Implementation of a Parallel Fault Simulation System using PODEM in a Hardware Accelerator using Python

Mahesh Bhat K[1], Namita Palecha[2]
[1] Department of Electronics and Communication Engineering, RV College of Engineering, Bangalore
[2] Assistant Professor, Department of Electronics and Communication Engineering, RV College of Engineering, Bangalore

Abstract: VLSI Testing is one of the essential domains in recent times. With the channel length of the transistor decreasing continually, the number of transistors in a chip increases, thus increasing the probability of defects or faults. Automatic Test Pattern Generator is one way to find such input test vectors to the circuit, which will help identify the faults if present. PODEM algorithm is one such algorithm used in this regard. This paper helps in reducing the runtime of this algorithm by the parallelism approach. Different stuck-at faults in the gate level circuit are simulated parallelly.

Keywords: ATPG, PODEM, VLSI, VLSI Testing

1. Introduction:

To pass or fail a circuit, appropriate input must be given to reveal all the possible fault defects. To do this, a test pattern generator is inevitable. A test pattern generation has the task of putting together a set of patterns so that a high fault coverage. A fault can either be untestable if it can't be distinguished with any examples or undetectable if it can't be detected within the given test vectors.

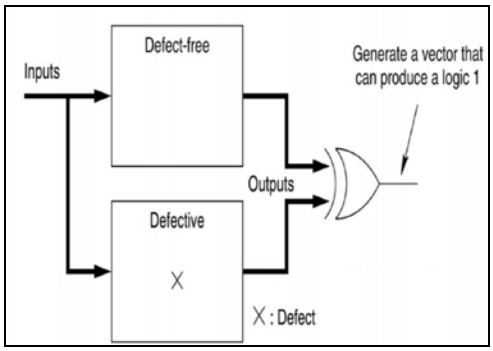

Fig 1.1 : Conceptual view of test generation [1]

Since diagnosing the possible issues in the circuit will help to cut down the testing cost, the physical defects are modeled. These models are known as Fault models. Several fault models were generated for various circuit levels. At the gate level, there are stuck at fault models, multiple stuck at faults, bridging fault models, and delay fault models (further divided into path delay and transition delay fault models). At the transistor level, there are models as stuck open and stuck short models. This occurs when the transistor always remains on and off respectively. Using this fault model, main testing systems are built. Many testing systems are built using the Automatic

Test Pattern Generator (ATPG). ATPG is mainly divided into random test pattern generation and Deterministic test pattern generation.

With the random test pattern generation, random test patterns are generated. These vectors are given to Circuit Under Test (CUT) and simulated. This process is quick and economical but not an accurate method to capture all faults. Therefore it is done with a Deterministic test pattern generator. This deterministic Test Pattern Generator (TPG) is divided into two categories, the algebraic method and the path based method. Using fault table to find the patterns or the boolean difference method is algebraic based, while Path sensitization method, D-Algorithm, Path Oriented DEcision Making (PODEM) are some of path based systems. Understanding these concepts and basics was possible due to the study of the textbook [1], paper [2].

For more understanding of the different ATPG algorithms, [3] was referred. The different ATPG algorithm of D-Algorithm, PODEM and FAN algorithm is explained. This is an old paper that was the basics for all the Automatic test pattern generation algorithms. It explains the most common terms in Automatic Test Pattern Generation like backtrace, backtrack, D-Frontier, Implication, and Justification. In the ATPG algorithms, line justification is an essential part of the generation of test patterns. In [4], the author puts forward a reconvergence aware testability measure to improve the process of the ATPG justification. The paper shows experimental results of the improvement of ATPG runtime by using Sandia Controllability/Observability Analysis Program (SCOAP) guided Path Oriented DEcision Making (PODEM). It truncates the runtime by 76%.

