

Behavioral Delay Fault Modeling and Test Generation

by

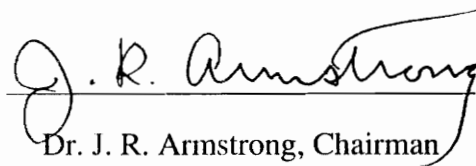
Anand Mukund Joshi

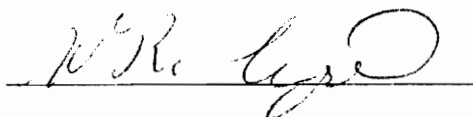
Thesis Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

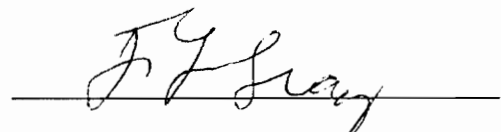
in

Electrical Engineering

APPROVED


Dr. J. R. Armstrong, Chairman


Dr. W. R. Cyre


Dr. F. G. Gray

May 1994

Blacksburg, Virginia

LD
5655
V855
1993
J674
C.2

Behavioral Delay Fault Modeling and Test Generation

by

Anand M. Joshi

Dr. J. R. Armstrong , Chairman

Electrical Engineering

(ABSTRACT)

As the speed of operation of VLSI devices has increased, delay fault testing has become a more important factor in VLSI testing. Due to the large number of gates in a VLSI circuit, the gate level test generation methodologies may become infeasible for delay test generation.

In this work, a new behavioral delay fault model that aims at simplifying the delay test generation problem for digital circuits is presented. The model is defined using VHDL. It is shown that each defined behavioral level delay fault can be mapped to a gate level equivalent fault and/or physical failure. A systematic way of representing a behavioral model in terms of a data flow graph is presented. A behavioral level input-output path is defined and a strategy to generate tests for delay faults along a behavioral path is presented. It is then shown that tests developed from the behavioral model can test a gate level equivalent circuit for path delay faults.

Acknowledgments

I am very much thankful to Dr. Armstrong for giving me opportunity to work on this interesting project. His patience and guidance was invaluable during the course of my graduate studies. I would also like to thank Dr. Gray and Dr. Cyre for serving as members of my committee.

It has been a pleasure to work with members of the VHDL group. Special thanks are due to Chang Cho and Dhawal Tyagi who spent a lot of time answering my stupid questions. Many many thanks are due to my lunch time partners, Chippur, Vik and Q. Without them, it would have been impossible to survive in the CEL lab. I would also like to thank other lab members, Shankar, RP, Rekha, Sanat, Gunjeet and John, whose sense of humor kept CEL alive.

I would like to dedicate this work to my parents, who make this life worth living. It would have been impossible for me to complete this work without their constant support and encouragement. Finally I would like to thank GOD for giving me the power to keep my conscience alive even in the most difficult times.

Table of Contents

Chapter 1. Introduction.....	1
1.1 Motivation.....	1
1.2 Contents.....	6
 Chapter 2: Gate Level Delay Fault Model.....	 8
2.1.1 Stuck-at Fault Model.....	8
2.1.2 Physical Failures in the Layout and Timing Considerations	10
2.2 Delay Faults: Gate Level Models.....	12
2.2.1 Localized Delay Fault Model.....	12
2.2.2 Distributed Delay Fault Model.....	14
2.3 Effect of delay faults.....	15
2.4 Test Generation for Delay Faults.....	19
2.4.1 Test Generation for a Localized Fault.....	19
2.4.2 Test Generation for a Distributed Fault.....	20
 Chapter 3 Behavioral Level Delay Fault Model.....	 23
3.1 Rationale for a high level fault model.....	23
3.2 Behavioral Level Delay Faults.....	26
3.2.1 Delay Faults in after statement.....	27
3.2.2 Transport/Inertial Delay Fault.....	29
3.2.3 Delay Fault in Generics.....	30

3.2.3 Delay Fault in wait statements.....	31
3.2.4 Delay fault in attributes.....	33
3.3 Fault Injection.....	35.
3.3.1 Fault injection in after clause.....	35
3.3.2 Transport/Inertial Delay Fault.....	36
3.3.3 Fault Injection in generics.....	37
3.3.4 Fault Injection in wait statements.....	37
3.3.5 Fault injection in attributes.....	38
3.4 Fault Sensitization	39
3.4.1 Sensitization of delayed after faults.....	39
3.4.2 Sensitization of fault in generic.....	42
3.4.3 Sensitization of fault in wait statement.....	42
3.4.4Sensitization of faults in attribute.....	43
3.5 Fault Sets.....	44
 Chapter 4. mapping of the behavioral level delay fault model.....	 48
4.1 Mapping of a behavioral level path to a gate level path.....	48
4.2 Mapping of a the behavioral level fault model to gate level fault model.....	56
4.2.1 Mapping of delayed after faults.....	56
4.2.2 A fault model for transport/inertial delay fault.....	63
4.2.3 Mapping of fault in Generic.....	65
4.2.4 Mapping of Fault in Wait statement.....	68
4.2.5 Mapping of faults in a attributes.....	69
4.3 Pure Behavioral Faults: Another Consideration.....	74

Chapter 5. Behavioral Level Input-Output Paths.....	77
5.1 Analysis of statements.....	77
5.1.1 A simple example of a DFG.....	78
5.1.2 If-then-else/ case statement.....	79
5.1.3 Graphical Representation of Wait Statement.....	81
5.1.4 representation of Attributes.....	81
5.2 Construction of behavioral paths: Examples.....	82
 Chapter 6. Behavioral Delay test generation.....	 102
6.1 Dummy Nodes.....	102
6.2 Test generation.....	106
6.2.1 General Considerations.....	106
6.2.2Strategy I.....	108
6.2.3 Strategy II.....	108
6.3 An example.....	111
6.3.1 Strategy I.....	112
6.3.2 Strategy II.....	119
6.4 Validation of the test at the gate level.....	126
6.4.1 Discussion.....	128
6.5 Generalization of the test procedure.....	129
6.5.1 Sensitization of successive nodes.....	129
6.5.2 A behavioral model.....	134
6.6 Test Generation for behavioral Path: Examples.....	137
6.7 Hierarchical Delay test generation.....	154

6.8 Test generation for MCM interconnect..... 158

Chapter 7. Conclusions and Future Scope..... 162

References..... 165

Vita..... 170

List of Illustrations

Figure 1: A typical MCM

Figure 2: A MCM

Figure 3: Failures in a NAND gate

Figure 4: Gate level representation of the physical failures

Figure 5: Nor gate

Figure 6 . Representation of a net as a transmission line

Figure 7: Localized Fault

Figure 8 : Localized Delay Fault Propagation

Figure 9. Distributed delay fault

Figure 10: Delay faults

Figure 11: Fault in the ready signal

Figure 12: Fault in bus causing bus contention

Figure 13: Example of delay test generation: localized fault

Figure 14: Path fault

Figure 15 : Non-robust test for a path

Figure 16: fault Sets

Figure 17: Data Flow Graph

Figure 18: A behavioral and gate level model

Figure 19: Bus delay fault

Figure 20: Connection and Disconnection fault

Figure 21. Fault in after/transport after statement

Figure 22: Fault model for transport / inertial fault

Figure 23. Transistor level schematic of Inverter

Figure 24. Global generic fault

Figure 25. Wait statement

Figure 26: And gate with interconnection delay

Figure 27: Pulse generator

Figure 28 A monostable multivibrator

Figure 29: A pure behavioral fault

Figure 30: Data Flow Graph of the statement $f = (u + v) * (v + w)$

Figure 31: if-then-else statement

Figure 32: Case statement

Figure 33: Representation of *Wait for* statement

Figure 34: Attribute representation

Figure 35: A DFG of a 2-to-1 mux

Figure 36: DFG corresponding to a MUX with Enable

Figure 37: An Arithmetic Logic unit

Figure 38: DFG of an ALU

Figure 39: Partial DFG of example 1

Figure 40: Example 5

Figure 41: Partial DFG of example 5

Figure 42: DFG of a buffer

Figure 43: 2-1 MUX: Gate level equivalent

Figure 44: A DFG of a 2-to-1 mux

Figure 45: DFG of mux with enable

Figure 46: Fault dat1

Figure 47: Fault daf4

Figure 48: Fault daf4

Figure 49: 2-1 mux: delayed after fault

Figure 50: Path A-1-3-4-5-7-8-O

Figure 51: Path SEL-7-8-9-10-O

Figure 52: Path A-3-5-O

Figure 53: Path SEL-4-5-O

Figure 54: Path SEL-4-5-O

Figure 55. Sensitization of two delay nodes

Figure 55: A control node followed by a delay node

Figure 56: A control node followed by a control node

Figure 57: Control node followed by a data node

Figure 58: DFG of a behavioral mode

Figure 59: Partial DFG of example 1

Figure 60: Path CON-1-6-LOAD

Figure 61: Path INCNT-7-OPCNT

Figure 62: DFG of a buffer

Figure 63: Path DS1-1-5-6-DO

Figure 64: Path DI-4-6-O

Figure 65: Fault in generic

Figure 66: Fault in generics

Figure 67: Fault in generics

Figure 68: DFG of a DFF

Figure 69: Fault along the path D-2-4-5-Q

Figure 70. A process at the behavioral level

Figure 71. Process Model Graph

Figure 72: Mixed level description of a MCM

Figure 73. Process Model Graph

Figure 74. Delay faults within a process

Chapter 1

INTRODUCTION

1.1 Motivation

The complexity of ICs has been increasing at a rapid rate over the past few years. Using modern VLSI technology, it is possible to produce a chip containing over a million transistors. For example, Intel's Pentium chip contains over 3 million transistors[1]. The trend in chip design is slowly moving towards Multi Chip Module (MCM) and a typical VLSI Chip On Silicon (VCOS) MCM may contain as many as 400 chips on a single substrate[2]. At the same time, the speed of operation is increasing. For example Intel's Pentium chip can operate at 66 MHz, ten times faster than the 8085 processor.

Testing is an experiment in which the system is exercised and its response is analyzed to ascertain whether it behaved correctly. The cost of locating a faulty component increases exponentially from the chip level to field level (Table I). If it costs 50 ₱ to locate a fault at the chip level then it costs \$ 500 to locate the same fault at the field level[3]. It is far more difficult to locate a fault at the field level than at the chip level. Thus it is an

important consideration that all the failures in a chip be detected as early as possible in order to minimize the cost of locating a faulty component.

Table I

Chip Level	50 ¢
Board Level	\$ 5
System Level	\$ 50
Field level	\$ 500

Increases in the speed and complexity in recent years has generated a heavy demand for the thorough testing of a chip. The traditional test generation approaches have relied on the stuck-at fault model. The model, although under considerable fire [4, 5], still remains the most predominant model in the industry. The model assumes that a line at the input or output of a gate is stuck at 1 or 0 value and modifies the behavior of that gate accordingly. Different test generation schemes for this fault are proposed in [6, 7, 8]. However, as the number of gates in an IC is increasing, it is becoming difficult and time consuming to generate tests using the gate level fault model.

There have been attempts in the past to model faults at the behavioral level and generate tests. The major goal of behavioral level test generation is to reduce the test generation complexity and computational time by using high level primitives in the test generation process[13, 14, 15]. Since there are fewer number of components, fewer evaluations are performed during the test generation. Also the number of signals in the high level

description is significantly reduced as compared to the gate level description. In addition, at this level it is possible to exploit hierarchy to generate tests.

As the speed of operation of a chip is increasing, tighter timing constraints have given rise to another type of faults called **delay faults** or **timing faults**. When a delay through an input-output path exceeds the normal value, it gives rise to a **delay fault**. These faults are related to the dynamic behavior of the circuit and two different fault models for this fault are proposed in [9, 10].

The **distributed fault model** proposed in [9] assumes that the delay fault exists along an input output path. The **localized fault model** proposed in [10] assumes that the delay fault exists at the input or output of a gate.

For the distributed fault model proposed in [9], it is desired that input-output paths in the circuit be tested for delay faults. Again with increasing number of gates, the number of input output paths increases exponentially with each fanout stem. This makes it impossible to generate test for each path in the circuit. To overcome this, strategies are developed in [11] to select a subset of all paths and tests are generated. Also it is shown in [13] that tests developed for a certain set of path should be able to give the 100% delay fault coverage.

This is the first motivation for the proposed work. The question considered here is: is it possible to define a behavioral level delay fault model using a hardware description language that can simplify the delay test generation ? If yes, then can it be mapped to physical failures in the circuit ?

Another motivating factor is MCM testing. An MCM consists of several chips connected on a silicon substrate. (Figure 1). Because of the very large number of gates involved, the gate level test generation principles followed today may become impractical at the MCM level.

As the number of vendors supplying a single functional chip for an MCM will increase, the gate level description may or may not be available for all the ICs. (Figure 2). In this case, it will be necessary to generate a test for the complete MCM using the mixed level description available. In such case a behavioral fault model would be extremely helpful in generating tests for an MCM. This model should consider the following aspects:

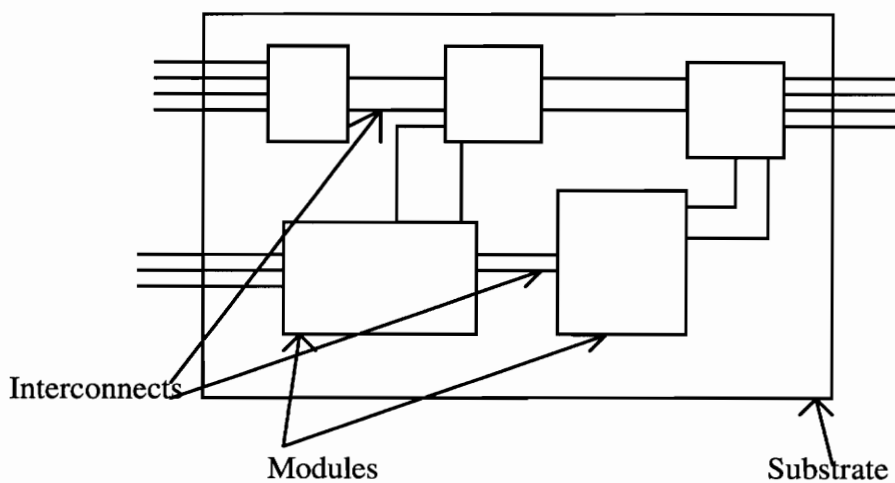


Figure 1: A typical MCM

1. be able to consider the behavioral description,
2. be able to map to physical failures in a circuit,

3. and, should simplify the delay test generation problem.

