Lebanese American University

School of Engineering

COE 599F Fall 2023

Project Report

Group 2:

Al Assaad Michael 201901542

Semaan Anthony 201900391

Date: 21/12/2023

Table of Contents

Table of Figures

Introduction

In the rapidly evolving domain of digital electronics, the robustness and reliability of integrated circuits (ICs) are paramount. This project, undertaken in the Fall of 2023 as part of COE 599F, delves into the process of testing digital integrated circuits through the development of a Simple Fault Simulator. Our journey begins with an analytical approach to understanding circuit descriptions, followed by a hands-on application of the stuck-at-fault model and culminating in the creation of a fault simulator designed to test and validate digital circuits.

GitHub: anthonySemaan01/digital-circuit-simulation (github.com)

Phase I – Benchmarks Parser

The first phase of our project, "Circuit Description and Parser Development," serves as the fundamental upon which the entire fault simulator is constructed. This critical phase revolves around the development of a comprehensive understanding of digital circuit descriptions and the creation of an effective parser tool.

Our primary objective in this phase is to develop a robust parser capable of accurately interpreting and processing circuit descriptions. Circuit descriptions are typically presented in a standardized format, such as netlists, which detail the various components and interconnections within a digital circuit. The parser's role is to read these netlists, decode the information, and transform it into a format that can be used for simulation and fault detection purposes.

In this phase, we face several challenges. Primarily, our parser must exhibit flexibility in handling various circuit description formats. Building upon the c17 digital circuit as a foundation, our goal is to ensure accurate parsing and internal representation of all other circuits. Additionally, efficiency is paramount—the parser needs to adeptly handle large, intricate circuits without errors. Lastly, the parser's design should prioritize expandability, enabling seamless integration of future sections. This multifaceted approach seeks to achieve precision, scalability, and reliability in circuit parsing.

To achieve the above-mentioned requirements, numerous classes have been implemented. First, the Circuit class represents the digital circuit itself, having as attributes a list of Input Wire objects), output Wire objects, internal Wires objects, and Gates objects which will be connected using the wires. Here is the constructor of the Circuit class:

```
class Circuit:
    def __init__(self):
        self.inputs: List[Wire] = []
        self.outputs: List[Wire] = []
        self.gates: List[Gate] = []
        self.wires: List[Wire] = []
```

Since the Circuit class references both the Wire and Gate class, we will be providing a detailed explanation of each:

Gate Class

The **Gate** class models a logical gate in a digital circuit, such as AND, OR, NOR… Here is the constructor of the Gate class:

```python
class Gate:
    def __init__(self, name: str, gate_type: str, fanin_wires: Union[List[Wire], None] = None,
                 output_wire: Union[Wire, None] = None):
        self.name = name
        self.gate_type = gate_type
        self.fanin_wires = fanin_wires  # List of Wire objects for fanin
        self.output_wire = output_wire  # Will be set later
```

Figure 1 - Gate Class

On creation, each gate is given:

1. name: being the number of the output wire.
2. gate type that represents the Boolean function performed by the gate.
3. Fanin_wires: the input wires object.
4. Output wire: the output wire object.

Based on the type of the gate, the output wire is given the correct value using the following function:

```python
def simulate(self):

    if self.output_wire.is_stuck_at:
        self.output_wire.value = self.output_wire.stuck_at_value
        self.output_wire.given_a_value = True
        self.output_wire.can_be_triggered = True
        self.output_wire.ensure_fanout_can_be_triggered()

    else:
        values_at_fanin_wires = []

        for input_wire in self.fanin_wires:
            if input_wire.is_stuck_at:
                values_at_fanin_wires.append(input_wire.stuck_at_value)
            else:
                values_at_fanin_wires.append(input_wire.value)

        if self.gate_type == "AND":
            value = all(values_at_fanin_wires)
        elif self.gate_type == "NAND":
            value = not all(values_at_fanin_wires)
        elif self.gate_type == "OR":
            value = any(values_at_fanin_wires)
        elif self.gate_type == "NOR":
            value = not any(values_at_fanin_wires)
        elif self.gate_type == "XOR":
            value = sum(values_at_fanin_wires) == 1
        elif self.gate_type == "NOT":
            value = not values_at_fanin_wires[0]
        elif self.gate_type == "XNOR":
            value = not (sum(values_at_fanin_wires))

        self.output_wire.value = value
        self.output_wire.given_a_value = True
        self.output_wire.can_be_triggered = True
        self.output_wire.ensure_fanout_can_be_triggered()
```