As the number of test patterns needed to test a circuit has been increasing due to the increase in complexity, [5] discusses the power consumption with respect to it. It uses the Tetramax tool by Synopsys, newly named as TestMax and then using the patterns to perform the power analysis. The paper uses the benchmark circuit of ISCAS'85 and ISCAS'89 to run the fault diagnosis. In [6], DFT and ATPG tool flow is explained and also the different tools given by semiconductor vendors. It shows that the ATPG is a essential part of the DFT and will help in finding input test patterns to simulate and to detect any faults in the fault sites. It also states the different fault models like stuck at fault models, transition faults, Path delay faults, IDDQ faults, Transistor faults.

With [7], a method to develop the ATPG algorithm from single stuck at faults to double stuck at faults is explained. It explains all the multiple stuck at faults also include the combination of single stuck at models. The paper tells that there are 3m -1 combinations possible if there are m-nets in the gate level circuit. This gave the idea of parallelism for this project and using it; different faults were simulated parallel.

To understand the concepts of parallel technique in ATPG, [8] was referred. The paper mainly explained two things; one is the use of genetic algorithms and how parallelism can be used in ATPG. It defines a new four valued logic. It represents two bits together i.e., 00 has 0, 11 has 1, 01 has V and 10 has N. Therefore using the parallel fault simulation accelerates the generation process. Since this project is software based and there is the availability of multiple processors to process the parallelism. Finally, Google Colab [9] was used to run the project on the Hardware Accelerator of GPU and TPU. Colab is Jupiter notebook hosted and runs on the cloud.

2. Methodology

The following are the steps taken up in the paper:
1. Inputs files of Netlist and Fault List.
2. From the netlist, generate a list of all the gates, including the net numbers of its respective inputs and outputs.
3. Ready the Parallel Processing system and call the function for the process to begin.
4. Perform the PODEM to the respective target of fault in the net, and it's stuck at value.
5. Output the test pattern to be used to detect each of the faults.

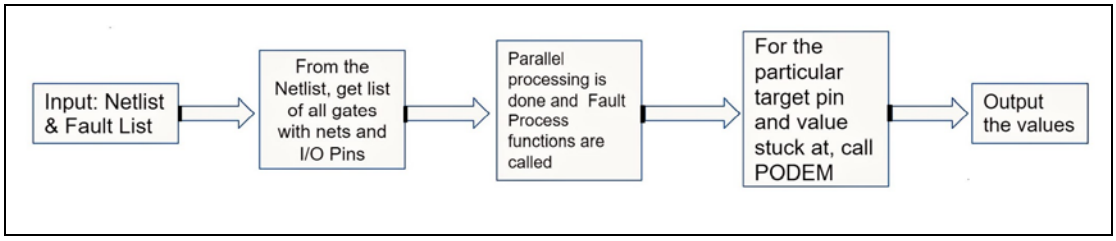

Fig 2.1 : Methodology and the flow of the implementation of PODEM

3. Path Oriented DEcision Making (PODEM)

To improve the D-Algorithm, Path Oriented DEcision Making (PODEM) came into being. In PODEM, D-Frontier is kept as in D-Algorithm. Since the decision or the choices is done only at the primary inputs, J-Frontier is not necessary.

```
1: initialize all gates to don't-cares;
2: D-frontier = ∅;
3: result = PODEM-Recursion(C);
4: if result == success then
5:    print out values at the primary inputs;
6: else
7:    print fault f is untestable;
8: end if
```

Fig 3.1.1

```
1: if fault-effect is observed at a PO then
2:    return (success);
3: end if
4: (g, v) = getObjective(C);
5: (pi, u) = backtrace(g, v);
6: logicSimulate_and_imply(pi, u);
7: result = PODEM-Recursion(C);
8: if result == success then
9:    return(success);
10: end if
11: /* backtrack */
12: logicSimulate_and_imply(pi, ū);
13: result = PODEM-Recursion(C);
14: if result == success then
15:    return(success);
16: end if
17: /* bad decision made at an earlier step, reset pi */
18: logicSimulate_and_imply(pi, x);
19: return(failure);
```

