Modeling for Fault Insertion and Parallel Fault Simulation

Zainalbedin Navabĩ*, Niranjan Cooray† and Ruchira Liyanage†
Department of Electrical and Computer Engineering
*University of Tehran; North Kargar Avenue / †Northeastern University; 409 Dana Building
Tehran, Iran 14399 / Boston, MA 02115
Fax: 98-21-688690

ABSTRACT
As the necessity for a high percentage of fault-free end products becomes a growing concern, development of better and faster fault simulators has also become a major issue. The speed at which the fault simulation is performed grows in importance as the circuits grow larger. In this paper, we propose a VHDL modeling technique for easy fault insertion and parallel fault simulation. The models require minor modification of non-fault structural models and make it possible to insert faults into a circuit at the deepest levels of nesting without requiring recompilation of the entire circuit for newly introduced faults. The modeling technique also enables us to perform parallel fault simulation of unlimited faulty circuits. We propose a set of utilities to be used, and a methodology to model circuits for achieving the above.

1.0 Introduction
Manufacturer's ultimate desire is to present a fault free end product to his customers. This can only be achieved by extensive testing for manufacturing defects. Fault simulation eases this testing procedure. Fault simulation is, simulation of the circuit under consideration, in the presence of faults. This can be used to compute the fault coverage for a given test pattern to confirm the credibility of the Automatic Test Pattern Generator (ATPG), to indicate areas of a circuit where fault coverage is inadequate, etc. To perform fault simulation, we should be able to inject faults to the circuit. However, it is undesirable if we have to recompile the entire circuit for the purpose of simulation whenever a fault is being introduced.

In parallel fault simulation one good circuit is simulated along with a number of faulty circuits simultaneously. Then the responses of faulty circuits can be compared with the response of the fault free circuit. Usually, this technique is based on parallel processing of bit oriented operations. Therefore, it is limited to computer systems or simulation programs that can perform logical instructions on number of bits in parallel. Since number of bits can be packed in same memory word this technique has the advantage of efficient memory use. This is a more time efficient fault simulation technique than series fault simulation, where faulty circuits are simulated one at a time. Since bits are packed into one memory word, the number of circuits that can be parallely simulated is limited by the word size of the machine. The simple technique of packing several bits into a word becomes impractical for multi-valued logic type. This will require other compacting techniques which depend on the number of values in the logic value system.

Parallel fault simulation using VHDL modeling encompasses these limitations. As we are modeling with a high level language, the model becomes machine independent. In this paper, we first develop a model that can be used for faulting and fault simulation. Then this model is overloaded for parallel fault simulations. Section 2 begins with an overview of faulting and fault simulation. It then presents a small example in which our modeling principles for fault insertion and fault simulation will be demonstrated and discussed. The next section discusses a more comprehensive example including a test bench and a complete simulation environment.
ENTITY xor_unfaultable IS
  PORT (a, b : IN BIT; z : OUT BIT);
END xor_unfaultable;

ARCHITECTURE structural OF xor_unfaultable IS
  COMPONENT gate PORT (a, b : IN BIT; z : OUT BIT); END COMPONENT;
  FOR ALL : gate USE ENTITY WORK.nand_2(structural);
SIGNAL im1, im2, im3 : BIT;
BEGIN
  g1 : gate PORT MAP (a, b, im1);
  g2 : gate PORT MAP (a, im1, im2);
  g3 : gate PORT MAP (b, im1, im3);
  g4 : gate PORT MAP (im2, im3, z);
END structural;

Figure 2. Structural description of the unfaultable XOR circuit

2.0 Faulting and Fault Simulation
Most widely studied and used fault model is the single stuck line fault model. In this fault modeling it is assumed that a single line of circuit is permanently shorted to either a '0' or a '1' and all the logical elements are functioning correctly. Although this fault modeling technique will not cover all the possible faults that can exist in a circuit, it models many different physical faults and can also be used to model other types of faults. Additionally, this modeling technique is independent of technology. In this paper, we will only be considering stuck at '0' and stuck at '1' fault models.

During fault simulation process, a given test set is applied to the fault-free circuit and to each of the faulty circuits. These faulty circuits are obtained by introducing certain different faults for each of them. Then the response of the circuit is analyzed to determine the faults detected by the test. If the response of the fault-free circuit differs from the response of a circuit with a fault, that particular fault is considered to be detected. With these results, fault coverage of a given test set can be calculated. If this fault coverage does not satisfy the requirement, the same process will be repeated with a different test set.

