Testing for Digital Integrated Circuits

Performance Measurement Analysis of Fault Simulator Algorithms

Objectives

- Analyze circuit description based on netlist
- Generate true-value simulation for different test vectors
- Apply stuck-at-fault model
- Experiment with benchmarks
- Develop fault simulator algorithms
- Perform analysis of different fault simulation algorithms
- Work efficiently within a team

Due Date

- Monday, December 9, 2024 – class time

Teams

- You may work in teams of at most 3 students.

Grading

- 30% of your final grade

Project Description

For this project, you will create a fault simulator that supports two fault simulation algorithms, and measures the performance of each against a set of benchmarks. The fault simulator can be developed in three consecutive stages.

You may use the high-level programming language of your choice (C++, Java, python, …).

Make sure you divide your work among the three phases; otherwise, you will not be able to complete the entire project on time.

Phase I: Benchmark Parsing

The first phase consists of parsing the ISCAS85 benchmark. (https://s2.smu.edu/~mitch/class/8389/assgnmt1/bench/iscas85_format.pdf).

The ISCAS85 benchmarks use a netlist to specify combinational circuits. *Bench* is a circuit description language used to describe the ISCAS85 and other benchmark circuits.

The general format is as follows:

- Inputs and outputs specification
- Gates specification

C17 is the simplest combinational circuit in the benchmark suite (source code at: https://pld.ttu.ee/~maksim/benchmarks/).

The following is C17 in *bench* format:

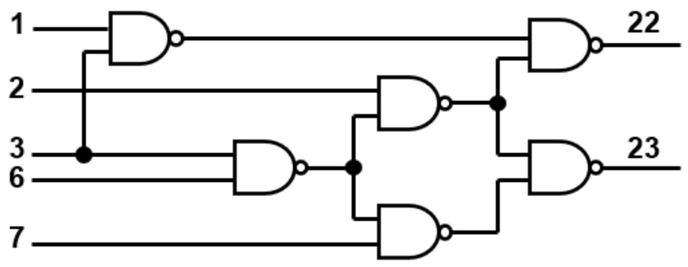
```
# c17
# 5 inputs
# 2 outputs
# 0 inverter
# 6 gates ( 6 NANDs )

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)
```

This corresponds to the following circuit:



Another example is the C432 which is a priority encoder and has the following header:

```
# c432
# 36 inputs
# 7 outputs
# 40 inverters
# 120 gates ( 4 ANDs + 119 NANDs + 19 NORs + 18 XORs )
```

- Given a bench input file, parse it using the language of your choice.
- Your code should read complex ISCAS combinational circuits.
- Design your code by properly interpreting the circuit and its gates and using an efficient data structure.
- Some suggested records you might want to maintain per gate (gate name, gate type, fanin list, fanout list, other attributes such as faults, …)
- Make sure you test your parser thoroughly, and not for just the C17 benchmark.
- This phase represents 30% of the project grade.

Phase II: Logic Simulation

In this phase, you are to add to Phase I by implementing a true-value logic simulator. Your program should levelize the circuit and then execute the provided vectors. You may assume the circuits consist of zero-delay Boolean gates.

You might want to take advantage of the fact that the ISCAS format lists each network node in levelized order. Also, it lists the fanout branches separately as distinct nodes (with distinct names) and specifies the connectivity for each fanout branch.

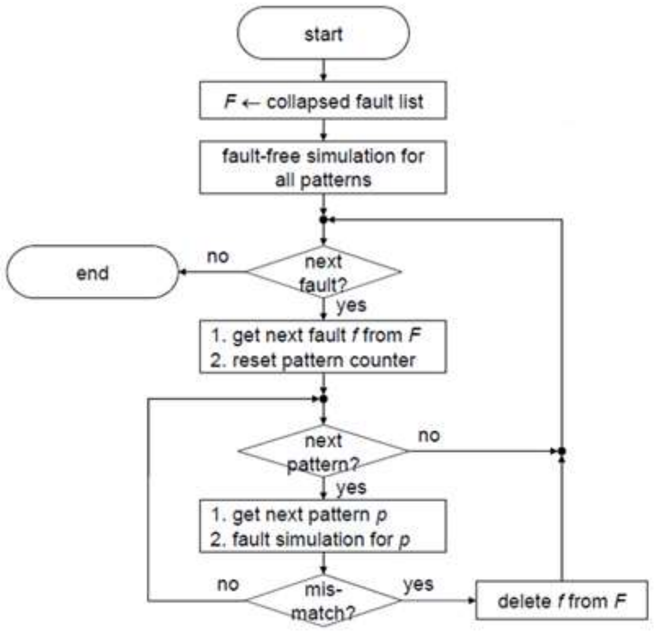
Given a test vector pattern and expected output, simulate your circuit and have your simulator verify the circuit.

This phase represents 20% of the project grade.

Phase III: Fault Simulator Implementations

Serial Fault Simulator

Implement a serial fault simulator. The fault simulator should take as input the circuit and a list of faults to be tested. For each fault, the simulator would inject the specified stuck-at-fault in the circuit and perform logic simulation over the modified netlist. The modified netlist should be simulated vector by vector, comparing the responses with good responses. You may follow the flowchart below and consider the collapsed fault list (initially) as the list of all the faults in the circuit.



Parallel Fault Simulator

Implement a parallel fault simulator. The fault simulator should take as input the circuit and a list of faults to be tested. Assume that the word length w = 32. Hence, every pass of the algorithm will be able to process 31 faults simultaneously. The modified netlist should be simulated vector by vector, comparing the responses with the good responses found in bit 0 of the word. If the circuit contains more than 31 faults, you will need to implement multiple passes of the algorithm.

Testing

To test your design, run the algorithms on the c17 benchmark.

In addition, create a bench netlist for a 2-bit ripple carry adder to test the algorithms. Refer to figure 7.1 on page 156 from your book that shows the structure for a 1-bit adder if needed.

Analysis

For each algorithm, your simulations must output the following:

- Time in seconds for the simulation to run.
- Fault coverage (= number of detected faults / total number of faults).

This phase should also compare the performance of both of these algorithms by addressing the following:

- Verify, using the testing data obtained above, that with *n* faults, the parallel fault simulator will run $(n + 1)(w - 1)/n$ times faster than a serial fault simulator. If that is not the case, explain why not.
- Plot the CPU time vs number of vectors for both circuits.
- Plot the CPU time vs the number of gates for both circuits.
- This phase represents 50% of your grade.

Project Deliverables

- Write-up showing documentation and implementation decisions (classes, data structures, …), user information, input formats, assumptions, limitations, sample runs and results, plots, link to your source code, …
- Demo discussing implementation details.