

ECE 538: VLSI System Testing

Assignment 4

Alexander Zapata

April 19, 2019

Duke Community Standard

By submitting this L^AT_EX 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	55.17%	30.75%	50.08%	49.82%	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	6.64%	49.23%	25.95%	44.24%	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	44.24%	45.63%	50.85%	51.03%	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:

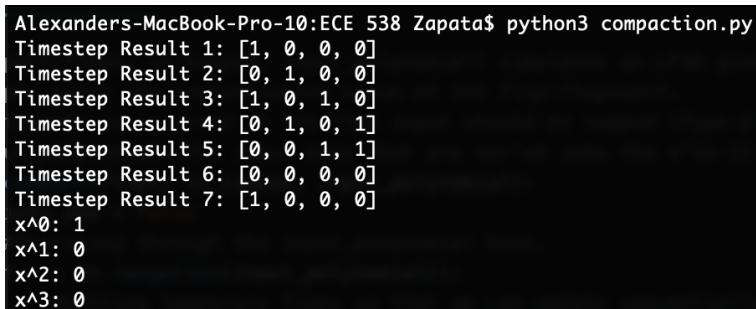
(c) Simulation Code and Results:

```

1
2 #compact(flops, is_tapped, input_polynomial) simulates an LFSR given input params.
3 #flops- gives the instantiation value of the flip-flopsself.
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  $x^{n-1}$  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^{0}".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:

To solve this problem, I used a heuristic BFD optimization algorithm (like discussed in class). The results and code are below.

	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, TAM width of 4

```

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.

Problem 4 soc Testing:

(a)

The code used for problem 3 (in Listing 2) was used to design wrappers of the appropriate sizes ($w = 2, 3, 4, 5, 6$), the results are as tabulated below.

	Wrapper SC 1	Wrapper SC 2
Wrapper Internal SCs	Included: - one 12-bit chain - one 8-bit chain	Included: - one 12-bit chain - one 8-bit chain
Wrapper Input Cells	8	8
Wrapper Output Cells	4	4
Scan-in, Scan-out Length	32	32

Table 7: Wrapper design for width of 2

	Wrapper SC 1	Wrapper SC 2	Wrapper SC 3
Wrapper Internal SCs	Included: - one 12-bit chain	Included: - one 12-bit chain	Included: - two 8-bit chain
Wrapper Input Cells	7	7	2
Wrapper Output Cells	3	2	3
Scan-in, Scan-out Length	22	21	21

Table 8: Wrapper design for width of 3

	Wrapper SC 1	Wrapper SC 2	Wrapper SC 3	Wrapper SC 4
Wrapper Internal SCs	Included: - one 12-bit chain	Included: - one 12-bit chain	Included: - one 8-bit chain	Included: - one 8-bit chain
Wrapper Input Cells	2	2	6	6
Wrapper Output Cells	2	2	2	2
Scan-in, Scan-out Length	16	16	16	16

Table 9: Wrapper design for width of 4

	Wrapper SC 1	Wrapper SC 2	Wrapper SC 3	Wrapper SC 4	Wrapper SC 5
Wrapper Internal SCs	Included: - one 12-bit chain	Included: - one 12-bit chain	Included: - one 8-bit chain	Included: - one 8-bit chain	None
Wrapper Input Cells	0	0	4	4	8
Wrapper Output Cells	1	1	1	1	4
Scan-in, Scan-out Length	13	13	13	13	12

Table 10: Wrapper design for width of 5

	Wrapper SC 1	Wrapper SC 2	Wrapper SC 3	Wrapper SC 4	Wrapper SC 5	Wrapper SC 6
Wrapper Internal SCs	Included: - one 12-bit chain	Included: - one 12-bit chain	Included: - one 8-bit chain	Included: - one 8-bit chain	None	None
Wrapper Input Cells	0	0	4	4	8	0
Wrapper Output Cells	0	0	0	0	4	4
Scan-in, Scan-out Length	12	12	12	12	12	4

Table 11: Wrapper design for width of 6

(b)