By integrating the fault simulator with ATPG, the number of iterations ATPG has to perform in generating the test set can be reduced. This is a very useful tool in overcoming the inefficiency of current ATPG programs.

2.1 Fault Insertion
We will first present a small example of a standard VHDL description of a circuit that does not have the capability of injecting faults to its internal nodes. As the concepts are developed, we will transform our simple example into a model that enables us to perform parallel fault simulation. The example is a 2-input XOR gate constructed using only 2-input NAND gates, as in Figure 1. Figure 2 shows the structural VHDL description of this circuit. In this description the structural description of the two input NAND gate is assumed to be in an entity called nand_2.

The model of Figure 2 must be modified such that a given node of the circuit can be faulted. To achieve this, 1) a new type which contains stuck-at-0 and stuck-at-1 values must replace the standard BIT type, 2) a resolution function must be developed to resolve faulted values for nodes that are being faulted,
and 3) generic parameters must be used in the model to be able to specify faults at circuit nodes. We will treat these issues as they apply to our simple XOR example.

2.1.1 New Signal Type

Usually, the enumerated type BIT with elements '0' and '1' is sufficient to model digital signals in a circuit. However, when faults are to be injected to a circuit under consideration, the basic type BIT becomes insufficient as the fault information (stuck at '0' and stuck at '1') needs to be included. Thus, a new enumerated type "fbits" with elements '0', '1', 'none', 'sa0', and 'sal' is defined to satisfy this requirement.

```plaintext
TYPE fbit IS ('0', '1', none, sa0, sal);
```

Elements '0' and '1' in the enumerated type fbit represent the usual logical values as in the enumerated type BIT. In addition to these logical values, the fault information of the signal under consideration is assigned using the next 3 elements of the enumerated fbit type. The fault free condition of the signal is indicated by assigning 'none' value to the signal, whereas the node or signal being stuck at '0' or stuck at '1' is indicated by assigning 'sa0' or 'sal' respectively.

In this example, the new type 'fbits' has been developed as an extension to the existing type 'BIT'. However, it must be noted that this expansion is not limited to the base type 'BIT'. Any other base type could also be expanded, by the addition of the three elements 'none', 'sa0' and 'sal' to the existing enumerated type.

2.1.2 Resolution Function

From the new type that was shown above, it is evident that all the faultable signals (that are defined using the type 'fbits') will have a logical value assignment as well as a fault value assignment. These two assignments must appear in the same architectural body, and one resultant logic value has to be evaluated for the signal, depending on the logical value and the fault value. This necessitates the use of a resolution function 'resolve' given in Figure 3.

When a logical assignment is made to a signal, this resolution function is called with the logical and fault value assignments. As soon as the resolution function identifies a 'sa0' or 'sal' being assigned to the signal, it assigns the logic values '0' or '1' respectively, irrespective of the logical signal assignment. On the other hand, if 'none' is assigned indicating a fault-free condition, a logic value is assigned depending on the logical signal assignment. A new subtype node, as shown below, is defined as the resolved subtype of 'fbits' using the resolution function of Figure 3. This new type is used to define all the faultable nodes/signals in the faultable circuit.

```plaintext
SUBTYPE node IS resolve fbit;
```
FUNCTION resolve (a : fbit_vector) RETURN fbit IS
  VARIABLE temp : fbit;
  VARIABLE flag : BOOLEAN := FALSE;
BEGIN
  FOR i IN a'Range LOOP
    IF a(i) = sa0 THEN
      temp := '0'; flag := TRUE; EXIT;
    ELSIF a(i) = sa1 THEN
      temp := '1'; flag := TRUE; EXIT;
    END IF;
  END LOOP;
  IF NOT flag THEN
    FOR i IN a'Range LOOP
      IF a(i) /= none THEN temp := a(i); EXIT;
    END IF;
    END LOOP;
  END IF;
  RETURN temp;
END resolve;

Figure 3. The resolution function

As mentioned above, for all the faultable signals there is a logical assignment as well as a fault assignment. The logical signal assignment is the same as that in the unfaulatable model. The fault assignment (sa0 or sa1 values) must be assigned to a node if it is to be faulted. Once the signals to be faulted have been decided, these fault assignments can be made inside the architecture itself. The major drawback of this technique is that when the faulty condition needs to be changed, the entire model needs recompilation. To avoid this recompilation, generic parameters will be used to pass the fault information to the underlying architecture(s) from a top level configuration. Thus, only the top level configuration has to be modified in order to change the faulty condition.

