Testing and Reliability of Digital Systems

Project Report 2

ECE 464

Signory Somsavath
Jonathan Wacker
Chris Escobar

Submitted 9 November 2020

Introduction

The main functionality and implementation of the simulator.

Functionality Requirements

The main objective of this project is the ability to use a test vector list. When using the simulator the user is given options to choose between different modes which includes n-bit counter or LFSR(with different taps). Once the PRPG(Pseudo-Random Pattern Generator**)** has been selected a list will be created and displayed with test vectors. The purpose of this program is to generate a list full of faults at every node around every gate being SA-0(stuck-at-1) or SA-1(stuck-at-1). Once the program is able to generate the test vectors and fault list from its given circuit it would then simulate the faults withs its TV's(test vectors). It would then be able to display the results of whether each fault was detected with its current TV. Once finished we are able to recursively come up with the minimum TV to cover the maximum number of faults. By creating a temporary list that holds the total number of faults we are able to keep track of the total number of faults deducing the TV list. All while producing a CSV file of all the results

Gate Operation

Gate input faults were created within the class using "fgateSA" as a part of the node to tell the fault type and "fgate" to tell if its fault input into the gate. Putting each fault respectively in a list. It was able to input and output the correct values given by the users around each gate. This was all done by creating and setting variables within the node class.

N-bit Counter Operation

Using an n-bit counter allows us to take in a user input via hex which is converted into binary as a seed. We then check the number of inputs that the specified circuit has, and take the minimum of 2 to the power of circuit inputs or 100, since large circuits. We cap this at 100 because 2^{CI} can get very very large for big circuits and could take days or even weeks to simulate.

Code Snippet(s):

# -- N-bit counter

```
        else:
            # Create full batch of test vectors
            for i in range(loops):
                # To account for overflow, reset TV back to zero
                if(x == 2**hexNumLen):
                    x = 0
                # Generate test vector and add to list
                binString = bin(x)[2:].zfill(numInputNodes)
                vector_list_main.append(binString)

                # Increment TV by 1
                x += 1
```

LFSR Operation

For creating the LFSR with different taps. It takes in the user's input of hex and seeds that values into a TV of the first 8 bits. The binary number is first made long enough to match the length of the number of inputs by adding onto itself until the length exceeds or matches the number of input nodes. Any extra bits are then cut off if it is longer than the number of inputs.

We then execute the LFSR option that the user specified. For all of these cases, the logic used fundamentally the same, just the bit manipulation changes dependent on the choice selected. For example, if the user chooses “(a)8-bit LFSRs with no taps (shifter)”, the program will use the binary number of the number of nodes as the first seed, and call the shifting function. The shifting function will then return the new shifted 8-bit number, and continue appending itself to the string of binary numbers until the binary number length meets the length requirement. If the length of the string is too large, the string will be cut appropriately to match specified length. We then append this new binary number into the vector_list, use the next seed, and repeat the process either 2^{CI} times or 100 times (whichever one is smaller). This also works for the other 8-bit LFSR options, except the bit shifting includes the corresponding taps in the shifting functions.

Code Snippet(s):

```
# -- 8-bit LFSRs with no taps (shifter)
        if(operation_choice.lower() == 'a'.lower()):
            # - Initial TV
            prior_test_vector = bin(x)[2:].zfill(numInputNodes)
            vector_list_main.append(prior_test_vector)

            # Create full batch of test vectors
            for i in range(0,loops-1):
                new_test_vector = ""

                # - Generate full string of vector
                for j in range(int(len(prior_test_vector)/8)):
                    new_test_vector = new_test_vector + 
D_Two_A(prior_test_vector[j*8:(j+1)*8])

                # - If length does not match evenly with 8 input, last one uses 
less inputs
                if len(prior_test_vector)%8 != 0:
                    # Run LFSR with extra zeros to compensate  for lack of input
                    full_content = 
D_Two_A(prior_test_vector[(len(prior_test_vector) - 
len(prior_test_vector)%8):(len(prior_test_vector))].zfill(8))

                    # pinch off what is not needed for TV
                    new_test_vector = new_test_vector + 
full_content[0:len(prior_test_vector)%8]

                prior_test_vector = new_test_vector
                vector_list_main.append(new_test_vector)
```

```
# -- LFSRs with no taps (shifter) helper function
def D_Two_A(originalSeed):  # 8-bit LFSRs with no taps (shifter)
    seedChunk = originalSeed
    newSeed = ''

    newSeed += seedChunk[7]
    newSeed += seedChunk[0]
    newSeed += seedChunk[1]
    newSeed += seedChunk[2]
    newSeed += seedChunk[3]
    newSeed += seedChunk[4]
    newSeed += seedChunk[5]
    newSeed += seedChunk[6]

    return newSeed
```

UI

UI created supports the requested order of operation with additional options for screen cleanliness. First prompt is to ask the user what circuit is to be used per requirements with an additional option to show the circuit node list if desired. Depending on the number of inputs in the circuit is greater than 6 or has 6 or less inputs, the user will have the option to either pick an LFSR type with a seed of choice or the code will run through all the test vectors respectively. Csv files data can be printed on request but, regardless of choice, the final print will include the current accumulated test batch and the history from prior loops. Once 10 runs or all test vectors are reached then the script allows the user to rerun the batches but with a different LFSR and seed depending on input.