Figure 2 - Gate Simulation Function

First, we are checking if we have any stuck at the output wire of the gate, and if the is the case, we are simply assigning the value to the wire. If not, we are checking the input wires and computing the respective Boolean function based on the type of the gate. Finally, we are updating the parameters of the wires, and we ensure that the output wire of the gate is now ready to be triggered as it has been given a value (This function will be discussed in more detail in subsequent sections).

Wire Class

The **Wire** class represents a wire in a digital circuit, which can carry signals between gates and can have various states and properties. Here is the constructor of the Wire class:

```
class Wire:
    def __init__(self, name: str, has_direct_connection_to_gate: bool = False,
                 direct_connect_to_gate: Union[str, None] = None,
                 seen_as_input_before: bool = False,
                 is_input: bool = False, is_stuck_at: bool = False,
                 stuck_at_value: Union[None, bool] = None,
                 can_be_triggered: bool = False, value: bool = False,
                 given_a_value: bool = False):
        self.name = name
        self.value = value
        self.given_a_value = given_a_value
        self.fanout: List[Wire] = []
        self.has_direct_connection_to_gate = has_direct_connection_to_gate
        self.direct_connect_to_gate: Union[str, None] = direct_connect_to_gate
        self.seen_as_input_before: bool = seen_as_input_before
        self.is_input = is_input
        self.is_stuck_at = is_stuck_at
        self.stuck_at_value = stuck_at_value
        self.can_be_triggered = False
```

Figure 3 - Wire Class

Similar to the Gate class, each wire:

1. has a name,
2. a value (True being ‘1’, False being ‘0’)
3. given a value is a Boolean that specifies whether the wire hold a value or not yet. This parameter is used to check if the gate that the wires are connected to can be triggered
4. fanout is a list of wires
5. is_input keeps track if the wire is an input or an internal signal
6. the remaining parameters are used internally to ensure smooth operation of the simulator (to be discussed later)

Now that the base code has been established, we developed the following parser:

```python
def parse_bench_file_with_unique_inputs(self, file_path: str):
    wires_usage_count: dict = {}
    output_wires_tracker: List = []

    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()

            if not line or line.startswith("#"):
                continue

            if line.startswith("INPUT"):
                new_input_wire = Wire(name=line.split('(')[1].split(')')[0], is_input=True,
                                      seen_as_input_before=False, has_direct_connection_to_gate=False,
                                      can_be_triggered=True)
                self.inputs.append(new_input_wire)
                self.wires.append(new_input_wire)

            elif line.startswith("OUTPUT"):
                output_wires_tracker.append(line.split('(')[1].split(')')[0])

            else:
                gate_info = line.split('=')

                gate_name = gate_info[0].strip()
                gate_def = gate_info[1].strip()
                gate_type = re.split(r'\(|,', gate_def)[0]

                created_gate = Gate(name=gate_name, gate_type=gate_type)

                fanin_names = [fanin_name.strip() for fanin_name in
                               re.findall(r'\(([^)]+)', gate_def)[0].split(',')]
```

