A VHDL Based Test Environment Including Models for Equivalence Fault Collapsing

Zainalabedin Navabi and Massoud Shadfar
Electrical and Computer Engineering Department
Faculty of Engineering / Campus No. 2 University of Tehran
14399, Tehran IRAN
Tel: +98-21-800-8485; Fax: +98-21-688690; Email: navabi_z@irearn.bitnet

Abstract

In the area of digital system test, fault collapsing is referred to the process of reducing faults in a circuit to only those that can be distinguished. In local fault collapsing, one fault is selected from each of the equivalent fault classes of logical gates. The selection will be based on the connections made to the ports of a gate. We have developed VHDL gate models for local equivalent fault collapsing. This paper presents a VHDL based test environment that uses fault collapsing VHDL models to generate a list of faults for test generation and fault simulation. Our VHDL modeling strategy and examples will be discussed.

1. Introduction

Digital system test involves various tools for simulation, fault list generation, fault reduction, and test generation. Because test generation programs are time consuming, it is best to be able to generate tests for only those faults that can be distinguished, so that the redundancy in test vectors is minimized [3]. Fault collapsing is the process of reducing faults in a circuit to only those that can be distinguished. An inexpensive method for fault collapsing is the local fault collapsing. This methods is particularly efficient for gate level combinational circuits. In local equivalent fault collapsing faults on ports of logical gates are collapsed such that only one fault from each equivalent class of faults remain on the ports of the gate. A typical test environment uses fault collapsing to obtain a reduced fault list, uses test generation to generate test vectors for the list of faults, and uses fault simulation for discarding faults detected by a generated test.

In order to be able to use VHDL in an integrated design, simulation and test environment, appropriate VHDL models must be developed. In such an environment, the same VHDL gate level description can be bound to simulation models for design verification, to test models for test generation, or it can be bound to fault collapsing models for finding an optimum fault list.

VHDL gate models for local fault collapsing start with a set of all possible faults of a gate. The gate models by communicating through their IO ports decide on a minimum list of distinguishable faults and report such faults to a text file. Fault collapsing can be done for a VHDL gate level description by binding the structures of this description to the fault collapsing models. A fault list report generated for this description specifies a reduced list of faults for the entire circuit.

This paper presents VHDL models for local equivalent fault collapsing in gate level descriptions. Section 2 discusses issues related to fault collapsing. In Section 3 we will present an environment that relates fault collapsing to other test issues. Section 4 describes rules used in our VHDL models for fault collapsing. In Section 5 VHDL implementation and coding will be discussed. At the end, an example and conclusions will be presented.

2. Equivalence Fault Collapsing

Test generation for digital circuits can be done by functional analysis of the circuit or by considering individual circuit faults. A fault oriented test generation method usually deals with single stuck at faults (ssf) when are either stuck at 0 or stuck at 1 (*sa0,sa1*). This implies that the test generation scheme generates a structural model of a circuit with a single fault on a circuit line [2]. For a *sa0* (*sa1*) fault the line being faulted is assumed to have a 0 (1) value, and test generation schemes generates an input pattern at primary input to cause the value of the fault to propagates to the output. Faults in a circuit may

0-8186-6215-8/94 $03.00 © 1994 IEEE

be such that no sequence can be found to distinguish between them. These faults are said to be functionally equivalent, which implies that the function of the circuit is the same when either of the faults occur [3]. A digital circuit contains many equivalent faults, and the process of reducing list of faults to contain only those that can be distinguished (not equivalent) is called fault collapsing. For example for a gate with n-input and one output there may be 2(n+1) possible single stuck at faults. This is contributed by *sa0* and *sa1* at each input and the output. For an AND gate, stuck at 0 faults at all inputs and the output are functionally equivalent and can not be distinguished from each other. Therefore, distinguishable faults in an n-input gate are limited to n+2 ssf. Expanding this concept to an entire circuits, it is obvious that a good number of faults in a circuits are equivalent. A test generation program will only need to generate test for such distinguishable faults.

3. Fault Collapsing in Test Environment

In the previous section we indicted that many faults in a circuit are equivalent and the process of fault collapsing can be used to reduce a fault list to only those that can be distinguished. This section presents an outline of the usage of fault collapsing in a test environment. A test environment may consist of tools for test generation, simulation, and testability analysis. A possible scenario for using equivalent fault collapsing in a test environment is to generate an optimum fault list to direct test generation process.