For an MCM, test generation for interconnects is another important consideration[2]. Each module can be tested separately before placing it on the substrate. Then interconnects can be tested for delay faults. Test generation for delay faults in interconnections using the behavioral level fault model can reduce the complexity of test generation problem significantly.

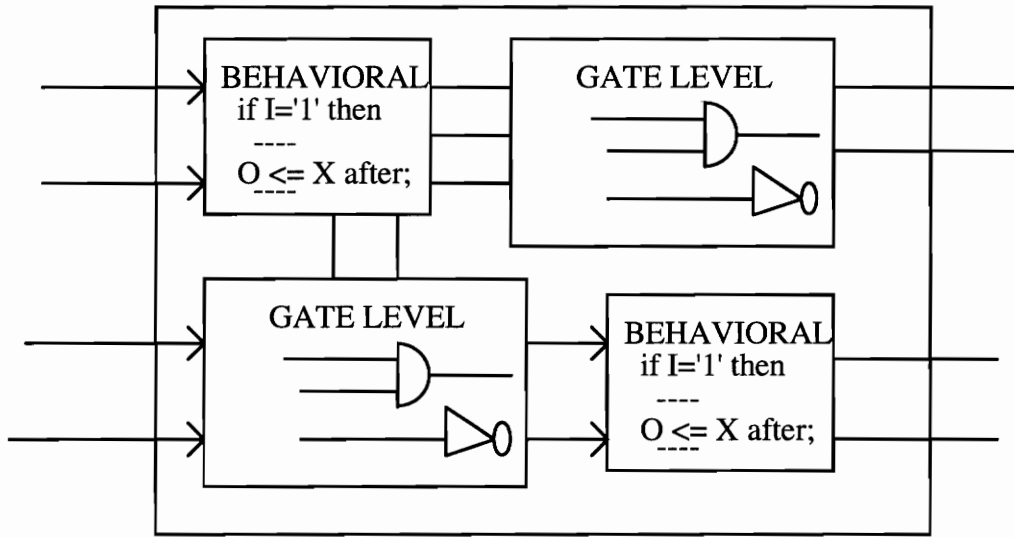


Figure 2: A MCM

To solve the problems stated above, in this work a new delay fault model is defined.

A hardware description language such as VHDL is capable of modeling at the behavioral as well as structural level of abstraction. In the present work, VHDL is used to define delay faults. Various timing constructs in the language are perturbed to define behavioral level delay faults. It is shown that each of these faults can be mapped to a gate level

equivalent fault or physical failure. A Data Flow Graph (DFG) representation of the behavioral model is used to define the input-output paths implied by the behavioral model. It is then proved that each path implied by the DFG maps to an equivalent path in the gate level description of the circuit. A systematic strategy to generate a list of all the behavioral i/o paths is presented. A strategy to generate delay tests for a behavioral level delay fault is presented. Application of the fault model for MCM interconnection is also discussed.

1.2 Contents

Chapter 2, "Gate Level Delay Fault Model", discusses the gate level delay fault model and the test generation strategy at the gate level.

Chapter 3, "Behavioral Level Delay Fault Model" presents the behavioral level fault model. Behavioral level faults are categorized and a systematic way to sensitize these faults is also presented.

Chapter 4, "Mapping of the Behavioral Level Fault Model" explains the possible mapping of the behavioral level fault model to lower level descriptions.

Chapter 5, "Behavioral Input-Output Paths" presents a systematic way to generate a list of all the behavioral level paths implied by the model.

Chapter 6 "Behavioral Delay Test Generation" discusses a test generation scheme for behavioral delay faults.

Chapter 7 "Conclusion and Future Scope" discusses the possibility of the automation of the delay test generation procedure.

Chapter 2

GATE LEVEL DELAY FAULT MODEL

2.1.1 Stuck-at Fault Model

The test generation of present VLSI circuits has mostly relied on the conventional stuck-at model. In this model, failures in the transistor level layout are mapped to a gate level equivalent representation [17, 18]. Consider faults in a CMOS NAND gate shown in Figure 3. The open at point a and the short between the ground and point b is likely to modify the behavior of the functional NAND gate as shown in Table II.

Table II

A	B	F	F faulty
0	0	1	1
0	1	1	0
1	0	1	1
1	1	0	0

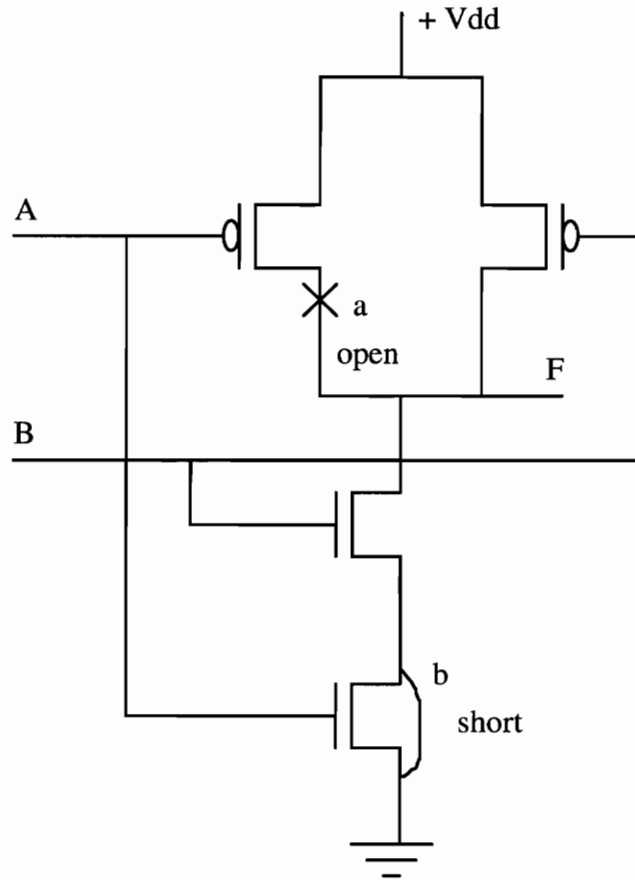


Figure 3: Failures in a NAND gate

At the gate level this behavior can be modeled using the stuck-at model as shown in Figure 4. This is an equivalent representation of physical failures in the transistor level layout of that gate. The input A, at the gate level, is assumed to be stuck at 1 and this represents the same behavior as that of the fault caused by physical failures shown in Figure 3. Here it is important to note that the line A stuck at 1 or stuck at 0 does not mean that the line is connected to V_{dd} or ground but is rather a representation of failures in the circuit.

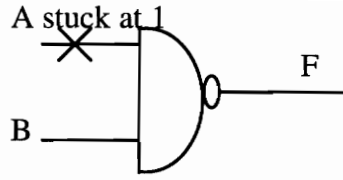


Figure 4: Gate level representation of the physical failures

2.1.2 Physical Failures in the Layout and Timing Considerations

As the speed of VLSI devices is increasing, timing is becoming a critical factor in the reliable operation of a circuit. The conventional stuck-at model is inadequate to model the dynamic behavior of a circuit. To test the delay and timing, new models are proposed in [9, 10]. These models consider the delay involved in a circuit and thus test the dynamic behavior of the circuit.

To explain timing failures caused by physical failures in the circuit, consider the NOR gate shown in Figure 5.

If an open is developed in the circuit at point a because of a process fault, then the floating gate of the pmos transistor affects the charging time at the output. It is shown in [18] that the output of the transistor shows six times the normal delay for a low to high transition. This gives rise to what is called a **delay fault**, i.e., a fault caused by a change in the point to point delay associated with a gate.

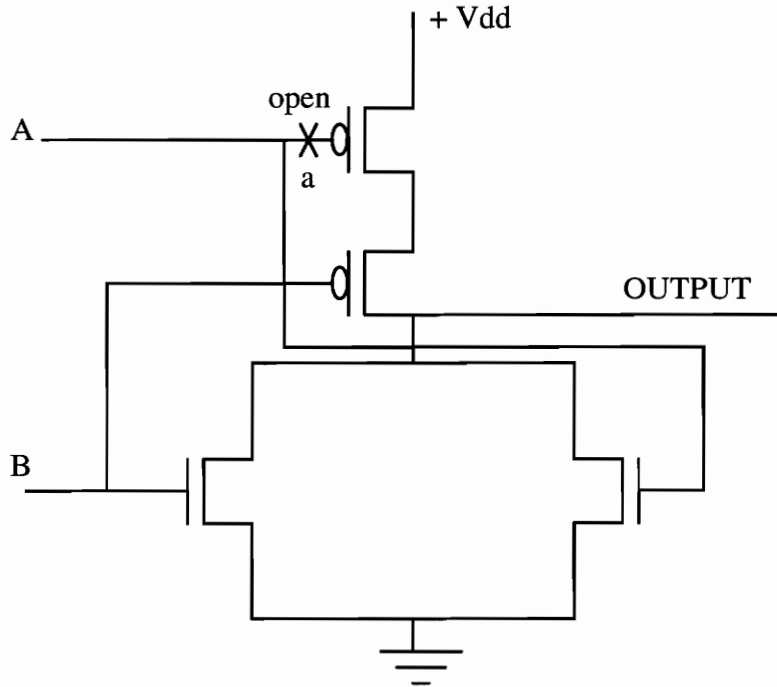


Figure 5: Nor gate

Another possible delay fault exists in an interconnection or net. An interconnection between two signals or a net can be modeled as a transmission line. The model has a resistance and capacitance associated with it (Figure 6). The change in the resistivity or capacitance is likely to cause additional delay for the signal to travel from point IN to OUT.

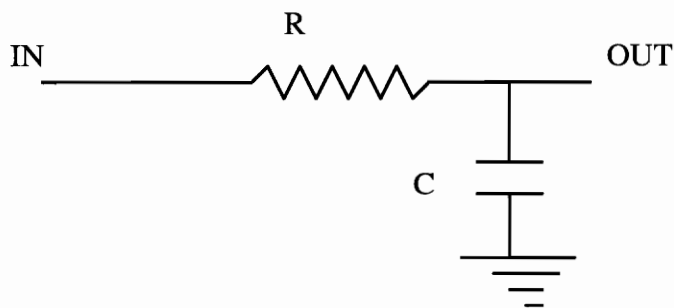
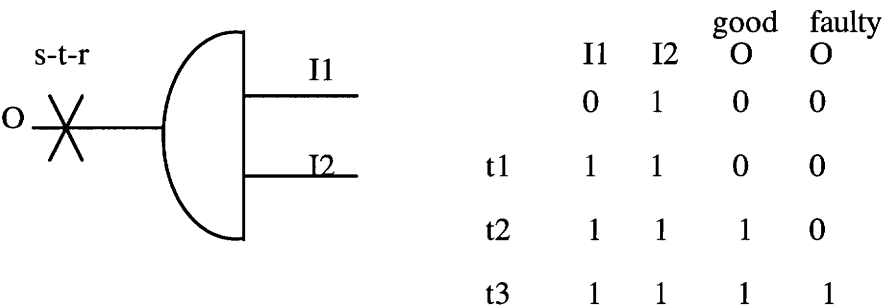


Figure 6 . Representation of a net as a transmission line

2.2 Delay Faults: Gate Level Models

2.2.1 Localized Delay Fault Model

A **localized delay fault** model assumes that a delay fault exists at the input or output of a gate. Thus all signals passing to the circuit output through that gate are delayed by the **size** of that fault. The size of a delay fault is the difference between the actual and the normal delay value.



Normal Delay = t2 - t1

Faulty Delay = t3 - t1

Figure 7: Localized Fault

Consider the AND gate shown in Figure 7. The normal propagation delay through the gate is t2 - t1. Thus in the normal case, when the input changes from "01" to "11", the

output is '1' at time t_2 . If a slow-to-rise fault exists at the output of the gate, then the output changes at time t_3 , where $t_3 > t_2$. Thus the faulty propagation delay is $t_3 - t_1$.

There are two types of faults defined at the gate level: **slow to rise** and **slow to fall**. If the delay fault is in the transition from logic '0' value to logic '1' value, then it is a *slow-to-rise* (*str*) fault. If the delay fault is in the transition from logic '1' value to logic '0' value, then it is a *slow-to-fall* (*stf*) fault.

If the output of a gate is a fanout stem, then the signal along all paths to the output is delayed by the size of that delay fault (Figure 8).

Tests for a localized s-t-r or s-t-f fault can be obtained by combining the tests for a s-a-1 and s-a-0 fault at that point. This simplifies the test generation problem for the localized fault in a combinational circuit. This is explained in the section 2.4.1.

