# Simulation and Boolean Expression Generation from ISCAS-Format Gate-Level Netlists

Urmit Kikani
*Dept. of Electronics and Communication Engineering*
*Institute of Technology, Nirma University*
Ahmedabad, India
22bec137@nirmauni.ac.in

Prince Viradiya
*Dept. of Electronics and Communication Engineering*
*Institute of Technology, Nirma University*
Ahmedabad, India
22bec144@nirmauni.ac.in

***Abstract*—This paper describes a simple and effective Python-based tool that helps understand how digital logic circuits work. These circuits are written in a special format called the ISCAS netlist, which is used to describe how different logic gates like AND, OR, NAND, and NOR are connected. The tool reads this file, understands the structure of the circuit, and performs a simulation to find out the output values for given input signals.**

**In addition to showing output values, the tool also creates Boolean expressions for each part of the circuit. These expressions help explain what logic the circuit is performing. This process is done by checking each connection step-by-step and combining them into full expressions. The tool is especially useful for learning, analysis, and testing of small digital circuits. It can be used by students, teachers, or engineers who want to understand the behavior of digital systems more clearly.**

***Index Terms*—Digital Logic Simulation, Boolean Expression, ISCAS Netlist, Gate-Level Design, Circuit Analysis**

## I. Introduction

Digital circuits form the foundation of modern electronic devices. They are made by connecting basic logic gates like AND, OR, NAND, and NOR [2]. Understanding how these circuits behave is very important for students, researchers, and engineers. One popular way to describe such circuits is using the ISCAS benchmark format [1]. These ISCAS-format netlists are plain-text files that describe how each gate is connected in the circuit. They are used all over the world for learning and testing digital design tools.

Many students and learners find it difficult to understand how a circuit described in a netlist works. Reading the netlist and manually tracking how each gate behaves can be time-consuming and confusing. That's why we created a simple Python tool. This tool takes an ISCAS-format netlist, processes the information, and simulates how the circuit will behave. It lets users enter input values and shows what the output values will be after passing through the logic gates.

In addition to just simulating outputs, the tool also builds Boolean expressions. These expressions help understand the logic behind each part of the circuit. For example, if an output is the result of an AND gate with two inputs, the expression would be written like (A & B). These expressions are useful for analysis, simplification, or even converting the logic to another form. This makes the tool a great help for educational purposes and small-scale research or testing.

## II. Related Work

ISCAS benchmark circuits have been widely used in digital design research and education. The original ISCAS'85 benchmark suite by Brglez and Fujiwara provided a standard format for logic-level circuit descriptions that allowed researchers to test their tools and algorithms on a common dataset [1].

Various tools and simulators have been developed to analyze these circuits. Many academic tools focus on logic synthesis, testing, and fault simulation using ISCAS-format netlists. Commercial Electronic Design Automation (EDA) tools offer more complex simulations but are often less accessible for learning purposes[6].

Compared to existing tools, this project offers a simple Python-based approach that focuses on ease of use and educational value. It allows users to simulate outputs and generate Boolean expressions without needing high-end software. While not intended for large industrial-scale circuits, the tool provides an effective foundation for teaching and understanding digital logic.

## III. Algorithm and Implementation

### A. Netlist Parsing

The tool reads an ISCAS-format netlist file and extracts gate-level connections [1]. It builds a data structure (dictionary) where each gate or node is represented by its type (AND, OR, NAND, NOR, INPUT, or FANOUT) and a list of its input nodes. Fanout nodes are redirected to their original signal source.

### B. Simulation Engine

Once inputs are assigned to the primary input nodes, the simulation iteratively evaluates each node using digital logic rules [2]. For every gate, the simulation checks whether all its inputs have known logic values. If so, it computes the output using the correct logic function (e.g., AND = 1 only if all inputs are 1).

### C. Boolean Expression Generator

A recursive function generates Boolean expressions by walking backward from each gate to its inputs. It uses signal names (like A, B, C...) and gate types to construct symbolic expressions. For example, if Node 5 is an AND gate with

inputs 2 and 3 named A and B, the expression becomes `(A & B)`. Fanouts are handled by treating them as aliases that point to the original signal source. The recursive function continues to resolve these references backward until it reaches primary input nodes [4]. This method allows complex expressions to be constructed for output nodes by composing smaller sub-expressions from earlier gates, thereby creating a full symbolic representation of the logic path. This approach ensures that the final Boolean expressions accurately reflect the hierarchical structure and logic flow of the entire combinational circuit.

### D. Algorithm Steps

1) Read and parse the ISCAS-format netlist.
2) Build a dictionary mapping node IDs to gate types and their inputs.
3) Identify and assign user-defined values to primary input nodes.
4) Simulate logic propagation:
   - Check for nodes whose inputs have known values.
   - Apply logic rules and assign output value.
   - Repeat until all nodes are evaluated.
5) For named nodes, recursively generate Boolean expressions from inputs to outputs.
6) Display signal values and Boolean expressions.

