

# ECE 538: VLSI System Testing

## Assignment 4

Alexander Zapata

April 19, 2019

### **Duke Community Standard**

By submitting this L<sup>A</sup>T<sub>E</sub>X document, I affirm that

1. I have adhered to the Duke Community Standard in completing this assignment.

## Problem 1 Path Delay and Small Delay Defect Testing using Synopsys TetraMax:

### a. Path Delay Faults

(i)

Number of Critical Paths	50	100	150	200	250	300
Total Faults	50	100	150	200	250	300
Detected	44	82	123	128	129	128
Test Coverage	88.00%	82.00%	82.00%	64.00%	51.60%	42.67%
Patterns	9	18	19	19	19	18
CPU Time	0.02	0.02	0.03	0.02	0.03	0.05

Table 1: Results for path-delay faults, 0.15ns clock period

(ii)

Number of Critical Paths	50	100	150	200	250	300
Total Faults	50	100	150	200	250	300
Detected	44	82	123	128	129	128
Test Coverage	88.00%	82.00%	82.00%	64.00%	51.60%	42.67%
Patterns	9	18	19	18	19	18
CPU Time	0.01	0.03	0.03	0.03	0.03	0.04

Table 2: Results for path-delay faults, 0.10ns clock period

The fault coverage for the 0.15ns/0.10ns path delay fault simulations were exactly the same. Having the same total faults, detected faults, and fault coverage means that— from one timing to the next— no additional delay faults were found on the critical paths tested (i.e., the paths not detected in the 0.15ns simulation had significant enough slack to also not be detected in the 0.10ns simulation). Between simulations, there was one more pattern for the 200 critical path simulation with 0.15ns clock than 0.10ns clock. This means that with the faster clock, fewer patterns were necessary to sensitize a delay long enough to detect. The CPU times were roughly the same for each simulation.

### b. Small Delay Defects

(i)

Slack	10%	15%	20%	25%	30%
Total Faults	4094	4094	4094	4094	4094
Detected	3994	3994	3994	3994	3994
Delay Effectiveness	0.11ns(55.17%)	0.165ns(30.75%)	0.22ns(50.08%)	0.275ns(49.82%)	0.33ns(53.68%)
SDQL	6289088.50	6126893.50	5438607.50	4897204.50	4477742.50
CPU Time	0.07	0.07	0.07	0.08	0.08

Table 3: Results for small delay defects, 1.1ns clock period

(ii)

Slack	10%	15%	20%	25%	30%
Total Faults	4094	4094	4094	4094	4094
Detected	3994	3994	3994	3994	3994
Delay Effectiveness	0.12ns(6.64%)	0.18ns(49.23%)	0.24ns(25.95%)	0.30ns(44.24%)	0.36ns(49.65%)
SDQL	5756102.00	5206344.50	5301291.50	4501716.00	4122875.25
CPU Time	0.08	0.06	0.07	0.07	0.08

Table 4: Results for small delay defects, 1.2ns clock period

The delay-effectiveness for almost all simulations using 1.2ns clock period (with the sole exception of 15% slack) was much lower than for the 1.1ns clock period simulations of the same slack-percentage. This means that fewer of the small delay defects could be detected in a circuit with a higher clock period. This is potentially because the small delay defects for a higher clock period (but with same percentage slack) have a longer path to sensitize— so fewer tests will be able to do so. The SDQL values for the 1.1ns simulations are relatively higher than those for their 1.2ns simulation counterparts. This is most-likely the case because of the delay-defect distribution (i.e., there are more small-delay defects that with little slack when the clock period is relatively smaller). The CPU times stayed relatively the same.

(iii)

Slack	10%	15%	20%	25%	30%
Total Faults	4094	4094	4094	4094	4094
Detected	3994	3994	3994	3994	3994
Delay Effectiveness	0.10ns(44.24%)	0.15ns(45.63%)	0.20ns(50.85%)	0.25ns(51.03%)	0.30ns(58.95%)
SDQL	6445672.00	5683769.50	5567790.00	5023122.00	4668992.50
CPU Time	0.07	0.08	0.08	0.08	0.08

Table 5: Results for small delay defects, 1.0ns clock period

The delay-effectiveness is generally higher (with the exception of 10% slack simulation) for the 1.0ns clock period simulations when compared to the 1.1ns clock period simulations. This most-likely means that there were shorter path lengths of the detected defects (within the slack-percentage), because the allowed slack was smaller. So, more shorter paths counted towards delay-effectiveness in the 1.0ns simulations. SDQL fluctuated for the 1.1ns and 1.0ns simulations for which was higher for a given slack-percentage. Delay-effectiveness and SDQL was generally lower for the 1.2ns simulations compared to the 1.0ns simulations (presumably for similar reasons to the ones discussed in part ii).

