Simulator Independent Fault Simulation Using WAVES

Todd A. DeLong
D. Todd Smith
Barry W. Johnson

Center for Semicustom Integrated Systems
Department of Electrical Engineering
University of Virginia
Charlottesville, Virginia 22903-2442

James P. Hanna
Electronics Reliability Division
Rome Laboratory
Griffiss AFB, NY 13441-5700

Abstract

This paper presents a novel serial VHDL fault simulation technique that uses IEEE standard 1029.1- 1996, Waveform and Vector Exchange (WAVES). This fault simulation technique has several advantages over previous techniques. First, the injection of a fault into the VHDL Unit Under Test (UUT) is performed entirely in VHDL, and therefore is completely simulator independent. Second, fault injection is accomplished using the WAVES standard so the designer can use the same tools to perform fault simulations on a VHDL model that are used for VHDL model verification. This technique utilizes two WAVES processes to accomplish fault simulation. One WAVES process resides in the test bench, applies the stimuli to the UUT, and compares the actual response of the UUT, reporting any discrepancies with the expected response as the simulation progresses. A second WAVES process is added to the architecture of the VHDL UUT. This second WAVES process supplies stimuli to fault mask signals that indicate the presence or absence of a given fault on a given signal in the model. The fault mask values are interpreted by a concurrent fault injection procedure which corrupts the fault-free port and internal signals according to the fault mask value and the fault model chosen by the designer. This paper discusses fault injection procedures that have been developed to support four common fault models: (1) the stuck-at fault model, (2) the bridging fault model, (3) the delay fault model, and (4) the stuck-open fault model. Since this technique inserts fault injection capability into the architecture of the UUT, the testbench and the entity declaration for the UUT need not be modified to perform fault simulations. Finally, since the technique is based on WAVES-96, it can be applied to any level of design abstraction from gate level to performance level modeling.

1. Introduction

The ever increasing complexity of digital designs makes the verification of the functional correctness of a digital device a challenging endeavor. The verification process is divided into two major phases, design verification (functional testing) and manufacturing verification (fault testing). The technique used for design verification is highly dependent on the design methodology. The design process begins by performing a design improvement iteration on a given device. Once the designer completes a design improvement iteration then a model of the device is produced. The device model is then simulated to determine if the design has the desired functional attributes. If the device model passes the functional test then the design process is complete and the model is used to build the designed device. If the device model fails the functional test then another iteration of design improvement and functional testing is performed. The iterative design refinement process continues until the modeled device passes the functional test.

The manufacturing verification process typically employs a well accepted technique referred to as the digital test paradigm (DTP). The DTP is performed after the device is designed but before the device is manufactured. The purpose of the DTP is to produce a set of test vectors that can be used to test for physical defects in the device during manufacturing. The DTP consists of four major components: (1) selection of a fault model, (2) generation of the fault list of interest, (3) generation of the test patterns to detect the faults contained in the fault list, and (4) estimation of the percentage of faults detected by the application of the test patterns. An important tool in the DTP is a fault simulator which is used to determine which faults are detected for a given set of test patterns. Conceptually, a fault simulator inserts a fault into the Unit Under Test (UUT), and then the faulty UUT is simulated. If the

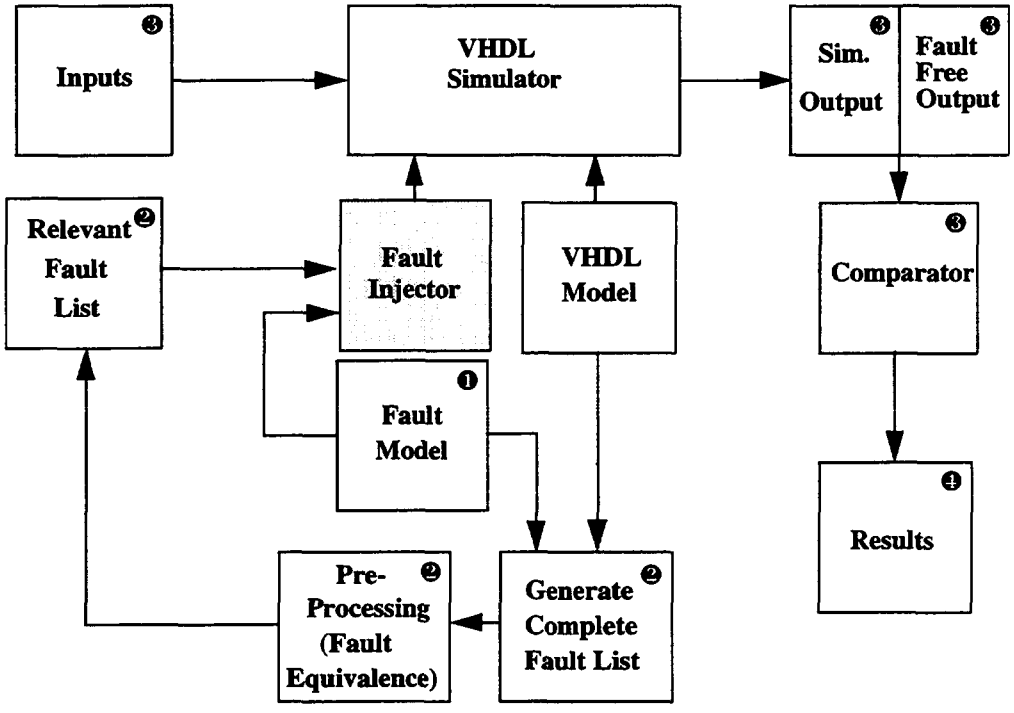


Figure 1: Structure of a simulator-independent VHDL fault simulator.