Operation of "Part A" of code

The code calls function full_coverage which return the array of faults. These faults are then output onto the shell.

full_coverage() uses the node_list, node.innames, and gateInputs to determine all of the possible faults in each circuit. The function first populates the list using all of the circuit inputs, stuck at 0 and stuck at 1 since those are always going to be given no matter what the circuit looks like. Then if takes the list of outputs, and uses this as the starting point for each new fault.

Within this, there is another loop which checks each of the current gates inputs that are not circuit inputs. Then for each of those gate inputs, we get "gateOutput-gateInput-x" where x is set to 0 and 1. This continues until all of the gates are traversed and each gate input has been covered as well. Before returning this list, we make sure that there are no duplicates by calling the helper function "remove_dup()", which is self explanatory from the function name. The final result is the final full fault list.

Code Snippet(s):

```
def full_coverage():
    gateList = [] #Gates Inputs for example g-'a'-1 or g-'b'-1
    outputGate = [] # Names of the Gate output 'g'-a-1
    allInputs = []
    faultLst = []
    i = 0
    j = 0
    k = 0
    for node in node_list:
        allInputs.append(node.name)
        for gateInput in node.innames:
            for target in node_list:
                if target.name == gateInput:
                    gateList.append(node.innames)
                    outputGate.append(node.name)
    while (j < len(allInputs)):
            a = ("{}-{}".format(allInputs[j],0))
            b = ("{}-{}".format(allInputs[j],1))
            faultLst.append(a),faultLst.append(b)
            j += 1
    while (i < len(outputGate)):
        k = 0
        while (k < len(gateList[i])):
            a = ("{}-{}-0".format(outputGate[i],gateList[i][k]))
            b = ("{}-{}-1".format(outputGate[i],gateList[i][k]))
            faultLst.append(a),faultLst.append(b)
            k += 1
        i +=1
    remove_dup(faultLst)
    return faultLst
#
...
#
# Helper Function # My Code Signory Somsavath
def remove_dup(x):
    i = 0
    while i < len(x):
        j = i +1
        while j < len(x):
            if x[i] == x[j]:
                del x[j]
            else:
                j+=1
        i += 1
```

Operation of “Part B” of code

The object is to associate all faults covered for each of the test vectors alone. This is done by comparing the simulation of a good circuit and a bad with a test vector and checking each associated with fault. This process populates an array with the content which is later used for printing and writing to a csv file.

Operation of “Part B2” of code

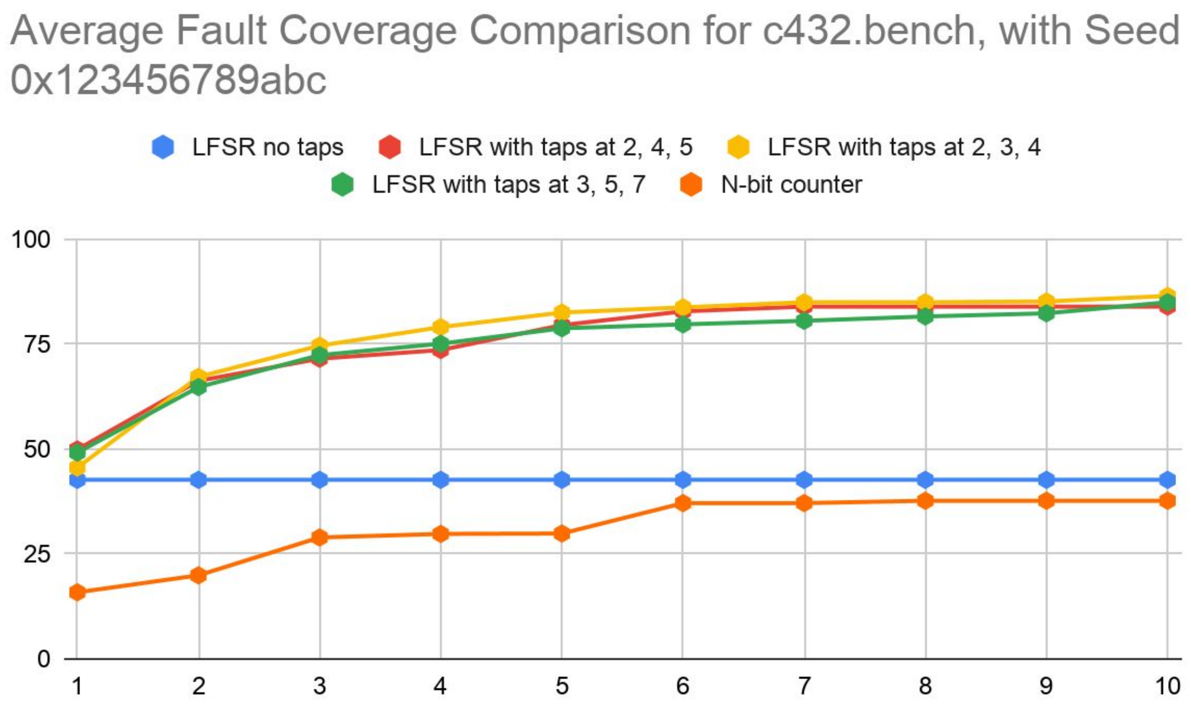
To create the best path to cover all faults, the script pulls in the information from “Part B” to recurs all possible paths for the shortest path with the most coverage. The function “locate_best_order” is used to achieve the objective. It first checks from the current list of available vectors and sees which one has the most coverage. After selecting the vector(s) that have the best current coverage for the remaining faults, the function then knows that there is still fault to be covered. From each test vector and remaining faults pool, the function is called again to see if there are any faults left to find. If there is no vector that covers any faults, then the path is complete and returned. If there is more than one, then there are multiple function calls to see which path is shortest. All paths are explored but the shortest path is passed back to the first call of the function. With the addition of test vectors that are not needed, the csv and print statements are done.