With the application of resolved node types and passing faults via generic parameters to the XOR circuit description in Figure 2, the faultable XOR circuit is modified as shown in Figure 4. The bold portions of code in Figure 4 signify the additions to the existing model in order to make it faultable.

ENTITY xor_faultable IS
  GENERIC (f1, f2, f3 : node);
  PORT (a, b : IN node; z : OUT node);
END xor_faultable;
ARCHITECTURE structural OF xor_faultable IS
  COMPONENT gate PORT (a, b : IN node; z : OUT node); END COMPONENT;
  FOR ALL : gate USE ENTITY WORK.nand_2(structural);
BEGIN
  im1 <= f1; im2 <= f2; im3 <= f3;
  g1 : gate PORT MAP (a, b, im1);
  g2 : gate PORT MAP (a, im1, im2);
  g3 : gate PORT MAP (b, im1, im3);
  g4 : gate PORT MAP (im2, im3, z);
END structural;

Figure 4. Structural description of the faultable XOR circuit

78
When modeling a circuit that is to be fault simulated, the designer has to decide which nodes are to be faulted, and a different generic parameter should be associated with each of these faultable nodes. In the circuit we have considered, all the internal nodes can be faulted by associating generic parameters to them. For this example we have selected the intermediate signals im1, im2, im3 to be faultable. For this to be done, the generic parameters f1, f2, f3 are assigned to each of these signals.

The structural architecture of xor_faultable in Figure 4 has two concurrent signal assignments to each of the faultable signals. This signal resolution is done by the previously mentioned resolution function 'resolve'. This resolution function takes in a vector of fbit values resulting from a multiple signal assignment and returns the appropriate logical value ('1' or '0') after considering fault/no-fault condition.

2.2 Single Circuit Fault Simulation

Faultable XOR model described in Figure 4 allows us to inject stuck at faults at any predefined faultable nodes/signals. In order to test whether these faults are being covered by a given set of test vector(s), input of the circuit must be provided with them. Thus, all the test vectors that have to be tested on, can be included in the test bench as in Figure 5 that is written on top of the entity.

```
ENTITY xor_test_bench IS
END xor_test_bench;

ARCHITECTURE testing OF xor_test_bench IS
  COMPONENT gate PORT (a,b : IN node; z : OUT node); END COMPONENT;
  SIGNAL a, b, z : node;
BEGIN
  g1 : gate PORT MAP (a,b,z);
  a <= '1' AFTER 500 NS, '0' AFTER 1 US;
  b <= '1' AFTER 250 NS, '0' AFTER 500 NS, '1' AFTER 750 NS, '0' AFTER 1 US;
END testing;

Figure 5. Test bench for a faultable single XOR circuit
```

The entity xor_faultable is bound to the test bench using a configuration as in Figure 6. This configuration that is written on top of the test bench is also used to pass the fault information as generic parameters.

```
CONFIGURATION binding OF xor_test_bench IS
  FOR testing
    FOR ALL : gate
      USE ENTITY WORK.xor_faultable(structural) GENERIC MAP ( f1 => sa0 );
    END FOR;
  END FOR;
END binding;

Figure 6. Configuration at top level
```

The major drawback of this technique is that the circuit has to be re-simulated every time a new fault is being introduced. That is, in order to fault all the signals (3) with 'sa0' and 'sa1', this circuit has to be simulated 6 times. This can be made more time and memory efficient by injecting these 6 faults to 6 different circuits and simulating them simultaneously. In other words, by performing parallel fault simulation.

2.3 Parallel Fault Simulation

In order to perform parallel fault simulation, more than one circuit has to be simulated simultaneously. This is achieved by generating a single model of n parallel circuits each of which is
generated by using unconstrained vectors of faultable nodes for all signals of the original single faultable circuit. Thus, all the signals need to be defined as a vector type of 'node' type, as shown below:

```
TYPE node_vector IS ARRAY (NATURAL RANGE <>) OF node;
```

When all ports and internal signals are declared as having node_vector type, for the logical operations to take place, such operators must be overloaded with functions performing appropriate logical functions on this new type.