faulty UUT produces an erroneous output, then the fault is detected by the test pattern set. A structure for a simulator-independent, general-purpose VHDL fault simulator that is used during the DTP is shown in Figure 1.

The DTP begins by applying a fault model to the UUT which generates a fault list of relevant faults. The list of relevant faults is generated by determining any fault equivalency that may exists in the complete fault list generated using the fault model. Typically, the designer assumes that the fault model is sufficient to represent all manufacturing defects. Thus, applying the fault model to UUT produces a list of all faults which are assumed to completely represent the set of possible manufacturing defects. The first two steps in the DTP are indicated by the boxes with a ❶ or a ❷ in Figure 1.

There are a wide variety of fault models described in literature. A brief description of the accepted faults models is presented here; the references provide more detailed information. The purpose of a fault model is to accurately represent the behavior of a faulty component. One possible fault scenario is a fault condition which adds extra delay to the signal propagation through a component. Delay fault models are used to represent faults which cause signal propagation delay [2, 9, 23]. Another common fault scenario is having two input signals shorted together. Bridging fault models are used to represent one signal corrupting another signal [1, 2, 9, 23]. Specifically, a bridging fault occurs when the value of signal A sets the value of signal B. There are two fault models which are commonly used to represent transistor-level fault conditions: (1) stuck-on/off fault model, and (2) stuck-open fault model [2, 9, 23]. For most digital logic architectures a transistor is used as a switching element. The stuck-on/off fault model represents the behavior of transistors which are permanently stuck-on/off. The stuck-open fault model is used to represent a fault condition which is common to CMOS logic. A CMOS circuit has failure modes which cause a signal line to be held at an erroneous logic value due to parasitic capacitance. Additionally, functional fault models are employed to evaluate models which are represented at a high-level of abstraction [2, 9, 23]. The major problem with functional fault models is that it is difficult to verify that a functional fault model accurately represents low-level physical defects in the UUT. For this reason, functional fault models have found limited acceptance. By far the most common fault model is the stuck-at fault model [6]. The stuck-at fault model represents a digital signal line being permanently stuck at a logic one or stuck at a logic zero value.

Test Pattern Generation (TPG) is the next step in the DTP. The test patterns will be used to drive the inputs to the model during the DTP. There are three common categories of TPG: (1) Automatic TPG (ATPG), (2) Random TPG (RTPG), and (3) Manual TPG (MTPG). The purpose of the test patterns is to detect each fault contained in the fault list. Conceptually, if the fault list contains F faults then the fault list can be represented by F faulty UUTs. The purpose of the test pattern set is to have each of the F faulty UUTs produce one or more output errors. A faulty UUT is detected during testing by comparing the fault-free output to the simulated output of the UUT. If a miscompare occurs then the UUT is faulty. The third step

in the DTP is indicted by the boxes with a ❷ in Figure 1.

The final step in the DTP is to calculate the percentage of faults in the fault list that are detected by the set of test patterns. This step is indicted by the box with a ❶ in Figure 1. The designer must decide if the percentage of detected faults is acceptable. If the percentage is acceptable, then the DTP ends. Otherwise, additional test patterns are produced via TPG and the test pattern evaluation process is repeated. The iterative addition of test patterns and evaluation via fault simulation continues until the percentage of detected faults becomes acceptable [2, 18].

The remainder of the paper is organized as follows. Section 2 reviews the previous work that has been done in the area of VHDL-based fault simulation. Section 3 discusses the mechanism developed for fault insertion in the simulator-independent, general-purpose VHDL fault simulator, indicated by the shaded box in Figure 1 labeled **Fault Injector**. Section 4 discusses the implementation of the fault models that are supported by the fault simulator, indicated by the box with a ❶ in Figure 1. Section 5 shows simulation results for the fault models from Section 4. Finally, Section 6 discusses conclusions and current research efforts.

2. VHDL-Based Fault Simulation

Very little research has been focused on performing VHDL-based fault simulation. There are essentially seven different VHDL-based fault injection techniques described in the literature. Each of the seven techniques is described briefly in the following paragraphs.