Knowing the acceptable widths of the wrappers, and the fact that each of the 8 SOC's are identical, greatly simplified this problem. The following code was used to maximize the minimum wrapper width for this TAM design (minimizing scan-in/scan-out length and therefore minimizing test-time).

```

1
2 def tam_designer(total_width, wrapper_widths, num_cores):
3     wrapper_widths.sort(reverse=True)
4     tam_routes = [0] * num_cores
5     for i in range(len(wrapper_widths)):
6         if((sum(tam_routes) == 0) or (sum(tam_routes) > total_width)):
7             for j in range(num_cores):
8                 tam_routes[j] = wrapper_widths[i]
9         while(sum(tam_routes) < total_width):
10             min_index = tam_routes.index(min(tam_routes))
11             tam_routes[min_index] += 1
12     return tam_routes
13
14 print(tam_designer(36, [2, 3, 4, 5, 6], 8))

```

Listing 3: Python code used to minimize TAM test time for the given SOC design

In the TAM design result, each embedded core wrapper is given their own TAM lines; however, the TAM width is not the same for each core because of the 36-bit TAM width constraint. No wrappers share TAM lines in this case, as preempting tests would result in longer maximum test time for the given widths. Four of the wrappers will have a TAM width of 5 and the other will have TAM widths of 4. This ensures the maximum scan-in/scan-out time is 16 clock cycles, corresponding to the BFD optimization in part (a) and assuming one clock cycle per single-bit shift-in/shift-out.

Problem 5 Test Compression:

(a)

To generate the fan-out design for this problem, I used a heuristic method of determining a compressed width set of test inputs given a pseudo-random test-pattern set. To do this, I generated a conflict-graph (as discussed in class), from the compatibility and inverse compatibility of the test patterns for each scan chain, and then heuristically found a graph coloring that would lead to a sufficiently compacted input-line set. To find a relatively small graph-coloring, the order of the graph-coloring was randomly assigned and the coloring was computed n-times for n randomized node sequences. The computed coloring with minimum number of colors was then chosen. Note: an exhaustive search for the chromatic-number of the conflict graph was not done as algorithms to compute an exact result are \mathcal{NP} -complete and on the time-order of $O(2^n n)$. The code used to find the pseudo-random test patterns, conflict graph, graph-coloring, and the resulting fanout decompressor are below:

```

1  import random
2
3  def generate_patterns(num_chains, chain_lengths, num_patterns):
4      pattern_holder = []
5      vals = []
6      random.seed(7)
7      for i in range(95):
8          vals.append('X')
9      for i in range(3):
10         vals.append('0')
11     for i in range(2):
12         vals.append('1')
13     random.shuffle(vals)
14     for i in range(num_patterns):
15         pattern_holder.append([])
16         for j in range(num_chains):
17             pattern_holder[i].append([])
18             for k in range(chain_lengths):
19                 pattern_holder[i][j].append(vals[random.randint(0, 99)])
20     return pattern_holder
21
22 def test_compare(test1, test2):
23     if(len(test1) != len(test2)):
24         return False
25     for i in range(len(test1)):
26         if((test1[i] != test2[i]) and (test1[i] != 'X') and (test2[i] != 'X')):
27             return False
28     return True
29
30 def inv_compare(test1, test2):
31     if(len(test1) != len(test2)):
32         return False
33     for i in range(len(test1)):
34         if((test1[i] == test2[i]) and not ((test1[i] == 'X') or (test2[i] == 'X'))):
35             return False
36     return True
37
38 def n_graph_coloring(color_num, graph):
39     final_color_dict = {}
40     for k in range(color_num):
41         random_list = list(range(len(graph)))
42         random.seed(k)
43         random.shuffle(random_list)
44         color_dict = {}
45         for i in random_list:
46             max_color = 0
47             for j in range(len(graph[i])):
48                 if(str(graph[i][j]) in color_dict.keys()):
49                     if(color_dict[str(graph[i][j])] > max_color):
50                         max_color = color_dict[str(graph[i][j])]
51             color_dict.update({str(i): max_color + 1})
52     if(not bool(final_color_dict) or (max(color_dict.values()) < max(final_color_dict.values()))):

```



```

53         final_color_dict = color_dict
54     return final_color_dict
55
56 def fanout_decompressor(pattern_holder, num_chains):
57     conflict_graph = []
58     pattern_holder_length = len(pattern_holder)
59     for i in range(num_chains):
60         conflict_graph.append([])
61         for j in range(num_chains):
62             if(i != j):
63                 same = True
64                 inv = True
65                 for k in range(pattern_holder_length):
66                     same = same and test_compare(pattern_holder[k][i], pattern_holder[k][j])
67                     inv = inv and inv_compare(pattern_holder[k][i], pattern_holder[k][j])
68                 if(not same and not inv):
69                     conflict_graph[i].append(j)
70     print("Conflict Graph:")
71     print(*conflict_graph, sep="\n")
72     return n_graph_coloring(16, conflict_graph)
73
74 pattern_holder = generate_patterns(16, 8, 100)
75 print("Heuristic choice for fanout coloring: " + str(fanout_decompressor(pattern_holder, 16)))

