UIC Fall 2020 ECE 464 Project 2: Fault Coverage Comparison of Various PRPGs

I. Introduction

Pseudo-Randomly generated test vectors can be obtained cheaply without running ATPG. But are they any good in covering all the faults in a circuit? The goal of this project is to examine the fault coverage of pseudo-random patterns, including counter and LFSRs.

In this project, you will continue to build your Python fault simulator, so that it becomes a powerful tool to help investigate the fault coverage (or, quality) of pseudo-randomly generated patterns. Your Python program will need to add the following main components:

- A) generate a full fault list and the ability to simulate any of them (including the new kind of fault: gate-input stuck-at faults);
- B) perform "batch" fault simulation on a single test vector - essentially to see how many (and which) faults among a list can be detected by a given test vector;
- C) perform "batch" fault simulation from a list of test vectors - starting with a full fault list, apply each test vector in order, and reporting how many faults have been covered / remained after applying each of the test vectors;
- D) implement various PRPGs (Pseudo-Random Pattern Generator) to build test vector lists, and perform the above experiments on some large circuit benchmarks.

You will build your Python simulator as a tool to collect many data and present the results to provide quantitative answers to the original question: **How good are the fault coverage of (pseudo-)randomly generated test vectors?**

II. Python simulator functionalities

A. Full Fault List Generation

Given a circuit bench file, produce the full fault list. For example:

```
# circuit.bench
INPUT(a)
INPUT(b)
INPUT(c)
OUTPUT(k)
f = NOT(b)
g = AND(a, b)
h = AND(f, c)
k = OR(g, h)
```

With this circuit your program should be able to generate the full list of single-stuck-at faults (without repetition):

```
a-0, a-1, b-0, b-1, c-0, c-1, k-0, k-1, f-0, f-1,
f-b-0, f-b-1, g-0, g-1, g-a-0, g-a-1, g-b-0, g-b-1,
h-0, h-1, h-f-0, h-f-1, h-c-0, h-c-1, k-g-0, k-g-1,
k-h-0, k-h-1 ...a total of 28 faults
```

B. Fault Coverage of a single test vector

Your program should be able to report which faults among the full fault list can be detected by any given single test vector.

For example, by iterating through all the possible 8 input combinations, your program should be able to derive for each test vector, all the faults that can be detected:

```
000:c-1,k-1,g-1,h-1,h-c-1,k-g-1,k-h-1
001:b-1,c-0,k-0,f-0,f-b-1,h-0,h-f-0,h-c-0,k-h-0
010:a-1,k-1,g-1,g-a-1,h-1,k-g-1,k-h-1
011:a-1,b-0,k-1,f-1,f-b-0,g-1,g-a-1,h-1,h-f-1,k-g-1,k-h-1
100:b-1,c-1,k-1,g-1,g-b-1,h-1,h-c-1,k-g-1,k-h-1
101:c-0,k-0,f-0,f-b-1,h-0,h-f-0,h-c-0,k-h-0
110:a-0,b-0,k-0,g-0,g-a-0,g-b-0,k-g-0
111:a-0,k-0,g-0,g-a-0,g-b-0,k-g-0
```

C. Fault Coverage of a list of test vectors

For a given circuit with its full fault list F, given a list of test vectors (say $t_0, t_1, t_2, ..., t_7$), your program should be able to perform fault simulation according to the order of the tv list according to the following algorithm flow:

```
• $T$ is the tv list, $F$ is the fault list
• iterate $\forall t \in T$ :
    – $D = \emptyset$
    – iterate $\forall f \in F$ :
          apply $t$ to $f$:
          if $t$ detects $f$, $D = D + \{f\}$
    – report $t$ covers $D$
    – $F = F - D$
```

In the end, your program should be able to show the number / list of faults covered by each of the test vectors, and the remaining number / list of faults.

