EE2703 - EE24B114 - ASSIGNMENT 4

Digital Logic Simulator

The goal of this assignment was to design a **combinational logic simulator** that reads a text-based *netlist* file and produces a JSON waveform description compatible with **WaveDrom**.

The `.net` file describes a small digital circuit using four sections:

- **INPUTS:** names of all input signals
- **OUTPUTS:** names of output signals
- **GATES:** logic expressions (AND, OR, NOT, XOR) defining intermediate or output signals
- **STIMULUS:** time-varying input values that change at specific time points

The program reads this file, simulates the logic behavior for each time step, and finally generates a JSON output showing the input and output waveforms

Program Implementation

a. Parsing the Netlist (`parse_netlist`)

- This function reads the `.net` file line by line, removes empty lines and comments, and then separates the contents into the four expected sections: `INPUTS`, `OUTPUTS`, `GATES`, and `STIMULUS`.
- The parser supports both inline (`INPUTS: A B C`) and block (`GATES:` followed by lines) styles.
- The parser also performs validation to ensure:
  - All sections are present and non-empty
  - Section names are valid
  - Each line belongs to a proper section
- If any issue is found (for example, a missing section or an undefined signal), the program raises a clear error message.

b. Simulating Logic (`simulate` and `eval_gate`)

- `simulate()` initializes a dictionary for all input signals and reads each line in the **STIMULUS** section to fill time-based input values.
- Check if the Stimulus data - has all the input values for every input signal

- The gate network is built from the **GATES** section. Each line like
  Eg: `T1 = OR(A, B)`
  defines a mapping of gate name -> logical expression.
- The program ensures time of input values is strictly in increasing order. If not raises error
- Before simulating, the program ensures that every signal used as an input is already defined (either as a primary input or as an output of a previous gate).
- `eval_gate()` executes gate logic in topological order. It supports:
  - **AND**, **OR**, **XOR** (binary gates)
  - **NOT** (unary gate)
- Each gate output is appended as a list of 0/1 values corresponding to time steps.
- After computation, results are merged back into `Input_values` so subsequent gates can use them.

c. JSON Generation (`to_wavedrom_json`)

- After simulation, all signals (inputs and outputs) are formatted into a JSON structure.
- Path(out_path).write_text(json.dumps(js, indent=1) + "\n")
- Converts the Python dictionary `js` (which holds your WaveDrom data) into a **JSON-formatted string**.
- The `indent=1` makes it look neat and readable, with indentation of 1 space.

Command-Line Interface (`main`)

- The program uses **argparse** to handle:
  - Required `.net` file input
  - Output path using `--out` or `-o`
- Example usage:
  `python3 digitalsim.py or_and_not.net --out or_and_not.json`

Using example : **or_and_not.net** as input . We get **or_and_not.json** below as output

```
INPUTS: A B C
OUTPUTS: Y
GATES:
  T1 = OR(A, B)
  T2 = NOT(C)
  Y  = AND(T1, T2)
STIMULUS:
  0  0 0 0
  1  1 0 0
  2  0 1 1
  3  0 0 1
```

```
{
 "signal": [
  {
   "name": "A",
   "wave": "0100"
  },
  {
   "name": "B",
   "wave": "0010"
  },
  {
   "name": "C",
   "wave": "0011"
  },
  {
   "name": "Y",
   "wave": "0100"
  }
 ]
}
```

Additionally, The program includes try and except block while reading the input file to raise “FilenotFound” Error.