The first VHDL fault simulation approach uses an event driven Single Pattern Multiple Fault Pattern (SPMFP) parallel fault simulation technique [14]. The authors note that the mapping of gate-level evaluations to single machine instructions is lost when high-level data structures are used to store signal values. Since VHDL signals are represented by complex data structures during simulation, it is typically not possible to evaluate W parallel gates with a single host processor instruction. No data is presented to indicate the increase in efficiency which results in using VHDL SPMFP fault simulation versus VHDL serial fault simulation [14]. Additionally, this technique requires that the VHDL model be modified in a significant fashion.

The second approach adapts critical path tracing for use with VHDL gate-level models [21]. The major drawback of this technique is that a reverse propagation phase is required to determine the critical signal lines. The reverse propagation conceptually entails sending information from the outputs of the UUT to the inputs of the UUT; that is, information flows in the reverse direction of the normal gate-level signal propagation [21]. Thus, all signal values in the UUT are bidirectional. The bidirectional flow required of gate-level signals is undesirable since the addition of bidirectionality can be viewed as changing the basic structure of the VHDL model.

The third technique entails the use of a behavioral VHDL fault model to derive the fault list of interest. The faults are injected into the UUT by modifying the VHDL source code [24]. There are two fundamental problems with this approach: (1) the compile time associated with each injected fault produces an unacceptable amount of overhead for large circuits, and (2) it is difficult to justify the completeness of the VHDL functional fault model. Specifically, it is difficult to correlate physical faults to the corruption of high-level behavioral VHDL statements. For this reason, it is difficult to use this technique to estimate the test coverage of the device for stuck-at fault models.

The fourth approach involves translating the VHDL model to a C program which is then used to propagate the effect of a fault forwards and backwards through the UUT. A functional fault model is used to derive the fault list. The approach provides the necessary fault simulation capability, but the technique requires a custom fault simulator to perform the fault simulations [15, 16, 17].

During the design of the fifth technique, a theoretical analysis is performed to determine the fundamental fault injection methods for VHDL models. The authors state that there are three fundamental techniques for performing VHDL-based fault simulation: (1) using bus resolution functions to corrupt the signal value and a fault insertion process to control when the bus resolution function injects a fault (referred to as saboteurs), (2) modifying the VHDL source code to introduce faults (referred to as mutation), and (3) using simulator specific features to change the value of signals and variables during a simulation [12, 20]. Both the saboteur and mutation methods require the modification of VHDL source code and a recompilation before fault simulation can occur. The recompilation of the VHDL model is needed every time the location of the fault is moved. The overhead associated with the model recompilation is one major problem with this approach. The saboteur bus resolution function also introduces overhead. The amount of overhead caused by the bus resolution function is both simulator dependent and related to the amount of activity associated with the signal [22]. The simulator specific fault insertion techniques do not require recompilation or any additional simulation overhead. However, to corrupt signals/variables with simulator specific features typically requires that the simulation of the UUT be halted, the desired fault condition is inserted, and the simulation restarted [12, 20]. Thus, simulator specific fault insertion has overhead associated with stopping and restarting the simulation to insert a fault. The other problem with simulator-based fault insertion is that the fault insertion is tied to a specific VHDL simulator.

The end result of the analysis is the development of the Multi-level Error/Fault Injection Simulation TOol (MEFISTO) [12]. The objective of MEFISTO is to provide a capability to perform fault grading during the design process. All three of the aforementioned fault insertion methods are used by MEFISTO. A 32-bit processor VHDL model is evaluated in [12] to demonstrate the feasibility of MEFISTO. The example fault simulation is performed on both a behavioral and structural VHDL model of a 32-bit microprocessor.

The sixth VHDL-based fault simulation technique relies on using a fault injection process to control a Bus Resolution Function (BRF) which injects a fault in a given signal [4]. This technique is most closely related to the saboteur method in the MEFISTO system. However, one of the goals in the development of this technique was to overcome the limitations in the MEFISTO system [4].