For example, the process of using all 8 tvs on the small benchmark is the following:

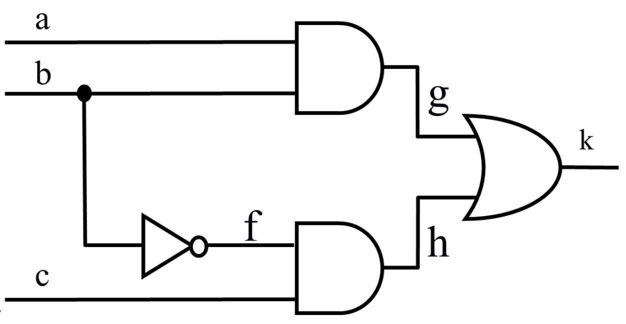

Fig. 1. Example of a small circuit.

```
INPUT(a)
INPUT(b)
INPUT(c)
OUTPUT(k)
f = NOT(b)
g = AND(a, b)
h = AND(f, c)
k = OR(g, h)

--- derived full fault list: ---
a-0, a-1, b-0, b-1, c-0, c-1, k-0, k-1, f-0, f-1,
f-b-0, f-b-1, g-0, g-1, g-a-0, g-a-1, g-b-0, g-b-1,
h-0, h-1, h-f-0, h-f-1, h-c-0, h-c-1, k-g-0, k-g-1,
k-h-0, k-h-1, ...a total of 28 faults.

tv list has 8 tvs:
['000', '001', '010', '011',
'100', '101', '110', '111']

Now applying 000 to the remaining 28 fault:
['a-0', 'a-1', 'b-0', 'b-1', 'c-0', 'c-1', 'k-0',
'k-1', 'f-0', 'f-1', 'f-b-0', 'f-b-1', 'g-0', 'g-1',
'g-a-0', 'g-a-1', 'g-b-0', 'g-b-1', 'h-0', 'h-1',
'h-f-0', 'h-f-1', 'h-c-0', 'h-c-1', 'k-g-0', 'k-g-1',
'k-h-0', 'k-h-1']...

Fault a-0 UNdetected by 000
Fault a-1 UNdetected by 000
Fault b-0 UNdetected by 000
Fault b-1 UNdetected by 000
Fault c-0 UNdetected by 000
Fault c-1 detected by 000
Fault k-0 UNdetected by 000
Fault k-1 detected by 000
Fault f-0 UNdetected by 000
Fault f-1 UNdetected by 000
Fault f-b-0 UNdetected by 000
Fault f-b-1 UNdetected by 000
Fault g-0 UNdetected by 000
Fault g-1 detected by 000
Fault g-a-0 UNdetected by 000
Fault g-a-1 UNdetected by 000
Fault g-b-0 UNdetected by 000
Fault g-b-1 UNdetected by 000
Fault h-0 UNdetected by 000
Fault h-1 detected by 000
Fault h-f-0 UNdetected by 000
Fault h-f-1 UNdetected by 000
Fault h-c-0 UNdetected by 000
Fault h-c-1 detected by 000
Fault k-g-0 UNdetected by 000
Fault k-g-1 detected by 000
Fault k-h-0 UNdetected by 000
Fault k-h-1 detected by 000

tv 000 detects the following 7 faults:
['c-1', 'k-1', 'g-1', 'h-1', 'h-c-1', 'k-g-1',
'k-h-1'] with 21 faults remaining.

Now applying 001 to the remaining 21 faults:
['a-0', 'a-1', 'b-0', 'b-1', 'c-0', 'k-0', 'f-0',
'f-1', 'f-b-0', 'f-b-1', 'g-0', 'g-a-0', 'g-a-1',
'g-b-0', 'g-b-1', 'h-0', 'h-f-0', 'h-f-1', 'h-c-0',
'k-g-0', 'k-h-0']...

Fault a-0 UNdetected by 001
Fault a-1 UNdetected by 001
Fault b-0 UNdetected by 001
Fault b-1 detected by 001
Fault c-0 detected by 001
Fault k-0 detected by 001
Fault f-0 detected by 001
Fault f-1 UNdetected by 001
Fault f-b-0 UNdetected by 001
Fault f-b-1 detected by 001
Fault g-0 UNdetected by 001
Fault g-a-0 UNdetected by 001
Fault g-a-1 UNdetected by 001
Fault g-b-0 UNdetected by 001
Fault g-b-1 UNdetected by 001
Fault h-0 detected by 001
Fault h-f-0 detected by 001
Fault h-f-1 UNdetected by 001