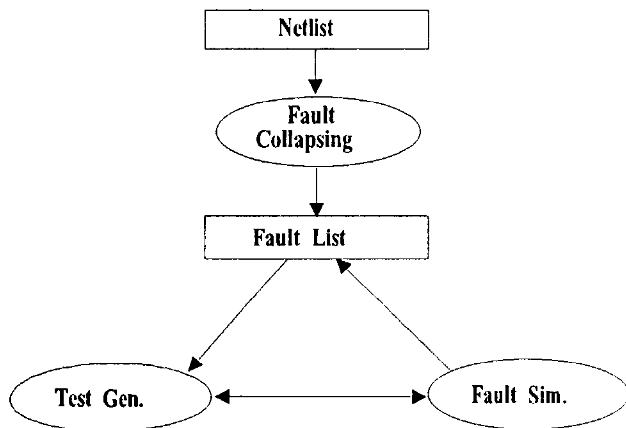

Figure 1. A general use of fault collapsing

Such a scenario, shown in Figure 1, uses a netlist for the input to the fault collapsing process. The result of this process is a reduced fault list in which only distinguishable faults are listed. A test generation program selects a target from a fault list and generates an appropriate test. This test is fed into a fault simulation program to remove all other faults that it detects. The process of test generation and fault simulation repeats until there are not any detectable faults in the fault list. Although fault collapsing can be based on fault equivalent and/or dominance relation we will only consider fault equivalent. A test pattern based on a fault list which is obtained this way will be able to find fault location as well as detecting single stuck at faults.

4. Local Fault Collapsing

Generally fault collapsing involves collapsing faults on adjacent nodes of a line, and removing equivalent faults on local I/O port of basic logic gates. This fault collapsing is usually refereed to as local gate level equivalent fault collapsing. A method based on local processing of gates and lines that can be efficiently implied by VHDL gate models will be presented here.

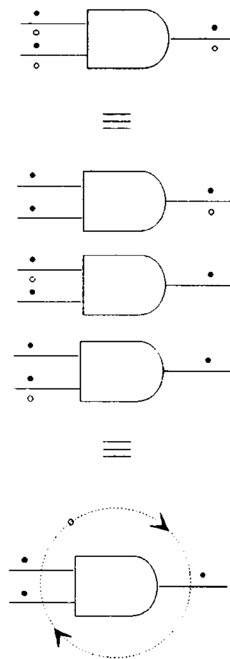

Figure 2. Float and Fixed faults

4.1. Isolated Structural Faults

Figure 2-a shows an AND gate with all possible faults at its inputs and outputs. A filled (hollow) circle

signifies an *sa0* (*sa1*) fault. It is clear that *sa0* fault at any of the inputs or at the output can not be distinguished when the output of the gate is observed or is propagated to next logic stages. Therefore a complete set of all distinguishable faults for this gate include *sa1*s (hollow circles) at the input and the output and a *sa0* at either of the input or the output, as shown in Figure 2-b. Another way of considering the three options in placement of the *sa0* fault is to consider it as a "*float fault*" that belongs to the AND gate and can go on any of its ports. This is shown in Figure 2-c. Non floats faults are referred to as *fixed faults* [4].

4. 2. Local Collapsing Rules

Float and fixed faults can be collapsed into their adjacent faults when gates are connected to fanouts, primary inputs or directly to outputs of the gates [5]. Figure 3 shows how various forms of these connections cause collapsing of gate input faults for a 2 input AND gate. These rules are local to each gate.