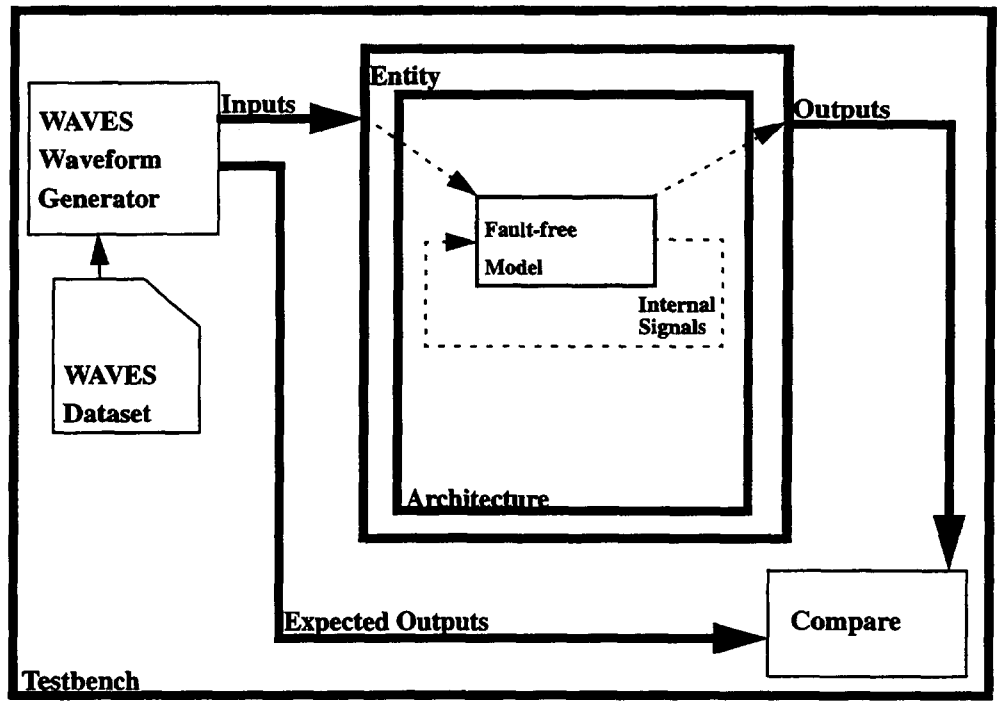


Figure 2: WAVES being used to test a UUT.

One of the most important features of this technique is that the technique separates the fault-free and faulty behavior so that they are independent. Thus, this technique can be used with existing models with minimal changes to the existing code, which can be made in an automated fashion. This feature also allows the technique to be used on any model at any level of abstraction. So, the technique can be used during the design process to incorporate fault simulation into the design process [8].

The seventh VHDL fault simulation technique uses the concept of a super entity to insert faults into the VHDL model [3]. A super entity is constructed by adding a data modulator to an existing VHDL entity. The purpose of the data modulator is to apply a mask vector to the output port of the super entity. The mask vector stores the type of fault corruption which is applied to the output port. The enable signal associated with the data modulator controls when the data modulator is active. The super entity approach allows for either permanent or transient faults to be inserted into the VHDL simulation [3]. The primary disadvantage of the superentity approach is that the fault insertion technique only allows for the insertion of faults into the output port of a VHDL entity. This is problematic because a designer may want to corrupt the internal signals contained in a given entity.

3. WAVES

In 1991, the IEEE Standard for Waveform and Vector Exchange (WAVES) [10] was established that defines how to apply test vectors to a VHDL model. For more details on WAVES-91, see [5, 7, 10, 19].

The WAVES standard is scheduled to be re-balloted in 1996, and several new features will be added to WAVES-96. One important feature is the ability to use multiple waveform generators to apply the test vectors to the UUT as opposed to a single waveform generator that is required with WAVES-91. One advantage of this feature is that it allows a VHDL model with mixed signal types (that is, bit vectors along with integers and enumerated types) to be tested using WAVES, an IEEE standard. This feature is also the key to the fault simulation technique presented in this paper. From hereon, WAVES refers to the proposed WAVES-96 standard.

3.1 WAVES Fault Simulation Technique

Figure 2 shows how WAVES is used to test a UUT under fault-free conditions. The test vectors that contain the stimuli to the VHDL model and the expected outputs are stored in a WAVES dataset in a format defined by the WAVES standard. The WAVES waveform generator reads the test vectors from the WAVES dataset, applies the stimuli to the inputs of the VHDL model, and provides the expected outputs to a comparator in the testbench that compares them with the actual outputs from the VHDL model.

In order to perform fault simulation using the same testbench, one must perform the desired corruption on the signals that are internal to the UUT. The signals are shown as dashed lines in Figure 2 and represent the port and internal signals for the UUT. One desirable attribute of the fault injection technique used to corrupt these signals is that the fault injection technique will not modify