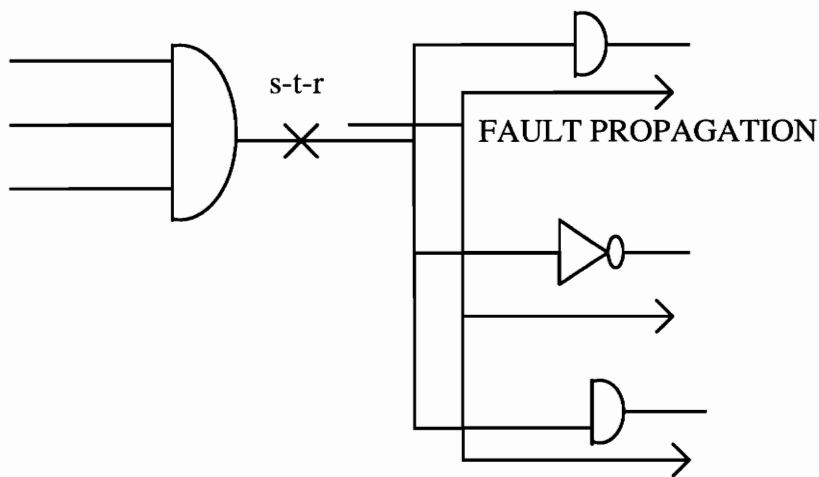


Figure 8 : Localized Delay Fault Propagation

2.2.2 Distributed Delay Fault Model

The **distributed delay fault** model assumes that the total delay of a certain path exceeds the normal delay and causes a delay fault (Figure 9). This could be attributed to any process fault that causes the delay of that path to change. If the path is faulty, then the total delay through that path exceeds the normal delay through the path.

Here unlike the localized fault model, it is assumed that the path is faulty and is responsible for the delay fault. The test generation strategy required for this fault is different than the one used for the localized fault and is difficult and time consuming. This is explained in section 2.4.2.

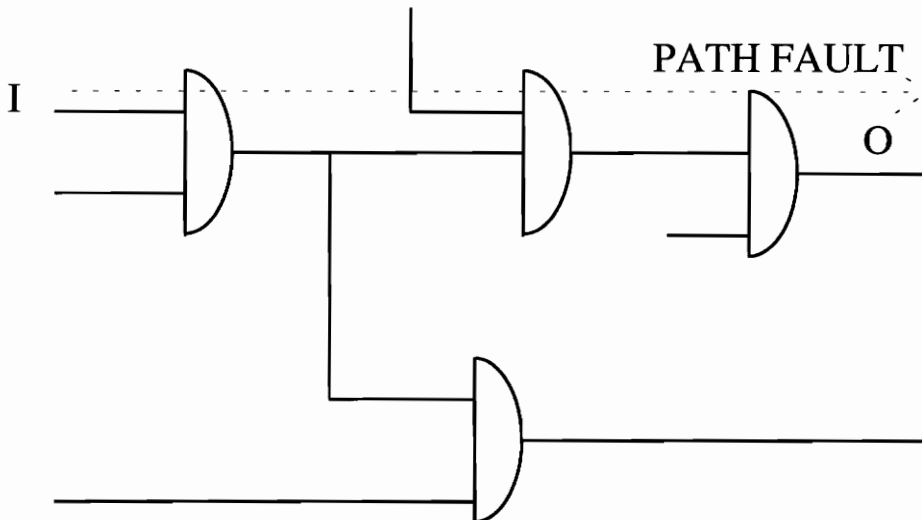


Figure 9. Distributed delay fault

2.3 Effect of Delay Faults

As a signal propagates from circuit input to output, there is a finite propagation delay involved along that path. If there is a fault in that path (either localized in a gate or distributed along the path) then the total propagation delay may exceed the normal delay through that path. This can lead to erroneous results in a digital circuit.

To illustrate the effect of delay faults at the gate level, consider the functional block shown in Figure 10 [19, 20]. This block has three units: input register, combinational block and output register. The maximum propagation delay through the combinational block is T_d . The output register is clocked at an interval greater than T_d to insure that the operation is performed in the combinational block and the proper value is stored in the output registers.

Now because of some physical failure (explained in section 2.1), if the total propagation delay in the combinational block exceeds the time T_d , then a wrong result may be latched in the output register and the result will be faulty. This effect is caused by a delay fault. It is not possible to model this effect using the conventional stuck at fault model, because the signal is not stuck at a certain value, but the delay associated with the signal is now changed.

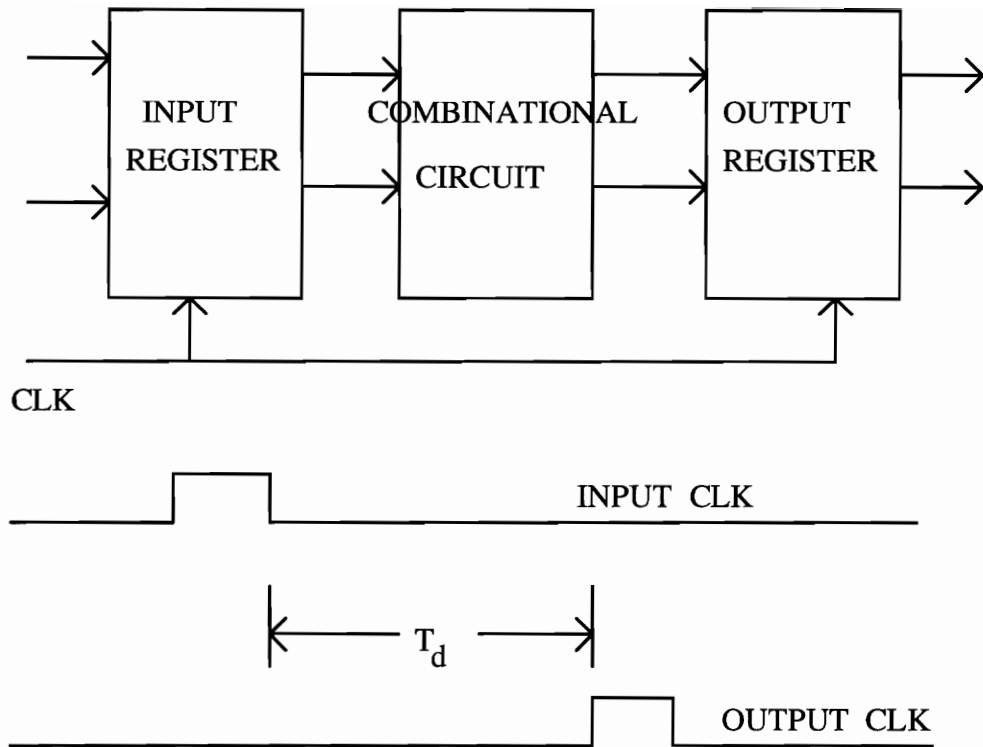


Figure 10: Delay faults

Another effect of delay fault is shown in Figure 11. The timing for the READY signal in an 8085 microprocessor is shown in Figure 11. As soon as the signal READY goes low, a wait state is inserted in the instruction. In the normal operation, the wait state will cause the process to be executed at a certain speed. But if there is a slow to rise fault in the signal, then additional wait states can be inserted in the instruction. This can reduce the system speed significantly, especially when the number of I/O operations performed is large.

A delay fault in the READY signal can be due to a fault in :

- (a) a signal path
- (b) a decoding circuitry (off-chip)
- (c) a decoding circuitry (on-chip)

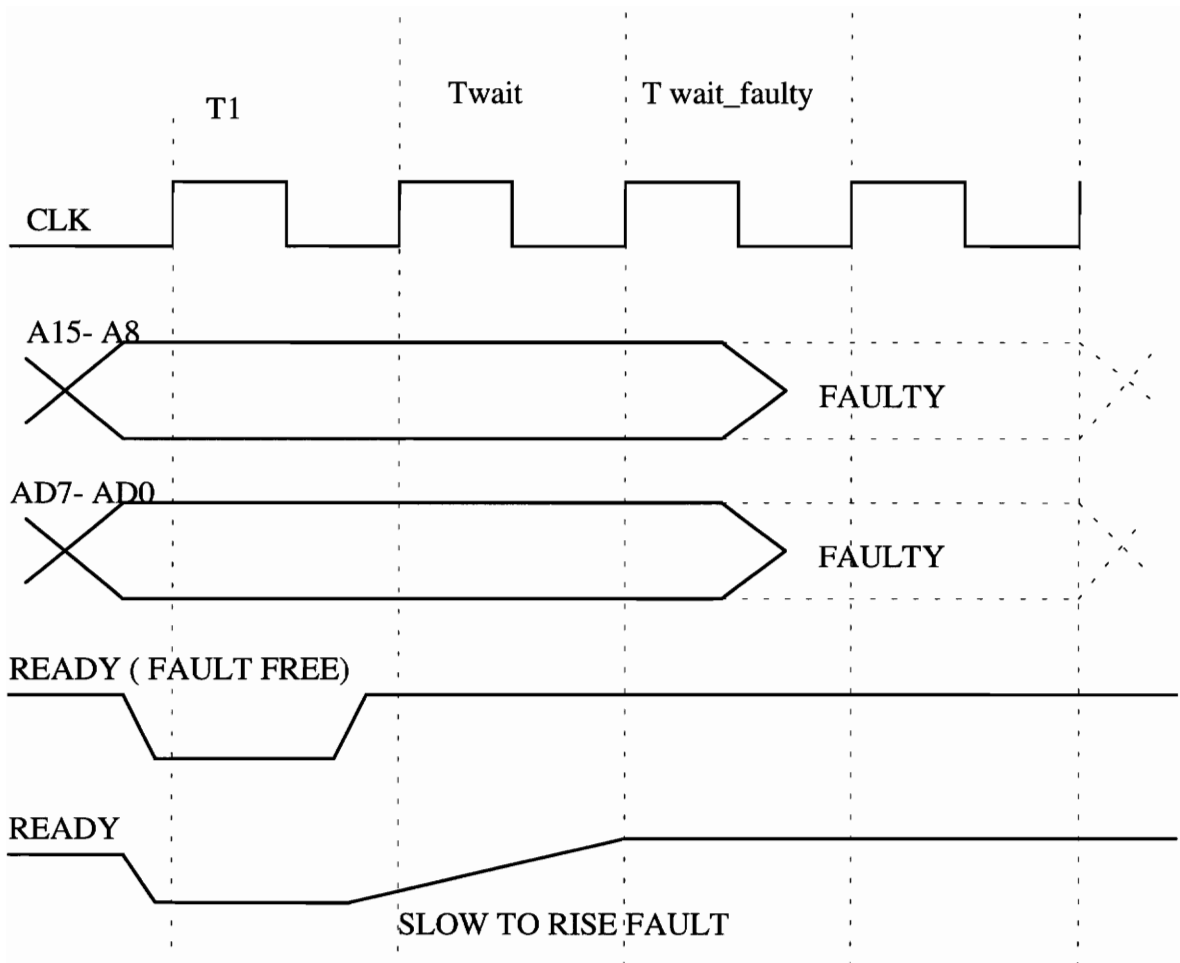


Figure 11: Fault in the ready signal

A bus can be characterized with a connection and disconnection delay. If there is a connection delay fault in a bus, then a bus contention can occur. This situation in an 8085 data bus is shown in Figure 12.

The data and address bus in an 8085 microprocessor are multiplexed. If there is a disconnection delay fault in the address bus, then as shown in Figure 12, there could be a bus contention for a small amount of time during T_{wait_faulty} . Figure 12 illustrates the situation.

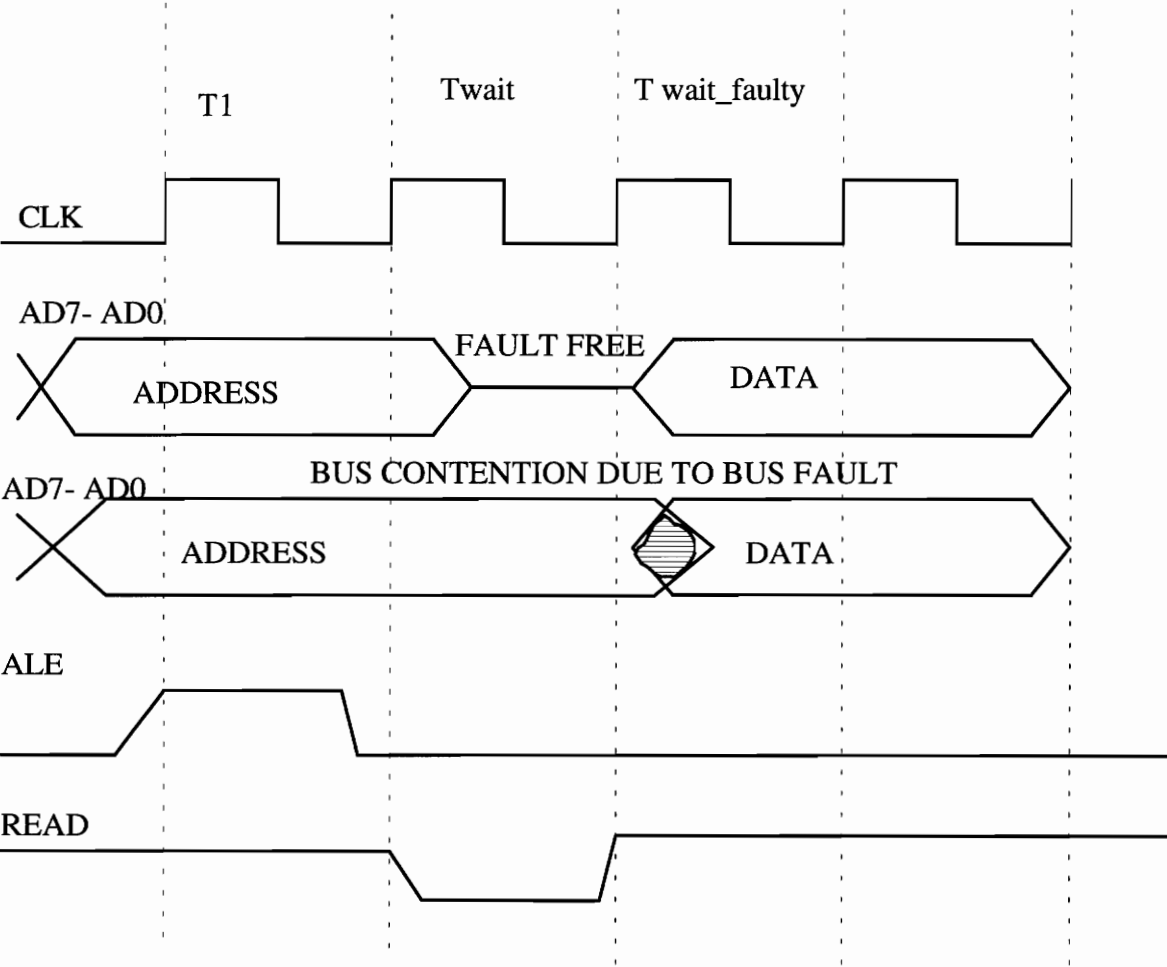


Figure 12: Fault in bus causing bus contention

2.4 Test Generation for Delay Faults

2.4.1 Test generation for a localized fault[10]

In this section a test generation strategy for a localized delay fault is briefly examined. The delay is associated with the transition from an initial value I_i to a final value F_i . To test a delay fault, an input pattern $\langle V1, V2 \rangle$ is required. This pair should satisfy the following criterion:

1. The pair should produce the necessary transition at the pin under test. For example, for a s-t-r fault, the test pair should generate a '0' to '1' transition at the pin under test.
2. The effect of the delay fault should be propagated to at-least one observable output. i.e., it should be possible to distinguish between the faulty and fault free value at that output.