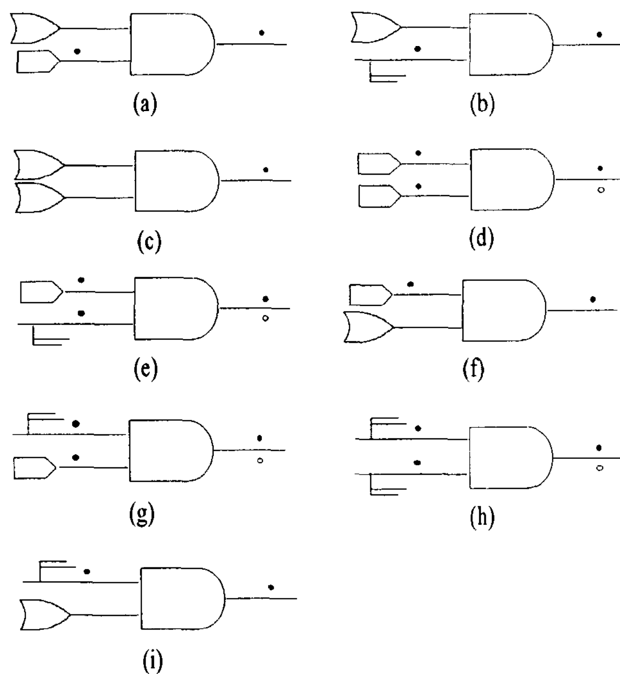

Figure 3. Fault collapsing based on input connections

If an input of a gate is driven by an output of another gate, the fixed fault at the gate input collapses into a similar fault at the output of the driving gate. For example in Figure 3-(a,b,c) if the driving gate connected to the input of the AND gate is output of an OR gate, the AND gate, based on the fact that the previous gate has the same fault at its output, removes the *sa1* fault at its input driven by the OR gate. Similarly the float fault that can go on any of the inputs or the output of the AND gate finds the output of the previous gate and appropriate place for itself to collapse into. There is a chance the driving gate can further collapse this fault into its driving input. If an input is connected to a primary input, the fixed fault at the gate input is collapsed into the same fault on the same connecting line driven by the primary input. This is illustrated in Figure 3-(d,e,f). The fault on gate input connecting to primary inputs is moved away from the gate to the primary input side of the connecting line. A float fault can collapse into faults on any of the ports of a gate. Since a primary input is a dead-end as far as further collapsing is concerned, we do not collapse a float fault into a primary input. For a gate with all its inputs connected to primary inputs, we have chosen the output to the gate to hold the float fault. This way, the float fault absorbs all similar faults at all primary inputs. Float fault in Figure 3-e finds a gate output at one of its inputs to collapse into. As shown in Figure 3-(g,h,i), float and fixed faults at gate inputs connected to fanouts behave like those of inputs connected to primary inputs.

Rules described in the previous paragraphs specify the placement of a fault on a specific end of a connecting line. Clearly, a fault can be placed on either end of a line. In the implementation of our models, faults are generated and reported by the gate that uses a line as input.

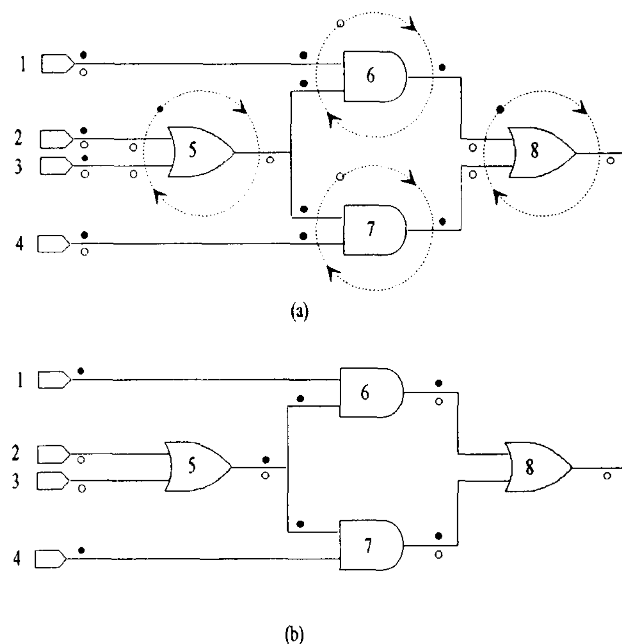

Figure 4. Example collapsing

Rules described here can be justified by use of a simple example. Figure 4-a shows a circuit with fixed and float faults at gate ports, primary inputs and fanouts. Figure 4-b shows the resulting faults after applying the above rules. Gate 8 has three fixed and a float fault. The fixed faults at its inputs are collapsed into the outputs of gates 6 and 7, and its float fault can be collapsed into any of these gates. Gate 6 collapses both its fixed faults into lines that it is connected to. The float fault on this gate absorbs both similar faults at the PI and the fanout, and appears only on the output of the gate. Application of our rules reduce the number of faults from 24 to only 13. As described earlier, faults on fanout stems can be reported by the gates connected to the stems.