Figure 4 - Benchmark parsing Function

```python
                fanin_wires_for_that_specific_gate = []
                # loop over each fanin found
                for fanin_wire_name in fanin_names:

                    # check if the fanin is already added to the circuit
                    if fanin_wire_name in self.get_all_wires_names():
                        wire: Wire = self.get_wire_based_on_name(fanin_wire_name)

                        # check if the wire has been seen as an input to a gate before
                        if wire.seen_as_input_before:
                            # the wire has been seen as an input before, but it does not have any fanout yet. This
                            # scenario happens when we have an input wire that has been connected to a gate, and now
                            # it should be connected to another gate
                            if len(wire.fanout) == 0:
                                wire_one = Wire(name=f"{fanin_wire_name}.{len(wire.fanout) + 1}",
                                                seen_as_input_before=True, has_direct_connection_to_gate=True,
                                                direct_connect_to_gate=wire.direct_connect_to_gate,
                                                can_be_triggered=True if wire.is_input else False)
                                wire.fanout.append(wire_one)

                                gate_connected_to_initial_wire_before_split = self.get_gate_connected_to_wire(wire)
                                gate_connected_to_initial_wire_before_split.fanin_wires[next(
                                    (i for i, inst in
                                     enumerate(gate_connected_to_initial_wire_before_split.fanin_wires) if
                                     inst is wire), None)] = wire_one

                                wire_two = Wire(name=f"{fanin_wire_name}.{len(wire.fanout) + 1}",
                                                seen_as_input_before=True,
                                                has_direct_connection_to_gate=True,
                                                direct_connect_to_gate=created_gate.name,
                                                can_be_triggered=True if wire.is_input else False)
                                wire.fanout.append(wire_two)
```

```
                        fanin_wires_for_that_specific_gate.append(wire_two)

                        wire.has_direct_connection_to_gate = False
                        wire.direct_connect_to_gate = None

                        self.wires.append(wire_one)
                        self.wires.append(wire_two)
                    else:
                        additional_wire = Wire(name=f"{fanin_wire_name}.{len(wire.fanout) + 1}",
                                               seen_as_input_before=True,
                                               direct_connect_to_gate=created_gate.name,
                                               has_direct_connection_to_gate=True,
                                               can_be_triggered=True if wire.is_input else False)
                        wire.fanout.append(additional_wire)
                        self.wires.append(additional_wire)
                        fanin_wires_for_that_specific_gate.append(additional_wire)

                else:
                    wire.has_direct_connection_to_gate = True
                    wire.direct_connect_to_gate = created_gate.name
                    wire.seen_as_input_before = True

                    fanin_wires_for_that_specific_gate.append(wire)
            else:
                wire = Wire(name=fanin_wire_name,
                            seen_as_input_before=True,
                            has_direct_connection_to_gate=True,
                            direct_connect_to_gate=created_gate.name)
                self.wires.append(wire)
                fanin_wires_for_that_specific_gate.append(wire)

        created_gate.fanin_wires = fanin_wires_for_that_specific_gate

        created_output_wire = Wire(name=created_gate.name, has_direct_connection_to_gate=False,
                                   seen_as_input_before=False, is_input=False, can_be_triggered=False)
        created_gate.output_wire = created_output_wire

        if created_gate.name in output_wires_tracker:
            self.outputs.append(created_output_wire)

        self.wires.append(created_output_wire)
        self.gates.append(created_gate)
```

First, it reads the file line by line.

1. Ignore empty lines and lines starting with #.
2. Processes lines starting with "INPUT" or "OUTPUT" or assume they represent gates.
    a. Lines starting with "INPUT" create a new input wire and add it to the list of inputs and wires.
    b. Lines starting with "OUTPUT" track the output wires.
3. For lines representing gates:
    a. Extracts gate name and definition.
    b. Creates a gate object and extracts its fanin (input) wires.
    c. Creates an output wire for the gate and sets it accordingly.
4. It iterates through the fanin wire names obtained from the gate's definition.
5. For each fanin wire:
    a. It checks if the wire is already part of the circuit.
    b. If the wire exists

i. If it has been seen as an input before:
    1. If it has no fanouts, it splits the connection to accommodate multiple gates by creating new wires (wire_one and wire_two) and updating connections accordingly.
    2. If it already has fanouts, it adds a new wire (additional_wire) and adjusts connections.
ii. If it hasn't been seen as input before, it connects it to the current gate.

c. If the wire doesn't exist in the circuit, it creates a new wire and connects it to the current gate.

