Fault Simulation

Group Project Phase #2

The second phase of the group project focuses on implementation of two different fault simulation methods: *deductive fault simulation* (DFS) and *parallel fault simulation* (PFS), in addition to a few modifications to your previous operations.

***Logic simulation* (with multiple inputs):** In phase-1, your logic simulator was called with a single input pattern. However, in this phase, you are asked to modify your implementation so `"LOGICSIM"` can run on a set of input patterns. And you are recommended to implemented Event-driven Simulation in phase2.

- An input line can have a value in {0, 1, **X**}. Please note that you may not implemented the simulation of 'X' values in your previous version.
- The input pattern file has the following format: the first line is the PI node numbers, separated by a comma, followed by the actual input patterns, each in a separate line. The sequence of input values is the same as the sequence in the first line, and they are separated by comma. Sample input file is provided for you alongside this instructions. The output pattern file has the same format as the input file, but this time for POs.
- Suggestion: as you don't know the number of input patterns in advance, using STL dynamic data structures, such as *vectors* can be very helpful.

***Reduced fault list* (RFL):** The RFL simply generates a reduced list of faults required for the ATPG of the circuit. In this function we only consider the *check point theorem* as a method to reduce the number of faults.
Follow these steps for the implementation of RFL:

- The new menu option is `"RFL"` which has one string arguments, the name of the output file.
- The output file format is a simple `.txt` file. Each row shows a single stuck at fault with this format: `<NODE-ID>@<FAULT>` such as 549@1 for node 549 stuck at 1.

***Deductive fault simulator* (DFS):**

- The DFS simulator simply has test patterns (one or many) as input and reports all the detectable (so not just the RFL) faults using these test patterns.
- Our suggestion is to first implement a method (maybe call it DFS_single?) that gets a single test pattern as input argument (maybe using vectors?), and runs DFS on

this single pattern. After verification of the results of this method, add a wrapper method that calls the previous method for each of the test patterns (and maybe call it DFS_multi?).

- The input test pattern file is the same as the input for your new modification of logic simulation. We will only use {0,1} values for fault simulation.
- The output is a simple `.txt` file. Each row shows a single stuck at fault with the same format as in RFL.
- The new menu option is `"DFS"` which has two string arguments, first the name of the input file and second the name of the output file.

***Parallel fault simulator* (PFS):**

- The PFS simulator simply gets a list of faults (it can be all the faults) and a list of test patterns (one or many) as inputs and reports which one of the faults can be detected with these test patterns using the PFS (single pattern, parallel faults simulation) method.
- Similar to DFS, our suggestion is to first implement a method that does PFS for a single test pattern and then write a wrapper around this to feed-in several test patterns.
- The input test pattern file format is the same as input file in DFS.
- The input fault list has the same format as the output of DFS.
- The output file is a list of the detectable faults, with the same format as the input fault list.
- The input fault list can have any length (F). You need to find out the processor bit-width (W) within your code (no hard-coding, we may not know if the machine that is running your simulator is 32 or 64 bits).
- PFS needs to pass the circuit $\lceil F/(W-1) \rceil$ times for the first test vector. However, you may argue that if *fault-dropping* is applied, we may remove some of the faults from our fault list when running PFS for other test patterns. The implementation of *fault-dropping* is not mandatory in this phase, but will be required in your final phase as one of the methods to accelerate your ATPG process.
- The new menu option is `"PFS"` which has three string arguments: input test pattern, input fault list, and output file.

Random test generation (RTG) and fault coverage (FC) calculation: For most combinational circuits, a small set of randomly generated test vectors can detect a substantial percentage of single stuck at faults. For most combinational circuits, this percentage lies in the 60–90% range [1]. A set of randomly generated tests of a given size typically provides

lower fault coverage compared to a set with an equal number of deterministically generated test vectors. Therefore, the main advantage of RTG is to detect many *easy* faults by just running fault simulation, and avoiding the expensive deterministic ATPG algorithms (such as D-Algorithm and PODEM). However, the rapid increase of fault coverage by running RTG will eventually saturate. The test process should detect this saturation and then run deterministic approaches for the remaining faults. This concept is illustrated in Fig. 1.
Follow these steps for the implementation of RTG:

- Fault coverage is defined as the ratio of all detected faults to all faults. The fault coverage starts from zero and increases by running fault simulation for different test patterns.
- The new menu option is `"RTG"` which has 4 arguments with this sequence: the number of total random test patterns to generate (n_{tot}), the frequency of FC report (n_{TFCR}), the name of the test-pattern report file, and finally the name of the FC report file.
- To draw a figure similar to Fig. 1, you need to report FC every n_{TFCR} test vectors. Just a note that you may have access to the current FC ratio at every step, however, as we will see in the next phase for detection of saturation, we may need a report for every n_{TFCR} patterns. Corner cases are $n_{TFCR} = 1$ or $n_{TFCR} = n_{tot}$. You can assume that n_{tot} is always a factor of n_{TFCR}.
- In this phase, you can avoid checking the random patterns being unique (repetition is accepted). However, make sure that you use a different seed for your generator function; a possible option is using time as a seed.
- The test-pattern report file has a similar format to the previous input files.
- The format of the FC report file is very simple, each line has the FC value after running for n_{TFCR} new test patterns. The values should be reported as percentage, with two decimal point accuracy and no % sign ("65.34").

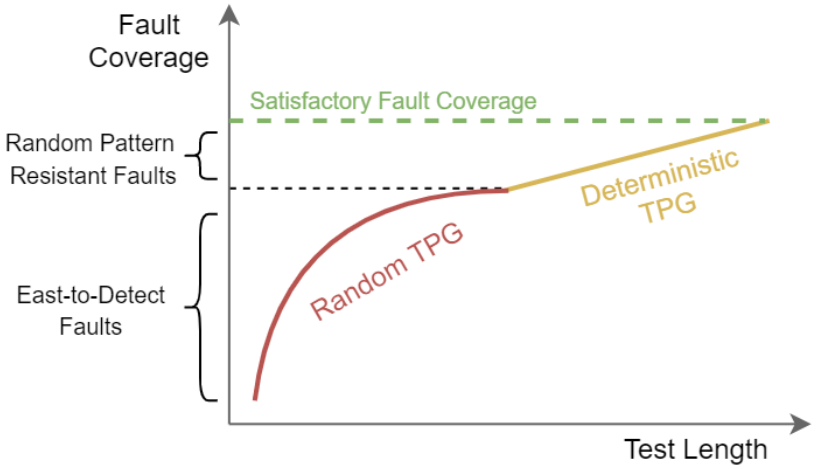

Figure 1: Fault coverage report by running RTG followed by ATPG

General guidelines for the simulator and submission:

- This project requires all the team members to be involved. Please read the description first and understand the requirements. Then divide the tasks among yourselves. Use Git!
- You need to discuss with your teammates about a good choice of data structure for representing a single fault and a fault list. When designing your data structures, consider the final ATPG simulator. We strongly suggest exploring different data structures, specially ones in C++ *standard template library* (STL) such as `vector` and `list`. Moreover, we encourage using the capabilities of *object oriented programming* (OOP) by designing meaningful classes for easier implementation of the next phase of the project, `D-Algorithm` and `PODEM`.
- We will NOT call `LEV` in this phase. If levelization is required for one of your operations, you are responsible to make sure your circuit had been already levelized.
- Starting this phase, file names can have an address and may not necessarily be in the same folder as your simulator executable file. For example, we may read a circuit with `READ ../../circuits/c17.ckt`). This also applies to output files.
- You can have different source code (`.c` or `.cpp`) or header (`.h`) files. However, you are responsible for providing on shell script `"compile.sh"` for compiling your project. The name of the executable file should be `"simulator"`.
- Please make sure your code can be compiled and executed in `viterbi-scf` servers.
- Please make a simple text file named `"members.txt"`, and only put the USC netID (username) of your group members in separate lines (do NOT include `@usc.edu` suffix).

- All files should be in one ".zip" folder named: "EE658_Phase2_⟨GroupNumber⟩". Only one person per group should submit the project.
- Our script will "unzip" your submission, run the "compile.sh" script and run the simulator with pre-defined commands. The complete sequence of commands and arguments are written down in a command file. By using the pipe "./simulator < command.txt", we specify the execution of the application ./simulator to read its information not from keyboard, but from the command.txt file. Our script compares the new generated files and compares them with the golden results.
- Use the discussion forum for possible doubts or questions, or visit office hours. Please avoid sending personal emails to the course staff.