```

Listing 4: Python code used to generate test-patterns; produce the conflict graph; and color the graph

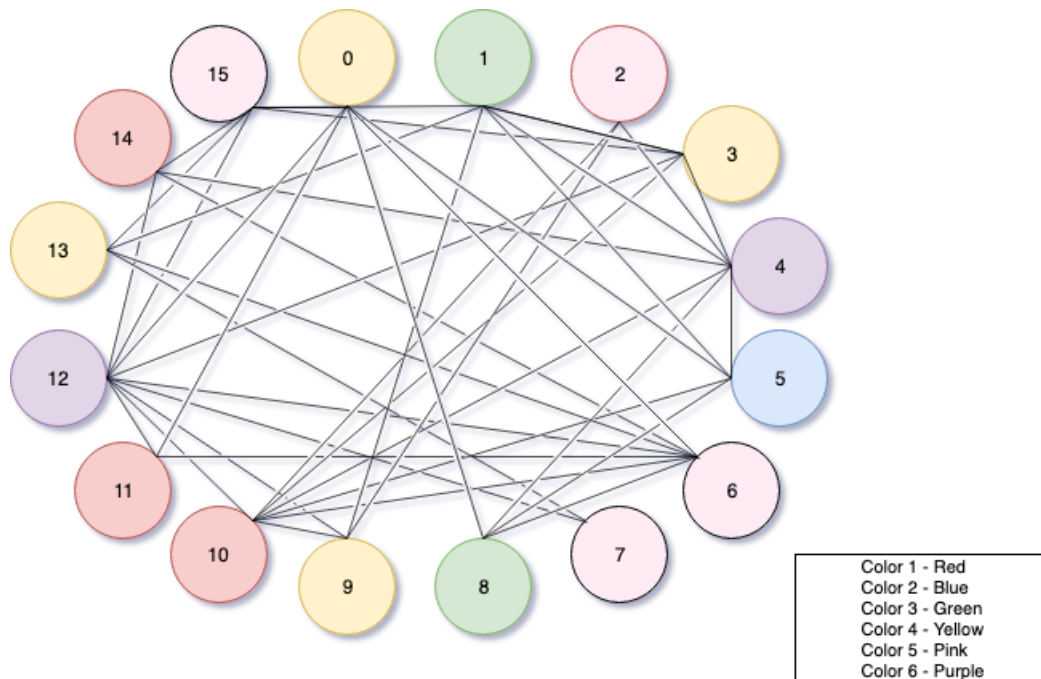


Figure 1: Conflict Graph generated from the pseudo-random test-pattern set

From this conflict graph, and a memoization of which kind of compatibility each pattern-set had with the others (normal or inverse), the following fanout structure was created:

