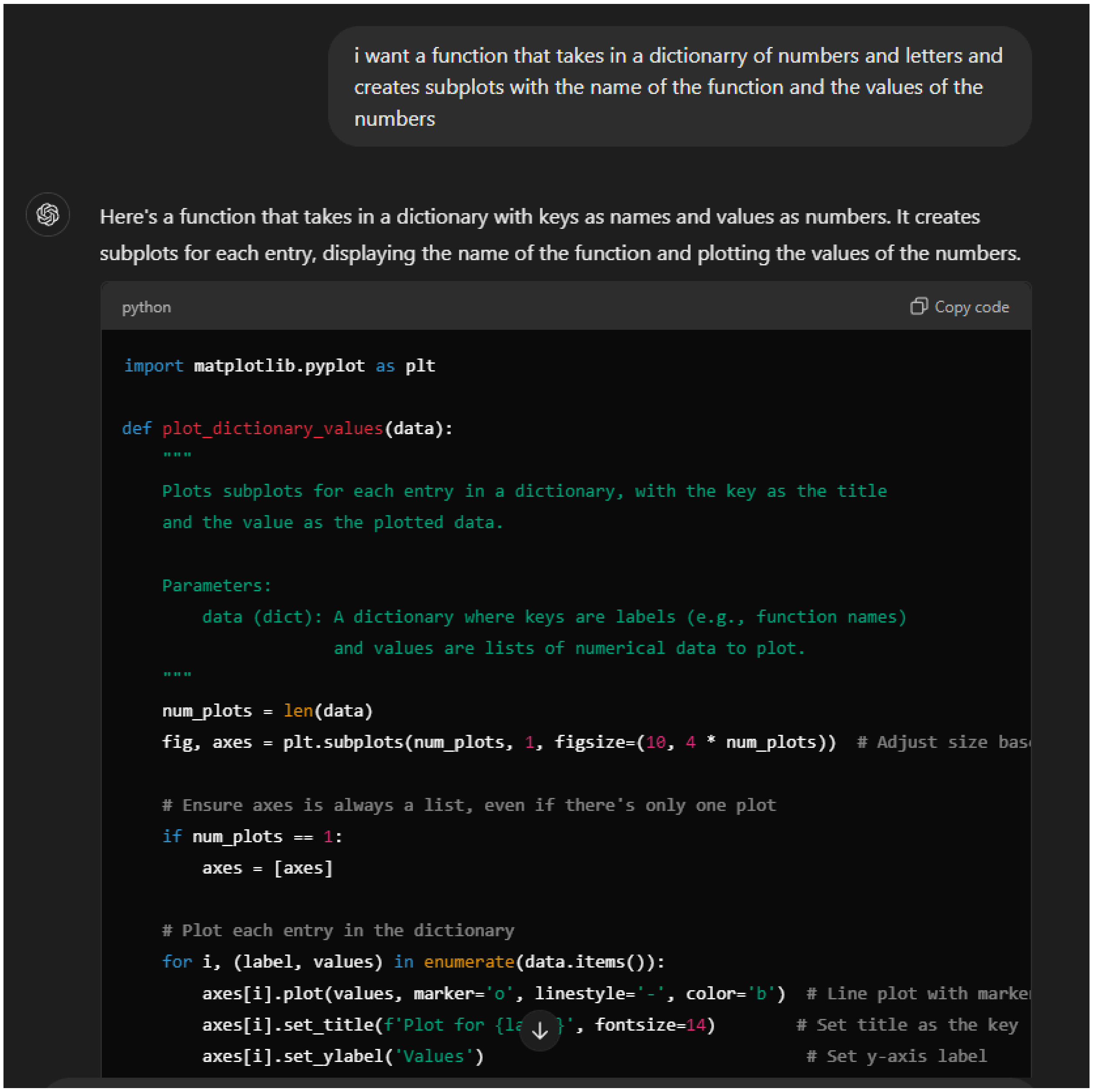


I want a python function that takes in files in this format (int, string , bool) and outputs a dictionary of them

Here's a Python function that reads a file containing data in the format `(int, string, bool)` and outputs a dictionary with this structure:

python Copy code

```python
def read_custom_file(file_path):
    """
    Reads a file with data in the format (int, string, bool) and returns a dictionary.

    Parameters:
        file_path (str): Path to the input file.

    Returns:
        dict: A dictionary where each integer key maps to a dictionary with a string and a
    """
    data = {}

    # Open and read the file line by line
    with open(file_path, 'r') as file:
        for line in file:
            # Strip whitespace and split each line by commas
            tokens = line.strip().split(',')

            # Parse the int, string, and bool values
            key = int(tokens[0].strip())
            string_value = tokens[1].strip()
            bool_value = tokens[2].strip().lower() == 'true'  # Convert to boolean

            # Store the parsed values ir   e dictionary
            data[key] = {'string': string_value, 'bool': bool_value}
```



i want a function that takes in a dictionarry of numbers and letters and creates subplots with the name of the function and the values of the numbers

Here's a function that takes in a dictionary with keys as names and values as numbers. It creates subplots for each entry, displaying the name of the function and plotting the values of the numbers.

```python
import matplotlib.pyplot as plt

def plot_dictionary_values(data):
    """
    Plots subplots for each entry in a dictionary, with the key as the title
    and the value as the plotted data.

    Parameters:
        data (dict): A dictionary where keys are labels (e.g., function names)
                     and values are lists of numerical data to plot.
    """
    num_plots = len(data)
    fig, axes = plt.subplots(num_plots, 1, figsize=(10, 4 * num_plots))  # Adjust size bas

    # Ensure axes is always a list, even if there's only one plot
    if num_plots == 1:
        axes = [axes]

    # Plot each entry in the dictionary
    for i, (label, values) in enumerate(data.items()):
        axes[i].plot(values, marker='o', linestyle='-', color='b')  # Line plot with marke
        axes[i].set_title(f'Plot for {la    }', fontsize=14)        # Set title as the key
        axes[i].set_ylabel('Values')                                # Set y-axis label
```