Operation of “Part C” of code

This simple section calculates the percentage of faults covered in the test batch. The number of runs and history document allows for the user to track how many batches have been done and easily copy the history of the document to add to a table.

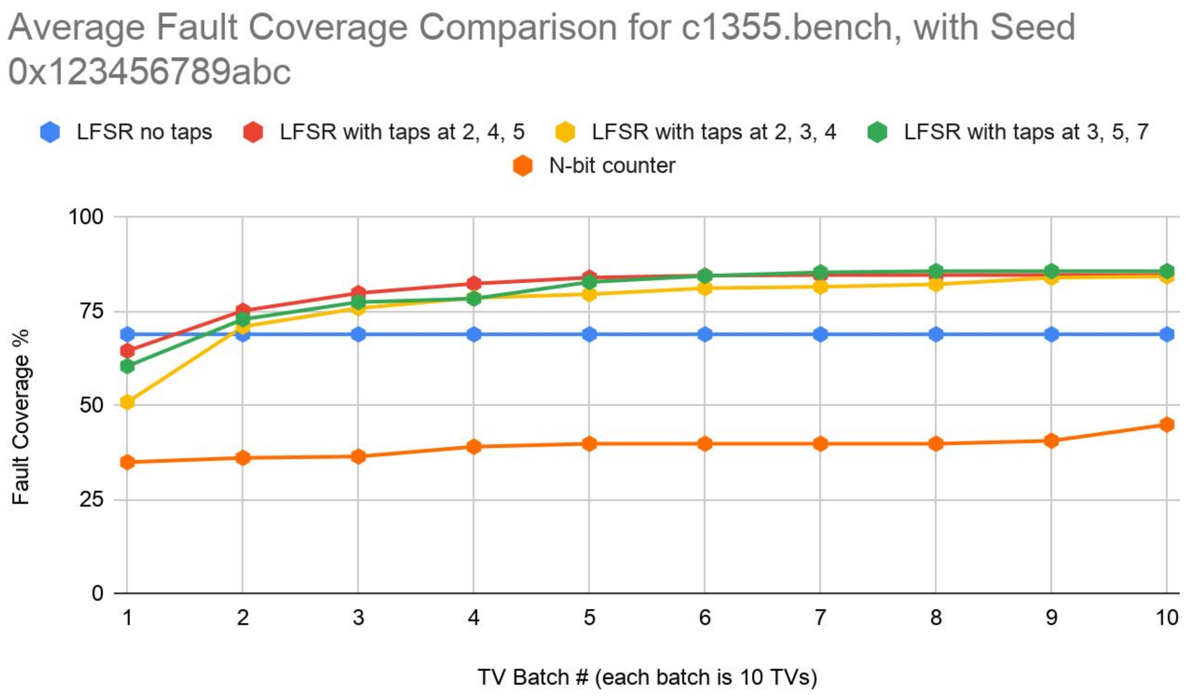
Fault Coverage Comparison Plot for c432.bench:

	1	2	3	4	5	6	7	8	9	10
LFSR no taps	42.66917293233083	42.66917293233083	42.66917293233083	42.66917293233083	42.66917293233083	42.66917293233083	42.66917293233083	42.66917293233083	42.66917293233083	42.66917293233083
LFSR with taps at 2, 4, 5	49.90601503759399	66.2593984962406	71.52255639097744	73.59022556390977	79.51127819548873	82.80075187969925	83.92857142857143	83.92857142857143	83.92857142857143	83.92857142857143
LFSR with taps at 2, 3, 4	45.58270676691729	67.19924812030075	74.62406015037594	79.04135338345864	82.5187969924812	83.7406015037594	84.9624060150376	84.9624060150376	85.15037593984962	86.46616541353383
LFSR with taps at 3, 5, 7	49.15413533834587	64.75563909774436	72.36842105263158	75.09398496240601	78.7593984962406	79.69924812030075	80.54511278195488	81.57894736842105	82.33082706766918	84.9624060150376
N-bit counter	15.88345864661654	19.924812030075188	28.947368421052634	29.793233082706767	29.887218045112785	37.12406015037594	37.12406015037594	37.68796992481203	37.68796992481203	37.68796992481203



Fault Coverage Comparison Plot for c1355.bench:

	1	2	3	4	5	6	7	8	9	10
LFSR no taps	68.86735311932163	68.86735311932163	68.86735311932163	68.86735311932163	68.86735311932163	68.86735311932163	68.86735311932163	68.86735311932163	68.86735311932163	68.86735311932163
LFSR with taps at 2, 4, 5	64.47607510599637	75.13628104179286	79.83040581465778	82.34403391883707	83.94912174439733	84.46396123561478	84.64566929133859	84.64566929133859	84.64566929133859	84.64566929133859
LFSR with taps at 2, 3, 4	50.847970926711085	70.9267110841914	75.80254391278012	78.58873410054512	79.52755905511812	81.13264688067837	81.49606299212599	82.16232586311327	83.91883706844337	84.25196850393701
LFSR with taps at 3, 5, 7	60.417928528164744	72.8649303452453	77.43791641429438	78.34645669291339	82.76801938219262	84.40339188370685	85.34221683827982	85.67534827377347	85.67534827377347	85.67534827377347
N-bit counter	34.887946698970325	36.03876438522108	36.402180496668684	39.00666262870987	39.794064203513024	39.794064203513024	39.794064203513024	39.794064203513024	40.58146577831617	44.881889763779526



Discussion

From the results, what is your evaluation of the performance of the 5 PRPGs? Are they good as test vector generators?

- LFSR no taps
    - This LFSR has immediate coverage with its test vectors. The period of 8 means that all unique test vectors are created with the first 10 test vector batches and subsequent test vectors are repeated. This burst of the test vectors in both c432.bench and c1355.bench covers a decent amount of the faults but not to a consistent satisfactory amount.
- LFSR with taps at 2, 4, 5
    - This LFSR has an excellent initial coverage at the first few test vector batches. Both c432.bench and c1355.bench, the LFSR has a relatively good start of coverage but does not have the best coverage after 10 batches.
- LFSR with taps at 2, 3, 4
    - This LFSR is inconsistent in its coverage per batch, appearing better in lower inputs but weaker in larger inputs. This LFSR configuration is not the best in comparison to the other LFSR in terms of consistency but can be the best one to use.
- LFSR with taps at 3, 5, 7
    - This LFSR starts out with great coverage for about the first 3 runs, but then its net fault coverage percent gain from runs 4 - 8 are pretty lackluster in both c432.bench and c1355.bench. It picks up again towards the end of the 10 runs, to match or even beat the other LFSR taps.
- N-bit counter
    - The N-bit counter is by far the worst method for generating test vectors due to its overly simple nature and conceptual poor coverage. Since the method used is incremental by 1, we get very similar results from one batch to the next batch since the test vector changes by a handful of bits at best. The lack of shifting really hurts this method, especially in large circuits like c1355.bench.

Conclusion

With the following results we can conclude that PRPG's performance are good for test vector generators because they provide full fault coverage and implementing it is more realistic as compared to running every test vector possible, saving time and saving memory on a given processor. LFSR with taps will perform relatively close and be more potent with a large pool of test vectors available. LFSR with no taps and a N-bit counter, in comparison, performs far worse with a large TV pool. Thus, LFSR with a large pool of vectors compared to using all test vectors is a more effective way to test a circuit.

Team contributions

Signory Somsavath: Created part A and worked jointly to create part D.

Jonathan Wacker: Created part B and part C from original requirements which include the generation of faults per test vector and the creation of optimal order of test vectors to cover most faults. Created UI and output including the creation of test vectors from the LFSRs functions and all test vectors depending on input, fault percentage calculation, and display format. Wrote relevant descriptions in the report.

Chris Escobar: Created the csv file outputs for circuit faults, worked jointly to create part D (n-bit counter & 8-bit LFSR) , and formatted the correct order of operations the user will be prompted with.