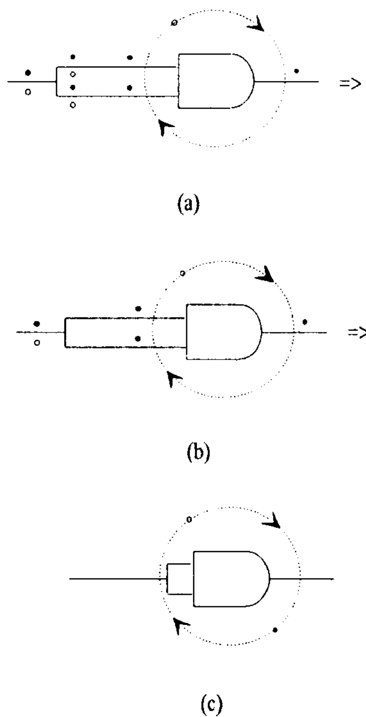

Figure 5. Faults for Special Input Connections

A special situation arises when a gate has input stems branched from the same fanout, as show in Figure 5. Application of our rules results in faults shown in Figure 5-b. Faults on lines A ,B, and are all equivalent and, can be represented by a similar fault on line A or the output. On the other hand, the float fault on the output of the gate can absorb the *sa0* fault on line A resulting into a single float fault. Circuit with the reduced faults is shown in figure 5(c). This figure shows two float fault for the entire gate structure. These faults can collapse into faults of connecting structure by applying rules of this section. It is worth while to mention that a buffer and an inverter, Figure 6, also have two float faults as is the structure of Figure 5.

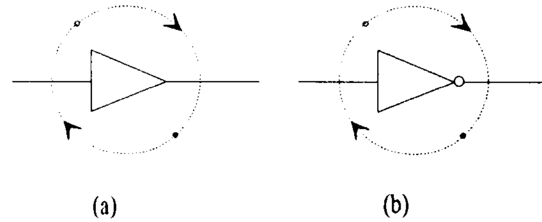

Figure 6. Float Faults In (a) Buffer (b) Inverter

5. VHDL Implementation

In the previous section we presented rules for a fault collapsing method. This method is a variation of several existing methods which have been adapted so that their combination can work as independent local VHDL models. In this section the general outline of VHDL codes for the implementation of these models will be discussed. We will present types used, resolution functions and a typical gate model [6],[7].

5.1. General Structures

A simulation run begins with all gates having all possible faults on their ports. The models are designed such that they report faults that remain on their input and output lines after fault collapsing has been completed. Primary inputs are treated as components that originally contain *sa0* and *sa1* faults on their only output line. These component also report collapsed faults if they drive fanouts. In order to be able to pass information to and from various components via their inputs and outputs, all signals are of INOUT mode and resolved.

```
TYPE carrier IS RECORD
  N:node_r;
  S:stem_r;
  I:id_r;
  F:nin_r;
END RECORD;
```

Figure 7. Signal Types

5.2. Signal Types

All interconnecting signals are of a record type shown in Figure 7. This record (*carrier*) contains fields

for passing enough information between gates so that each gate can decide on faults that collapse on its ports. Base type declaration for the fields of this record are depicted in Figure 8.

```
TYPE node IS (branch, lin, tree);
TYPE stem IS (gate_in, gate_out, primary_in);
SUBTYPE id  IS natural;
SUBTYPE nin  IS natural;
```

Figure 8. Base Type of Fields

The four types shown in this figure use resolution functions identified by *node_f*, *stem_f*, *id_f*, and *nin_f*. The base type for field *N* is *node*. The *node_f* resolution function is shown in Figure 9. This field is used for distinguishing between simple lines and fanouts.

All component ports report to this function. The function counts the number of connecting lines and returns *lin* if there are only two drivers on the end of a line, and returns *tree* if there are more than two connections (fanout). Since rules of Section 4 treat fanout differently, this information is needed by the models.

```
FUNCTION node_f (drivers: node_vector) RETURN
node
IS
 BEGIN
   IF drivers'LENGTH > 2 THEN
     RETURN tree;
   ELSE
     RETURN lin;
   END IF;
END node_f;
```