6. Finally, it updates the fanin wires for the specific gate (created_gate) with the processed fanin wires (fanin_wires_for_that_specific_gate).

Based on the code above, whenever we have a fanout, the following naming convention is followed:

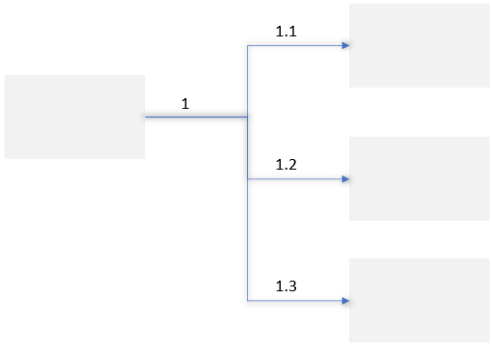

Figure 5 - Fanout Representation

Parsing Endpoint:

```
def build_circuit(self, file_name: str):
    circuit = Circuit()
    if file_name not in os.listdir(self.path_service.paths.benchmarks):
        raise Exception("Filename not available")
    circuit.parse_bench_file_with_unique_inputs(
        file_path=os.path.join(self.path_service.paths.benchmarks, file_name))

    return {
        "inputs": [wire.get_wire_parameters() for wire in circuit.inputs],
        "all_wires": [wire.get_wire_parameters() for wire in circuit.wires],
        "all_gates": [gate.get_gate_parameters() for gate in circuit.gates],
        "outputs": [wire.get_wire_parameters() for wire in circuit.outputs]
    }
```

Figure 6 - Circuit Class

Phase 2 – True Value Simulation

In this phase, we aim to construct a true-value simulation tool utilizing the parser developed in phase 1. This tool will not only execute true-value simulations but will also possess the capability to introduce a stuck-at fault at a specified signal within the circuit, potentially influencing the overall behavior of the system.

Flag Ready to be Triggered Signals

Before proceeding, the 'ensure fanout can be triggered' has been called without providing proper documentation. Here is the function definition:

```
def ensure_fanout_can_be_triggered(self):
        if self.can_be_triggered:
            for fanout_wire in self.fanout:
                fanout_wire.ensure_fanout_can_be_triggered()  # Recursively call
                for each fanout wire
                fanout_wire.can_be_triggered = True

                if not fanout_wire.given_a_value:  # Check if fanout wire
                already has a value
                    if fanout_wire.is_stuck_at:
                        fanout_wire.value = self.stuck_at_value
                    else:
                        fanout_wire.value = self.value
                    fanout_wire.given_a_value = True  # Mark fanout wire as given
a value

                    fanout_wire.ensure_fanout_can_be_triggered()  # Recursively
assign value to fanout wires
```

In this function, we are ensuring that each fanout wire of a ready to be triggered wire can be triggered by recursively looping over each connection.

Implementation

To achieve this objective, we have developed the following functionality:

```
def simulate_circuit(self, input_vector: List[InputParam], place_stuck_at: Union[None, StuckAt] = None):
    stuck_at_wire = None
    if place_stuck_at:
        stuck_at_wire = self.get_wire_based_on_name(place_stuck_at.wire_name)
        stuck_at_wire.is_stuck_at = True
        stuck_at_wire.given_a_value = True
        stuck_at_wire.value = place_stuck_at.value
        stuck_at_wire.stuck_at_value = place_stuck_at.value
        stuck_at_wire.can_be_triggered = True
        stuck_at_wire.ensure_fanout_can_be_triggered()
```

