

EE677

Assignment 1 Report

Prajwal Nayak

29th August, 2025

Github : <https://github.com/PrajwalNayak2506/logicSimulatorForDigitalCircuits>



Digital Logic Circuit Simulator – Project Report

1. Introduction

This project implements a command-line based Digital Logic Circuit Simulator in C++. Its primary function is to simulate combinational digital circuits defined in VHDL files by converting them into a `.bench` intermediate format, simulating test input vectors, and producing output results.

2. Project Objectives

- Convert VHDL circuit descriptions to `.bench` format.
 - Parse `.bench` files representing the circuit netlist.
 - Construct a graph-based model of the circuit.
 - Levelize circuit nodes ensuring correct gate evaluation order.
 - Simulate combinational logic on provided input vectors.
 - Output simulation results to a file.
-

3. Limitations and Conditions for the Assignment to Work (**IMPORTANT!!**)

Steps to Run the Simulator

1. Prepare the VHDL file describing your combinational logic circuit following the specified syntax rules.
2. Compile the source code using a C++17 compatible compiler, for example:

```
g++ -std=c++17 *.cpp -o Bencher
```

3. Place the compiled executable and the VHDL file in the same directory.
 4. Ensure a `testbench.txt` file is present in the directory containing one binary input vector per line corresponding to the primary input nodes.
 5. Run the simulator executable with the command:
`./Bencher`
 6. The program will:
 - Convert the VHDL file to a `.bench` format.
 - Parse the `.bench` file.
 - Simulate the circuit using the input vectors in `testbench.txt`.
 - Output results will be written to `sim_output.txt`.
-

Limitations and Conditions on the VHDL File (IMPORTANT)

The simulator makes several assumptions and has conditions for properly parsing VHDL files:

- **Strict Syntax Requirement:** The VHDL file must contain one statement per line; multi-line statements are not supported.
- **No Behavioral Constructs:** It does not support complex behavioral VHDL constructs such as `PROCESS`, `IF-THEN-ELSE`, or `CASE` statements.
- **Simple Logic Only:** Expressions with more than one operator (e.g., combined AND/OR) and operator precedence are not handled.
- **Naming Assumptions on Ports:** Output ports must start with 'o' or 'O'. Ports named otherwise (e.g., q, sum, result) may be misinterpreted as inputs.
- **Unsupported Features:** The parser does not handle `INOUT` ports, assignments from constants (e.g., `led <= '1'`), or `GENERATE` statements.
- **Gate Identification:** Only supports a fixed set of gates — `AND`, `OR`, `NAND`, `NOR`, `XOR`, `XNOR`, and `NOT`.

Because of these restrictions, the tool works best with simple combinational circuits written in a strictly formatted subset of VHDL.

4. Overall Architecture and Workflow



Conversion Phase

- The tool reads a VHDL file, extracts inputs, outputs, and gate definitions.
- Converts the information into a `.bench` file listing inputs, outputs, and gate statements.

Parsing Phase

- The `.bench` file is parsed to identify gates, inputs, outputs, and their connections.
- Data structures representing gates and their interconnections are populated.

Graph Construction

- Circuit nodes (gates) are represented as objects with properties like type, inputs, outputs, parents, and children.
- Connections between gates form a directed graph reflecting circuit signal flow.

Levelization

- Using a BFS-based approach, nodes are assigned levels.
- Levels represent topological order ensuring inputs are processed before dependents.

Simulation

- The simulator reads input test vectors from a file.
- Propagates the input signals level-wise evaluating each gate.
- Writes the outputs for each test vector to a simulation output file.

4. Key Data Structures

- Node Class: Represents a gate with attributes such as `id`, `type` (gate type), `inputs` (signal IDs), `parents`, `children`, `level` (evaluation order), and `evaluate` (current logic value).
- Vectors: Store gate IDs, types, inputs, outputs, and test vectors in ordered lists.

- Unordered Maps: Map signal names and output IDs to Node objects for efficient lookup.
- Queue: Used for BFS traversal during graph levelization.

5. File-wise Functional Explanation of the Digital Logic Circuit Simulator

bencher.cpp

This is the main driver file of the project. Its primary responsibilities are:

- Input Processing: Reads the given VHDL file (whose name is to be given as input) line-by-line.
- VHDL to .bench Conversion: Parses the VHDL syntax to extract inputs, outputs, and gate definitions, then converts this information into the `.bench` format.
- Signal Management: Maintains mapping from signal names to unique integer IDs, ensuring consistent signal references across the circuit.
- Gate Identification: Recognizes standard logic gates from VHDL assignments and constructs gate objects for conversion.
- File IO: Writes the generated `.bench` file.
- Control Flow: After conversion, it triggers parsing of the `.bench` file and initiates simulation.