Fig 3.1.2

```
1: if fault is not excited then
2:    return (g, v̄);
3: end if
4: d = a gate in D-frontier;
5: g = an input of d whose value is x;
6: v = non-controlling value of d;
7: return (g, v);
```

Fig 3.1.3

```
1: i = g;
2: num_inversion = 0;
3: while i ≠ primary input do
4:    i = an input of i whose value is x;
5:    if i is an inverted gate type then
6:       num_inversion++;
7:    end if
8: end while
9: if num_inversion == odd then
10:   v = v̄;
11: end if
12: return(i, v);
```

Fig 3.1.4

Fig 3.1 : Pseudo Code of PODEM [1]

The basic flow of PODEM is illustrated in Figures 3.1.1 and 3.1.2. It is based on a branch and bound search, but the decisions are restricted to the primary inputs. All internal nets within the circuit obtain their value of logic high or low from logic simulation from the decision points.

As per the PODEM algorithm, it starts with an objective as a target, and it backtraces from the goal to a primary input with the help of the best way. If the target fault becomes unexcited or there are no more D-frontier gates, then a substandard decision had been taken beforehand, and a inversion of portion of the past choices is required.

Three fundamental functions in PODEM-Recursion() are getObjective(), backtrace() and logicSimulate and imply(). The getObjective() function repeats the following ob□jective ATPG should endeavor to legitimize. Before the target fault is energized, the objective is to set the net on which the target fault resides to the value contradicting the stuck-up value given by the fault list. Once the fault is excited, the getObjective() function selects the best fault from the D-frontier to propagate to the following stage. The pseudo-code for getObjective() is shown in Fig 3.1.3.

The backtrace() function returns to the recursion function a input assignment from where there is a way of unjustified gates to the current objective as shown in Fig 3.1.4. Thus, backtrace() won't ever cross through a path containing atleast one justified gates. Finally, the logicSimu□late and imply() function can simply be a regular logic simulation routine.

4. Implementation of the Parallel Fault Simulation of PODEM

As discussed in previous section, the Path Oriented DEcision Making (PODEM) algorithm is implemented. A slight modification is done in the algorithm so that multiple fault sites are run parallelly. Before running the algorithm, a list of the different types of Gates such as INVERTER, AND, OR, NAND, NOR, XOR, BUFFER gates.Normally in the PODEM algorithm, a fault is taken and evaluation is done. Here, using the concept of parallel processing the fault list is divided into two and two fault evaluation is run parallelly. This speeds up the time and thereby decreasing the runtime. Figure 4.1 shows the steps involved in parallel processing.

```
1063
1064    # multiprocessing pool object
1065    pool = multiprocessing.Pool()
1066
1067    # pool object with number of element
1068    pool = multiprocessing.Pool(processes=2)
1069    #print(fault_list)
1070
1071    #print(type(fault_list[0][0]))
1072    # map the function to the list and pass
1073    # function and input list as arguments
1074    pool.map(fault_process, fault_list)
1075
1076    t1 = time.time()
1077    print("Time",t1-t0)
1078
```

Fig 4.1 : Parallel Processing of the Faults

Fault Process is a function that encloses all the functions involved in the PODEM. These include the evaluation logic function, PODEM function which includes D-frontier, objective, backtrace function as discussed in the pseudo code. Figure 4.2