### E. Flowchart

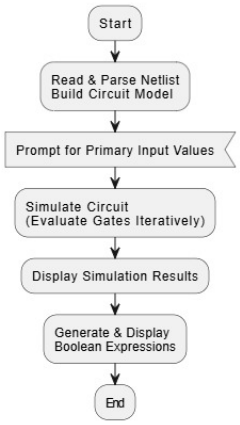


Fig. 1: Flowchart of Simulation and Boolean Generation Tool

### F. Input and Output Behavior

TABLE I: Sample Input and Output Table

| Input Description | Expected Output |
|---|---|
| ISCAS netlist file (e.g., AND gate with inputs 2, 3) | Parsed gate with type AND and inputs [2, 3] |
| User inputs: 1 for input 2, 0 for input 3 | Signal values computed across all nodes |
| Node with AND gate of inputs 2, 3 | Boolean Expression: (A & B) |
| Fanout node connected to another | Redirected signal used for Boolean expression |

## IV. Results and Discussion

### A. Example: C17 Circuit

The C17 circuit is a well-known benchmark from the ISCAS'85 suite. It consists of five inputs, two outputs, and six logic gates. Fig.2 shows the gate-level schematic of the C17 circuit.

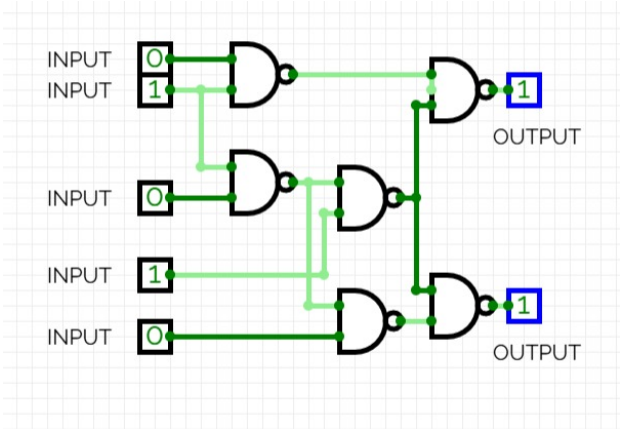


Fig. 2: Gate-Level Diagram of C17 Circuit

### B. Full Truth Table of the C17 Circuit

TABLE II: Truth Table for C17 Circuit

| N1 | N2 | N3 | N4 | N5 | J (N10) | K (N11) |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 |

The ISCAS-format netlist for the C17 circuit is shown below:

```
1,inpt,inpt,N1,0
2,inpt,inpt,N2,0
3,inpt,inpt,N3,0
4,inpt,inpt,N4,0
5,inpt,inpt,N5,0
6,gat,nand,N6,2,1,3
7,gat,nand,N7,2,3,4
8,gat,nand,N8,2,2,4
9,gat,nand,N9,2,5,5
10,gat,nand,N10,2,6,7
11,gat,nand,N11,2,8,9
```

Using this netlist, the tool evaluates the signals and generates the following simulation result for a given set of input values:

```
--- Simulation Output ---
Signal 1: 0
Signal 2: 1
Signal 3: 0
Signal 6: 1
Signal 7: 0
Signal 8: 0
Signal 9: 0
Signal 10: 1
Signal 11: 1
Signal 14: 1
Signal 15: 1
Signal 16: 0
Signal 19: 1
Signal 20: 0
Signal 21: 0
Signal 22: 1
Signal 23: 1
```

Boolean expressions generated for intermediate and output nodes are:

```
---BooleanExpressionsforOutputNets---
N22=~((N1&N3)&(N2&~(N6&N3)))
N23=~((N7&~(N6&N3))&(N2&~(N6&N3)))
```

The output demonstrates the correct functioning of the logic simulation and expression resolution, verifying that the tool processes ISCAS-format netlists accurately and transparently.

## V. Limitations

While the tool is effective for small-scale circuits and educational purposes, it has limitations when applied to larger and more complex designs. Currently, it supports only a subset of gate types and assumes combinational logic with no clocked elements or sequential behavior. Additionally, the tool may become inefficient for very large circuits due to its recursive nature and lack of parallel processing. Advanced features such as fan-in/fan-out balancing, fault modeling [5], and timing analysis are not yet implemented [7]. Integrating GUI interfaces and comprehensive error handling would further improve usability for wider audiences.

## VI. Conclusion

This work demonstrates an effective method to simulate and extract Boolean expressions from ISCAS-format netlists [1]. It offers insight into the internal working of gate-level circuits and opens the door for enhancements like truth table generation, fault simulation, and logic synthesis. The simplicity of the Python-based approach makes it accessible and ideal for educational use [3].