Figure 7 - True Value Circuit Simulation

Initially, we verify whether the user intends to introduce any stuck-at faults into the circuit. We update the parameters accordingly based on this input. If no stuck-at faults are requested, the simulation will proceed as a true-value simulation.

```
186             # assign the desired values to input wires
187             for input_wire in self.inputs:
188                 for input_param in input_vector:
189                     if input_param.wire_name == input_wire.name:
190                         if stuck_at_wire is input_wire:
191                             input_wire.value = input_wire.stuck_at_value
192                             input_wire.stuck_at_value = input_wire.stuck_at_value
193                         else:
194                             input_wire.value = input_param.value
195                         input_wire.given_a_value = True
196                         input_wire.can_be_triggered = True
197                         input_wire.ensure_fanout_can_be_triggered()
198                         break
199 
200             simulated_gates = []
201             non_simulated_gates = self.gates
202 
203             while non_simulated_gates:
204                 remaining_gates = []
205                 for gate in non_simulated_gates:
206                     if gate.can_be_triggered():
207                         gate.simulate()
208                         simulated_gates.append(gate)
209                     else:
210                         remaining_gates.append(gate)
```

The initial 'for' loop is responsible for assigning respective values to all input wires. In cases where an input wire has a fanout, these values are prepared for triggering and adopt the value of the input wire.

Subsequently, the 'while' loop iterates continuously over the list of gates that have not yet been simulated. This loop operates until there are no gates in the circuit that remain untriggered. When the list of gates yet to be simulated becomes empty, the simulation concludes. At this stage, the wires retain the desired values established during the simulation.

Simulation Endpoint

```
def simulate(self, simulate_circuit: SimulateCircuit):
    circuit = Circuit()
    if simulate_circuit.file_name not in os.listdir(self.path_service.paths.benchmarks):
        raise Exception("Filename not available")
    circuit.parse_bench_file_with_unique_inputs(
        file_path=os.path.join(self.path_service.paths.benchmarks, simulate_circuit.file_name))
    circuit.simulate_circuit(input_vector=simulate_circuit.input_params, place_stuck_at=simulate_circuit.stuck_at)

    circuit_output_values = circuit.get_circuit_output_values()
    wires = circuit.wires

    wires_values = {}
    for wire in wires:
        wires_values[wire.name] = wire.value

    return {
        "circuit_output_values": circuit_output_values,
        "wires_values": wires_values
    }
```

Figure 8 - Simulation Endpoint

Demo Phase 2 - 1

For this phase of the project, we will be implementing the following:

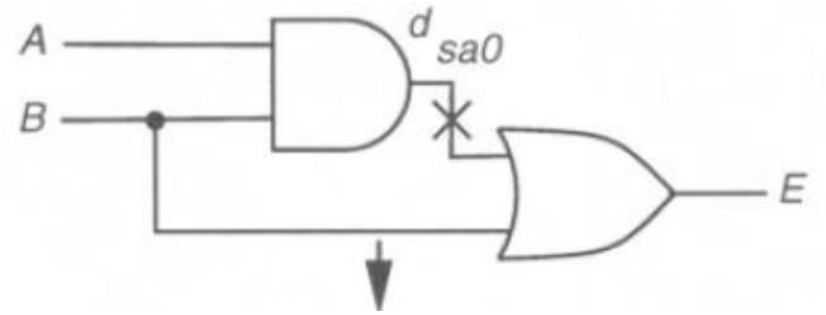

Figure 9 - True Value Simulation Demo

Benchmark code representation:

```
# 2 inputs
# 1 outputs
# 0 inverter
# 2 gates ( 1 AND, 1 OR )

INPUT(1)
INPUT(2)
OUTPUT(4)

3 = AND(1,2)
4 = OR(3,2)
```

The API request is as follows:

```
{
  "file_name": "self_developed.txt",
  "input_params": [
    {
      "wire_name": "1",
      "value": false
    },
    {
      "wire_name": "2",
      "value": false
    }
  ],
  "stuck_at": {
    "wire_name": "3",
    "value": true
  }
}
```

We assigned input 1 a value ‘0’; input 2 a value ‘0’; and a stuck at fault ‘1’ at 3 (the output of the AND gate). We obtained the following results after simulation:

```
{
  "circuit_output_values": {
    "4": true
  },
  "wires_values": {
    "1": false,
    "2": false,
    "3": true,
    "4": true,
    "2.1": false,
    "2.2": false
  }
}
```


As expected, we obtained a ‘1’ at the output as we have a stuck at 1 at the output of the AND gate.

