Ramy Shehata- 900222862
Mohammed Khaled- 900225303
Omar Ganna- 900222646

Logic Event Simulator Report

Digital Design 1- Project 1- Spring 2024

1- 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 circuit file (.cir), and stimuli file (.stim file) and taking the details of each gate, such as the number of inputs and the delays from the library file (.lib), 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 provide details regarding the implementation of this simulator and how it works on a deeper level.

2- Program Design

The Digital Logic Circuit Simulator is a tool designed to simulate the behavior of digital logic circuits. The primary function of the simulator is to process input files containing information about the circuit components, such as gates and wires, as well as input stimuli. The simulator computes the logic states of the wires and produces output that reflects the behavior of the circuit over time.

2.1- Parsing input files:

The simulator reads input files containing descriptions of the circuit components and stimuli. These files include information about gate types, input and output connections, delay times, and input stimuli with timestamps.

1-Library File Parsing: "parseLibraryFile"

This parsing task involves reading a library file that contains descriptions of different logic gate components.

Each line in the library file represents a single gate component and includes details such as the gate name, number of inputs, output expression, and delay.

Here, we created a class named Gates, that stores all the information from each line into their own separate variables. Such as the gateName, numOfInputs, outputExpression, timeDelay.

The simulator parses this information to create a collection of gate objects, each representing a specific logic gate component with its associated properties. This is done using the fstream library for C++.

2-Stimuli File Parsing: "parseStimuliFile"

The simulator also parses a stimuli file that contains input signals with associated timestamps. Each line in the stimuli file represents a single input stimulus and includes details such as the timestamp, input name, and logic value.

Here, we also created a class named Stimuli, that stores all the information from each line into their own separate variables. Such as the inputName, logicValue, and timeDelay. This is also done using the fstream library for C++. By parsing this file, the simulator extracts the input signals along with their corresponding timestamps, which are crucial for determining the timing behavior of the circuit during simulation.

3-Circuit File Parsing: "parseCircuitFile"

In addition to the library and stimuli files, the simulator parses a circuit file that describes the interconnections between gates and wires in the circuit. The parseCircuitFile function plays a crucial role in the simulation process by extracting information from the circuit file. The circuit file includes sections for defining input and output connections, as well as the components and their connections. By parsing this file, the simulator constructs a data structure which is a vector<pair<string, vector<wire>>> that represents the circuit layout, including the connections between gates and wires, the inputs and outputs and the names of the gates. It also correlates all the stimuli with the inputs .In addition, it also checks if the gate names are actually in the lib file, otherwise it produces an error. This parsing task is essential for simulating the behavior of the circuit, as it provides the necessary information about the circuit topology and component interactions.

All parsing makes sure that there is coverage for whitespaces and commas in the lines to make sure that the functions accurately take all the information from their respective files.

2.2- Logic computation:

Once the input files are parsed, the simulator computes the logic states of the wires in the circuit over time. It applies the logic operations defined by the gate components to determine the output states based on the input states. This is the main heart of our code which is done by the function **computingLogic.** This is the function that does the simulation after it has received all the inputs and gates information.

1-Gate Evaluation:

The function iterates through each gate in the circuit using a loop over the ioComponents vector. Within each iteration, it identifies the gate type based on the string identifier (it->first). The identification is done using the gate name, it then understands the output expression using an evaluation of the infix expression given in the lib file and determines how each gate is handled specifically. For each gate type, specific logic operations are performed to compute the output state based on the number of inputs and input states. It also handles major errors such as if the logic gate is not defined.

2-Input/Output Handling:

Input and output wires are managed within the ioComponents vector, where each gate is associated with its input and output wires. The function retrieves the state of input wires from the ioComponents vector and applies logic operations to compute the output state. Output states are updated within the ioComponents vector, reflecting changes in the circuit's behavior. It also evaluates where the inputs and outputs are in the gates and handles logic errors if there are invalid inputs and outputs.

3-Delay Handling:

Delays associated with gates and wires are considered when updating the output states.

