Fault Simulation

Group Project Phase #2

The second phase of the group project focuses on the implementation of two different fault simulation methods: *deductive fault simulation* (DFS) and *parallel fault simulation* (PFS), applying techniques to reduce the fault list in a netlist, and also a few modifications to your previously developed logic simulator.

1 Random Test Pattern Generator (RTPG)

Using this command enables our simulator to generate random test patterns and store them in a file.

```
[cmd$] RTPG <tp-count> <mode:b|t> <tp-file>
```

- The *test pattern* (tp) file will be used in many different commands, and has the following format (referred to *as tp-file-format)*. The first line is introducing the PI nodes by their IDs (e.g. an integer in self format). IDs are separated by comma and sorted in increasing order. Each of the following lines is the description of one tp in which the values are ordered based on the first line and are separated by comma.

```
1,2,3,6,7
0,1,1,0,0
1,0,x,0,x
```

A sample test pattern description (tp-file-format) for c17

- Mode can be either 'b' (binary) or 't' (ternary). In binary format, input values can be either 0 or 1, while in ternary, 'x' can be also considered a valid value.
- 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.

2 Logic Simulation

In phase-1, your logic simulator was called with a single binary tp.In this phase, you need to modify your implementation so `"LOGICSIM"` can run on a set of multiple input patterns.

```
[cmd$] LOGICSIM <input-file> <output-file>
```

- An input line can have a value in {0, 1, **x**}. This is different to the previous implementation where inputs were considered to be binary.
- The input and output files both follow the tp-file-format.

3 Reduced Fault List

Reduced fault list (RFL) refers to a list of faults that is reduced by applying the *check point theorem* (CPT).

```
[cmd$] RFL <output-fl-file>
```

- The output file format (referred to as *fl-file-format*) is a simple text file that contains a list of faults. Each row shows a single stuck-at fault with this format: `<NODE-ID>@<FAULT>`.

4 Deductive Fault Simulation (DFS)

The DFS simulator simply has test patterns (one or many) as input and reports all the detectable (and not just the RFL) faults using these test patterns.

```
[cmd$] DFS <input-tp-file> <output-fl-file>
```

- Our suggestion is to first implement a method (maybe calling it dfs_single) that gets a single tp as an input (maybe using vectors?) and runs DFS on it. After verification of the results of the single method, implement a wrapper that calls the previous method for each of the test patterns (maybe calling it DFS_multi).
- The format of the input test pattern file is tp-file-format (only binary values), and the output file follows the fl-file-format.

5 Parallel Fault Simulation (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.

```
[cmd$] PFS <input-tp-file> <input-fl-file> <output-fl-file>
```

- Similar to DFS, our suggestion is to first implement a method to run PFS for a single tp, verify, and then write a wrapper to apply all tps in the input file.
- The format of input tp file tp-file-format. The input fault list and the output detected faults both follow the fl-file-format.
- 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.
- Note: don't forget to test your PFS implementation by running large circuits, so the total number of faults would be larger than the word length of your computer. Don't rely on the small circuits used for auto-grading on GitHub.

6 Fault Coverage (FC) with Random Patterns

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.

```
[cmd$] TPFC <tp-count> <freq> <output-tp-file> <report-file>
```

- The new menu option is `"TPFC"` 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.

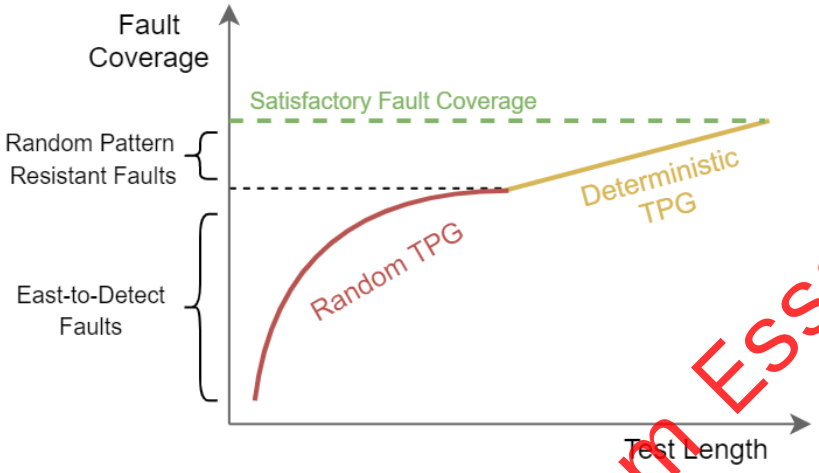

Figure 1: Fault coverage report by running RTG followed by ATPG

- 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 simulations for different test patterns.
- 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} pattern. 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 output test pattern file follows tp-file-format. 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 percentages, with two decimal point accuracy and no % sign, similar to the box below.

```
24.60
45.43
58.09
66.44
70.10
73.18
75.57
```

7 General Guidelines

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. Just a few notes:

- Use Git! All members of the team should be able to use basic git commands.
- Discuss with your teammates a good choice of data structure for representing a node, a single fault and a fault list.
- We strongly suggest exploring different data structures, especially 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`.
- If levelization process is a prerequisite for one operation (e.g. PFS), you are responsible to make sure this is handled within the operation, and NOT by the operator running `"LEV"` before running another command. Note: `"LEV"` is still a command that needs to work properly.
- File names can include their path 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 should have different source code (`.cpp`) and header (`.h`) files for reading the input circuit, LEV, LSIM, DFS, PFS, RTPG, etc. You are also responsible for providing the `CMakeLists.txt` 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.
- Use the discussion forum for possible doubts or questions or visit office hours. Please avoid sending personal emails to the course staff.
- Do not forget to add the descriptions of your new commands to HELP.