### 2.3.1 Faulting Individual Circuits

Although the above methodology changes a single model to n parallel models, the methodology for faulting ports or signals is left unchanged. The ability to fault the circuits from top level configuration is also preserved. This modeling technique should provide us with the flexibility of faulting a particular signal in one of the n parallel circuits and to leave this signal unfauluted in another circuit. Implementation of this faulting methodology becomes more convenient with several utility subprograms that will be discussed here.

#### 2.3.1.1 Utilities for single circuit faulting

Fault values will be assigned to the circuit nodes through the generic parameters. Since all the signals are defined as node_vector, all the generic parameters that are assigned to them should also be defined as node_vector. In order to fault a single node of n parallel nodes, a fault value must be assigned to just one element of this vector, and the other elements must be left unfauluted. This can be achieved by using the function in Figure 7. This function is called to generate a node_vector whose all but one element have the 'none' fault value. The one element has the value associated with the 'fault' parameter of the function. The 'circuits' parameter indicates the number of circuits that are to be simulated in parallel. The 'circuit' parameter identifies the particular circuit that needs to be faulted at a specified node.

#### 2.3.1.2 An expansion utility

Since our circuit model consists of n parallel circuits, for assigning logic values '1' or '0' to a node, or for assigning 'none' to all same nodes of n circuits, a vector of the specified values must be generated. The length of these vectors depends on the number of circuits that are to be simulated in parallel. Therefore, the formation of these vectors of '0's, '1's and 'none's must be done dynamically with the information on the number of circuits. The function 'vector' given in Figure 8 is employed for this purpose.
FUNCTION vector (a : node; circuits : INTEGER) RETURN node_vector IS
  VARIABLE result : node_vector (1 TO circuits);
BEGIN
  FOR i IN 1 TO circuits LOOP result(i) := a;
  END LOOP;
  RETURN result;
END vector;

Figure 8. Utility function

To form vectors of '0's, '1's and 'none's, that are to be applied to n parallel circuits, this function may be called as follows.

vector("0", n) => "000...0"
vector("1", n) => "111...1"
vector(none, n) => "none&none&none&...&none"

Using the methodology described in Section 2.3, the modified description of faultable XOR, for parallel fault simulation is shown in Figure 9.

ENTITY xor_faultable IS
  GENERIC (f1, f2, f3 : node_vector); PORT (a, b : IN node_vector; z : OUT node_vector);
END xor_faultable;

ARCHITECTURE structural OF xor_faultable IS
  COMPONENT gate PORT (a, b : IN node_vector; z : OUT node_vector); END COMPONENT;
  FOR ALL : gate USE ENTITY WORK.nand_2(structural);
  SIGNAL im1, im2, im3 : node_vector(1 TO f1'LENGTH));
BEGIN
  im1 <= f1;
  im2 <= f2;
  im3 <= f3;
  g1 : gate PORT MAP (a, b, im1);
  g2 : gate PORT MAP (a, im1, im2);
  g3 : gate PORT MAP (b, im1, im3);
  g4 : gate PORT MAP (im2, im3, z);
END structural;

Figure 9. Structural description of XOR circuit for parallel fault simulation

2.3.2 Parallel Simulation
For generating a simulatable model, a test bench needs to be written on top of our n-circuit parallel model. Since we need to apply the same test vectors to all the parallel circuits simultaneously, only one test bench is needed for all the parallel circuits. This test bench makes use of the function ‘vector’ in applying test vectors to an unconstrained number of circuits underneath. In order to form corresponding vectors, the function ‘vector’ needs the number of circuits that are to be simulated in parallel. Since it is undesirable to recompile the test bench for new number of parallel circuits, this information is passed as a generic parameter to the test bench. At the simulation time, the value for this generic parameter is passed as a simulation parameter from the command line.