Figure 9. Node_f Resolution Function

The next field of the *carrier* record is of type *stem*. This field (*S*) uses the *stem_f* resolution function shown in Figure 10 and is used for identifying the logic driver of a line. The values this field can take are *gate_in*, *primary_in* or *gate_out*. If a line is connected to a gate output, the *S* field value becomes *gate_out*, and if it is connected to a primary input *S* becomes *primary_in*.

```
FUNCTION   stem_f (drivers: stem_vector)   RETURN
stem
IS
 BEGIN
    FOR i IN drivers'RANGE LOOP
      IF drivers(i) = primary_in THEN
        RETURN primary_in;
      ELSIF drivers(i) = gate_out THEN
        RETURN gate_out;
      END IF;
    END LOOP;
        RETURN  gate_in;
 END stem_f;
```

Figure 10. Stem_f Resolution Function

The *I* field of the *carrier* record is of type *id_r*, and returns the id number of the gate or PI where output is connected to a line. The id number of each gate is a unique natural number produced by an up-counter. Input lines assign 0 to the *id_f* resolution function and this function, shown in Figure 11, returns the only non zero driving value.

```
FUNCTION id_f ( drivers: id_vector)  RETURN id is
BEGIN
    FOR i IN drivers'RANGE LOOP
      IF drivers(i)  >  0
      THEN  RETURN  drivers(i);
      END IF;
    END LOOP;
    RETURN  0;
END id_f;
```

Figure 11. Id_f Resolution Function

The *id* field is necessary for the implementation of rules for collapsing faults on several fanout stems driving the inputs of the same gate, Figure 5. Such a gate knows of its special case when it finds the same id on both its inputs.

```
FUNCTION nin_f ( drivers: nin_vector)  RETURN nin IS
BEGIN
   RETURN  (drivers'LENGTH -1);
END nin_f;
```

Figure 12. Nin_f Resolution Function

The *F* field is of type *nin_r* which is used with the *nin_f* resolution function shown in Figure 12. The *nin_f* resolution function returns the total number of its connecting lines. This information is used by gates with more than one input connected to the same fanout. Reading *nin*, such a gate will recognize if fanout is also feeding other gate structures. Only in this case a fanout is treated the same as other fanouts. The example of Figure 13(a) shows a fanout which we will treat as a regular node, while Figure 13(b) shows a fanout node of two stems.

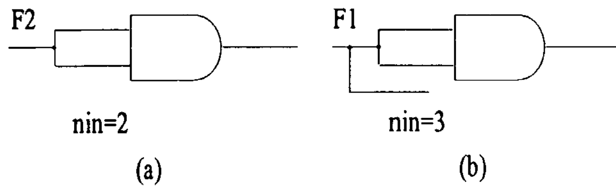

Figure 13. Use of *nin* for identifying real fanouts (a) Not a fanout, (b) A fanout

5.3. Gate Architecture

Using types and resolution functions described above, architecture for basic logical gates have been developed. Each gate is identified by a unique id number passed to it via a generic parameter. This number appears in all fault reports in a text file. Gates wait for a certain amount time that is proportional to their *gate_id* before reporting their faults to the text file. Therefore fault report is sorted according to the gate id numbers.

As shown in Figure 14 each gate provides information regarding its line type, id, and input or output line on all its bi-directional IO ports. This is done by three concurrent signal assignments to the input and output ports of a two-input gate. The two-input NAND gate of Figure 14, for example, places *branch* on the node field of its ports. The gate reads from this field either *lin* or *tree* indicating a fanout of it exists. On the *stem* field (for clarity, field is being referred to by its type, instead of its name) the output places *gate_out* and the inputs place *gate_in*. The *gate_id* is only placed on the *id* and *nin* fields of the outputs. The information provided to the ports will be accumulated by the appropriate resolution functions and the collected information will become available to all gates connected to a node. This information appear in the N, S, or I fields of the ports and specify the type of the gate connected to a line, e.g., PI, PO, gate_output, or fanout.

```
ENTITY nand2_fc IS
  GENERIC (gate_id: NATURAL);
  PORT (i1, i2, o1: INOUT status);
END nand2_fc;
--
ARCHITECTURE collapse OF nand2_fc IS
BEGIN
  o1 <= (branch, gate_out, gate_id, gate_id);
  i1 <= (branch, gate_in, 0, 0);
  i2 <= (branch, gate_in, 0, 0);
 collapsing: PROCESS
 BEGIN
  . . .
  WAIT FOR gate_id*1 NS;

  IF (i1.N = lin) AND (i1.S = gate_out)
  THEN
    IF (i2.N = lin) AND (i2.S = gate_out)
      THEN  . . . report Case 3-c
    ELSIF (i2.N = tree) OR
           ((i2.N = lin) AND (i2.S = primary_in))
      THEN . . . report Case 3-b or 3-a
    END IF;
  ELSIF (i1.N = lin) AND (i1.S = primary_in)
  THEN
    IF (i2.N = lin) AND (i2.S = gate_out)
      THEN . . . report Case 3-f
    ELSIF (i2.N = tree) OR
           ((i2.N = lin) AND (i2.S = primary_in))
      THEN . . . report Case 3-e or 3-d
    END IF;
  ELSIF (i1.N = tree) THEN
    IF (i2.N = lin) AND (i2.S = gate_out)
      THEN . . . report Case 3-i
    ELSIF (i2.N = lin) AND (i2.S = primary_in)
      THEN . . . report Case 3-g
    ELSIF (i2.N = tree) AND (i2.I /= i1.I)
      THEN . . . report Case 3-h
    ELSIF  (i2.N = tree) AND (i2.I = i1.I) AND
            ((i2.S = gate_out) OR (i2.S = primary_in))
            AND (i2.F = 2 )
      THEN . . . report Case 13-a
    ELSIF
         (i2.N = tree) AND (i2.I = i1.I) AND (i2.F > 2 )
      THEN . . . report Case 13-b
    END IF;
  END IF;
  WAIT;
 END PROCESS collapsing;
END collapse;
```