shows the fault process and the function within it. Finally, after the algorithm is run the output is seen on the console.

```
def fault_process(fault_list):
    global Tot_pins
    def eval_logic():  # 5 valued Imply Function and call other gates logic inside here      Call No-11 ···
    def PODEM(target_pin1,stuck_at1):   #PODEM Function                                      Call No-12 ···
    def create_D_front(d_list):                                                              #called inside PODEM ···
    def d_front(pin_no,d_list):                                                              #called inside create D Front ···
    def find_gate(pin):  # Find which gate has the target pin as the output                  #Call inside d_front ···
    def objective(target_pin,stuck_at):                                                      #called inside find_gate ···
    def backtrace(target_pin,pin_val):                                                       #called inside Objective ···
    def mod_backtrace(pin_no):                                                               #called inside backtrace ···
    def pin_assgn(pin_val):                                                                  #called inside mod_backtrace ···
    def find_inv_par(): # Find Inversion parity of the path                                  #called inside pin_assign ···
    def output_write(all_output):                                                            #call no-13 ···
    def input_write(all_input):                                                              #call no-14 ···

    global target_pin,val_stuck_at
    target_pin=int(fault_list[0])
    val_stuck_at=int(fault_list[1])
```

Fig 4.2 : Fault Process in PODEM

5. RESULTS

The following circuit, as shown below in Fig 5.1, is tested for the algorithm. It is a combinational circuit with fan outs and multiples primary inputs and primary outputs. Fig 5.2 is the netlist for the same circuit.

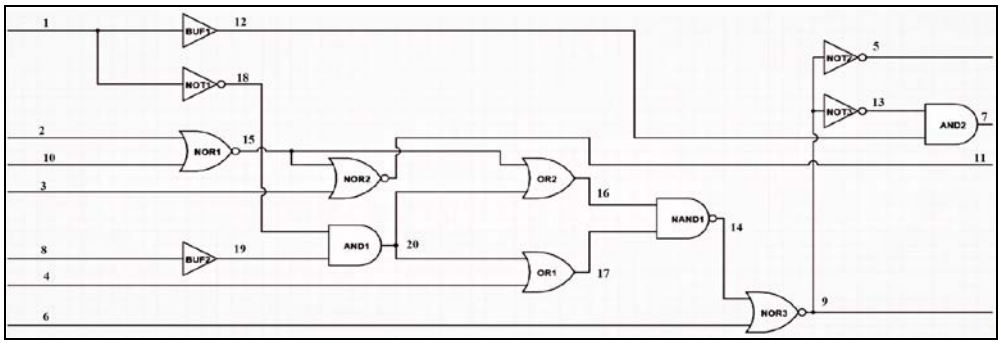

Fig 5.1 : The Combinational Circuit testing the algorithm

netlist1.txt

```
1   INV 1 18
2   BUF 1 12
3   NOR 2 10 15
4   BUF 8 19
5   NOR 3 15 11
6   AND 18 19 20
7   OR 4 20 17
8   OR 15 20 16
9   NAND 16 17 14
10  NOR 6 14 9
11  INV 9 13
12  AND 12 13 7
13  INV 9 5
14  INPUT  1 2 3 4 6 8 10 -1
15  OUTPUT  7 9 11 5 -1
```

Fig 5.2 : Netlist of the circuit

Fig 5.3 is the fault list. Here two faults have been given as an example and have been simulated. For user-friendly purposes, the input and output pins are enlisted. For each of the fault cases, the values obtained at the primary outputs for the input test vectors are listed. Finally, the output of the PODEM algorithm is written on the console, as shown in Fig 5.4.

