Final Project

1 Project Description

You will implement an automatic test generator for single stuck-at faults. It will include (a) fault collapsing; (b) a circuit simulation algorithm, and (c) a test generation algorithm. We assume a single-stuck-fault (SSF) model. ***Python is strictly required for this project for the ease of testing of your code.*** The following are the requirements for the program:

1. The program must run in an interactive mode.
2. The input to the program is a gate-level net-list.
3. The generator must parse the gate-level netlist and then report the following:
   (a) The final list of fault classes after performing fault collapsing
   (b) For each detectable fault, show the list of test vectors that will detect such faults.
   (c) The list of undetectable faults (if any).
   (d) The list of detectable faults for a given test vector revealed by circuit simulation.

Figure 1 shows the general flow of the test pattern generator.

On program invocation, the following interactive menu should be presented to the user:

```
[0] Read the input net-list
[1] Perform fault collapsing
[2] List fault classes
[3] Simulate
[4] Generate tests (D-Algorithm)
[5] Generate tests (PODEM)
[6] Generate tests (Boolean Satisfaibility)
[7] Exit
```

These options are explained with more details below.

- First it reads the circuit (Option `0`) and generates appropriate data structures.
- Option `1` performs fault collapsing and creates the fault classes.
- By selecting `2`, the found fault classes are displayed, one for each line.
- Option `3` simulates the circuit. It takes an input test vector and a set of faults, and show faults propagated to one of the outputs at the end of simulation. If no faults are provided, it simulates the circuit with the provided test vector to show the output values.
- Options `4` to `6` provide different test generation algorithms.

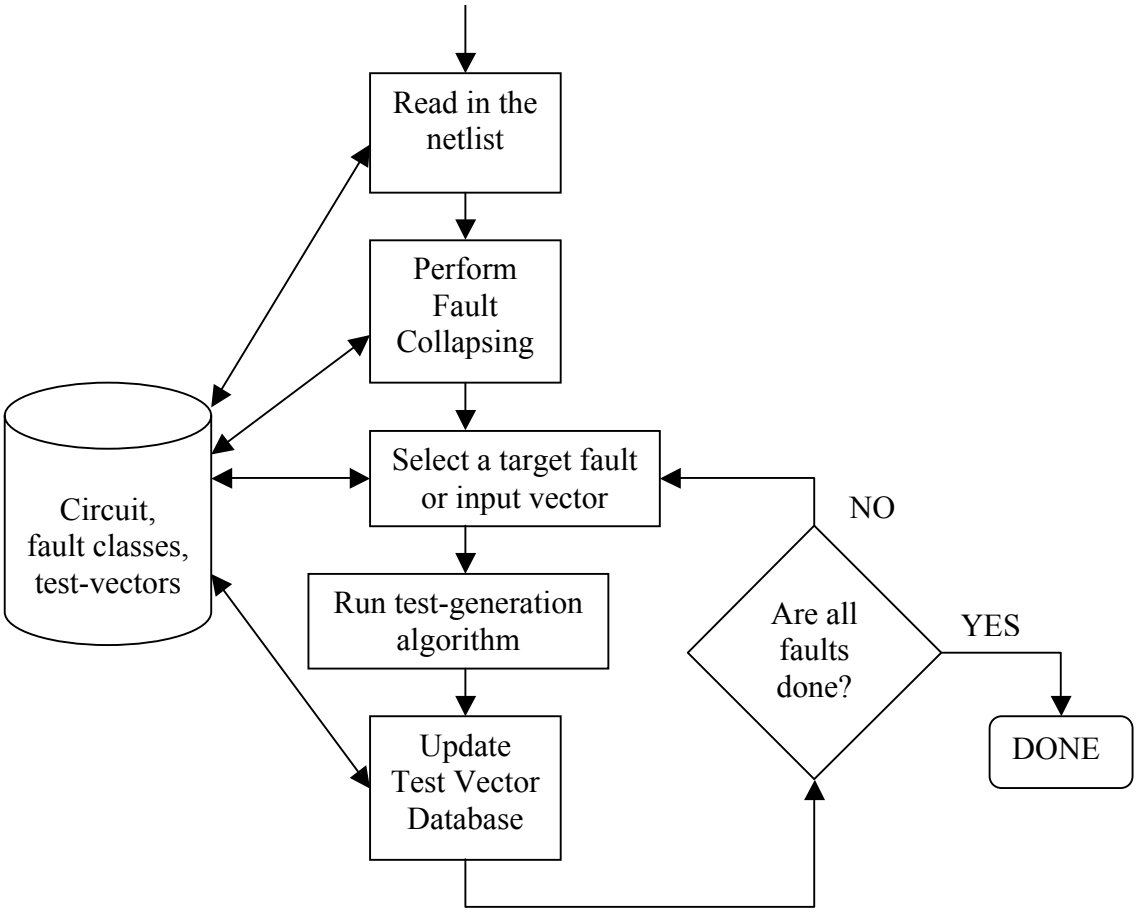

Figure 1: High level task flow for the ATPG.