Figure 14. NAND Fault Collapsing Model

If a line is connected to a fanout, the number of fanouts and the gate identification of all gates connected to that fanout will also be provided. All port field information will be used by the collapsing process of a gate.

In the collapsing process of a gate architecture, a wait statement delays fault calculation and reporting by (gate_id * 1 FS). This is necessary so that the final report will be sorted according to the gate identification number. The collapsing process continues by checking port fields for cases of Figures 3 and 13. When a case is found, faults will be reported.

In the collapsing process, after the WAIT statement, an IF statement checks the following:

(i1.N = lin) AND (i1.S = gate_out)

This condition is true if *i1* is not a fanout (lin for a simple line, and tree for fanout), and it is driven by a gate output. Referring to Figure 3, this condition separates cases 3-a, b, or c from all other cases of this figure. Since faults on the ports of a gate for the case of Figure 3-c is different than that of Figures 3-a and b, a second IF statement is used to separate case of Figure 3-c from the other two cases. The condition used for this purpose is:

(i2.N = lin) AND (i2.S = gate_out)

This condition will be true if the second input of the gate being considered is a simple line driven by another gate output. If the conditions mentioned above are both true, case of Figure 3-c is separated from all other cases, and a fault report indicating a sa0 fault on the output will be generated. An example report is shown below. Note that differences in faults shown in Figure 3 and those described in this section are due to the use of NAND gate in this section as opposed to the AND gate in the example of Figure 3.

[I1:none; I2:none; O1:sa_0]

Since faults on the ports of a gate for cases shown in Figures 3-a and 3-b are the same, a single If statement separates these cases and produces the appropriate report. The condition checked is:

(i2.N = tree) OR ((i2.N = lin) AND (i2.S = primary_in))

and the produces report for cases of Figures 3-a and b is:

[I1:none; I2:sa_1; O1:sa_0]

The collapsing process continues with checking all remaining cases of Figure 3 and producing appropriate reports. Various cases of this figure are shown in Figure 14 in Bold. The last section of this process will be reached if both inputs of a gate are fanouts. If the fanouts of the two inputs are independent, case of Figure 3-h is observed and the following report will be generated:

[I1:sa_1; I2:sa_1; O1:sa_0,sa_1]

On the other hand, if the two inputs are connected to the same none, the I fields of the input ports are checked for equality. If these fields are equal, and the number of fanouts at the node is 2 (F field of any of the inputs is 2), then a situation similar to that of Figure 13-a is detected and according to the analysis of Figure 5, this reduces to no faults on the ports of the gate. If the I fields are equal and the fanout at the common node is more than 2, then a case similar to that of Figure 13-b is found. In this case the following report will be generated for a NAND gate:

[I1:none; I2:none; O1:sa_0, sa_1]