The delay value is retrieved from the gate's library component (libComponents) and added to the computation time. Output states are updated with the calculated delay to ensure that logic state transitions occur at the correct time intervals. This is done by taking the intervals between each input in the stimuli file and This is done by taking the intervals between each input in the stimuli file and mapping them to the corresponding gates in the circuit. When a gate's output state is determined, its associated delay is taken into account. For instance, if a gate has a delay of 2 units, the output state change will be scheduled to occur 2 units of time after the inputs are processed. This scheduling ensures that the simulation reflects the real-world behavior of digital circuits, where signal propagation takes a finite amount of time.

4-Gate-Specific Operations:

Specific operations are performed based on the gate type identified within the ioComponents. For example, for an AND gate ("AND"), the function computes the logical AND operation between input states to determine the output state. This is done using the evaluation of the output expression that is parsed from the library file. Similar operations are performed for other gate types such as OR, NOT, NAND, NOR, and XOR, each following their respective logic functions. Our function can also handle any input number for each gate.

5-Main Loop of The Simulation

The function iterates over the ioComponents vector, which contains the components and their associated wires.For each component, it extracts the gate expression, evaluates it, and updates the output wires accordingly. It uses a stack-based evaluation approach for logical expressions, handling operators like ~ (negation), & (AND), and | (OR). After evaluating the expression, it updates the output wire's type and delay, pushing the result into F_output.

2.3- Output generation:

After computing the logic states of the wires, the simulator generates output data that represents the behavior of the circuit. This output data can be written to an output file (.sim) for each circuit for visualization. Simulation results can be visualized using Python. The simulator automatically generates events in a waveform format, enabling users to visualize changes in the circuit's behavior over time effectively. This visualization aids in understanding complex circuit dynamics and identifying patterns or anomalies in the simulation results.

2.4- Running the simulation

Our simulation can run through the terminal and using CMake and github actions. The procedures are found in the github repo on how to change the simulation files such as the.lib and .stim and so on.

3- Data structures and algorithms

Our code uses a multitude of data structures and algorithms that enhance how we simulate the circuits. These data structures include:

Structs:

wire: Represents a wire in the circuit with attributes like name, type (boolean), delay, and an initial state stack. This struct helps encapsulate the properties of wires.

Vectors:
vector<pair<string, vector<wire>>> ioComponents: Stores pairs where each pair consists of a gate type (string) and a vector of wires associated with that gate. This vector represents the input/output components of the circuit.
vector<Gates> libComponents: Stores information parsed from the library file, where each element represents a gate with properties like name, number of inputs, output expression, and delay. This vector holds the library components used in the circuit.
vector<Stimuli> stimuli: Contains stimuli parsed from a stimuli file, where each element represents a stimulus with properties like timestamp, input wire name, and logic value. This vector stores stimuli for simulation.
vector<int> timeScale: Stores timestamps extracted from stimuli, used to control the simulation time scale.
vector<string> inputs: Stores the inputs of the circuit file (the ones under the word INPUTS:)
vector<Stimuli> F_output: Stores the simulation output, consisting of stimuli with timestamps, input wire names, and logic values.
Stack:
stack<int> initial (inside wire struct): Stores the initial state of a wire, which is pushed onto the stack when the state changes during simulation.
stack<char> operators: process the operators like &, ~, | and the other bitwise operations.
stack<bool> operands: process the operands and does on it the operations to compute the logic
We also used some **algorithms** such as :
Parsing Algorithm:
parseLibraryFile: Parses the library file containing gate information and creates a vector of Gates objects.
parseStimuliFile: Parses the stimuli file and creates a vector of Stimuli objects.
parseCircuitFile: Parses the circuit file, extracting input/output components and their connections.
Utility Functions such as: getMax, getMin, removeSpaces, removeCommas, getWire, getDelay
Error Handling Functions: checkInputs, checkGates.