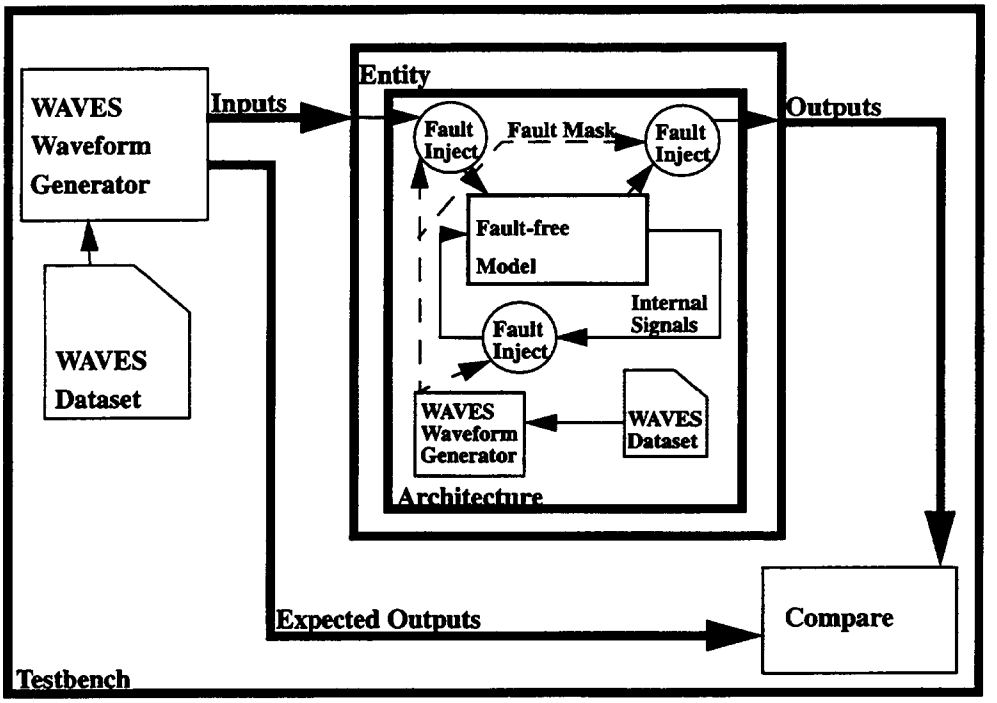


Figure 3: WAVES being used to fault simulate unit under test.

the fault-free behavior of the VHDL model. One way to do this is to intercept the signals before they are used, corrupt them, and then pass them on to the fault-free model. This is shown in Figure 3. The circles labeled **Fault Inject** inside the architecture represent concurrent procedure calls (from hereon referred to as fault injection procedures) that are used to corrupt the port and internal signals. The corruption is performed by modifying each fault-free signal value according to a corresponding fault mask value. The fault mask values are provided to the fault injection procedures by a WAVES waveform generator that is internal to the architecture (also shown in Figure 3). This second waveform generator reads the fault mask values from a second WAVES dataset (shown in Figure 3) and drives the fault masks with the corruption information from the WAVES dataset. The corruption information (that is, the fault mask values) indicates how each signal is to be corrupted by the procedure before passing it to the fault-free model according to the fault model chosen.

4. Fault Models

The fault injection procedure corrupts a signal based on the value of the corresponding fault mask and the fault model chosen by the designer. A pictorial representation of the fault injection procedure is shown in Figure 4. The inputs to the procedure are the fault-free data, the fault mask, and the fault model, which is implemented in VHDL as an n-dimensional table. The fault-free data and the fault mask are indices into the table. The number of indices depends on the fault model.

4.1 Stuck-At Fault Model

The simplest fault model is the stuck-at fault model. For this fault model, signal values are stuck at a logic value that is defined by the fault mask. The type definition for a 2-dimensional table along with a portion of the 2-dimensional table representation of the stuck-at fault model for MVL9 logic values is shown in Figure 5(a). The two indices for the stuck-at fault model table are the fault-free signal value and the fault mask value. Essentially, the result is assigned the fault mask value except when the fault mask value is 'U'. A mask value of 'U' represents fault-free behavior (that is, no corruption occurs). So, the result receives the fault-free signal value in this case. The VHDL implementation of the fault injection procedure for the stuck-at fault model is shown in Figure 5(b).

4.2 Bridging Fault Model

In the bridging fault model, the values of two signals are 100% correlated. The VHDL implementation is very similar to the stuck-at fault model, so the VHDL code has been omitted. The 3-dimensional table supports 6 types of correlation: (1) X-dominance where both signals receive a value of 'X' if either of the fault-free values equals 'X', (2) 0-dominance where both signals receive a value of '0' if either of the fault-free values is '0', (3) 1-dominance where both signals receive a value of '1' if either of the fault-free values is '1', (4) Z-dominance where both signals receive a value of 'Z' if either of the fault-free values is 'Z', (5) signal-1-dominance where signal 2 receives the value of signal 1, and (6) signal-2-