Fault h-c-0 detected by 001
Fault k-g-0 UNdetected by 001
Fault k-h-0 detected by 001

tv 001 detects the following 9 faults:
['b-1', 'c-0', 'k-0', 'f-0', 'f-b-1', 'h-0', 'h-f-0',
'h-c-0', 'k-h-0'] with 12 faults remaining.

Now applying 010 to the remaining 12 faults:
['a-0', 'a-1', 'b-0', 'f-1', 'f-b-0', 'g-0', 'g-a-0',
'g-a-1', 'g-b-0', 'g-b-1', 'h-f-1', 'k-g-0']...

Fault a-0 UNdetected by 010
Fault a-1 detected by 010
Fault b-0 UNdetected by 010
Fault f-1 UNdetected by 010
Fault f-b-0 UNdetected by 010
Fault g-0 UNdetected by 010
Fault g-a-0 UNdetected by 010
Fault g-a-1 detected by 010
Fault g-b-0 UNdetected by 010
Fault g-b-1 UNdetected by 010
Fault h-f-1 UNdetected by 010
Fault k-g-0 UNdetected by 010

tv 010 detects the following 2 faults:
['a-1', 'g-a-1'] with 10 faults remaining.

Now applying 011 to the remaining 10 faults:
['a-0', 'b-0', 'f-1', 'f-b-0', 'g-0', 'g-a-0',
'g-b-0', 'g-b-1', 'h-f-1', 'k-g-0']...

Fault a-0 UNdetected by 011
Fault b-0 detected by 011
Fault f-1 detected by 011
Fault f-b-0 detected by 011
Fault g-0 UNdetected by 011
Fault g-a-0 UNdetected by 011
Fault g-b-0 UNdetected by 011
Fault g-b-1 UNdetected by 011
Fault h-f-1 detected by 011
Fault k-g-0 UNdetected by 011

tv 011 detects the following 4 faults:
['b-0', 'f-1', 'f-b-0', 'h-f-1']
with 6 faults remaining.

Now applying 100 to the remaining 6 faults:
['a-0', 'g-0', 'g-a-0', 'g-b-0', 'g-b-1', 'k-g-0']...

Fault a-0 UNdetected by 100
Fault g-0 UNdetected by 100
Fault g-a-0 UNdetected by 100
Fault g-b-0 UNdetected by 100
Fault g-b-1 detected by 100
Fault k-g-0 UNdetected by 100

tv 100 detects the following 1 faults:
['g-b-1'] with 5 faults remaining.

Now applying 101 to the remaining 5 faults:
['a-0', 'g-0', 'g-a-0', 'g-b-0', 'k-g-0']...
Fault a-0 UNdetected by 101
Fault g-0 UNdetected by 101
Fault g-a-0 UNdetected by 101
Fault g-b-0 UNdetected by 101
Fault k-g-0 UNdetected by 101

tv 101 detects the following 0 faults:
[] with 5 faults remaining.

Now applying 110 to the remaining 5 faults:
['a-0', 'g-0', 'g-a-0', 'g-b-0', 'k-g-0']...
Fault a-0 detected by 110
Fault g-0 detected by 110
Fault g-a-0 detected by 110
Fault g-b-0 detected by 110
Fault k-g-0 detected by 110

tv 110 detects the following 5 faults:
['a-0', 'g-0', 'g-a-0', 'g-b-0', 'k-g-0']
with 0 faults remaining.

Now applying 111 to the remaining 0 faults:
[]...

tv 111 detects the following 0 faults:
[] with 0 faults remaining.

Final UNDETECTED: 0 faults: []
```