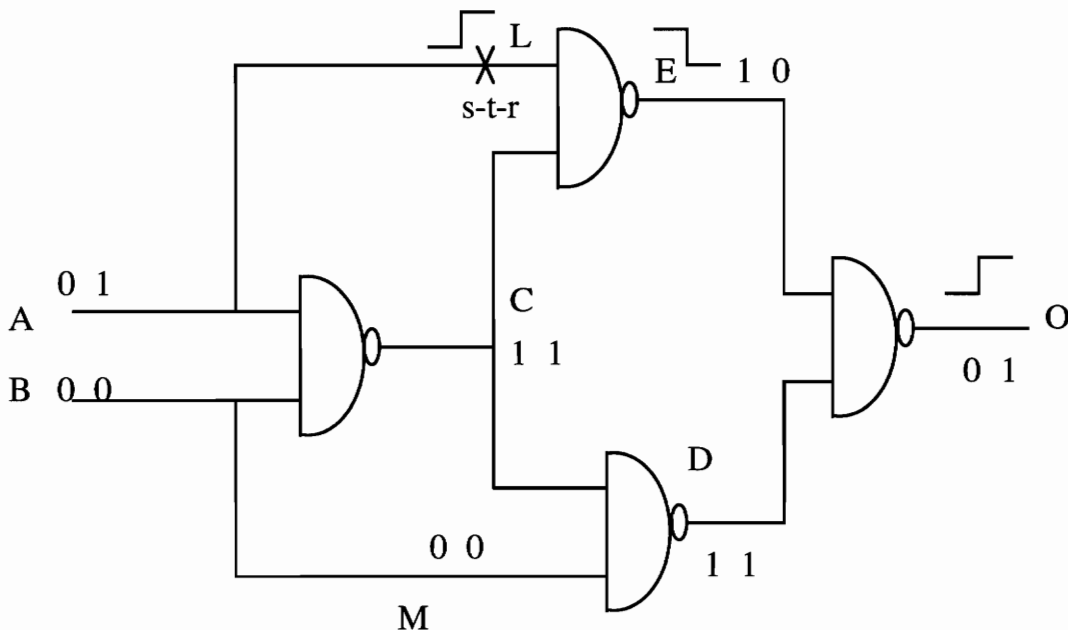


Figure 13: Example of delay test generation: localized fault

Consider the circuit shown in Figure 13. If the test sequence for a s-t-r fault at pin L is to be constructed, then according to criterion 1, it should create the corresponding transition at that pin. Thus input A should change from 0 to 1 to create the transition. Input B should be maintained at '0' so that the effect of the transition propagates to the output. The corresponding response at all nets for this input pair is shown in Figure 13.

In order to test the circuit, the first pattern is applied at $t = 0$. The circuit is allowed to stabilize and at time t_1 a second pattern is applied. The output is sampled (latched) at the time $t_1 + \text{normal propagation delay}$. The faulty and fault free response should be different at this time. In Figure 13, the faulty value is '0' and the fault free value is '1'.

It is possible to generate a test for a localized delay fault using the strategy used for stuck-at faults. This is because, the test for a s-t-r fault can be obtained by combining the test for stuck-at-1 and stuck-at-0 faults at that pin. These two can generate the necessary transition at the faulty pin and propagate the effect to the output.

2.4.2 Test generation for a distributed fault[20]

The strategy to generate a test for a distributed fault differs from that of a localized fault. To illustrate the difference, consider the circuit shown in Figure 14. Consider the path ABCD. If we want to sensitize the path, A can be changed from '0' to '1'. This transition should propagate to output along the path ABCD. Thus for this purpose E, F, G should be maintained at '1' value so that the transition can propagate to the output D.

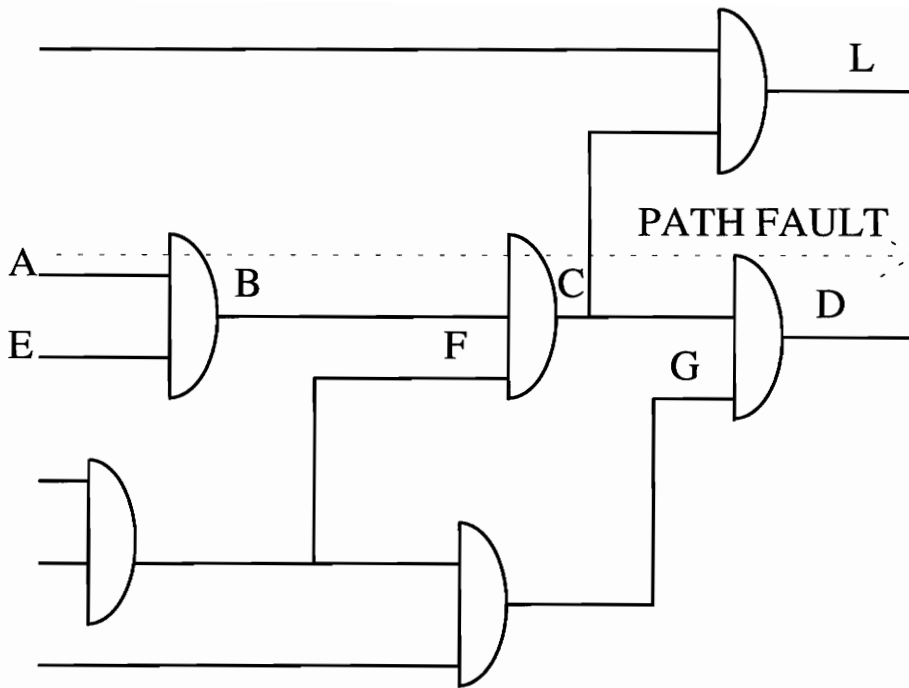


Figure 14: Path fault

Now consider a s-t-r fault at the pin B. This fault can be sensitized by either changing A from '0' to '1' and maintaining E at '1' or by changing E from '0' to '1' and maintaining A at '1'. This is different from the test generation principles used for distributed delay fault test generation. In distributed fault test generation, the primary input on the path must be changed from an initial value to a final value to sensitize the path, whereas in localized fault test generation, any PI can be changed from an initial value to a final value to sensitize the fault, provided that the transition reaches the faulty pin.

The propagation strategies are also different. In Figure 14, the test for s-t-r fault at pin B can propagate the effect of transition to either primary output L or primary output D. Whereas a test for the path ABCD must propagate the effect to the primary output D.

The pair $\langle V1, V2 \rangle$ is a test for a path if

- (a) it sensitizes the entire path and
- (b) propagates the response to the primary output on the path.

Consider the circuit shown in Figure 13 again. Here first step is to find out all input-output paths involved. Thus six paths (ALEO, ACEO, ACDO, BMDO, BCEO and BCDO) can be found. If the test for the s-t-r fault along the path ALEO is to be constructed, then input A initially is set to value '0'. Then the next value '1' is applied at the input A. This sensitizes the path. This transition should propagate along that path to at least one observable output. Thus if pin C is maintained at '1' and pin D is maintained at '1', then the transition can propagate to the output along the path ALEO. Thus the test $\langle 00, 10 \rangle$ will test distributed delay fault along the path ALEO.

This is an example of the robust test for the path ALEO. This test propagates the transition only along the path ALEO and no other path is sensitized. A test is a **robust test** if it propagates the transition to a primary output only along that path, i.e., does not sensitize any other path.

A test is **non robust** if it sensitizes more than one path. For example, consider a test for the path ACDO. The input test pair $\langle 11, 01 \rangle$ creates a transition at the input and propagates to the primary output along the path (Figure 15). But this pair also creates a transition along the path ALEO and ACEO. Thus in addition to the path under test, two other paths (ALEO, ACEO) are sensitized. Since three transitions arrive at the gate 2 at three different times, a hazard could be generated at the output E of the gate (This can

happpen if there is a delay fault along the path A-L-E. In addition to the path under test, the test is sensitizing an additional delay fault along path A-L-E). This hazard could further appear at the primary output of the circuit and create problems in the validation of the test. Figure 15 illustrates this.

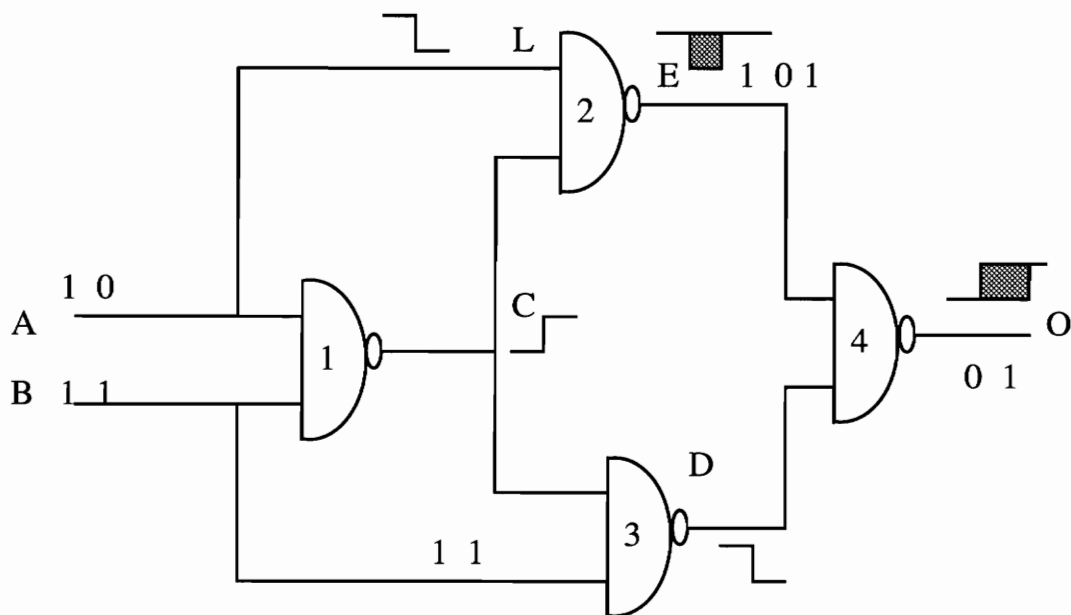


Figure 15 : Non-robust test for a path

It is desired that the test be robust for a path. But for a certain number of paths in a circuit, it may not be possible to generate a robust test. Here it is not possible to generate a robust test for the path ACDO.

Chapter 3

BEHAVIORAL LEVEL DELAY FAULT MODEL

3.1 Rationale for a High Level Fault Model

In a digital circuit, a physical fault may cause excessive delay through the circuit. In [8], it is shown that CMOS NOR gate, with a floating gate shows six times the normal delay for a low to high transition. Change in the resistivity or capacitance associated with a net can also change the delay.

A gate level delay fault model is the representation of these physical faults at the next higher level of abstraction. Two gate level delay fault models are discussed in section 2.2. These models consider the effect of the fault at the gate level and accordingly modify the behavior, i. e., change the delay associated with the faulty gate/path. In a VLSI circuit, because of the large number of transistors involved, a gate level fault model is preferred over the transistor level fault model. It is not practical to construct delay test for a fault in

each transistor and then simulate the transistor level fault model for test verification. Thus lower level faults are mapped to the next level of abstraction and tests are generated.

As the complexity of VLSI devices is increasing, the number of gates per IC is increasing. Also, the number of input-output paths increases exponentially in a gate level model with each fanout stem. Thus if a distributed delay fault is considered, it is very time consuming and inefficient to generate tests for all input-output paths involved. To overcome this, strategies are developed [11] to select a set of paths that is likely to sensitize a delay fault in the circuit under consideration, and then generate tests for these paths. This means that, unlike stuck at faults, it is not necessary to consider all path delay faults in the device for delay test generation [13] and tests developed for certain paths should be able to give 100% delay fault coverage.

The technology trend is slowly moving towards the Multi Chip Module (MCM). In a MCM, a number of small modules are connected on a single substrates to form a functional IC. An MCM could be far more complex than the present single VLSI chip and can contain many times the number of gates and transistors than the present chip. It will not be practical to generate delay tests for a MCM using the gate level delay fault model. Also at this level, in addition to the delay in gates, the delay in the interconnection has to be considered. Finally, as the number of vendors supplying a single functional module for a MCM increases, the gate level equivalent of a single module within a MCM may or may not be available. In such a situation, it is necessary to generate delay tests for a MCM using the functional description available. At this point one can ask a question: is there any way to define a fault model that can take care of low and high level delay faults and yet be simpler than the existing model? Also, one would hope that tests generated

from this model should be able to give satisfactory coverage of delay faults due to physical failures existing in the circuit.