This file handles initial preprocessing and acts as the entry point for the entire workflow, bridging between human-readable hardware description (VHDL) and machine-parsable netlist (`.bench`).

parser.cpp / parser.h

This module processes the `.bench` file generated by `bencher.cpp` and extracts circuit component information:

- Parsing: Reads `.bench` lines, ignoring comments and empty lines.
- Gate Extraction: Identifies gate declarations, inputs, and outputs.
- Data Structuring: Populates vectors for gate IDs, gate types, input node IDs, and output node IDs.
- Whitespace & Symbol Handling: Cleans input lines by removing spaces and parsing essential tokens like gate types and connected nodes.
- Maintains a logical representation of the circuit in basic data structures to pass on for node creation.

This stage transitions from a textual circuit description format to structured raw gate data.

nodify.cpp / nodify.h

The role of this module is to construct Node objects that represent each gate or IO element of the circuit:

- Node Creation: For each gate parsed, create a Node instance with properties like ID, type, input signal IDs, etc.
- Data Organization: Stores nodes in an unordered map keyed by output IDs for fast lookup.
- Initialization: Sets attributes such as gate type (AND, OR, NOT, etc.), inputs, and prepares nodes for graph building.
- This module transforms flat parsed data into organized, object-oriented entities reflecting circuit components.

It serves as a bridge between parsing raw gate data and creating a connected circuit graph.

graphiphy.cpp / graphiphy.h

This module builds the directed graph representation of the circuit from nodes:

- Parent-Child Linking: Uses the input-output IDs in Nodes to set up pointers:
 - `parents` link to predecessor nodes providing input signals.
 - `children` link to successor nodes receiving output signals.
- Graph Connectivity: Establishes edges that represent signal flow between nodes based on the circuit topology.
- Ensures each node knows its connected gates to propagate signals during simulation.

By the end of this phase, the circuit is structurally represented as a DAG (directed acyclic graph) suitable for topological processing.

levelizer.cpp / levelizer.h

This module assigns topological levels to nodes to determine evaluation order:

- BFS Levelization: Uses a Breadth-First Search starting from input nodes (level 0).
- Level Assignment: Ensures each node's level is greater than its parent nodes' level.
- This produces a layering of nodes, where signals are evaluated level by level in simulation to respect dependencies.
- Handles nodes with initially unknown levels and updates them dynamically.

This phase prepares the circuit graph for efficient simulation execution.

simulator.cpp / simulator.h

The core simulation logic is implemented here:

- Test Vector Input: Reads binary input vectors from `testbench.txt`.
- Level-wise Evaluation: For each vector:
 - Assign inputs to primary input nodes.
 - Traverse nodes in increasing level order, computing gate outputs based on parent node values.
- Gate Logic Implementation: Evaluates standard gates (AND, OR, NOT, NAND, NOR, XOR, XNOR) with appropriate truth tables.

- Output Collection: Gathers values of output nodes after evaluation of each test vector.
- Result Storage: Writes simulation outcomes to `sim_output.txt` for every input vector.

This module performs the actual digital logic behavior modeling using the constructed data structures and ordering.

gatesIdentifier.cpp

- Provides a mapping between gate name strings (e.g., “AND”, “NOT”) and internal integer codes.
 - This mapping standardizes gate representation for use in logic evaluation and simplifies switch-case statements during simulation.
 - Helps maintain a consistent gate identification mechanism across modules.
-

node.h


Defines the Node class structure representing every element in the circuit:

- Attributes include:
 - Unique `id`
 - `type` (gate kind encoded as integer)
 - Lists of input signal IDs
 - Pointers to parent and child Node objects
 - `level` for evaluation order
 - `evaluate` to hold computed logic value during simulation

This class is the foundational data structure linking together all gate information, connectivity, and state for simulation.

6. Logical Flow Across Modules

1. Input VHDL file name → Handled by *bencher.cpp* converting it to `.bench`.

- 
2. `.bench` Parsing → *parser.cpp* processes the netlist, generates gate and IO info.
 3. Node Objects Creation → *nodify.cpp* creates Node instances from gate data.
 4. Graph Building → *graphiphy.cpp* links nodes via parent/child pointers.
 5. Level Assignment → *levelizer.cpp* applies BFS to order nodes topologically.
 6. Simulation → *simulator.cpp* evaluates gates level by level over test vectors given in `testbench.txt`.
 7. Output Generation → Simulation results written to file `sim_output.txt`.
-

7. Summary

This modular simulator efficiently converts circuit descriptions from VHDL to a standardized intermediate format, models the circuit as a directed graph, applies graph algorithms for evaluation order, and simulates logic behavior for given test inputs. The use of clear abstractions like the Node class and maps supports scalability and future enhancements.