As you can deduce, the fault coverage result depends on the order of the tv list. For example, the above process goes through a list of 8 test vectors in the order of 000 to 111, and its coverage of the full fault list can be summarized in the following order:

```
000:c-1,k-1,g-1,h-1,h-c-1,k-g-1,k-h-1
001:b-1,c-0,k-0,f-0,f-b-1,h-0,h-f-0,h-c-0,k-h-0
010:a-1,g-a-1
011:b-0,f-1,f-b-0,h-f-1
100:g-b-1
101:
110:a-0,g-0,g-a-0,g-b-0,k-g-0
111:
```

However, when the tv list is applied in reversed order (starting with 111, going down to 000), you will see a different result, summarized below:

```
111:a-0,k-0,g-0,g-a-0,g-b-0,k-g-0
110:b-0
101:c-0,f-0,f-b-1,h-0,h-f-0,h-c-0,k-h-0
100:b-1,c-1,k-1,g-1,g-b-1,h-1,h-c-1,k-g-1,k-h-1
011:a-1,f-1,f-b-0,g-a-1,h-f-1
010:
001:
000:
```

This captures the dynamics (and complexity) in applying test vectors in such a process, if your goal is to come up with the minimum tv list to cover the maximum number of faults.

D. *TV list generation via PRPGs*

For small circuits with PI width n that is manageable, we can afford to generate and use all 2^n test vectors. For large circuits, we'll have to rely on a small set of tv generated pseudo-randomly. Here, we'll fix the size of tv list to 100.

When a tv is applied to the circuit, the leftmost bit should be assigned to the first input (for example $t_0 = 011$ applied to the above small circuit should have `abc` = 011)

*1) **n-bit Counter**:* For any given circuit with PI of width n, with any given starting seed, assuming using a counter for PRPG, your program should be able to generate 100 test vectors (with seed as t0), or 2^n when $2^n < 100$.

For example, circuit `c499.bench` has $n = 41$ inputs. A user should be able to set seed via a hex input. For example, If seed is set to `0x123456789abc`, this (more than the 41-bit input) should be expanded into 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100, and the first 41 bits will be used as tv0 to assign to each of the PI's, with 0, 0, 0, 1, going to the first, second, third, forth input of the circuit bench, etc:
t_0: 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1
t_1: 0001 0010 0011 0100 0101 0110 0111 1000 1001 1011 0
t_2: 0001 0010 0011 0100 0101 0110 0111 1000 1001 1011 1
t_3: 0001 0010 0011 0100 0101 0110 0111 1000 1001 1100 0
... all the way to t_{99}.

*2) **8-bit LFSRs**:* Here, we assume as many as possible 8-bit one-to-many LFSRs are used to feed all the PI's of a circuit. The first LFSR will have its LSB (left most bit) feeding to the circuit's first input. All the LFSR's will have the same configuration (as specified below). However, they will be seeded differently, according to a global seeding scheme: seed = `0x123456789abcdef0` (if this is not large enough to feed a circuit's input, repeat the same pattern for as many times as needed.)

Similar to the above counter case, your program should be able to generate 100 (or 2^n, whichever is smaller) test vectors, from the initial seed, and use this tv list to perform fault simulation and obtain the fault coverage.

2a) 8-bit LFSRs with no taps (shifter)
$h_0 : h_7 = 10000000$
2b) 8-bit LFSRs with taps at 2, 4, 5
$h_0 : h_7 = 10101100$
2c) 8-bit LFSRs with taps at 2, 3, 4
$h_0 : h_7 = 10111000$
2d) 8-bit LFSRs with taps at 3, 5, 7
$h_0 : h_7 = 10010101$

III. UPDATED - RESULTS

A. *UI and Functionality*

Input: your program should allow the user to specify:

1) Which circuit to run.
2) For large circuit with more than 6 PI's, which of the 5 LFSR options to apply.
3) A seed in hex.

Output: showing on screen the detail information such as the ones shown on page 2.

B. *Fault coverage report via csv files*

For a specific configuration, your program should generate the following two csv files:

1) a `circuit_all.csv` file to show the fault coverage per single tv, for example:
```
000,c-1,k-1,g-1,h-1,h-c-1,k-g-1,k-h-1
001,b-1,c-0,k-0,f-0,f-b-1,h-0,h-f-0,h-c-0,k-h-0
```