Phase 3 - Serial Simulation

The final phase of the project intends to perform a serial simulation. The implemented Serial Simulation follows the following flowchart:

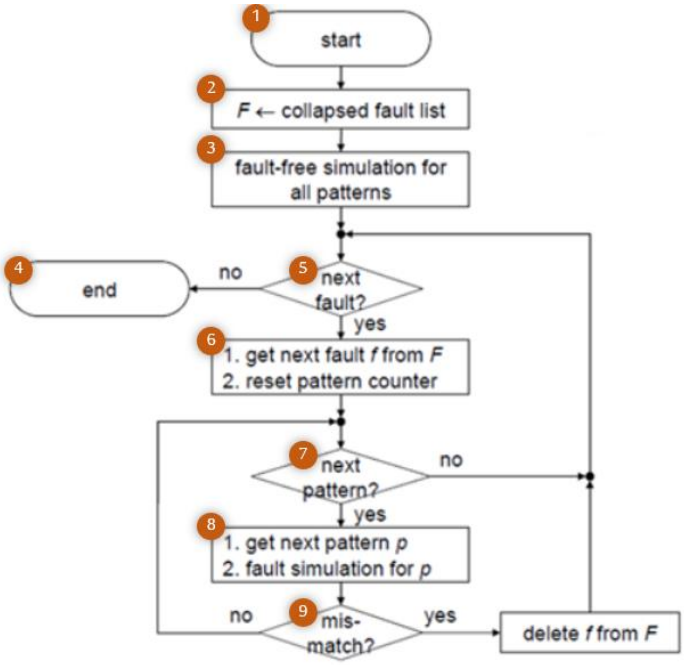

Figure 10 - Serial Fault Simulation Flowchart

Step 1: Commence the serial simulation by converting the benchmark file into the custom class representation.

Step 2: Generate the collapsed fault list, encompassing all potential faults within the circuit.

Step 3: Generate all potential input patterns and execute fault-free simulation, also known as true-value simulation.

Steps 5, 6, 7, 8, 9: Iterate through each identified fault within the circuit. Insert the fault and initiate simulation using the identified input patterns. Upon identifying a discrepancy between the true value and the faulty value, promptly detect the fault and perform an early dropout (cease the simulation upon fault detection). This optimization step enhances the system's performance.

Input Pattern Generation

To generate all potential input patterns of the circuit, the following function has been developed.

```
def generate_wire_patterns(self):
    def generate_patterns_helper(wires, index, pattern, patterns):
        if index == len(wires):
            patterns.append(pattern.copy())
            return

        pattern[wires[index].name] = False
        generate_patterns_helper(wires, index + 1, pattern, patterns)

        pattern[wires[index].name] = True
        generate_patterns_helper(wires, index + 1, pattern, patterns)

    patterns = []
    generate_patterns_helper(self.inputs, 0, {}, patterns)
    return patterns
```

Figure 11 - Test Vectors Generation Function

The function recursively loops over input wires until all input patterns are developed.

Faults Identification

Since the Circuit class is keeping track of all the wires in the circuit, the identification of all faults in the circuit has been developed using the following method:

```
def get_all_faults_in_the_circuit(self):
    all_the_faults_in_circuit = []
    for wire in self.wires:
        all_the_faults_in_circuit.append(StuckAt(wire_name=wire.name, value=True))
        all_the_faults_in_circuit.append(StuckAt(wire_name=wire.name, value=False))

    return all_the_faults_in_circuit
```

Figure 12 - Getting Faults in the Circuit Function

Implementation

After performing the above-mentioned steps, the following block is being called:

```
for stuck_at_fault in stuck_at_faults:
    for index, input_pattern in enumerate(new_input_patterns):
        # have the input parameter list compatible with simulate circuit
        input_parameters: List[InputParam] = []
        for input_name, input_value in input_pattern[index].items():
            input_parameters.append(InputParam(wire_name=input_name, value=input_value))

        copy_of_circuit = copy.deepcopy(circuit)
        copy_of_circuit.simulate_circuit(input_vector=input_parameters, place_stuck_at=stuck_at_fault)
        circuit_output = copy_of_circuit.get_circuit_output_values()

        if circuit_output != circuit_results_per_input_pattern[index]:
            faults_detected.append({"fault": stuck_at_fault,
                                    "vector_used": input_parameters,
                                    "true_value": circuit_results_per_input_pattern[index],
                                    "faulty_value": circuit_output})
            number_of_detected_faults += 1
            break
```