In the future, we plan to extend this tool by supporting sequential logic, optimizing simulation efficiency, and adding a graphical interface for better circuit visualization. Integration with logic minimization tools and the ability to generate full truth tables for arbitrary netlists will make this simulator even more versatile. This project sets the stage for further development in interactive digital circuit education and lightweight design verification environments.


## Acknowledgment

We thank the faculty at Nirma University for their guidance and support in completing this project.


## References


[1] F. Brglez and H. Fujiwara, *A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran*, ISCAS 1985.

[2] M. Mano, *Digital Logic and Computer Design*, Prentice Hall, 1995.

[3] Python Software Foundation, *Python Language Reference*, https://www.python.org

[4] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, Pearson, 2003.

[5] M. Abramovici, M.A. Breuer, A.D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, 1990.

[6] D. D. Gajski, *Principles of CMOS VLSI Design*, Prentice Hall, 1994.

[7] J. Bhasker, *Static Timing Analysis for Nanometer Designs*, Springer, 2009.

## APPENDIX

```
def parse_netlist(file_path):
    circuit = {}
    fanouts = {}
    primary_inputs = []

    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip().replace('', ',')  # Fix malformed characters
            parts = line.strip().split(',')
            idx = int(parts[0])
            category = parts[1]
            gate_type = parts[2]

            if category == 'gat':
                input_count = int(parts[4])
                inputs = [int(p) for p in parts[5:5 + input_count]]
                circuit[idx] = {'type': gate_type, 'inputs': inputs}
                if gate_type == 'inpt':
                    primary_inputs.append(idx)

            elif category == 'fan':
                source = int(parts[3])
                fanouts[idx] = source

    # Merge fanouts into the circuit
    for fan, source in fanouts.items():
        if fan not in circuit:
            circuit[fan] = {'type': 'fan', 'inputs': [source]}

    return circuit, primary_inputs

def evaluate_gate(gate_type, inputs):
    if gate_type == 'and':
        return int(all(inputs))
    elif gate_type == 'or':
        return int(any(inputs))
    elif gate_type == 'nand':
        return int(not all(inputs))
    elif gate_type == 'nor':
        return int(not any(inputs))
    elif gate_type == 'inpt':
        return inputs[0]
    elif gate_type == 'fan':
        return inputs[0]
    else:
        raise ValueError(f"Unknown gate type: {gate_type}")

def simulate_circuit(circuit, primary_inputs, input_values):
    signal_values = {}

    for idx, val in zip(primary_inputs, input_values):
        signal_values[idx] = val

    while len(signal_values) < len(circuit):
        for node, info in circuit.items():
            if node in signal_values:
                continue
            inputs_ready = all(inp in signal_values for inp in info['inputs'])
            if inputs_ready:
                input_vals = [signal_values[inp] for inp in info['inputs']]
                signal_values[node] = evaluate_gate(info['type'], input_vals)

    return signal_values

def build_boolean_expression(circuit, node):
    info = circuit[node]
    gate_type = info['type']

    if gate_type == 'inpt':
        return f"N{node}"
    elif gate_type == 'fan':
        return build_boolean_expression(circuit, info['inputs'][0])
```

```
    input_exprs = [build_boolean_expression(circuit, inp) for inp in info['inputs']]

    if gate_type == 'and':
        return '(' + ' & '.join(input_exprs) + ')'
    elif gate_type == 'or':
        return '(' + ' | '.join(input_exprs) + ')'
    elif gate_type == 'nand':
        return '~(' + ' & '.join(input_exprs) + ')'
    elif gate_type == 'nor':
        return '~(' + ' | '.join(input_exprs) + ')'
    else:
        raise ValueError(f"Unknown gate type in expression: {gate_type}")

def find_output_nodes(circuit):
    used_as_input = set()
    for info in circuit.values():
        for inp in info['inputs']:
            used_as_input.add(inp)
    outputs = [node for node in circuit if node not in used_as_input]
    return outputs

if __name__ == "__main__":
    netlist_file = "iscas2.txt"  # Your ISCAS-format netlist file
    circuit, primary_inputs = parse_netlist(netlist_file)

    print(f"Primary inputs found: {primary_inputs}")
    input_values = []

    print("Enter values for primary inputs:")
    for inp in primary_inputs:
        while True:
            try:
                val = input(f"Value for N{inp} (0 or 1): ").strip()
                if val not in ['0', '1']:
                    raise ValueError
                input_values.append(int(val))
                break
            except ValueError:
                print("Invalid input. Please enter 0 or 1.")

    signal_values = simulate_circuit(circuit, primary_inputs, input_values)

    print("\n--- Simulation Output ---")
    for k in sorted(signal_values.keys()):
        print(f"Signal N{k}: {signal_values[k]}")

    output_nodes = find_output_nodes(circuit)

    print("\n--- Boolean Expressions for Output Nets Only ---")
    for node in sorted(output_nodes):
        expr = build_boolean_expression(circuit, node)
        print(f"N{node} = {expr}")
```