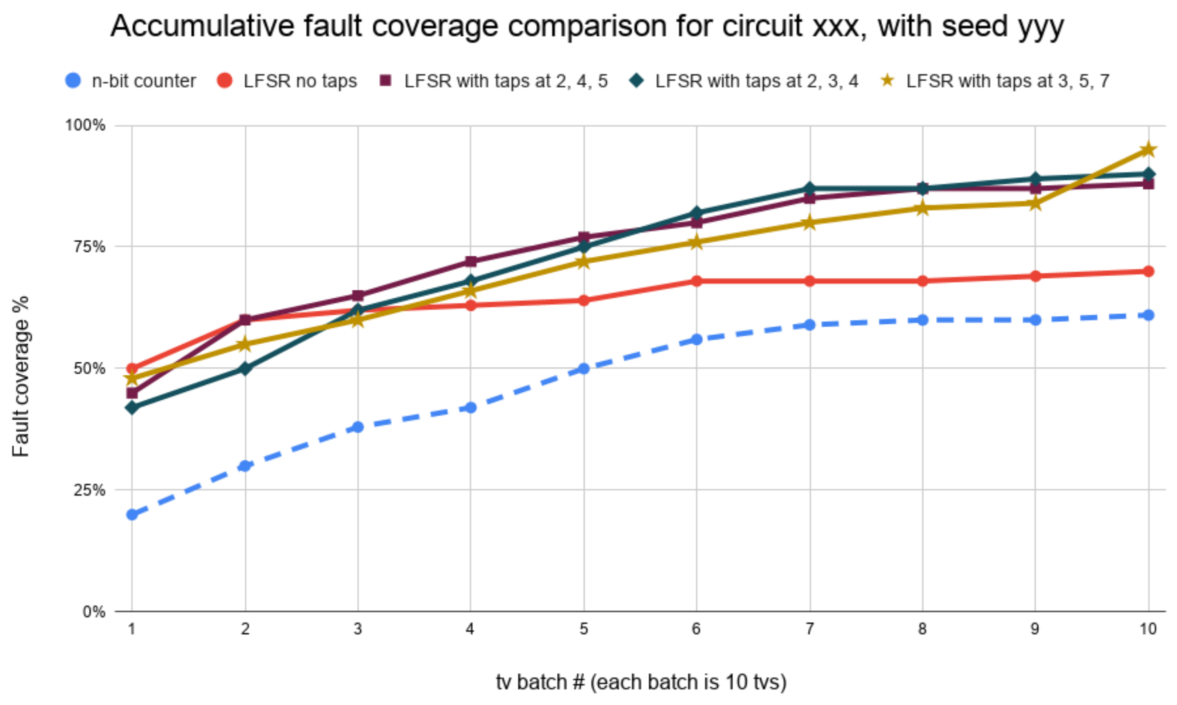

Fig. 2. Example of a comparison plot of a given seed.

```
010,a-1,k-1,g-1,g-a-1,h-1,k-g-1,k-h-1
011,a-1,b-0,k-1,f-1,f-b-0,g-1,g-a-1,h-1,h-f-1,k-g-1,k-h-1
100,b-1,c-1,k-1,g-1,g-b-1,h-1,h-c-1,k-g-1,k-h-1
101,c-0,k-0,f-0,f-b-1,h-0,h-f-0,h-c-0,k-h-0
110,a-0,b-0,k-0,g-0,g-a-0,g-b-0,k-g-0
111,a-0,k-0,g-0,g-a-0,g-b-0,k-g-0
```

2) a `circuit_list.csv` file to show the fault coverage according to a tv list, for example:

```
000:c-1,k-1,g-1,h-1,h-c-1,k-g-1,k-h-1
001:b-1,c-0,k-0,f-0,f-b-1,h-0,h-f-0,h-c-0,k-h-0
010:a-1,g-a-1
011:b-0,f-1,f-b-0,h-f-1
100:g-b-1
101:
110:a-0,g-0,g-a-0,g-b-0,k-g-0
111:
```

C. *Fault coverage comparison plot*

For a given large circuit, you should show via a plot, a comparison of fault coverage results of the 5 configurations (1 counter, 4 LFSRs).

Plot setting:

- On the x-axis, there should be 10 data points, representing 10 “tv batches”. Each batch consists of 10 tvs.
- The y-axis represents the accumulated fault coverage of all the tv batches so far.

An example plot (for illustration only) is shown in Fig. 2. This shows the result of applying the same seed (yyy) to generate 100 test vectors using 5 PRPGs on a circuit (xxx). The accumulated fault coverage are measured after every 10 tvs (thus there are 10 batches, each is 10 tv).

D. *Extra Credit: average plot*

Produce the comparison plot with the average result of 1000 runs, each time a random seed is used.

IV. Report and Submission Details

A. *File Submission on Blackboard*

1) a PDF report file
2) Your python program file.
3) Result csv files for circuit c17.bench
4) Result csv files for circuit c432.bench
5) Result csv files for another large bench file (select among c1355, c1908, c2670, c3540, c5315, c6288, c7552)

B. *PDF report content*

1) Introduction of the main functionality and implementation of your simulator.
2) Fault coverage comparison plot for c432.bench
3) Fault coverage comparison plot for another large bench file (select among c1355, c1908, c2670, c3540, c5315, c6288, c7552)
4) Discussion and conclusion: from the results, what is your evaluation of the performance of the 5 PRPGs? Are they good as test vector generators?

C. *Submission deadline*

Nov 8th 11:59pm on Blackboard, for both instructor evaluation and peer evaluation.