The behavioral level fault model defined here is aimed at solving these problems. A hardware description language such as VHDL is capable of modeling at the gate level as well as behavioral level of abstraction. Timing constructs in VHDL are used to define delay faults. In the presented approach, a data flow description of the device is considered to define the behavior of a model. A data flow graph (DFG) represents the flow of signal from input to output and the timing constructs in the language define the delay associated with these paths. By perturbing the timing construct, a delay fault is sensitized.

In the section 3.2, a theoretical approach to defining delay faults at the behavioral level is presented. In section 3.3 sensitization of these faults is discussed. A possible mapping of these faults to low level faults is presented in chapter 4.

3.2 Behavioral Level Delay Faults

The hardware description language VHDL provides rich language constructs to define behavior of the model. In the language, many statements are defined that control the timing of a digital device. By perturbing these, it is possible to inject a delay fault in the model.

If we consider a behavioral level description and the timing constructs in VHDL, two categories of faults can be defined:

Definition : A *pure behavioral delay fault* refers to a fault that changes the timing associated with the behavioral model. but it is not possible to justify this fault using the gate level fault model. These faults are not considered in this work.(An example of this type of fault is presented in section 4.3)

Definition : A *gate level equivalent behavioral delay fault* refers to a fault that can be mapped to an equivalent gate level fault. From here on a behavioral level delay fault refers to a gate level equivalent behavioral delay fault. This is the main concentration of the present research. All faults defined below represent gate level equivalent behavioral level delay faults.

Gate level equivalent behavioral delay faults can further be classified as follows:

1. Faults in after statements.
2. Faults in generics.
3. Faults in wait statements.
4. Faults in attributes.

3.2.1. Delay faults in after statements

Definition : A *delayed after fault* is a fault that changes the delay associated with the after statement while being simulated. This delay value is more than the normal value. The fault causes the time expression in the 'after' statement to change. This value may be a generic value specified in the entity declaration or a constant.

An after clause can appear in a signal assignment statement as follows[21]:

```
signal_assignment_statement ::=  
target <= [transport] waveform;  
target ::  
    = name | aggregate  
waveform ::=  
    waveform_element {, waveform_element}  
waveform_element ::=  
    value_expression [after time_expression] | null [after  
time_expression]
```

If there is no after clause in the signal assignment statement, then in VHDL implicit 'after 0 ns' is assumed. The effect of a delayed after fault is to change the value of the *time_expression*.

We call the expression on the left side of the signal assignment statement the **target** while the expression on the right side of the assignment statement the **source**. The signal assignment statement represents the assignment of the source value to the target value and the after clause represents the delay involved in that action. Thus a change in the value of the source is propagated to the target with a delay defined by the after clause.

In this context, different types of source and target can be considered and following cases are possible:

1. Source and target both are signals. The delayed after fault can represent a fault in the interconnection between the source and target signal.

2. Source is an expression. Here again two cases are possible:

(a) Source is a logical expression. In VHDL, the information from the lower level is exploited at the higher level and thus logical functions can be used in the source expression. Thus the delayed after fault in this case can represent a fault in the logical element corresponding to that function.

(b) The source expression is an arithmetic expression or a function. (Such as $A + B$ or $ADD(A, B)$). The after statement represents a delay involved in that operation. The delayed after fault here can represent a fault in that particular component.

3. Source and target may represent the connection between a module and a data bus. An after clause in such a statement may represent the delay involved in the connection and disconnection of the module and the data bus. Thus a delayed after fault can represent a connection and/or disconnection delay fault in a bus. An example of disconnection delay fault causing bus contention in a microprocessor is presented in Section 2.3.

3.2.1.1 Transport/Inertial delay fault

As a special case of fault in after statements, a transport/inertial delay fault is defined. This fault is defined for a signal assignment statement in which both source and target are signals.

Definition : A transport/inertial delay fault exists when a transport (inertial) delay in the signal assignment statement is replaced by inertial (transport) delay.

Two categories of this fault can be defined:

- (a) Transport delay replaced by an inertial delay. In this fault, if a fault free statement implies transport delay, then the faulty statement implies inertial delay. This represents a change in the capacitance associated with that interconnection.
- (b) Inertial delay replaced by a transport delay. In this fault, if a fault free statement implies inertial delay, then the faulty statement implies a transport delay and in the faulty case, after clause is removed from the statement. This represents a short circuit between the two signals.

3.2.2. Delay Faults in generics

A delay fault in generics represents perturbation in the value of the generic. Generic values are passed to different entities from the test bench.

Definition: A delay fault in a generic is defined as a fault that perturbs the value of a generic in a behavioral model. A generic may be used to control structural, data flow or behavioral descriptions of a block[21]. In this context, the following generic kinds are perturbed to define a delay fault:

1. Generics of type TIME.
2. Generics that occur in a time_expression in any VHDL statement (need not be of type time).

3. Generics, that are passed to a function that returns result in a variable of type TIME.

A generic fault differs from a fault in the after statement. A generic fault is more global than a fault in the after statement.

In a model, a single component¹ can appear at many places. We define *local generic delay fault* as a fault that affects the delay of a single component and exists within that component only. This may represent a fault developed in that particular component due to a physical failure.

A delay fault in a generic that perturbs the timing of all components of the same type in a model is defined as a *global generic delay fault*. A global generic fault may imply that the design of that particular component is faulty. This could be attributed to a fault in the process. This is discussed in Chapter 4.

3.2.3. Delay faults in wait statements

One of the VHDL language constructs used frequently in a behavioral model to control timings is the wait statement.

Definition : A delay fault in the wait statement is defined as a fault that changes the delay associated with the statement. This value can be more than the normal value. This value can either be a generic or a constant.

¹ A component here refers to a component in the top level description of the model.

A wait statement format can be written as:

```
wait statement ::=  
wait [sensitivity_clause] [condition_clause]  
[timeout_clause]  
sensitivity_clause ::= on sensitivity_list  
sensitivity_list ::= signal_name, {signal_name}  
condition_clause ::= until condition  
condition ::= boolean expression  
timeout_clause ::= for time_expression
```

The fault in a wait statement can be injected by changing the `signal_name` or `boolean_expression` or `time_expression`. The perturbation in the value of `signal_name` in the sensitivity list or `boolean_expression` represents a pure behavioral fault and is not considered here. The faults in these statements perturb the timing of the circuit and do not affect the point to point delay. The clause *for* in a VHDL model represents the delay between two points and thus defines a gate level equivalent delay fault.

The *for* clause defines the delay between the behavioral block before and after the wait statement. If one of the signals in the behavioral block before the wait statement appears in the behavioral block after the wait statement, then the wait statement defines the delay in the interconnection of this signal. Thus a delay fault in the wait statement can represent a delay fault involved in the interconnection. This is explained in Chapter 4.

3.2.4. Delay faults in attributes

A predefined attribute in VHDL denotes a value, function type and range associated with various kinds of entities [12]. A delay fault is defined for the attributes that are related to a signal.

An attributed can be perturbed in two ways: the delay of the parameter T in the attribute can be perturbed (attribute modification) or an additional attribute '*delayed*' can be inserted in the statement (attribute insertion). The later case defines the delayed attribute fault and is defined only for statements having fixed format. This is explained in section 3.2.4.1. The faults defined by modifying an attribute are explained below.

Definition : A delay fault in an attribute is a fault that changes the value of the delay implied by the attribute. This value can be either specified as a generic or a constant. The effect of this fault is to change the normal delay value.

A typical attribute can be represented as:

$$\begin{aligned} & \text{signal_name}'[\text{attribute}][\text{attribute}] \\ & \text{attribute} ::= \text{attribute_name}[\text{time_expression}] \end{aligned}$$

The fault in an attribute can be injected by changing the value of time_expression.

The following predefined attributes are used to define this fault:

1. S'delayed(T):

Parameter: A static expression of type TIME that evaluates to a non negative value. If omitted, the default is 0 ns.

Result: A signal equivalent of signal S delayed T units of time.

Fault Injection: Perturb the value of T.

2. S'stable (T):

Parameter: A static expression of type TIME that evaluates to a non negative value. If omitted, the default is 0 ns.

Result: A signal that has value TRUE when an event has not occurred on the signal for T units of time.

Fault injection: Perturb the value of T.

3. S'last_event:

Result: The amount of time that has elapsed since the last event (transaction) occurred on the signal S.

Fault Injection: In this case, it is not possible to change the value of the attribute. But if the attribute is used in condition statement, then it is possible to change the value on the right hand side of the conditional operator to inject the fault.

For example: if S'last_event > T ns then

----Statements

end if;

In this expression, it is possible to perturb T to inject a fault.

A fault in an attribute can represent a fault in the interconnection. A fault in attribute can represent a set up or hold time fault in sequential primitives. These examples are discussed in chapter 4.

3.2.4.1 Delayed attribute fault

As a special case of faults in attributes, **delayed-attribute** fault is introduced. An example of this fault is shown below.

If the fault free statement is $CLK = '1'$ and $CLK'event$ then in the presence of delayed attribute fault, the faulty statement takes the form $CLK'_{delayed}(T) = '1'$ and $CLK'_{delayed}(T)'event$. This maps to a clock skew in sequential circuits. This fault is defined only for a statement of the form shown above.

3.3 Fault Injection

In this section, a fault injection strategy for each type of fault is discussed.

3.3.1 Fault injection in after clause

An after clause can appear in a signal assignment statement as follows:

```
signal_assignment_statement ::=  
target <= [transport] waveform;  
target ::
```

```

        = name | aggregate

waveform ::=

        waveform_element {, waveform_element}

waveform_element ::=

        value_expression [after time_expression] | null [ after
time_expression]

```

Fault in the after clause can be injected by changing the value of *time_expression*. The effect of delay fault (wrong value at the output at the time of sampling) is caused because of the excessive delay involved. To simulate this effect, the value of *time_expression* should be greater than the normal value during fault injection.

3.3.2 Transport/Inertial delay fault

(a) Transport delay replaced by an inertial delay:

If a statement with transport delay appears in the architectural body, then this fault can be injected by removing transport delay.

For example:

Fault Free: Y <= transport A after 5 ns;

Fault Injection: Y <= A after 5 ns;

(b) Inertial delay replaced by a transport delay:

If a statement with inertial delay appears in the architectural body, then fault can be injected by removing the after clause and any delay involved.

For example:

Fault Free: $Y \leq A$ after 5 ns;

Fault Injection: $Y \leq \text{transport } A$;

3.3.3 Fault injection in generics

As mentioned in section 3.2, only certain kinds of generics can be perturbed to inject a fault. A generic value, greater than normal value can be passed to an entity from test bench to inject fault in a generic.

3.3.4 Fault injection in wait statements

A wait statement can be written as:

wait statement ::=

***wait** [sensitivity_clause] [condition_clause]*

[timeout_clause]

*sensitivity_clause ::= **on** sensitivity_list*

sensitivity_list ::= signal_name, {signal_name}

*condition_clause ::= **until** condition*

condition ::= boolean expression

*timeout_clause ::= **for** time_expression*

As mentioned in section 3.2, only the *for* clause is considered in the present research. A fault in the *for* clause can be injected by changing the value of *time_expression*. Again this value should be greater than normal value.

3.3.5 Fault injection in attributes

A typical attribute can be represented as

$$\begin{aligned} & \text{signal_name}'[\text{attribute}][\text{attribute}] \\ & \text{attribute} ::= \text{attribute_name}[\text{time_expression}] \end{aligned}$$

A fault in an attribute can be injected by changing the value of *time_expression*. This value should be greater than the normal value.

For the attributes 'last_event', the fault injection strategy is slightly different. It is not possible to inject fault in these attributes by changing their value. But if they appear in conditional statement, then it is possible to change the value on the right hand side of the conditional expression to inject the fault.

For example:

```
if S'last_event > T ns then
----Statements
end if;
```

In this expression, it is possible to change the value of *T* to inject a fault.

The **delayed attribute** fault can be injected by inserting an additional attribute 'delayed in the conditional expression as shown below.

Fault Free: CLK='1' and CLK'event

Faulty: CLK'delayed (T) and CLK'delayed(T)'event.

3.4 Fault Sensitization

Definition: Sensitization is an assignment of the value(s) to signal(s) to generate the necessary transition at the fault under test.

Definition: A test is an input pair $\langle V1, V2 \rangle$ that sensitizes the fault under test and propagates the effect of that fault to at least one observable output.

Test inputs are applied at two different times. At $t = 0$, the initial vector $V1$ is applied and the effect is allowed to propagate to the output. At $t = t1$, the next input $V2$ is applied. This creates the necessary transition at the fault under test. The output is sampled at time $t1 + \text{normal propagation delay through the circuit}$. In order to detect the fault, the faulty and fault free response should be different at this time.

For example, a test pair for a certain behavioral block and the output response is shown below:

	INPUT		OUTPUT	
	X1	X2	Fault Free	Faulty
			Y	Yf
t= 0	0	0	0	0
t= t1	0	1	0	0
t= t2	0	1	1	0

The test pattern is <00, 01>. The change in the value of X2 from '0' to '1' sensitizes the fault. The response is observed at t = t2. The faulty value is '0' whereas the fault free value is '1'. Thus this test detects the fault under test.

3.4.1 Sensitization of delayed after faults

An after clause appears in a signal assignment statement. Thus by sensitizing the signal assignment statement, a delayed after fault can be sensitized. The source can be a signal or expression. The following cases can be considered:

(a) Source is a signal:

A delayed after fault can be sensitized by changing the value of the source signal from an initial value V_i to a final value V_f .

For example in the statement

O <= I after 5 ns;

by changing I from '0' to '1' , it is possible to sensitize a delayed after fault in that interconnection.

(b) Source is a logic expression:

A delayed after fault can be sensitized by changing the value of one of the signals in source from an initial value V_i to a final value V_f and selecting other value of the signal in the source expression such that the value of target can change.

For example in the statement

$O \leq I1 \text{ and } I2 \text{ after } 5 \text{ ns};$

a delayed after fault can be sensitized by setting $I1=0'$ (at $t=0'$) and $I1 = 1'$ (at $t= t1$) maintaining $I2 = 1'$.