Teaming. You can work in a team with **up to** 2 people.

Evaluation

This project carries 35% of the final grade, and it is evaluated as follows.

1. **Correctness** Whether your code produces correct results for a set of test cases. A set of 5 circuits will be provided to you. You make sure that your code should run correctly for them. Additional 5 circuits (not provided to you) will be run to check your code during your project demonstration. So make sure your code implements the test generation algorithm correctly instead of specializing your code for those provided test circuits.
2. **Coding style** Your code should have good *Readability* and *Modularity*.
3. **Error handling** Your code should exits gracefully in presence of wrong input entry (such as mistype, wrong option, non-existent file name, etc.)

1.1 Deliverables

1. Submit a final report summarizes your project, *i.e.* code structure, compile instructions, results from applying your generator to the five benchmark circuits.
2. Upload archived code in a single zipped file including a above final report, Makefile, and other necessary files.

1.2 Input Netlist format

The input netlist (self-explanatory) will be given in a format illustrated by the following example. Note that "$" denotes the start of a comment. An example is given at the end of document or by this link.

2 Project Assignment

[**CIS 4930**] Complete the following menu items: $1, 2, 3, 4, 5, 7$.

[**CIS 6930**] Complete all seven menu items.

2.1 Additional Information for Item 6

To implement option `6` in the menu, you need to convert a test generation program into the corresponding Boolean satisfiability problem, and then use an existing Boolean SAT solver to find the tests. Some of the modern efficient solvers can be found at `http://www.satlive.org/solvers/`. *Make sure you pick one of the CDCL solvers.* Another interesting and good solver is the `Z3` solver which also has a Python distribution (`https://pypi.org/project/z3-solver/`).

3 Additional Projects for CIS 6930

For students enrolled in CIS 6930, you have options to choose one of ideas below for the final project.

The first idea is to detailed study modern debug infrastructures embedded in processors and systems-on-chip design and understand they they facilitate silicon validation (*aka* post-silicon validation) and debug. Some prominent examples include *ARM CoreSight* and *Intel TraceHub*.

Another project idea is to perform a slightly deeper study on using machine learning to help test generation. In this project, you will not be limited to test generation for physical

fault. Instead, you will consider the test generation problem for functional testing, *i.e.* checking if logic functionality of a design is correct. A key challenge for functional testing is to reduce test set while achieve a satisfying coverage for efficiency. As the starting point, you should refer to Section 7 of *"Machine learning for Electronic Design Automation: A survey"* for some recent work on this subject. A more specific example topic is hardware Trojan detection using reinforcement learning.

- From the above three subjects, study of ARM Coresight, study of Intel TraceHub, and study of test generation with machine learning, you will select one subject.
- Search for related articles published recently and other necessary information, study them, and write a technical report on your findings. This technical report should be written as you would write a technical paper for a conference in terms of format and structure.
- At the end of the semester, a 30-minute oral exam will be given to every student. During the oral exam, you make a presentation about your study and answer questions raised by the instructor.
- Evaluation of your final project is based on the quality of your technical report and the oral exam.

Teaming. This is an individual project.

4 Input Netlist format – An Example

```
$c17 iscas example (to test conversion program only)
$-----------------------------------------------
$
$  total number of lines in the netlist ..............    17
$  simplistically reduced equivalent fault set size =     22
$        lines from primary input  gates .......     5
$        lines from primary output gates .......     2
$        lines from interior gate outputs ......     4
$        lines from **     3 ** fanout stems ...     6
$
$        avg_fanin  =  2.00,     max_fanin  =  2
$        avg_fanout =  2.00,     max_fanout =  2
$
1gat                                    $... primary input
2gat                                    $... primary input
3gat                                    $... primary input
6gat                                    $... primary input
7gat                                    $... primary input
                                        $... primary input
$
$

22gat                                   $... primary output
23gat                                   $... primary output
                                        $... primary output
$
$
$       Output  Type    Inputs...
$       ------  ----    ---------
        10gat   nand    1gat    3gat
        11gat   nand    3gat    6gat
        16gat   nand    2gat    11gat
        19gat   nand    11gat   7gat
        22gat   nand    10gat   16gat
        23gat   nand    16gat   19gat
```