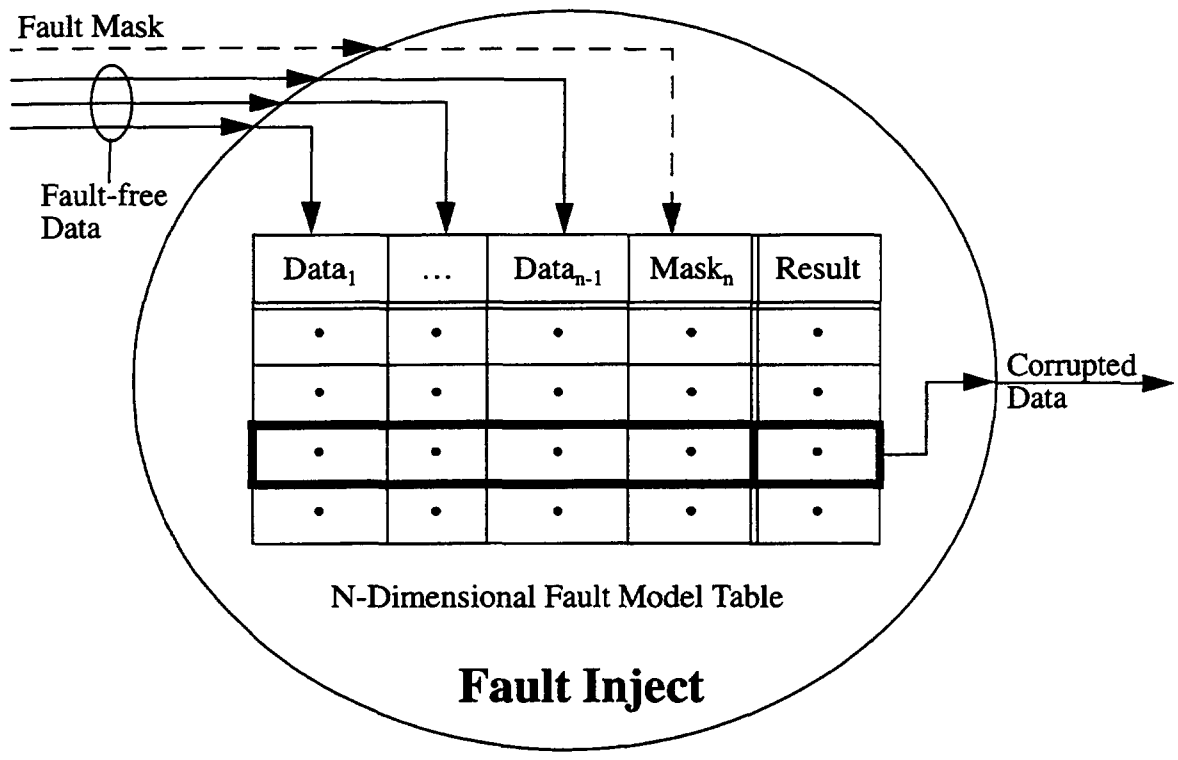


Figure 4: Concurrent procedure call that performs the data corruption.

dominance where signal 1 receives the value of signal 2.

4.3 Delay Fault Model

In the delay fault model, the update of a signal is delayed by a specified value. With regard to using WAVES to perform the fault simulation, the delay value represents the mask for the fault simulation. So, the WAVES LOGIC_VALUE is defined to be a subtype of time (as opposed to a subtype of std_ulogic as with the previous fault models). Again, the VHDL implementation has been omitted.

4.4 Stuck-Open Fault Model

The stuck-open fault model represents transistor-level faults that can occur in a CMOS implementation of logic gates which cause the logic gates to retain the previous output of the device if the fault is activated by certain inputs. The effect of a stuck-open fault is that the combinational logic gate becomes a sequential component. Since most digital logic is implemented using NOR or NAND gates, fault models for these two gates are the only ones supported at this time. A CMOS implementation of a NOR gate is shown in Figure 6. Also shown in Figure 6 is a state table that represents the fault-free and faulty behavior of the gate for the stuck-open fault model. The first three columns list the inputs, A and B, and the fault-free output, C. The remaining three columns list the output under various fault conditions. For example, the fourth column lists the output for the NOR gate when a stuck-open fault occurs on transistor 4. For a stuck-open fault on transistor 4, the n-channel device no longer provides a path to VSS for the input combination of A = '1' and B = '0'. So, instead of generating the fault-free output equal to '0', the gate actually produces the previous output value of the gate (shown as C_p in the table), which will cause an error if the previous value was '1'. The remaining columns list the output of the gate for stuck-open faults on the other three transistors. Figure 7 shows similar information for a CMOS implementation of a NAND gate. Figure 7 shows the CMOS implementation of a NAND gate along with the state table that represents the fault and fault-free behavior of the gate.

5. Simulation Results

Results are shown in Figure 8 for a serial fault simulation example of a 4-bit counter using the stuck-at fault model. The first column lists the simulation time. The second column lists the values for the testbench signal that represents the actual output from the 4-bit counter. The third column lists the values for the testbench signal that represents the expected output from the 4-bit counter. The fourth column lists the values for the fault mask which are provided to the model by a second WAVES waveform generator. The fifth column lists the fault-free values of the output for the 4-bit counter before the signal is corrupted by the fault model. Finally, the sixth column lists the (possibly) faulty values of the port signal for the 4-bit counter after it is (possibly) corrupted by the fault model. As shown in Figure 8, the stuck-at fault model is activated at 406 ns by driving the fault mask signal with a value of "U0101UUUU". This represents a stuck-at value of "0101" on **DataOut**, which is passed to the test-

```
type FaultTable2D is
   array (logic_value,logic_value) of logic_value;

constant stuck_at_fault_model: FaultTable2D:=
-- Fault-free  Mask          Result
(
   .
   .
   '0' =>   (
               'U' =>        '0',
               'X' =>        'X',
               '0' =>        '0',
               '1' =>        '1',
               'Z' =>        'Z',
               'W' =>        'W',
               'L' =>        'L',
               'H' =>        'H',
               '-' =>        '-'
            ),
   .
   .
);
```

Figure 5(a): Stuck-at fault model table.