_ENTITY xor_test_bench IS GENERIC (circuits : INTEGER := 1); END xor_test_bench;

ARCHITECTURE testing OF xor_test_bench IS
  COMPONENT gate PORT (a,b : IN node_vector; z : OUT node_vector; END COMPONENT;
  SIGNAL a, b, z : node_vector(1 TO circuits);
BEGIN
  g1 : gate PORT MAP (a,b,z);
  a <= vector(1', circuits) AFTER 500 NS,
    vector(0', circuits) AFTER 1 US;
  b <= vector(1', circuits) AFTER 250 NS,
    vector(0', circuits) AFTER 500 NS,
    vector(1', circuits) AFTER 750 NS,
    vector(0', circuits) AFTER 1 US;
END testing;

Figure 11. Test bench for parallel circuit simulation

The binding of all the entities described above is done in the configuration declaration given in Figure 12. It uses the function ‘one_fault’ to inject faults to the circuits. This configuration is written in order to simulate 3 circuits in parallel, where the first circuit is considered to be the fault-free circuit.
CONFIGURATION binding OF xor_test_bench IS
  FOR testing
    FOR ALL : gate
      USE ENTITY WORK.xor_faultable(structural)
      GENERIC MAP ( f1 => one_fault (3, sa0, 2),
                       f2 => one_fault (3, sa1, 3),
                       f3 => vector (none, 3) );
    END FOR;
  END FOR;
END binding;

Figure 12. Configuration for parallel fault simulation

If a particular node has to be faulted in more than one circuit, the function 'one_fault' cannot be used. Instead, an array of fault values must be associated with the particular generic parameter. For example for simulating 3 parallel circuits, if node im2 (associated with f2) needs to be stuck at '0' in circuit 2 and stuck at '1' in circuit 3, the association list for the generic parameter f2 in the configuration declaration should be done as follows:

\[ f2 => \text{none} \& \text{sa0} \& \text{sa1} \]

3.0 A Complete Example

As a complete example, we will consider a 4 bit binary counter. This comprises of 4 JK flip-flops, 2 AND gates and a clock. Each of these individual components is described behaviorally. The 4 bit counter is structurally described using virtual components that are later bound to the above mentioned building blocks. Therefore in this description of 4 bit counter, we are using various levels of abstraction. Testing of this counter is done by a test bench written on top of the structural description. Finally, a configuration is written in order to bind all the above mentioned entities together and complete the description of the testable 4 bit counter. Since all the component bindings are done in the top level configuration, it is possible to pass the fault information to any desired level. Thus, all the nodes in the circuit are faultable regardless of their depth of abstraction.

Figure 13. 4 bit binary counter

Figure 14 shows the necessary utility package. It includes all the new type declarations and overloading of logic operators that is needed. The resolution function and all the faulting functions are also included in this package. A new procedure 'probe' is written and added to this package in order to facilitate the comparison of faulted and fault-free responses. This probing procedure is utilized in the test bench on all the vectored (parallel) outputs. This procedure compares the output of the same node/signal on all the parallel circuits, and reports any errors that are detected (assuming that the circuit 1 is a fault free circuit), during simulation. The output of the simulation with the probing procedure can be found at the end of this example.
The clock is described behaviorally, in the description in Figure 15. A generic parameter is assigned to the clock so that it becomes faultable. Since the nodes are declared as of node_vector type this
description represents multiple copies of the same clock. Thus, by associating a fault value to an element in the generic parameter (node_vector f0), the output of each individual clock can be faulted.

```
USE WORK.utilities.ALL;
ENTITY clock IS
  GENERIC ( f0 : node_vector);
  PORT ( en : IN node_vector; ck : OUT node_vector);
END clock;

FOR i IN 1 TO f0'LENGTH LOOP

ARCHITECTURE behavioral OF clock IS
  SIGNAL state : node_vector (1 TO f0'LENGTH);
BEGIN
  ck <= f0;
  PROCESS
  BEGIN
    IF NOW > 1 US THEN WAIT;
    ELSE IF en(i) = '1' THEN state(i) <= NOT state(i); END IF;
    WAIT FOR 50 NS;
    END IF;
    END PROCESS;
    ck <= state;
  END behavioral;
END clock;
```

Figure 15. The behavioral description of a clock

```
USE WORK.utilities.ALL;
ENTITY jk_flipflop IS
  GENERIC ( circuits : INTEGER);
  PORT (j, k, ck, reset : IN node_vector; q : OUT node_vector);
END jk_flipflop;

ARCHITECTURE behavioral OF jk_flipflop IS
  SIGNAL state : node_vector(1 TO circuits);
BEGIN
  clk : BLOCK (ck(1)='0' AND NOT ck(1)'STABLE)
  BEGIN
    ff : FOR i IN 1 TO circuits GENERATE
    state(i) <= GUARDED '0' WHEN reset(i)="1" ELSE
    '0' WHEN j(i)='0' AND k(i)="1" ELSE
    '1' WHEN j(i)="1" AND k(i)='0' ELSE
    NOT state(i) WHEN j(i)="1" AND k(i)="1" ELSE
    state(i);
    END GENERATE;
    END BLOCK clk;
    q <= state AFTER 10 NS;
  END behavioral;
END jk_flipflop;
```

Figure 16. The behavioral description of a JK flipflop.