## Problem 2 Response compaction using LFSRs:

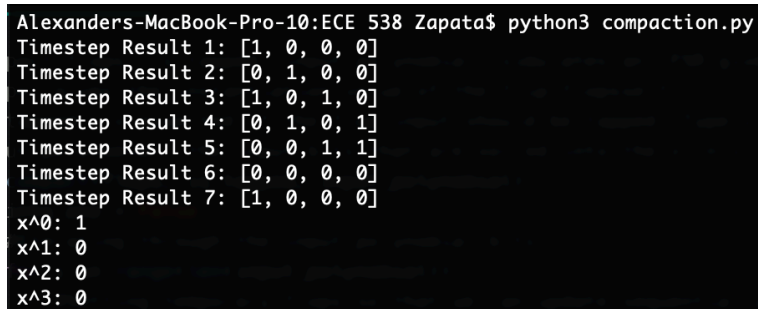
### Simulation Code and Results:

```

1
2 #compact(flops, is_tapped, input_polynomial) simulates an LFSR given input params.
3 #flops- gives the instantiation value of the flip-flopself.
4 #is_tapped- is asserted if the flop input should be tapped (Type-2 LFSR) high-order -> low-order.
5 #input_polynomial- the input bits that are xor'ed into the xn-1} flop in order.
6 def compact(flops, is_tapped, input_polynomial):
7     into_one = False
8     #Iterating through the input_polynomial bits.
9     for i in range(len(input_polynomial)):
10         #Setting temporary flops so that we can update sequentially.
11         temp_flops = flops.copy()
12         #Update each flop depending on tap value and value of previous flop.
13         for j in range(len(flops)):
14             if(is_tapped[j] == 1 and j != 0):
15                 temp_flops[j] = flops[len(flops) - 1] ^ flops[j - 1]
16             else:
17                 if(j != 0):
18                     temp_flops[j] = flops[j - 1]
19         temp_flops[0] = input_polynomial[i] ^ flops[len(flops) - 1]
20         #Set current values to updated values.
21         flops = temp_flops
22         #Printing result.
23         print("Timestep_Result_" + str(i + 1) + ":-" + str(flops))
24     #Printing final flop values.
25     for i in range(len(flops)):
26         print("x".format(i) + ":-" + str(flops[i]))
27
28 #Calling the function with the values for the homework.
29 compact([0, 0, 0, 0], [1, 0, 0, 1], [1, 0, 1, 0, 1, 1, 1])

```

Listing 1: Python code used to simulate LFSR compaction



```

Alexanders-MacBook-Pro-10:ECE 538 Zapata$ python3 compaction.py
Timestep Result 1: [1, 0, 0, 0]
Timestep Result 2: [0, 1, 0, 0]
Timestep Result 3: [1, 0, 1, 0]
Timestep Result 4: [0, 1, 0, 1]
Timestep Result 5: [0, 0, 1, 1]
Timestep Result 6: [0, 0, 0, 0]
Timestep Result 7: [1, 0, 0, 0]
x^0: 1
x^1: 0
x^2: 0
x^3: 0

```

As can be seen in Figure 1, the result of the LFSR compaction was found to be a 1 in the  $x^0$  register (i.e., a 1). This is the same result that was found by hand on the attached sheet.

### Problem 3 SOC Test Infrastructure Design:

	Wrapper SC 1	Wrapper SC 2	Wrapper SC 3	Wrapper SC 4
Wrapper Internal SCs	Included: - one 12-bit chain - one 6-bit chain	Included: - one 12-bit chain - one 6-bit chain	Included: - two 8-bit chains	Included: - one 8-bit chain - two 6-bit chain
Wrapper Input Cells	3	2	4	0
Wrapper Output Cells	2	3	3	3
Scan-in, Scan-out Length	23	23	23	23

Table 6: Wrapper design generated for embedded core C

```

1  import math
2
3  def minimize_function(wrapper_chains, objects_to_add, activity_monitor, name):
4      for i in range(len(objects_to_add)):
5          min_chain = wrapper_chains.index(min(wrapper_chains))
6          max_chain = wrapper_chains.index(max(wrapper_chains))
7          minimum_chain_val = math.inf
8          minimum_chain_num = -1
9          for j in range(len(wrapper_chains)):
10             value_to_minimize = wrapper_chains[max_chain] - (objects_to_add[i] + wrapper_chains[j])
11             if((value_to_minimize >= 0) and (value_to_minimize < minimum_chain_val)):
12                 minimum_chain_val = value_to_minimize
13                 minimum_chain_num = j
14             if(minimum_chain_num == -1):
15                 wrapper_chains[min_chain] += objects_to_add[i]
16                 activity_monitor[min_chain].append("{}:~{}".format(name, objects_to_add[i]))
17             else:
18                 wrapper_chains[minimum_chain_num] += objects_to_add[i]
19                 activity_monitor[minimum_chain_num].append("{}:~{}".format(name, objects_to_add[i]))
20
21  def design_wrapper(tam_width, primary_input_num, primary_output_num, internal_scan):
22      wrapper_chains = []
23      activity_monitor = []
24      for i in range(tam_width):
25          wrapper_chains.append(0)
26          activity_monitor.append(["Wrapper-Chain~{}".format(i + 1)])
27      #Step 1
28      internal_scan.sort(reverse = True)
29      minimize_function(wrapper_chains, internal_scan, activity_monitor, "sc")
30      #Step 2
31      primary_inputs = [1] * primary_input_num
32      minimize_function(wrapper_chains, primary_inputs, activity_monitor, "pi")
33      #Step 3
34      primary_outputs = [1] * primary_output_num
35      minimize_function(wrapper_chains, primary_outputs, activity_monitor, "po")
36      activity_monitor.append("Final_wrapper_chains:~{}".format(wrapper_chains))
37      print(*activity_monitor, sep = "\n")
38      return wrapper_chains
39
40  design_wrapper(4, 9, 11, [12, 12, 8, 8, 8, 6, 6, 6, 6])

```

Listing 2: Python code used to generate wrapper design.