For each fault:

1. For each input pattern:
    a. Simulate the circuit by inserting the fault along with the desired inputs.
    b. Compare the results obtained from the true-value simulation to the results after the fault insertion.
        i. If the results differ, halt the simulation for this specific fault, flagging the fault as detectable.
        ii. If the results are identical, proceed to the next input pattern.
    c. If all input patterns have been exhausted without identifying the fault, deem the fault as redundant (non-detectable).

Demo Phase 3 - 1

In the following section, we will be going through a sample circuit example to showcase the full functionality of the code explained above.

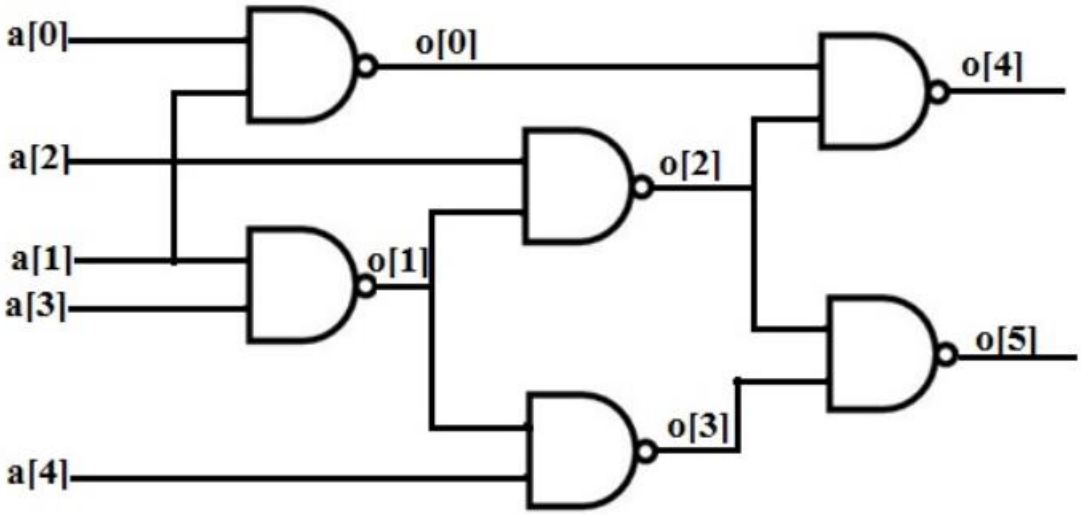

Figure 13 - c17 Circuit Design

The following is the c17 benchmark that we will process:

```
# c17
# 5 inputs
# 2 outputs
# 0 inverter
# 6 gates ( 6 NANDs )

INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)
11 = NAND(3, 6)
16 = NAND(2, 11)
19 = NAND(11, 7)
22 = NAND(10, 16)
23 = NAND(16, 19)
```

Figure 14 - c17 Benchmark File

After parsing the c17 benchmark file, we would need to simulate the circuit to detect faults. To do so, we call the “serial simulation” endpoint that would perform this task and the response to this API call is presented below:


```
{
  "total_time": "48.758507 ms",
  "total number of faults": 34,
  "number of redundant faults": 0,
  "coverage": "34/34",
  "fault_coverage": 1,
  "fault_efficiency": 1,
  "input_patterns": 32,
  "faults_not_detected": []
}
```


Hence, we can deduce that all faults have been detected.