Similarly a delayed after fault sensitized by setting $I2=0'$ (at $t=0$) , $I2=1'$ at ($t= t1$) and maintaining $I1 = 1'$.

(c) Source is a function:

A delayed after fault can be sensitized by changing the value of one of the function arguments from an initial value V_i to a final value V_f and selecting other arguments such that the function value can change.

For example in the statement

```
O<= ADD(A, B, C) after 5 ns;
```

The function ADD represents a single bit adder with carry. A delayed after fault can be sensitized by changing A from '0' to '1' and maintaining B and C at '0'.

3.4.2 Sensitization of faults in generics

Generics of type TIME or generics that occur in a time_expression in any VHDL statement (need not be of type time) or generics that are passed to a function that returns result in a variable of type TIME are perturbed to define this delay fault.

A generic can appear in different statements (signal assignment, wait or attribute). The strategies developed in this section to sensitize a delay fault in these statements can be used to sensitize all the statements. Sensitizing all these statements (wherever the faulty generic appears) will sensitize the fault in a generic.

3.4.3 Sensitization of faults in wait statements

Only faults in the *for* clause are considered in the present research. As explained in section 3.2, the *for* clause defines the delay between the behavioral block before and after the wait statement. Thus in order to sensitize a delay fault in the wait statement, the signals which appear before and after the wait statement should be changed from an initial value V_i to a final value V_f .

For example consider the following wait statement

```
A<= B or C;  
wait for 100 ns;  
D<= A;
```

The wait statement defines the delay associated with the interconnection of the signal A and D. Thus, if A is changed from '0' to '1', then a delay fault can be sensitized.

3.4.4 Sensitization of faults in attributes

A fault in an attribute can be sensitized by changing the signal value associated with that attribute from an initial value Vi to a final value Vf .

For example in the attribute

$$Y \leq S'\text{delayed}(T)$$

the fault can be sensitized by changing the signal S from '0' to '1'. The sensitization and response of the statement are shown below.

	S	Y	Y (faulty)
t=0	0	unknown	
t=t1	1	0	0
t=t1 + T	1	1	0

At $t=0$, the signal S is set to 0. At $t = t_1 > T$, the signal S is set to '1'. If there is no fault, the output should change after $t_1 + T$ units of time, else the output will maintain the old value '0'.

The same strategy will sensitize delayed-after fault. A list of attributes and the strategy to sensitize them is given below:

$S'_{\text{delayed}}(T)$	Change S from V_i to V_f
$S'_{\text{stable}}(T)$	Change S from V_i to V_f
$S'_{\text{last_event}}$	Change S from V_i to V_f
$S'_{\text{last_value}}$	Change S from V_i to V_f

3.5 Fault Sets

If we represent each of the fault category as a set, then the set of faults in generics will always intersect the set of faults in other categories (Figure 16(a)). This is because a generic can appear in any of these statements. If generic appears only once in the entity in either signal assignment or wait statement or in attribute, then the fault in generic and the fault in that statement are *equivalent faults*. (Figure 16(b))

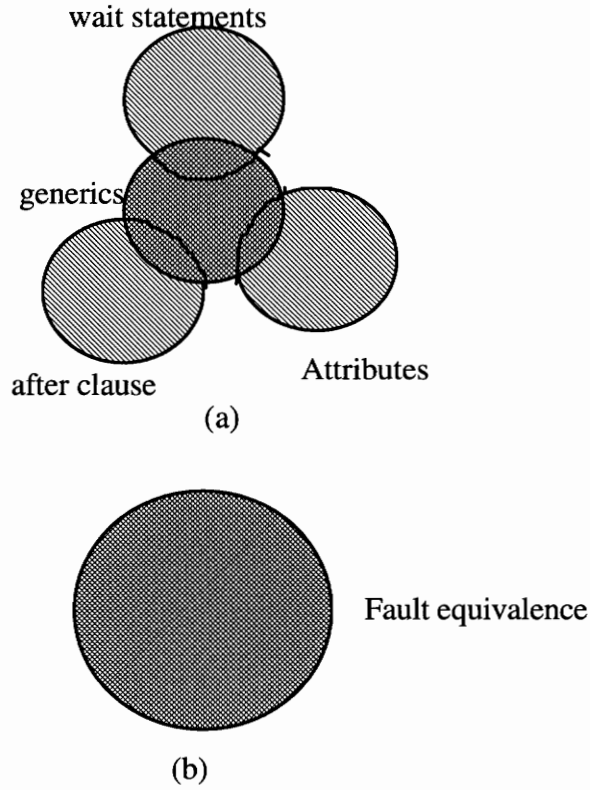


Figure 16: behavioral level delay faults

Definition: Two faults are said to be *equivalent* if their faulty and fault free response is the same for all test vectors that detect the fault.

Theorem 3.5.1(a) A fault in a generic is equivalent to a fault in the after statement if (i) the generic appears only once in the architectural body of an entity and (ii) that appearance of a generic is in the after statement.

Theorem 3.5.1(b) A fault in a generic is equivalent to a fault in the wait statement if (i) the generic appears only once in the architectural body of an entity and (ii) the appearance of the generic is in the wait statement.

Theorem 3.5.1(c) A fault in a generic is equivalent to a fault in an attribute if (i) the generic appears only once in the architectural body of an entity and (ii) that appearance of a generic is in an attribute.

Proof: Fault injection in a generic involves perturbing the value of a generic. A generic can appear in a signal assignment or wait statement or an attribute. If the generic appears only once in an entity in one of the faulty constructs mentioned above, then fault injection for a fault in the generic involves perturbing the value of a generic in that construct. The sensitization of this fault in generic involves sensitizing the construct as explained in section 3.3.2. Since the generic appears only once in the entity, the fault sensitization involves sensitizing only one construct. This is same as the injecting and sensitizing the fault in that construct. A test that sensitizes a fault in generic will also sensitize a fault in the construct. The response of a fault in the generic or a fault in the construct will be same for all the test vectors which sensitize the fault. Thus the two faults are equivalent.

Theorem 3.5.2 A transport/inertial delay fault can always be detected by a test developed for a delayed after fault for the same statement

.

Proof: A transport/inertial delay fault is defined for a signal assignment statement when both source and targets are signals. A signal assignment statement with after clause implies a certain delay value, and without after clause implies a delay value of 0 ns. A delayed after fault perturbs this value. Sensitizing the delayed after fault involves changing the source from an initial value to a final value.

Sensitizing a transport/inertial delay fault involves changing the value of source from an initial value to a final value. Thus a test that sensitizes the delayed after fault will also

sensitize the transport/inertial delay fault. The same propagation strategy can propagate the effect of both the faults to a primary output. Thus the transport/inertial delay fault can always be detected by a test for delayed after fault. Note that these two faults are not equivalent, because the output response is different for two cases.

Chapter 4

MAPPING OF THE BEHAVIORAL LEVEL FAULT MODEL

In this chapter, with the help of examples, it is shown:

1. How a behavioral level path can be mapped to a gate level path.
2. How physical defects can be represented by a behavioral level fault model.
3. How other timing faults such as clock skew are reflected in the behavioral level fault model.

4.1 Mapping of a Behavioral Level Path to a Gate Level Path

The distributed delay fault model (gate level) assumes that the delay of an input-output path is distributed along the path and that there is a fault at one or more points in that path. Thus in this model, all input to output paths are identified from a gate level description, and the total delay along such a path is defined as the sum of the delay at each gate in that path.

At the behavioral level, it is possible to define a similar path fault model. A behavioral model can be represented in terms of a Data Flow Graph (DFG). Using DFG, all the input output paths implied by a behavioral model can be found out. What needs to be shown is:

- (a) each input-output path represented by a DFG maps to a gate level equivalent path and
- (b) any fault in the behavioral component along such a path maps to an equivalent gate level fault.

To explain this, a simple example of the DFG representation of a behavioral model is considered. It is shown that each path implied by a DFG represents an equivalent path at the gate level.

Definition: A data flow graph[29, 39, 40, 41] is a directed graph $G(v,t)$, where v is the set of nodes and t is the set of directed edges. In a behavioral level VHDL description, a node v_i is defined by statements in a model. A set of directed edges t_{ij} , $t_{ij} \in t$, establishes connection between nodes v_i and v_j , where $v_i, v_j \in v$. The direction of each edge t_{ij} in a DFG represents graphically the data dependency between the node v_i and v_j , i.e., a directed edge between v_i and v_j defines that v_j depends directly on v_i . Thus a DFG defines the order in which the operations are performed as determined by the data dependencies in the model.

The following are salient features of a DFG:

1. In a DFG, all primary inputs to a model are defined as *starting nodes* and all primary outputs are defined as *terminating nodes*. The intermediate nodes are defined by statements in the model. The direction of edges represents the sequence of the operation

followed within the model. Thus a DFG represents a flow of information from a primary input to a primary output in a VHDL model.

2. A DFG can be either cyclic or acyclic. A DFG of a VHDL behavioral model is acyclic if the model has (a) no ports of type inout and (b) the behavioral description is written in data flow style. A DFG is cyclic when ports of type inout or loop statements appear in the behavioral description. In this research, only acyclic DFG are considered.

Definition: A behavioral level input-output (io) path is defined as a path defined by a DFG (constructed from a behavioral model) between a primary input I and primary output O.

A simple example that illustrates the concept of a DFG is shown below:

Example 1:

The following operation is performed:

$O1 = I1 * (I1 + I2).$

$O2 = I1 * I2$

To illustrate the concept of DFG, an additional signal I is inserted.

entity EX is

port (I1, I2: in INTEGER; O1, O2: out INTEGER);

end EX;

architecture EX_ARCH of EX is

signal I: INTEGER;

begin

I <= I1 + I2;

*O1 <= I * I1;*

*O2 <= I1 * I2;*

end EX_ARCH;

The corresponding DFG is shown in Figure 17.

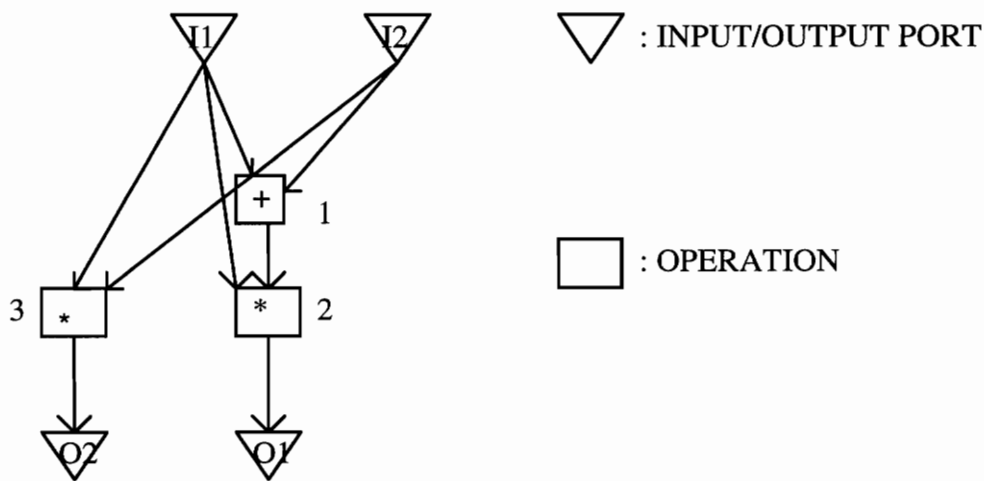


Figure 17: Data Flow Graph

Here, inputs I1 and I2 are *starting nodes* and O1 and O2 are *terminating nodes*. The rectangles define operations performed within the behavioral block.

Using this graph it is possible to construct all the input-output paths defined by the behavioral model. For example, there are two paths from I1 to O1 and one path from I1 to O2.

Different representations may be required to construct a DFG of a more complex behavioral model. The example shown here merely illustrates the input-output paths defined by a behavioral model. A systematic way of generating DFG of a data flow style behavioral description is presented in chapter 5.

Theorem 4.1: Each behavioral level io path represents an equivalent path in a gate level model.

Proof: For a behavioral model B, let G be the gate level equivalent. Let P be a path defined by the DFG of the behavioral model B (Figure 18) from input X to output Y.

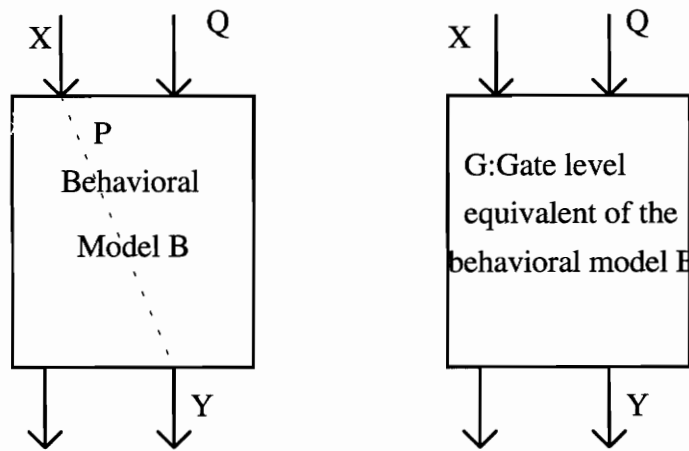


Figure 18: A behavioral and gate level model

For a particular input value I_i at the input X, let O_i be the output value at output Y. If the input value changes to some other value I_j at the pin X, then a path $P \in \text{DFG}$ of the behavioral model is sensitized. The input value at the other input Q is maintained such that the transition propagates to the output Y. This transition changes the output value at

pin Y from an initial value O_i to some other value O_j . Thus the change at the input X propagates to the output Y through the path P in the behavioral model.