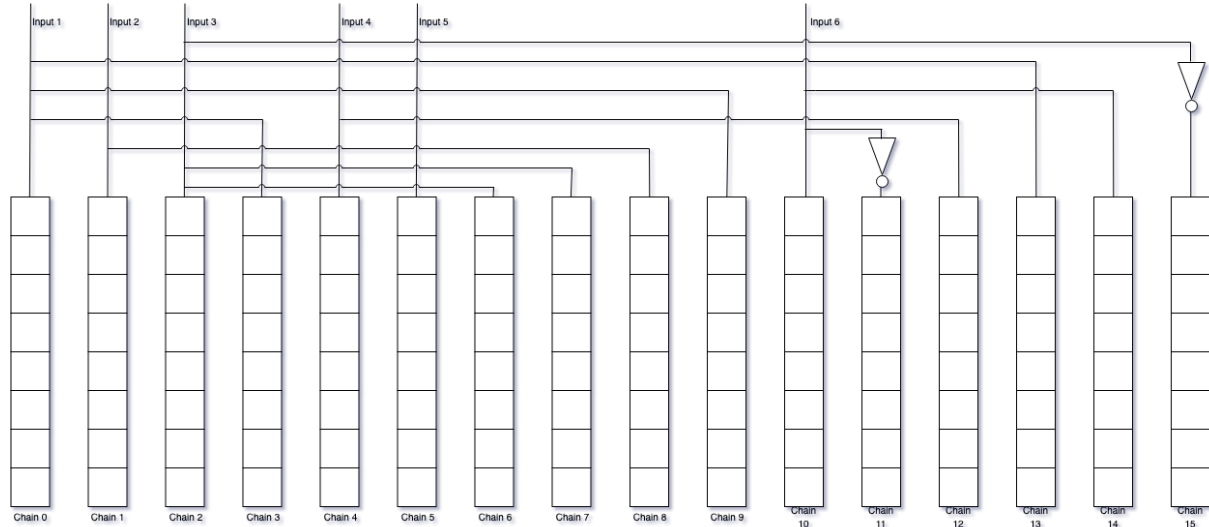


Figure 2: Fanout-based decompressor generated for Part (a)

(b)

An extension of the algorithm used in Part (a) was written, so that a fanout structure could be created for two configurations (each with 50 test patterns). The code and result are below:

```

1      from compressor import generate_patterns, fanout_decompressor
2
3  def reconfigurable_compressor(pattern_holder1, pattern_holder2, num_chains):
4      print("Configuration_1_coloring:_" + str(fanout_decompressor(pattern_holder1, num_chains)))
5      print("Configuration_2_coloring:_" + str(fanout_decompressor(pattern_holder2, num_chains)))
6
7  def split_pattern_holder(pattern_holder):
8      half_holders = [[], []]
9      for i in range(len(pattern_holder)):
10         if(i <= len(pattern_holder)/2):
11             half_holders[0].append(pattern_holder[i])
12         else:
13             half_holders[1].append(pattern_holder[i])
14     return half_holders
15
16  pattern_holder = generate_patterns(16, 8, 100)
17  half_holders = split_pattern_holder(pattern_holder)
18  pattern_holder1 = half_holders[0]
19  pattern_holder2 = half_holders[1]
20
21  reconfigurable_compressor(pattern_holder1, pattern_holder2, 16)

```

Listing 5: Python code used to generate test-patterns; produce the conflict graph; and color the graph in two configurations

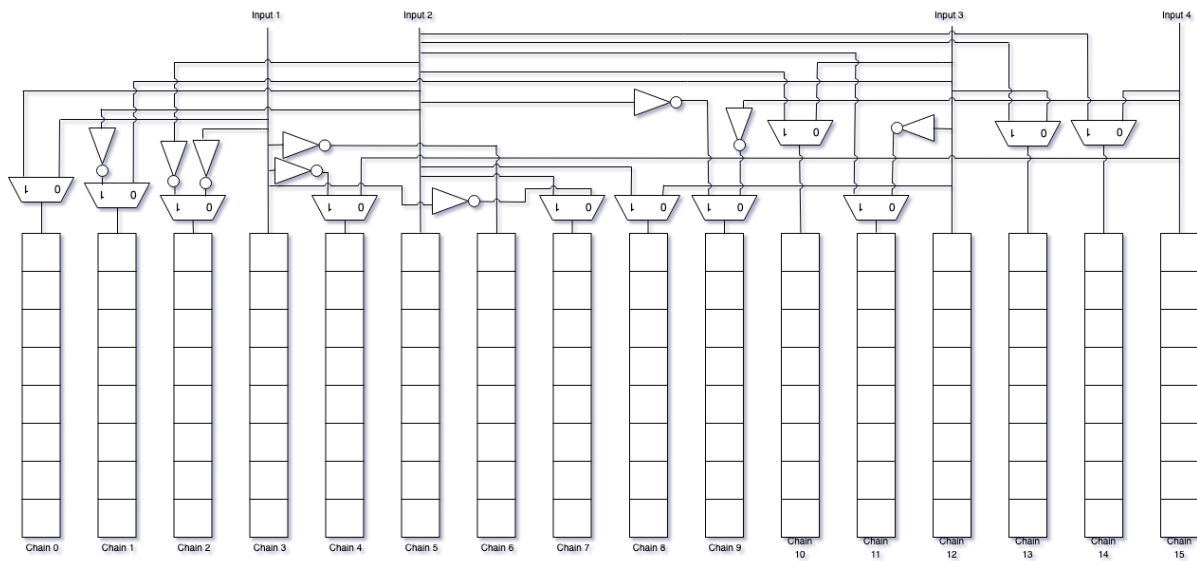


Figure 3: Reconfigurable decompressor with two configurations (starts with all multiplexers at 0, then switch to 1 after the first fifty test-patterns)