Demo Phase 3 - 2

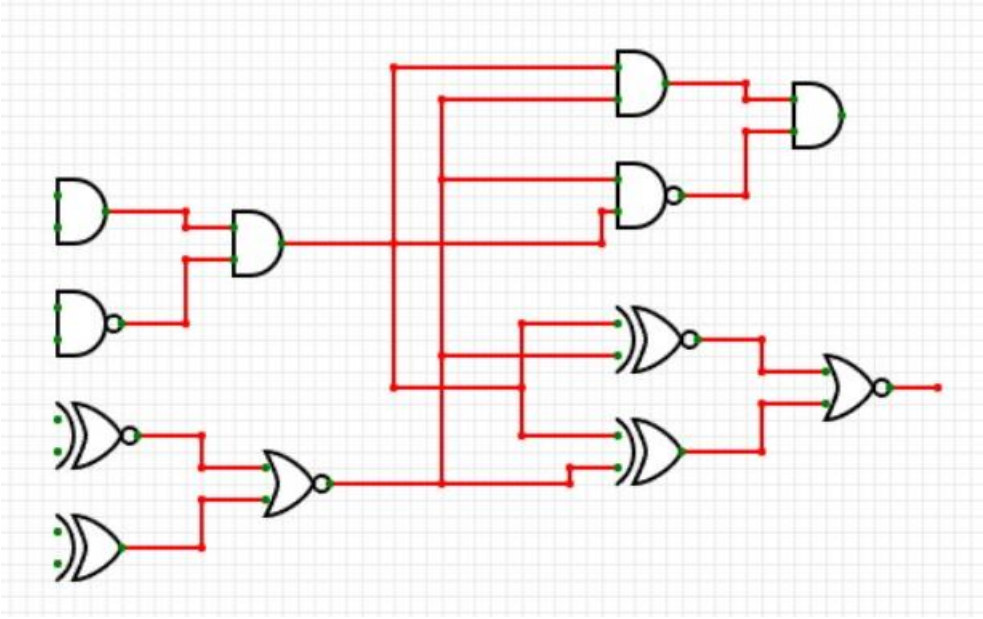

Figure 15 - Complex Circuit Fault Simulation

In this demo, we will demonstrate the testing of serial fault simulation on a more complex circuit composed of ANDs, NORs, XORs and XNORs.

Below is the benchmark file:

```
# 8 inputs
# 2 outputs
# 12 gates

INPUT(1)
INPUT(2)

INPUT(3)
INPUT(4)

INPUT(5)
INPUT(6)

INPUT(7)
INPUT(8)

OUTPUT(19)
OUTPUT(20)

9 = AND(1, 2)
10 = NAND(4,3)
11 = XNOR(5,6)
12 = XOR(7,8)

13 = AND(9,10)
14 = NOR(11,12)

15 = AND(13,14)
16 = NAND(13,14)

17 = XNOR(13,14)
18 = NOR(13,14)

19 = AND(15,16)
20 = NOR(17,18)
```

The result of the simulation is shown below:


```
{
  "total_time": "1965.307474 ms",
  "total number of faults": 56,
  "number of redundant faults": 10,
  "coverage": "46/56",
  "fault_coverage": 0.8214285714285714,
  "fault_efficiency": 0.696969696969697,
  "input_patterns": 256,
  "faults_not_detected": [
    {
      "wire_name": "15",
      "value": false
    },{
      "wire_name": "13.1",
      "value": false
},{
      "wire_name": "13.2",
      "value": true
    },{
      "wire_name": "14.1",
      "value": false
    },{
      "wire_name": "14.2",
      "value": true
    }, {
      "wire_name": "16",
      "value": false
    }, {
      "wire_name": "13.4",
      "value": true
    }, {
      "wire_name": "14.4",
      "value": true
    },{
      "wire_name": "18",
      "value": false
    },  {
      "wire_name": "19",
      "value": false
    }
  ]
```

Conclusion

In brief, this report mentioned our progress throughout the project in its three phases. We began by understanding the benchmark file and parsing it to have a modular representation that would lay the foundation for the second phase that performs true value simulation of the circuit. The last stage was to perform serial fault simulation where we would extract the fault coverage and undetected faults.