RUNNING HEAD: DD\_1 PROJECT (1)

**Logic Gates Simulator**

Adham Ali / 900223243

Omar Saqr / 900223343

Ebram Thabet / 900214496

Digital Design I (Spring 2024)

The American University in Cairo

Table of Contents

1. Introduction
2. Used data structures and Algorithms
3. Challenges
4. Testing
5. Contribution

# **Introduction:**

Digital circuits are the building blocks of all electronic devices, from calculators to computers. Our project aims to make a logic gate simulator. In this simulator, you can choose a circuit you want, and it will give you the output with its propagation delay. However, you must make three files: lib, cir and stim. The lib file is responsible for the propagation delay of the gates, as well as the expressions and names of the gates. The CIR file shows the inputs and where every gate's inputs are. The last stim file shows how the inputs vary over time. Moreover, our program can identify any unstandardised gate. How? The program reads the expression that is present in the lib file.

**Used Data Structures and Algorithms:**

1. **Unordered\_map**

Usage: This data structure is extensively used to store and access variables and their values, component functions, and propagation delays associated with wires.

Key Features: It offers average constant-time complexity for insertions, deletions, and searches. The use of hashing allows for fast access to element values using keys.

Examples in Code:

unordered\_map variables: Maps variable names to their boolean values for the logical expression evaluation.

unordered\_map input\_map: Maps input wire names to their current binary values (0 or 1).

unordered\_map component\_functions: Maps component names to the number of parameters they have.

unordered\_map delay\_map: Maps wire names to their propagation delays in picoseconds.

unordered\_map component\_library: Maps component names to their properties (Component struct instances), including the number of inputs, delay, and logical operation.

1. **Vector**

Usage: This data structure stores sequences of elements, such as inputs, components, and operands for logical expressions. It allows dynamic resizing and direct access to elements.

Key Features: Provides random access to elements, efficient insertion/removal at the end, and the ability to change size dynamically.

Examples in Code:

vector inputs: Stores the names of input wires for the circuit.

vector> components: Stores the circuit components, where each component is represented by a vector of strings detailing its configuration.

vector inp: Used temporarily to store boolean values representing the inputs to a logic gate or function.

1. **Stack**

Usage: Used in the LogicalExpressionEvaluator class to implement the algorithm for evaluating infix logical expressions.

Key Features: Provides Last-In-First-Out (LIFO) access to elements, supporting operations like push, pop, and top.

Examples in Code:

stack operands: Stores the boolean values of operands as the expression is evaluated.

stack operators: Stores the operators encountered in the expression, aiding in the proper sequencing of operations.

1. **Tuple**

Usage: To store grouped elements of different types together. It's particularly used for storing readings from a file with multiple value types.

Key Features: Allows storage of a fixed-size collection of heterogeneous elements.

Examples in Code:

vector> readings: Stores the time, wire identifier, and value for each stimulus reading from a file.

1. **Struct Component**

Usage: Represents the properties of a component in the circuit library.

Structure: Contains fields such as num\_inputs (number of inputs to the component), delay\_ps (propagation delay in picoseconds), and logic (the logical operation performed by the component).

Example in Code: Used within the component\_library unordered\_map to store detailed information about each component type available for circuit simulation.

These data structures are fundamental to the program's ability to efficiently process and evaluate logical expressions and simulate the behaviour of digital circuits based on dynamic inputs and component configurations.

**Description of some important functions:**

Library File Parsing: We load component definitions from a library file, mapping component names to their logic functions and properties. This enables the simulator to dynamically recognise and process various components.

Circuit File Parsing: The circuit file describes the circuit layout, including inputs and how components are connected. Our parsing logic constructs an internal representation of the circuit from this description used in the simulation.

Stimuli File Processing: Stimuli files define external inputs to the circuit over time. We simulate the impact of these inputs on the circuit, capturing the sequence of events as the circuit responds to changes.

The simulation core iterates through events (changes in input values), updating the circuit’s state based on the logic defined for each component. This involves calculating the output of each gate based on its current inputs and propagating these outputs through the circuit, respecting the defined delays.

**Challenges:**

One of the core challenges was accurately simulating propagation delays, a fundamental aspect of digital circuits where signals take time to travel through components. Achieving realistic simulations of these delays was critical, as they can significantly affect the behaviour and timing of circuit outputs. We tackled this by integrating delay parameters into our logic functions and developing a scheduling system that could manage and execute events based on their timing. Ensuring that this system accurately reflected a real circuit's sequential and concurrent operations required extensive testing and refinement. Moreover, the logic of the unstandardised gates makes us change the algorithm. Our previous algorithm only read the gate name from the lib file and performed the logical operation as the C++ language has operations that perform the gate output. Unfortunately, c++ has only the operation of the standard gates, but if the lib file has a gate called (wert), it will not read it, so we make the code read the expression that is in the lib file, and we cancelled the idea of the operations of c++. The last challenge was the continuous integration; we do not have the time to do it, as we knew the meaning of CI before the deadline by 2 days.

**Testing:**

We developed comprehensive tests using predefined circuits with known outcomes to ensure our simulator's accuracy. This process involved comparing the simulator’s output against expected results for various input sequences, allowing us to identify and correct discrepancies.

**Contributions:**

* Adham Ali

1. Calculating the Propagation delay algorithm with the initial state of the variables.
2. Organizing the code by adding some comments.
3. Writing the Report.
4. Debugging some errors in the algorithm of read/write files.
5. Drawing the graphs for the propagation delay for all circuits.

* Omar Saqr

1. Reading the files to use them in the code.
2. Making the logic of calculating the propagation delay, but when the inputs vary.
3. Making the logic of unstandardised gates.
4. Debugging and fixing some errors in the propagation delay.

* Ebram Thabet

1. Visualize the simulation output graphically in terms of waveforms.
2. Making the lib, cir and stim files.
3. Making the logic of calculating the propagation delay, but when the inputs vary.