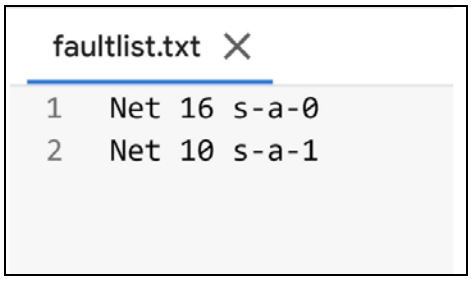

Fig 5.3 : Fault list

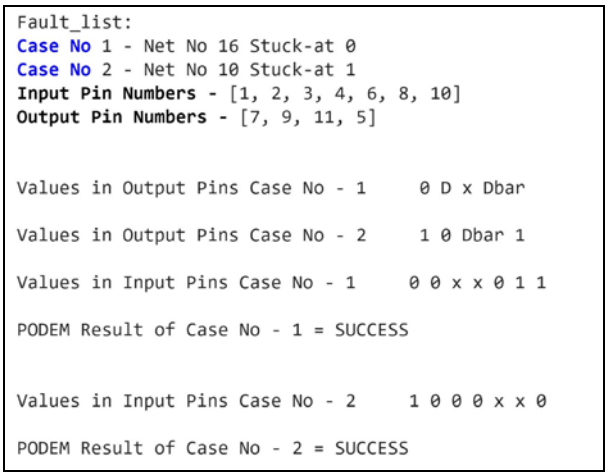

Fig 5.4 : Output of the PODEM Algorithm

The Table 5.1 shows the runtime of the algorithm. The algorithm is run twice and then averaged to find the reduced runtime. The percentage of reduction is found to be 67.54% normally. With respect to GPU, TPU based running in Google Colab Notebook it is about 77.52% and 75.19%.

	Normal Runtime (in sec)	Reduced Runtime (in sec)	Reduced Runtime in GPU (in sec)	Reduced Runtime in TPU (in sec)
Trail No 1	0.1129	0.0332	0.0292	0.0285
Trail No 2	0.1185	0.0419	0.0228	0.0289
Average	0.1157	0.03755	0.026	0.0287

Table 5.1 : Runtime of the Algorithm in Various Situation

6. Conclusion

The PODEM algorithms have been implemented successfully. Modification has been done to PODEM to speed up the process. As a result, the run time of the process has been reduced by about 67.54%, which may vary depending on the size of the circuit and the netlist. If it is run on hardware accelerators such as GPU and TPU, the reduction is more.

7. References:

1. Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. 2006. VLSI Test Principles and Architectures: Design for Testability (Systems on Silicon). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
2. S. A. Al-Arian, "Functional level ATPG and fault coverage," IEEE Proceedings of the SOUTHEASTCON '91, 1991, pp. 104-108 vol.1, doi: 10.1109/SECON.1991.147714.
3. T. Kirkland and M. R. Mercer, "Algorithms for automatic test-pattern generation," in *IEEE Design & Test of Computers*, vol. 5, no. 3, pp. 43-55, June 1988, doi: 10.1109/54.7962.
4. K. Chen, C. Chen and J. Huang, "Testability Measures Considering Circuit Reconvergence to Reduce ATPG Runtime," 2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2019, pp. 1-2, doi: 10.1109/DDECS.2019.8724660.
5. C. N. Kumar, A. Madhumitha, N. S. Preetam, P. V. Gupta and J. P. Anita, "Fault Diagnosis Using Automatic Test Pattern Generation and Test Power Reduction Technique for VLSI Circuits," 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), 2019, pp. 412-417, doi: 10.1109/ICOEI.2019.8862751.
6. J. Chauhan, C. Panchal and H. Suthar, "Scan methodology and ATPG DFT techniques at lower technology node," 2017 International Conference on Computing Methodologies and Communication (ICCMC), 2017, pp. 508-514, doi: 10.1109/ICCMC.2017.8282741.
7. P. Wang, C. J. Moore, A. M. Gharehbaghi and M. Fujita, "An ATPG Method for Double Stuck-At Faults by Analyzing Propagation Paths of Single Faults," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 65, no. 3, pp. 1063-1074, March 2018, doi: 10.1109/TCSI.2017.2765721.
8. M. Sabry, A. Wahba and H. Mahdi, "A parallel technique for ATPG using genetic algorithms," Proceedings of the Tenth International Conference on Microelectronics (Cat. No.98EX186), 1998, pp. 71-73, doi: 10.1109/ICM.1998.825571.
9. Colaboratory – Google. [Online]. Available: https : / / research . google . com / colaboratory/faq.html.