The above discussion along with the partial VHDL code of Figure 14 cover all cases that can occur in a two-input NAND gate. Code for other logic gates are similar to that of Figure 14.

```
ARCHITECTURE collapse OF pi_fc IS
BEGIN
  o <= (branch, primary_in, pi_id, pi_id);
  collapsing: PROCESS
  BEGIN
    . . .
    WAIT FOR pi_id*1 NS;

    IF ( o.N = tree )
      THEN . . . report [ sa_1, sa_0 ]
     ELSE . . . report [  none  ]
     END IF;
     WAIT;
  END PROCESS collapsing;
END collapse;
```

Figure 15. PI Fault Collapsing Model

Another structure that deserves attention is a Primary input. Figure 15 shows the pseudo code for a PI. As in the NAND gate, the only port of the PI (its output port) reports all its properties to the gates connected to it. Also, a WAIT statement delays fault reporting according to the gate identification number. In the collapsing process of a PI, the N filed of the PI output is checked for

fanout. If a fanout exists, sa0 and sa1 are reported on the primary input. Otherwise, no faults are reported.

When all gates of a gate level description are instantiated, a sorted report of all collapsed faults will be generated.

```
...
G108: and2
        GENERIC MAP(108)
        PORT MAP(y(5),y(3),y(9));
G109: or2
        GENERIC MAP(109)
        PORT MAP(y(6),y(7),y(8));
G110: nor2
        GENERIC MAP(110)
        PORT MAP(y(10),y(9),y(11));
G111: nor2
        GENERIC MAP(111)
        PORT MAP(y(8),y(2),y(12));
G112: not1
        GENERIC MAP(112)
         PORT MAP(z(1),z(3));
...
```

Figure 16. Instantiation List

6. An ALU Example

Using our fault collapsing models, gate level circuit of a 74LS181 four bit ALU was analyzed. The main body of the ALU model is simply an instantiation list, a portion of which is shown in Figure 16.

```
Gate_TYPE (id_number)
      [ SSF list for gate nodes      ]
----------------------------------------
...
AND2 ( 108   )
      [ I1:none;   I2:sa_1;   O1:sa_1 ]
OR2  ( 109   )
      [ I1:none;   I2:none;   O1:sa_0 ]
NOR2 ( 110   )
      [ I1:none;   I2:none;   O1:sa_1 ]
NOR2 ( 111   )
      [ I1:none;   I2:sa_0;   O1:sa_1 ]
NOT  ( 112   )
      [ O:sa_1, sa_0 ]
...
```

Figure 17. Partial Fault Report

This circuit has 121 components consisting of NAND, NOR, PI and PO structures. For fault collapsing this architecture is instantiated in a test bench and a report file is generated. The simulation ceases after all gates report faults that remain on their ports after fault collapsing. Figure 17 shows a partial report generated for this circuit. A total of 288 faults were reported, while the number of original uncollapsed faults was 426. This is a 30% reduction in a circuit that contains a relatively high number of fanouts.

CONCLUSIONS

This paper presented VHDL models for local equivalent fault collapsing. This and other test related applications benefit from VHDL ability for hierarchical and configurable hardware descriptions. Concurrency in VHDL was particularly useful for modeling for fault collapsing. Such modelings in VHDL make this language and its related tools, a platform that can provide simulation, verification, synthesis and test in the same environment.

REFERENCES

[1] P. Goel. " Test generation costs analysis and projections." in *Proc. DAC-17. 17th IEEE ACM Design Automation. Conf.* . June 1980, pp. 77-84.

[2] J. A. Abraham, "Fault Modeling in VLSI," in *VLSI Testing*, ed. T. W. Williams, North Holland, Amsterdam, The Netherlands, 1986.

[3] E. J. McCluskey and F. W. Clegg, "Fault Equivalence in Combinational Logic Networks," *IEEE Trans. Computers*, Vol. C-20, pp. 1286-1293, November 1971.

[4] D. R. Schertz and G. Metze, " A New Representation for Faults in Combinational Digital Circuits," *IEEE Trans. Computers*, Vol. C-21, pp. 858-866, August 1972.

[5] K. To. "Fault folding for irredundant and redundant combinational circuits."*IEEE Trans. Comput.*, Vol . C-22, no. 11.pp.1008-1015, Nov. 1973.

[6] Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems, McGraw Hill, New York, 1993.*

[7] Z. Navabi, N. Cooray and R. Liyanage. " Using VHDL in parallel fault simulation " *proc. scs., International Conference on Simulation in Engineering Education. January 17-20, 1993,*Vol . C-25, no. 3.pp.198-203.