Now assume that the DFG represents a path non existent in a gate level model. This means that the change at pin X from initial value I_i to final value I_j does not sensitize any path in the gate level model. The gate level model represents an equivalent behavior defined by the model B, and for the input value I_i at pin X the output value at pin Y must be O_i . If the input value at pin X changes from I_i to I_j , then since there is no path sensitized at the gate level, the output Y in the gate level circuit will retain its old value O_i . But this is not possible since the gate level circuit G represents a behavior defined by the model B. Thus our assumption is wrong and there has to be at least one path in the model that propagates the effect of input value change to the output.

The total delay along a behavioral level io path is the sum of the delay along each node in the DFG. This delay is represented by different behavioral level timing constructs. For example, in Example 1, an after clause can represent the delay at each operation as follows:

```
architecture EX_ARCH of EX is  
signal I: INTEGER;  
begin  
s1: I <= I1 + I2 after 5 ns;  
s2: O1 <= I * I1 after 5 ns;  
s3: O2 <= I1 * I2 after 5 ns;  
end EX_ARCH;
```

The statements s_1 , s_2 and s_3 define the delay along the node 1, 2 and 3 respectively. These together define the delay along the path. For example, delay along the path I1-1-2-O1 is 10 ns and I1-2-O1 is 5 ns. Any fault in the after clause represents a fault in the component that performs addition or multiplication operation. Thus a delayed after fault in these statements can be mapped to a fault in a gate level equivalent circuit that performs this function.

Cor 4.1.1: If a behavioral model B represents a behavioral equivalent of a gate level fanout free circuit, then the behavioral i/o paths in the DFG can represent all the input-output paths in a gate level equivalent circuit.

Proof: A fanout free circuit represents a unique path from the input to output. This means that at the gate level, there is only one path from an input to output. At high level the DFG has to define at least one path between the input $I[i]$ and output $O[j]$ (so that the effect of the change at the input can propagate to the output). This path will define the equivalent path from $I[i]$ to $O[j]$ at the gate level. (There could be more than one path at the behavioral level representing the same gate level path. While generating a delay test from a behavioral model, this fact should be considered)

Cor 4.1.2: If a behavioral model A represents an internal fanout free gate level circuit (i.e. a circuit in which only primary inputs may be fanout stems), then using the strategy developed in [43], it is possible to identify fanout points in a behavioral level model and thus the equivalent representation can cover all the paths in the gate level equivalent.

Proof: In any internal fanout free circuit, if there are n branches at the input, then there are n paths from that input to output. In previous research[43], it is shown that it is possible to identify fanout points in a behavioral description. A port or signal is a fanout stem when it satisfies one of the following conditions:

1. A port or signal appears in a source expression more than once or appears in more than one source expression.
2. A port representing a primary output appears in at least source expression (port of type inout, represents feedback path in the network). This case is not considered in the present research.

Thus if the fanout stems are identified at the input then it is possible to identify all the paths in a model.

Comment: Consider the following example [43]:

```
process (COND, A, B, C, D )  
begin  
s1: if COND='1' then  
s2:   X<= A and B;  
else  
s3:   X<=A or C;  
end if;  
end process;
```

In this example, the signal `A` appears more than once in the source expression (in statements `s2` and `s3`) and thus is a fanout stem.

A systematic strategy to find input-output paths implied by the model using a DFG is explained in Chapter 5 .

4.2 Mapping of the Behavioral Level Delay Fault Model

In this section, with the help of examples[23-28], it is shown that each behavioral level fault defined can be mapped to an equivalent gate level fault and/or a physical failure.

4.2.1 Mapping of delayed after faults

4.2.1.1 Mapping to a fault in an interconnection

A delayed after fault may represent a fault in an interconnection. In a signal assignment statement, if the source is a signal, then the statement implies connection between the source and the target. The delay involved in this connection is defined by an after clause.

For example, in the statement

`Y <= X after 5 ns;`

any fault in the after clause can be mapped to a delay fault in the interconnection between `X` and `Y`. This could be attributed to a change in the resistance or capacitance associated with the net.

4.2.1.2 Mapping to a fault in a gate

A delayed after fault may represent a fault in a gate. In a signal assignment statement, if the source is a logical expression, then source implies a gate with the functionality implied by the logical expression. The delay involved in this gate is defined by an after clause. Thus any fault in the after statement maps to a fault in a gate performing that logical operation.

For example, in the statement:

$$A \leq B \text{ and } C \text{ after } DEL \text{ ns};$$

if the change in the source signal B is from '0' to '1' and the delayed after fault is injected, then it maps to a slow-to-rise fault and if the change is from '1' to '0' then it maps to a slow-to-fall fault.

4.2.1.3 Mapping to a fault in a functional block

A delayed after fault may represent a fault in a functional block. In a signal assignment statement, if the source is a function, then the source implies a component performing that operation. The delay involved in this component is defined by an after clause. Thus any fault in the after statement can be mapped to a fault in this functional block.

For example in the statement

$$SUM \leq ADD(A, B) \text{ after } 5 \text{ ns};$$

the function ADD represents an adder. Thus the fault in the after statement is mapped to a fault in the component.

A logical expression can be passed as a parameter to a function. For example consider the statement:

SUM<= ADD(A and C, B) after 5 ns;

The delayed after fault here implies a fault in either the component or the gate. If the signal A or C changes and a delayed after fault exists, then it represents a fault in the AND gate or in the ADD component. If the signal B changes, and the delayed after fault exists, then it represents a fault in the component ADD.

4.2.1.4 Mapping to a fault in a bus

If multivalued logic is used in the simulation, then it is possible to simulate the effect of the delay fault involved in transition from an initial value V_i to final value V_j using delayed after fault. In this context, a special case which involves transition from an active state to a high impedance state ('0' to 'Z' or '1' to 'Z') is important. Consider a data bus driven by two components (Figure 19):

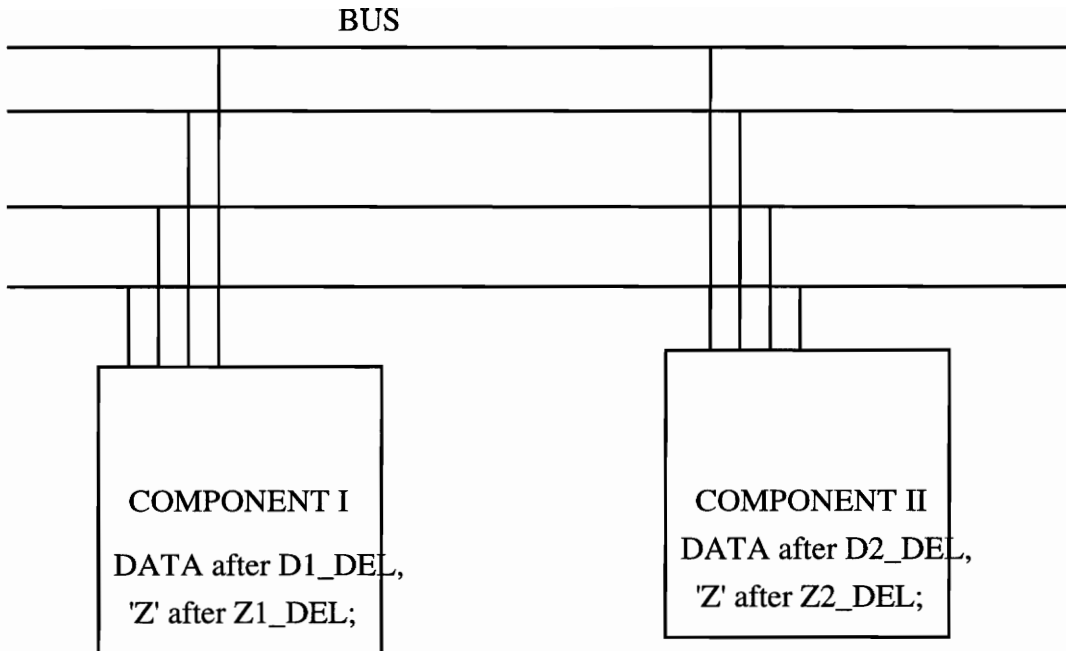


Figure 19: Bus delay fault

Only one component at a time is expected to drive the bus. Thus when component I is active, component II is in high impedance state and when component II is active, component I is in high impedance state. (The high impedance state is represented by 'Z'). There is a finite connection and disconnection delay involved. The connection delay for component I is represented by D1_DEL and disconnection delay is by Z1_DEL.

If there is a fault in the disconnection delay of component I (Represented by 'Z' after Z1_DEL statement), then component II will get connected before component I is disconnected. Thus there will be a bus contention for a small duration of time. The delayed after fault here represents the disconnection fault in the data bus. This can be attributed to change in resistance or capacitance associated with that bus. (In VHDL, statements *null* or *disconnect* can also be used to represent disconnection delay).

Similarly there could be a delay fault in the connection. If there is a fault in the connection delay of component I (Represented by *DATA after DI_DEL* statement), then component II will get disconnected and the bus will retain 'Z' (high impedance value) for a longer period of time (till component I puts data on the bus). If the timing is critical in the device, then this may lead to an erroneous result. The delayed after fault in this case represents a connection fault in the bus.

To illustrate the above situation, consider a bus X driven by two blocks A and B in Example 2. Only one block at a time drives the output. The normal connection and disconnection delay for block A and block B is 5 ns. When SET = '1' and RESET = '0', the output X is '1' and block B is disconnected, and when RESET = '1' and SET = '0', the output X is '0' and block A is disconnected.

A fault can be injected in statement s2 (representing disconnection delay of block A), by changing the delay to 10 ns. This can cause bus contention for a small duration of time as shown in Figure 20. This is because block B puts data on the bus before block A is disconnected. Thus delayed after fault can represent a disconnection fault in the bus.

Similarly a fault can be injected in the statement s3 (representing connection delay of block B) by changing the delay to 10 ns. This will maintain the data bus in high impedance state 'Z' for a some time as shown in Figure 20. This is because, component A gets disconnected and because of the fault, component B takes longer time to put data on the bus. Thus delayed after fault can be mapped to a connection fault in the bus.

Example2 :

```
use work.all;
use work.USER_TYPES.all;

entity NULA is
port (SET, RESET: in DotX; X: out DotX);
end NULA;

architecture NUL_ARCH of NULA is
signal X1: DotX register;

begin
A: block(SET='1')

    begin
    process(GUARD)
        begin
            if GUARD then
s1:                X1<= '1' after 5 ns;
                else

s2:                X1<= 'Z' after 5 ns;
                end if;
            end process;
        end block;
```

end block A;

B: block(RESET='1')

begin

process(GUARD)

begin

if GUARD then

s3: X1 <= '0' after 5 ns;

else

s4: X1 <= 'Z' after 5 ns;

end if;

end process;

end block B;

X <= X1;

end NUL_ARCH;

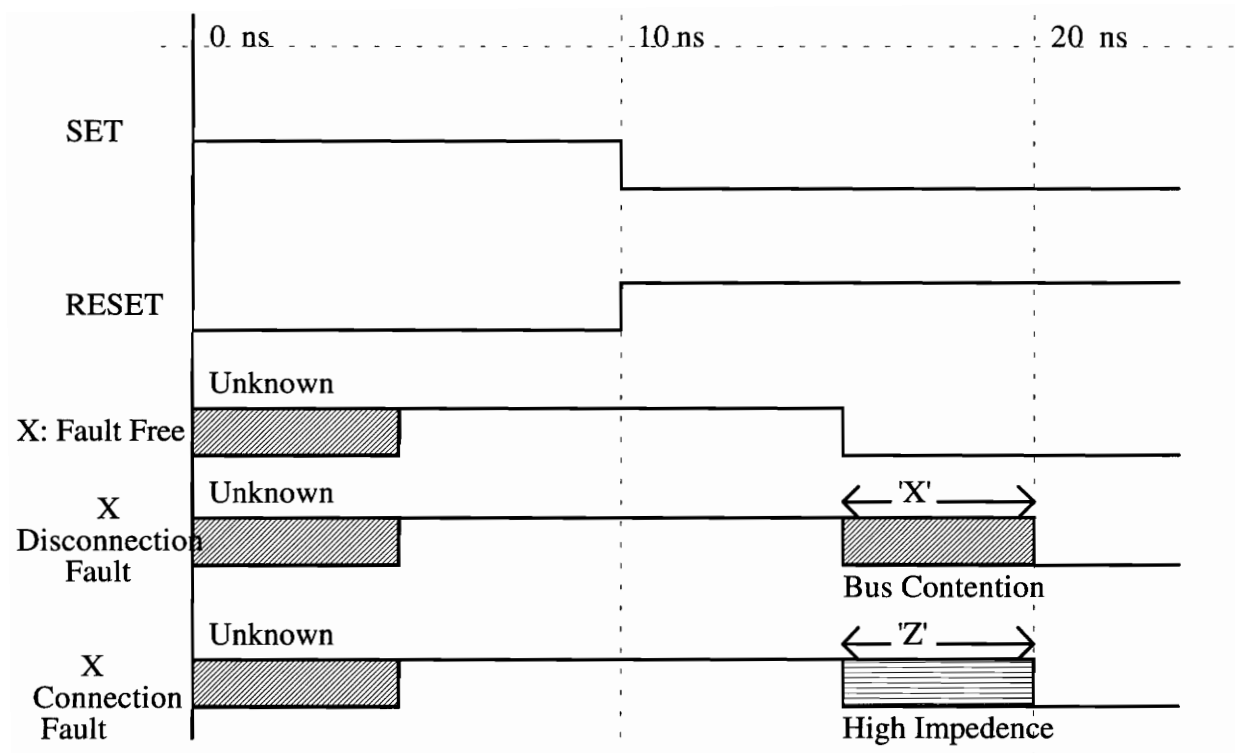


Figure 20: Connection and Disconnection fault

4.2.2 A fault model for transport/inertial delay fault

4.2.2.1 Transport delay replaced by an inertial delay

In this fault the 'transport after' clause is replaced by an 'after' clause in a behavioral model. This can represent a fault model shown in Figure 21.