The JK flipflop is described behaviorally in Figure 16. Generic parameters are not passed to this description with fault information, as faulting of the inputs/outputs of the flipflops are provided in the structural description of the nibble counter. However, a generic parameter is passed with the information
on the number of circuits simulated in parallel. Although, the general structure of the flipflop remains
the same as a single flipflop, a GENERATE statement is added to make multiple parallel unconstrained
copies of this flipflop.

USE WORK.utilities.ALL;
ENTITY nibble_counter IS
  GENERIC (f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13 : node_vector);
  PORT (en, reset : IN node_vector; q0, q1, q2, q3 : OUT node_vector);
END nibble_counter;

ARCHITECTURE dataflow OF nibble_counter IS
  COMPONENT jk_ff
    GENERIC (circuits : INTEGER);
    PORT (j, k, ck, reset : IN node_vector; q : OUT node_vector);
  END COMPONENT;
  FOR ALL : jk_ff USE ENTITY WORK.jk_flipflop(behavioral);
  COMPONENT clock
    GENERIC (f0 : node_vector);
    PORT (en : IN node_vector; ck : OUT node_vector);
  END COMPONENT;
  FOR ALL : clock USE ENTITY WORK.clock(behavioral);
  SIGNAL j0, k0, j1, k1, j2, k2, j3, k3, ck, im0, im1, im2, im3, im4 :
    node_vector(1 TO f1'LENGTH);
BEGIN
  -- set fautable signals
  im0 <= f1; im1 <= f2; im2 <= f3; im3 <= f4; im4 <= f5;
  j0 <= f6; k0 <= f7; j1 <= f8; k1 <= f9; j2 <= f10; k2 <= f11; j3 <= f12; k3 <= f13;
  -- clock instantiation
  clk : clock GENERIC MAP (f0) PORT MAP (en, ck);
  -- flipflop instantiation
  ff1 : jk_ff GENERIC MAP (f1'LENGTH) PORT MAP (j0, k0, ck, reset, im0);
  ff2 : jk_ff GENERIC MAP (f1'LENGTH) PORT MAP (j1, k1, ck, reset, im1);
  ff3 : jk_ff GENERIC MAP (f1'LENGTH) PORT MAP (j2, k2, ck, reset, im2);
  ff4 : jk_ff GENERIC MAP (f1'LENGTH) PORT MAP (j3, k3, ck, reset, q3);
  j0 <= vector('1', f1'LENGTH);
  k0 <= vector('1', f1'LENGTH);
  j1 <= im0; k1 <= im0;
  j2 <= im3; k2 <= im3
  j3 <= im4; k3 <= im4;
  im4 <= im0 AND im1;
  im5 <= im4 AND im2;
  q0 <= im0; q1 <= im1; q2 <= im2;
END dataflow;

Figure 17. Structural description of the 4 bit counter

The nibble counter is described structurally in Figure 17, using the clock and the JK flipflop
described previously. The bindings of the clock and the JK flipflop to the virtual component declarations
are also being done here. This could instead be done in the top level configuration. Provisions have been
provided in this structural description to fault all the nodes and intermediate signals. This is achieved
through the assignment of generic parameters f1 to f14. The individual clock signals in a single circuit,
going to different flipflops may not be faulted in this description. This is due to the limitation in VHDL in using predefined attributes.