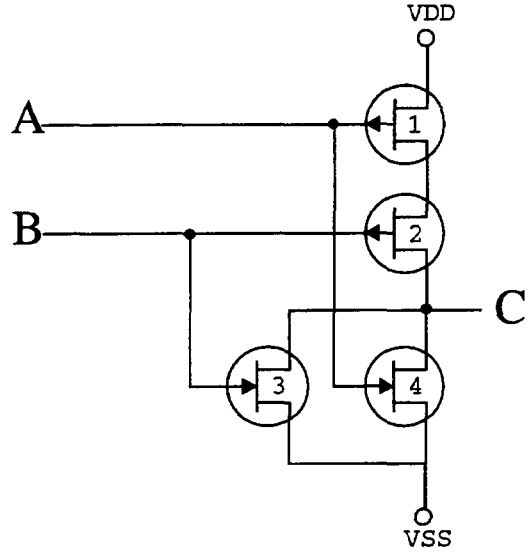
```
   .
   .
package body WAVES_FAULT_DECLARATIONS is

   procedure fault_inject (
      signal data : in std_ulogic;
      signal mask : in logic_value;
      constant fault_model: in faulttable2d;
      signal result : out std_ulogic) is

   begin
--
-- This procedure will corrupt the signal, result, according to the table,
-- fault_model, which defines the faulty behavior. The signals, data and
-- mask, are used as indices into the table, fault_model.
--
   begin
      result <= fault_model(data,mask);
   end fault_inject;

end WAVES_FAULT_DECLARATIONS;
```

Figure 5(b): Fault injection procedure for the stuck-at fault model.



A	B	C	C_4	C_3	$C_{1,2}$
0	0	1	1	1	C_p
0	1	0	0	C_p	0
1	0	0	C_p	0	0
1	1	0	0	0	0

C_P = previous output value

Figure 6: CMOS implementation of a NOR gate.

bench signal, **ActualDataOut**. The corrupted value causes a miscompare between **ActualDataOut** and **ExpectedDataOut**, as shown in Figure 8 at 410 ns.

Similar results were obtained for the bridging, delay, and stuck-open fault models. However, the simulation results are not shown due to paper length limitations.

6. Conclusions

This paper has presented a technique for serial fault simulation using WAVES, IEEE standard 1029.1-1996. The technique uses WAVES to apply stimuli to fault masks, which are added to the architecture of the VHDL model, in a fashion similar to how it is currently used to apply stimuli to the test pins in a testbench. The fault mask values are interpreted by a concurrent procedure to corrupt port and internal signals according to the fault model chosen by the designer. Also presented were the four fault models that are currently supported: (1) stuck-at fault model, (2) bridging fault model, (3) delay fault model, and (4) stuck-open fault model. Finally simulation results were presented for the stuck-at fault model.

Current research efforts are focused on two areas. The first area deals with the automation of the fault simulation technique. A tool is under development which will automate the modifications to the VHDL model that are necessary to perform the fault simulations. This includes, for example, the addition of the fault mask signals and the concurrent procedure calls that perform the corruptions. The second area deals with decreasing the total time required to perform the fault simulations for a given VHDL model. Towards this end, the application of WAVES to fault simulation techniques other than serial, such as concurrent or deductive [2], is being investigated with the hope of reducing the time per fault simulation.

7. References

[1] Abramovici, M. and P.R. Menon, "A Practical Approach to Fault Simulation and Test Generation for Bridging Faults", *IEEE Transactions on Computers*, Vol. C-34, No. 7, July 1985, pp. 658-663.

[2] Abramovici, M., M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, AT&T Bell Laboratories and W. H. Freeman & Company, 1990.

[3] Center for Digital Systems Engineering, Research Triangle Institute, "Mission-Critical Failure Effects Analysis Using Quantitative Techniques," Final Report, Contract No. F30602-93-C-0009, Rome Laboratory/ERM, Griffiss AFB, NY 13441-4514, April 1994.

[4] DeLong, T. A, B. W. Johnson, and J. A. Profeta, III, "A Fault Injection Technique for VHDL Behavioral-Level Models," *IEEE Design and Test*, submitted for publication.

[5] Drager, Steven L., James P. Hanna, and Robert G. Hillman, "VHDL Model Verification and System Life Cycle Support", *VHDL International Users' Forum Notebook of Sessions*, February 28-March 2, 1996, pp. 305-311.

[6] Eldred, R. D., "Test Routines Based on Symbolic Logic Statements," *Journal of the ACM*, Vol. 6, No. 1, January 1959, pp. 33-36.

[7] Flynn, Christopher J., Frederick G. Hall, James P. Hanna, and Mark T. Pronobis, "Using

WAVES in a Top-Down Design Methodology", *VHDL International Users' Forum Notebook of Sessions*, November 13-16, 1994, pp. 12.1-12.6.

[8] Ghosh, Anup, *A Methodology and Application of Fault Simulation in the Design Process of Large-scale Systems*, Ph.D. Dissertation, Department of Electrical Engineering, University of Virginia, May 1996.