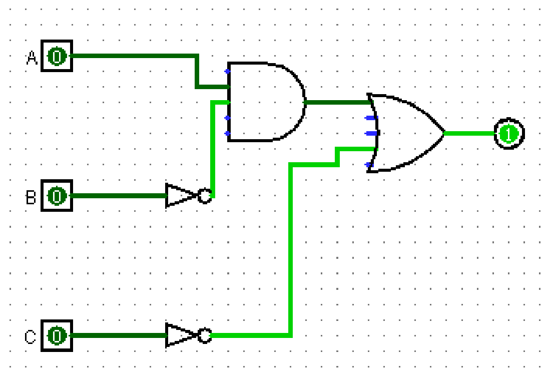
4-Testing

We designed a comprehensive set of test cases to evaluate the functionality and performance of the simulation code across a range of circuit configurations. These test cases encompass various logic gates and input combinations to ensure thorough coverage of the simulation capabilities.We created 5 test cases that have different components, 1 of which is a 4 input based circuit and the rest are 3 input-based circuits. These circuits cover all the gates that can be done in the simulation. For AND2 up to OR3 and AND3, however, our code can take more than 3 inputs.

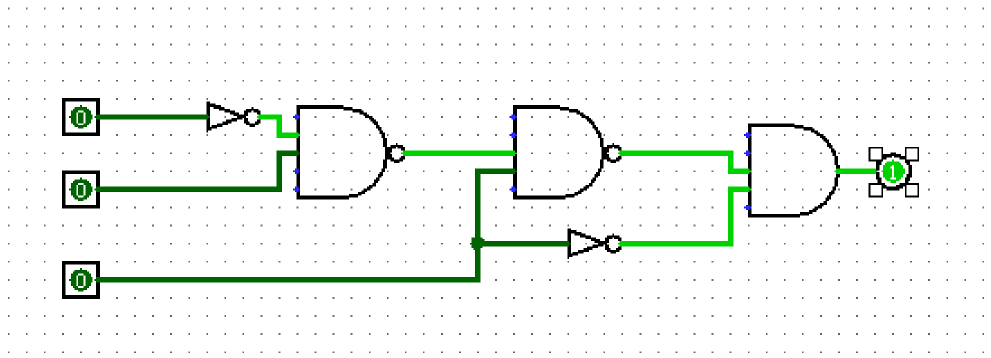
4.1 Test Cases

Now, we will show out 5 test cases images and evaluate how our simulation did the delays

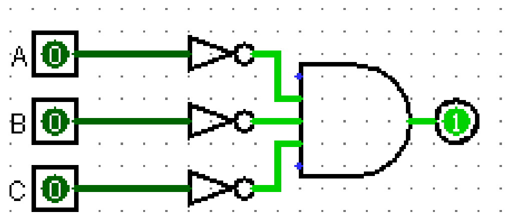
Test Case 1:



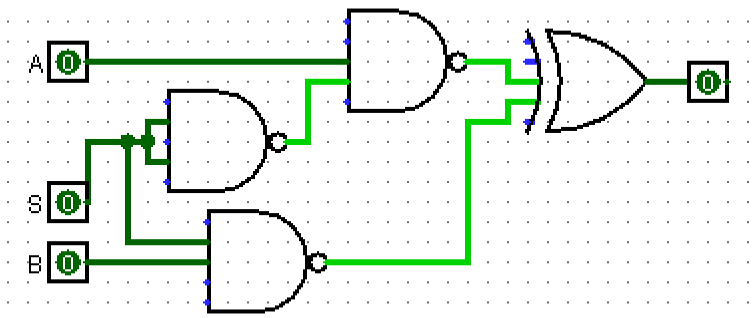
Test Case 2:



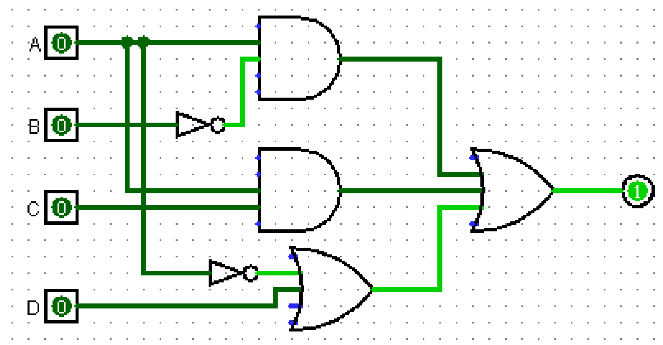
Test Case 3:



Test Case 4:



Test Case 5:



4.2 Test Simulation

Each test case was executed using the simulation code, with output generated in a sim file for their respective test circuit folder to simulate different logic scenarios. Each test case is simulated and checked with the manually done delay first to ensure the credibility of our simulator, where the simulation code was executed with the provided input files for each test case. The test environment was also configured to accurately represent the behavior of the circuits under test, ensuring reliable results.

In addition, our code is able to run from the terminal using the guide listed in our github, where the code works through the terminal of any OS.

5- Challenges

Throughout the project, we grappled with several challenges that influenced our simulation's design and implementation. Initially, aligning on the best approach was daunting, leading to numerous revisions of the design code and algorithm. Dividing tasks among us highlighted differing implementation ideas, necessitating frequent redesigns to merge concepts effectively.

Initially, reaching an agreement on how to effectively model the components was challenging. However, through collaborative brainstorming and experimentation, we introduced a struct representation for wires, coupled with a vector of pairs of strings and wire structs. Additionally, our simulation was mainly a time-based simulation. However, it was really hard to implement an event driven using that approach. We resorted to changing our implementation into one that only depends on an event-driven approach, resolving inefficiencies and improving overall accuracy.

During testing, minor discrepancies were identified in some test cases compared to the manually done tests, primarily related to timing issues and edge cases. These issues were promptly investigated, and appropriate adjustments were made to the simulation parameters to rectify the discrepancies. One of these issues was how the time delay for the outputs differ based on whether the gates change from 1 to 0 or from 0 to 1. Our code was facing major problems regarding this specific scenario. And we have solved it by making sure each gate can calculate whether it is changing from 0 to 1 or from 1 to 0. This is done in order to calculate the delay that comes out of the gate effectively, however it still doesn't work on all test cases. This resulted in us calculating the maximum delay for any inputs entering the gate.

Another major challenge was regarding the sequential circuit. We tried many different approaches in order to solve the problem regarding the processing of the output and returning it into the stim file. And we also tried matching the inputs and outputs to detect the sequential process. But sadly due to time constraint and the convolution of our implementation, it is not possible to implement a sequential circuit that works well with our implementation, however our code can basically run any variation of combinational circuit.

6- Team Contributions

Our team worked closely together on every aspect of the project. We held regular meetings to plan our approach and make decisions together. Although each member focused more on certain parts of the project, everyone contributed ideas and effort to ensure its success. For instance, Ramy Shehata mainly focused on doing the parsing of all the files and was responsible for creating the Gates and Stimuli classes. However, Mohammed Khaled assisted in the entire process when there were bugs, he solved most of the bugs Ramy encountered when he was doing that part. He also assisted heavily in creating the main logic part of the function alongside Omar Ganna. Omar was the initiator of the logic implementation we have in our code and was the first to propose and write the idea of this logic function. However, Mohammed assisted throughout the way for each function in the code from the logic and parsing to the rest of the functions. Omar Ganna also was responsible for creating the simulation on the graph by Python, although it has some problems but it does the job correctly. All 3 team members created the test circuits together and calculated the time delay for all of them together. We also like to thank ChatGPT for assisting us throughout the code. We used it to debug some of the code during the parsing phase, such as how our gate names had extra spaces and commas that were unnecessary. We also used it to debug the time delay functions we had and it was successful in identifying the problems we had. We also used ChatGPT for helping in using the #include algorithm parts such as using functions like std::find, and other vector functions. ChatGPT also was a heavy assistance in guiding us to make the run through the terminal and the use of CMake and github actions for CI/CD flow.