The test bench that is written on top of all the parallel circuits is given in Figure 18. A generic parameter with the information of the number of parallelly driven circuits needs to be passed. The previously mentioned procedure 'probe' is being used in this test bench. Whenever an event occurs on any of the outputs (q0...q3), a comparison of the outputs will be done by this procedure.

```
USE WORK.utilities.ALL;
ENTITY counter_test IS
  GENERIC( circuits : INTEGER := 1);
END counter_test;

ARCHITECTURE test OF counter_test IS
  COMPONENT nibble_counter
    PORT (en, reset : IN node_vector; q0, q1, q2, q3 : OUT node_vector);
  END COMPONENT;
  SIGNAL enable, r : node_vector(1 TO circuits);
  SIGNAL q0, q1, q2, q3 : node_vector(1 TO circuits);
BEGIN
  cc : nibble_counter PORT MAP (enable, r, q0, q1, q2, q3);
  enable <= vector('1', circuits);
  r <= vector('0', circuits), vector('1', circuits) AFTER 490 NS,
     vector('0', circuits) AFTER 510 NS;
  probe(q0);
  probe(q1);
  probe(q2);
  probe(q3);
END test;
```

Figure 18. Test bench for the parallel simulation of nibble counter

The top level configuration declaration that binds all the entities underneath is given in the Figure 19. It is also being used to pass the fault information. In this configuration it was decided to simulate 6 circuits in parallel, with the circuit 1 being a good circuit. The following single faults have been injected to the circuits 2 through 6. All the other nodes are left as fault free.

```
im1 in circuit#2 - stuck at '0'
im2 in circuit#3 - stuck at '1'
im3 in circuit#4 - stuck at '0'
im4 in circuit#5 - stuck at '1'
im5 in circuit#6 - stuck at '0'
```

This configuration uses the function one_fault to inject faults to the circuits. However, if a particular node has to be faulted in more than one circuit, this function cannot be used. In order to achieve this, an array of fault values needs to be formed using concatenation. For example, to fault signal im3 (generic parameter f4 is assigned to this signal - Fig 17) to stuck-at-0 in circuit 2 and stuck-at-1 in circuit 3, the following has to be associated with f4 generic parameter. This association assumes 6 parallel circuits and can be included in a configuration declaration similar to that of Figure 19.

```
f4 => none&sa0&sa1&none&none&none
```
USE WORK.ALL;
USE WORK.utilities.ALL;
CONFIGURATION parallel OF counter_test IS
  FOR test
    FOR ALL : nibble_counter
      USE ENTITY WORK.nibble_counter(dataflow)
      GENERIC MAP(f0=> vector(none,6),
                    f1=> one_fault(6,sa0,2),
                    f2=> one_fault(6,sa1,3),
                    f3=> one_fault(6,sa0,4),
                    f4=> one_fault(6,sa1,5),
                    f5=> one_fault(6,sa0,6),
                    f6=> vector(none,6),
                    f7=> vector(none,6),
                    f8=> vector(none,6),
                    f9=> vector(none,6),
                    f10=> vector(none,6),
                    f11=> vector(none,6),
                    f12=> vector(none,6),
                    f13=> vector(none,6),
                    f14=> vector(none,6));
  END FOR;
END FOR;
END parallel;

Figure 19. The configuration at top level

The output of the simulation run is given in the Figure 20. When the output response of a circuit differs from the response of the fault-free circuit, the procedure 'probe' writes the simulation time and the number of the circuit in which error is being detected.

<table>
<thead>
<tr>
<th></th>
<th>NS: fault is detected in circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
</tr>
<tr>
<td>210</td>
<td>3</td>
</tr>
<tr>
<td>210</td>
<td>2</td>
</tr>
<tr>
<td>310</td>
<td>2</td>
</tr>
<tr>
<td>310</td>
<td>3</td>
</tr>
<tr>
<td>410</td>
<td>5</td>
</tr>
<tr>
<td>410</td>
<td>2</td>
</tr>
<tr>
<td>410</td>
<td>3</td>
</tr>
<tr>
<td>410</td>
<td>4</td>
</tr>
<tr>
<td>410</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 20. Simulation output

4.0 CONCLUSIONS

In this paper, we have presented a methodology of using VHDL in modeling digital circuits for parallel fault simulation. First, the faulting concepts were introduced using a simple example. Then a more complete example was used to illustrate the techniques of inserting faults from a top level configuration and performing parallel fault simulation. We have introduced a set of utility procedures to facilitate this parallel fault simulation. This new technique of fault simulation illustrates that the standard
VHDL language and its simulator can efficiently and conveniently be used for the demanding tasks such as fault simulation.

5.0 REFERENCES