[9] Hayes, J. P., "Fault Modeling ", *IEEE Design and Test of Computers*, Vol. 2, No. 2, April 1985, pp. 88-95.

[10] *IEEE Standard for Waveform and Vector Exchange (WAVES)*, IEEE Standard 1029.1-1991, IEEE Computer Society/IEEE Standards Coordinating Committee 20, 1991.

[11] *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164)*, IEEE Standard 1164-1993, IEEE Computer Society, 1993.

[12] Jenn, E., J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool, "*24th Fault Tolerant Computing Symposium Proceedings*, June 1994, pp. 66-75.

[13] Johnson, Barry W., *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley Publishing Co., 1989.

[14] Navabi, S., N. Cooray, and R. Liyanage, "Modeling for Fault Insertion and Parallel Fault Simulation", *VIUF Conference Proceedings*, April 1993, pp. 75-89.

[15] Pitchumani, V., P. Mayor, and N. Radia, "A System for Fault Diagnosis and Simulation of VHDL descriptions", *28th Design Automation Conference Proceedings*, June 1991, pp. 144-150.

[16] Pitchumani, V., P. Mayor, and N. Radia, "Fault Diagnosis of VHDL descriptions using Functional Fault Models", *VHDL Users' Group Conference*, April 1991, pp. 29-33.

[17] Pitchumani, V., P. Mayor, and N. Radia, "Fault Diagnosis using Functional Fault Models for VHDL descriptions", *International Test Conference Proceedings*, October 1991, pp. 327-337.

[18] Pradhan, D. K., *Fault Tolerant Computing Theory and Techniques*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[19] Pronobis, Mark T., Robert Hillman, and Christopher Flynn, "Test Insertion Without Being a Test Expert", *VHDL International Users' Forum Proceedings*, October 15-18, 1995, pp. 10.1-10.8.

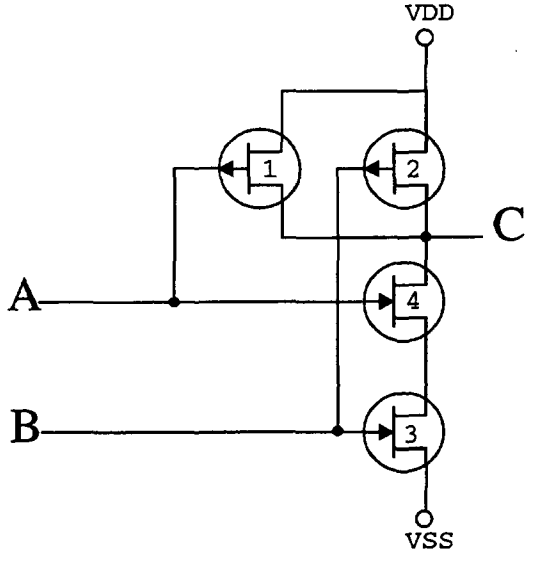
[20] Ramakrishnan, T. and L. Kinney, "Extension of the Critical Path Tracing Algorithm", *27th Design Automation Conference Proceedings*, June 1990, pp. 720-723.

[21] Shadfar, M., A. Peymandoust, and S. Navabi, "Using VHDL Critical Path Tracing Models for Pseudo Random Test Generation", *VIUF Conference Proceedings*, April 1995, pp. 4.1-4.10.

[22] Voss, Andrew P., Robert H. Klenke, and James H. Aylor, "The Analysis of Modeling Styles for System Level VHDL Simulation", *VHDL International Users' Forum Proceedings*, October 15-18, 1995, pp. 1.7-1.13.

[23] Wadsack, R. L., "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *Bell System Technical Journal*, Vol 57, May-June 1978, pp. 1449-1474.

[24] Ward, P. C. and J. R. Armstrong, "Behavioral Fault Simulation in VHDL ", *27th Design Automation Conference Proceedings*, June 1990, pp. 587-593.



A	B	C	C_1	C_2	$C_{3,4}$
0	0	1	1	1	1
0	1	1	C_p	1	1
1	0	1	1	C_p	1
1	1	0	0	0	C_p

C_p = previous output value

Figure 7: CMOS implementation of a NAND gate.

```
Time   ActulDataOut  ExpectedDataOut   FaultMask    FaultDataOut   DataOut
----   ------------  ---------------   ---------    ------------   -------
.      .             .                 .            .              .
.      .             .                 .            .              .
405    1010          ----              UUUUUUUUUU   1010           1010
406    0101          ----              U0101UUUU    1010           0101
410    0101          1010              U0101UUUU    1010           0101
415    0101          ----              U0101UUUU    1010           0101
425    0101          ----              U0101UUUU    1011           0101
426    1011          ----              UUUUUUUUUU   1011           0110
430    1011          1011              UUUUUUUUUU   1011           0110
435    1011          ----              UUUUUUUUUU   1011           0110
.      .             .                 .            .              .
.      .             .                 .            .              .
```

Figure 8: Simulation results for the stuck-at fault model.