Logic Circuits Simulator

Overview:

The provided program is a circuit simulation tool implemented in C++, capable of reading circuit descriptions, stimuli, and component libraries to simulate the behavior of digital circuits. The simulation results are then visualized through waveform animations saved as GIFs.

Overall data structures and algorithms

In our project, we used a few data structures and algorithms to help us complete the code. The first data structure we used was a hashmap to organize the components, stimulus input, and circuit flow. We have also used a breadth-first search to evaluate the boolean expression we will write out to our simulation files. The get_stimulus() function works by reading the file containing the stimuli and storing all the stimuli in an array by order. The get_component() works by reading the library file and storing the components in a hashmap from the component name to the component. The get_circuit() function works similarly to the get_component() function, except that it reads the circuit components with their inputs and outputs. The evaluate_expression() function loops over the boolean expression for the gate to evaluate the bitwise operation. The simulate() function loops over the stimuli checking the inputs that changed values and then checks the circuit for the changed values to change the needed outputs.

Algorithms

1. get_stimulus Function:

This function reads stimuli data from a file and parses it into a vector of **Stimulus** structures.

Algorithm:

1. Open the file specified by the **filename**.
2. Read each line from the file.
3. Parse the line to extract **time_stamp_ps**, **input**, and **logic_value**.
4. Store the parsed data into a **Stimulus** structure and add it to the vector.
5. Return the vector containing all parsed stimuli.

Data Structures Used:

- **vector<Stimulus>**: Stores parsed stimuli data.

2. evaluateExpression Function:

This function evaluates Boolean expressions provided as strings.

Algorithm:

1. Parse the input expression and store it in a modified form.
2. Iterate through each character of the modified expression:
    - If the character is an operand (0 or 1), push it onto the operand stack.
    - If the character is an operator (&, |, ~), push it onto the operator stack.
    - If the character is '(', push it onto the operator stack.
    - If the character is ')', pop operators and operands from the stacks and evaluate the expression within the parentheses.
3. After processing all characters, evaluate the remaining expression using the stacks.
4. Return the final result of the expression evaluation.

Data Structures Used:

- **unordered_map<string, bool>**: Stores input values.
- **stack<bool>**: Stack for operands.
- **stack<char>**: Stack for operators.

3. get_component Function:

This function reads component information from a file and stores it in a map.

Algorithm:

1. Open the file specified by the **filename**.
2. Read each line from the file.
3. Parse the line to extract **name**, **num_inputs**, **output_expression**, and **delay_ps**.
4. Create a **Component** structure and populate it with the parsed data.
5. Store the component in the map with its name as the key.
6. Repeat steps 2-5 for each line in the file.

7. Return the map containing all parsed components.

Data Structures Used:

- **unordered_map<string, Component>**: Stores parsed component data.

4. evaluate_and_propagate Function:

This function evaluates component output values and propagates them through the circuit.

Algorithm:

1. Iterate over each component in the circuit.
2. Evaluate the output expression of the component using the **evaluateExpression** function.
3. Update the output value of the component.
4. Update input values of other components connected to the output of the current component.
5. Repeat steps 1-4 for each component in the circuit.

Data Structures Used:

- **map<string, circ_comp>**: Stores circuit components.
- **unordered_map<string, Component>**: Stores component information.

5. get_circuit Function:

This function reads circuit information from a file and stores it in maps.

Algorithm:

1. Open the circuit file specified by the **filename**.
2. Read each line from the file.
3. Parse the line to determine the section (inputs or components).
4. Parse input signals and store them in a map.
5. Parse circuit components and store them in another map.
6. Repeat steps 2-5 for each line in the file.
7. Return the maps containing parsed circuit components and input signals.

Data Structures Used:

- **map<string, circ_comp>**: Stores circuit components.
- **map<string, bool>**: Stores input signals.

6. simulate Function:

This function simulates the circuit behavior based on provided stimuli and component specifications.

Algorithm:

1. Open a file for writing simulation data.
2. Iterate over each stimulus in the provided vector.
3. For each stimulus:
    - Update input values in the circuit based on the stimulus.
    - Evaluate and propagate signals through the circuit.
    - Write simulation data to the output file.
4. Close the output file.

Data Structures Used:

- **vector<Stimulus>**: Stores stimuli data.
- **map<string, circ_comp>**: Stores circuit components.
- **unordered_map<string, Component>**: Stores component information.

7. main Function:

Algorithm:

1. Iterate through different values of **n**.
2. Construct file paths for the current iteration.
3. Parse component information and circuit description.
4. Simulate the circuit using parsed data.
5. Repeat steps 2-4 for each value of **n**.

Testing:

The program's testing procedure involves parsing input circuit descriptions, stimuli, and component libraries from files. It then simulates the circuit behavior based on provided stimuli and component specifications. The simulation data is written to an output file, and waveform animations are generated for visualization.

Functionality Testing: The program reads circuit components, evaluates expressions, and propagates signals correctly, ensuring accurate simulation results.

Integration Testing: Integration between various components like stimuli parsing, circuit evaluation, and output visualization is tested to ensure seamless operation.

Robustness Testing: The program handles invalid input formats gracefully, providing informative error messages for easier debugging.

Challenges:

The biggest challenge that we faced during this project was how are we going to model everything in the code. We went into the project not knowing what the best way to model our data flow was. However, when we started working on the code we realized that we organize the components and how they are connected in the circuits using hashmaps. Our second challenge is how we will make the coding process the data we have made read from our files and output the simulation. Nevertheless, we soon knew we should model our stimulus inputs as an ordered array and loop over the array scanning the circuit to see which outputs would change at what time.

Visualization: Generating accurate and visually appealing waveform animations posed a challenge, requiring synchronization of simulation data with the animation frames.

Contributions:

Saif: Test Circuits 1, 2,3, simulate function, evaluate expression function, get circuit component function, evaluate and propagate function, structs for component, circuit component, and stimuli. Also, the graph.py code for visualization of waveforms

Adham: Test circuits 4, and 5, get components function, get stimuli function, time graph diagrams, simulate function.