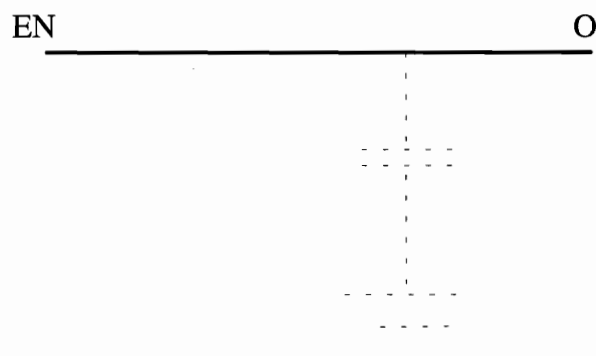


Figure 21. Fault in after/transport after statement

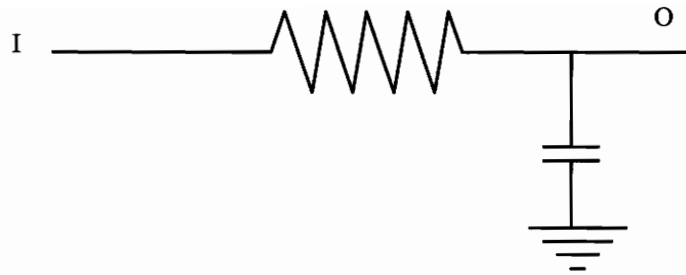
Consider a transmission line. A finite delay is involved when signal travels from EN to O. This can be represented by a transport delay. Now because of some process fault, if a capacitance is developed in the transmission line, then it will take a finite time for the capacitor to charge and change state. The input signal has to be steady for a certain amount of time to charge or discharge the capacitance. The same effect can be observed if the resistance increases beyond the normal value. This is represented by an inertial delay.

Fault free: $O \leq \text{transport EN after 3 ns};$

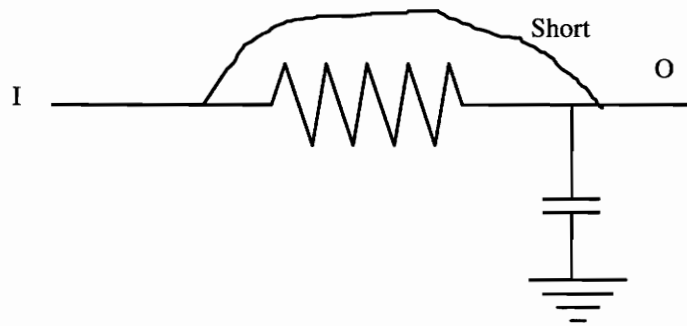
Faulty: $O \leq \text{EN after 3 ns};$

4.2.2.1 Inertial delay replaced by a transport delay

In this fault, an inertial delay is replaced by a transport delay. This may represent the following situation. Consider a RC network shown in Figure 22. In the absence of any fault, the output changes its value only after the capacitor C at the output is charged or discharged. This fault represents a short circuit between the input and output and thus there is no delay involved in the transmission of a signal from point I to O.



FAULT FREE $O \leq I$ after 5 ns;



FAULTY $O \leq \text{transport } I$;

Figure 22: Fault model for transport / inertial fault

4.2.3. Mapping of faults in Generics

4.2.3.1 Mapping of a local generic fault

Using generics, it is possible to pass information regarding physical parameters to a behavioral description and thus study the effect of these physical parameters on the behavior.

Consider a transistor level model of an inverter shown in Figure 23[44]:

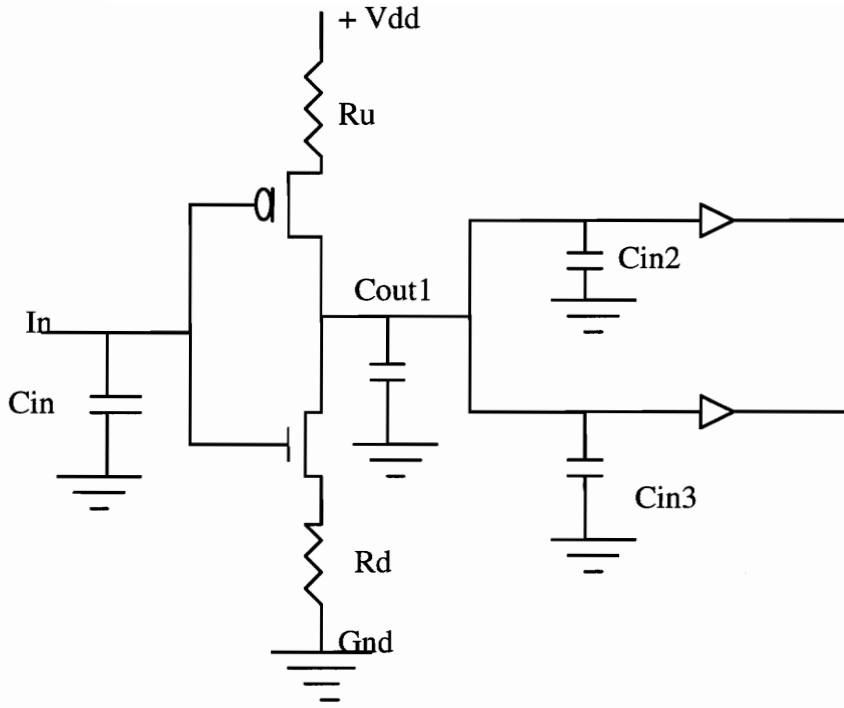


Figure 23. Transistor level schematic of inverter

The propagation delays are defined as:

$$tp_{0-1} = k * Ru * (Cout1 + Cin2 + Cin3)$$

$$tp_{1-0} = k * Rd * (Cout1 + Cin2 + Cin3)$$

Where Ru , Rd , $Cout1$, $Cin2$, $Cin3$ are the resistance and capacitance values involved with the inverter as shown in Figure 23 and k is technology constant. The values of Ru , Rd , $Cout1$, $Cin2$, $Cin3$ can be passed as generics to the entity. These can appear in a signal assignment statement as follows :

Example 3:

architecture RC_ARCH of RC is

```

signal O1: BIT;
begin
process(I)
variable ONEW, OOLD: BIT;
begin
ONEW:= not I;
      if ONEW = '1' and OOLD='0' then
        O<= ONEW after (RU*(COUT1+CIN2+ CIN3))*(1 ns);
      elsif ONEW = '0' and OOLD='1' then
        O<= ONEW after (RD*(COUT1+CIN2+ CIN3))*(1 ns) ;
      end if;
OOLD:= ONEW;
      end process;
end RC_ARCH;

```

If one of the generic is perturbed, then it is possible to reflect the effect of that perturbation on timings in the behavioral model. Thus a fault in generic can represent a fault in that physical parameter.

4.2.3.2 Mapping of a local/global generic fault

A component may appear in more than once place in a top level description of a model. Consider the following example in Figure 24. In this model, the data is latched temporarily in the buffer and is then transferred to the main memory. If there is a local generic fault, then it implies a delay fault in any one of the AND gates. This could be

attributed to a local fault developed in one of the transistors in that gate. A global generic fault implies a delay fault in both AND gates. This could be attributed a fault in mask preparation of all the AND gates or there could be an error in the transistor level layout of the gate.

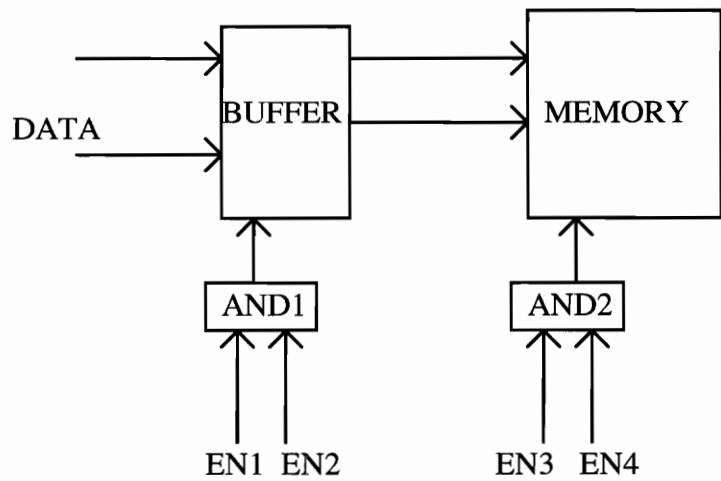


Figure 24. Global generic fault

4.2.4. Mapping of faults in wait statements

A fault in a wait statement may represent a fault in an interconnection. A wait statement with a for clause can be viewed as shown in Figure 25. The for clause represents a delay between the behavioral block before the statement and after the statement.

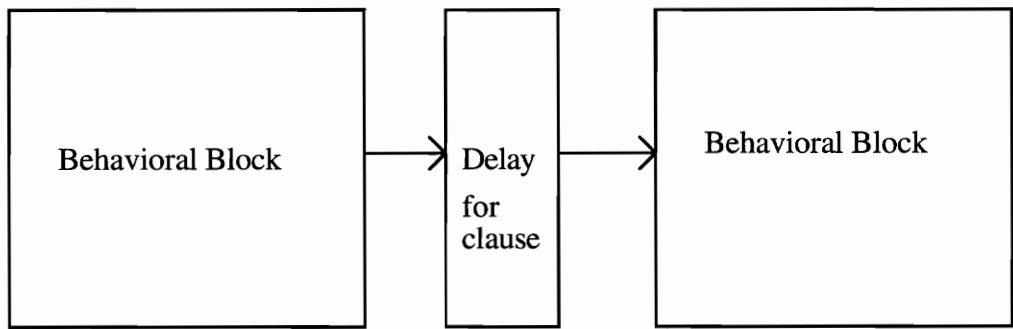


Figure 25. Wait statement

Consider the following code:

Example 4:

```
A<= B or C;  
wait for 100 ns;  
D<= A;
```

Here the wait clause represents a delay in the interconnection of signals A and D. Thus any fault in the wait statement can be mapped to a fault in the interconnection

4.2.5. Mapping of faults in attributes

4.2.5.1 Mapping to a fault in an interconnection

The attribute 'delayed can represent a delay associated with an interconnection. Thus a fault in an attribute can represent a fault in an interconnection.

For example: if it is desired to model delay through interconnection for an AND gate(Figure 26), an attribute 'delayed can be used as:

```
C<= A'delayed( A_D ns) and B'delayed(B_D ns) after O_D ns;
```

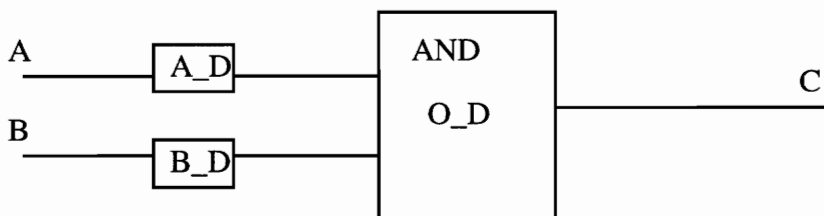


Figure 26: And gate with interconnection delay

The attribute 'delayed represents the delay through the interconnection and the after clause represents the delay through the AND gate. Thus any delay fault in the attribute maps to a fault in the interconnection.

4.2.5.2 Mapping to a clock skew

A behavioral model for a D flip-flop is described in example 5. At the rising edge of the pulse, new value is loaded in the D flip-flop. If the expression, (CLK= '1' and NOT CLK'stable), is changed to (CLK'delayed(5 ns) = '1' and NOT CLK'delayed(5 ns)'stable) then it represents a delayed attribute fault. This delay can represent a delay involved in the interconnection between the CLK pin in the model and the wire that carries the global clock. This represents a local clock skew. Using attributes, the exact value of clock skew for each sequential element can be substituted in the time_expression and the fault simulation can be performed.

Example 5:

architecture DAT_ARCH of DAT is

signal TMP: BIT;

begin

TMP<= D when (CLK='1' and NOT CLK'stable) else TMP;

O<= TMP after 5 ns;

end DAT_ARCH;

4.2.5.3 Mapping to a fault in sequential circuits

Attributes are often used to check setup/hold time in sequential circuits. Faults in these attributes can represent a set up or hold time fault in a sequential circuit. In example 6, the data is loaded in the flip-flop if only there is no set up or hold time fault.

Example 6:

```
process(CLK'delayed(HT), D)

begin

if CLK'delayed(HT)'event and CLK'delayed(HT) = '1' then
    if D'stable(ST + HT) then
        O<= D;
    end if;
end if;

end process B;
```

The data at the input is required to be stable for the time defined by the generics ST and HT which define set up and hold time respectively. In the expression $D'stable(ST + HT)$ if a fault is injected by changing the value of the ST and HT, then the data is now required to

be stable for more time than in fault free case. This represents a setup-time or hold time fault in the sequential element.

A sequential element requires a certain minimum pulse width at its reset or set pins. This is because the effect of the change at that pin takes a finite time to propagate to the output and thus the input should not change during that period of time. If a gate in this path is faulty, then a longer pulse width may be required at the reset or set pin to reset or set the element. In example 7, the attribute `RESET'delayed'stable(MPW)` checks for the minimum pulse width requirement and the output is reset only if the minimum pulse width requirement is satisfied. Any fault in this attribute represents a fault in the path between the RESET pin and the output.

Example 7:

```
process(CLK'delayed(HT), D, RESET'delayed(MPW))

begin

if CLK'delayed(HT)'event and CLK'delayed(HT)='1' then
    if D'stable(ST + HT) then
        O<= D;
    end if;
end if;

if RESET='1' and RESET'delayed'stable(MPW) then
```

```
O<= '0'
```

```
end if;
```

```
end process B;
```

4.2.5.4 Mapping to a fault in a component

A fault in attribute may represent a fault in a component. Consider a pulse generator shown in Figure 27. The delay element has a delay of 5 ns and thus the output pulse duration is 5 ns. The behavioral model for this circuit using an attribute 'delayed' can be written as shown in Example 8.

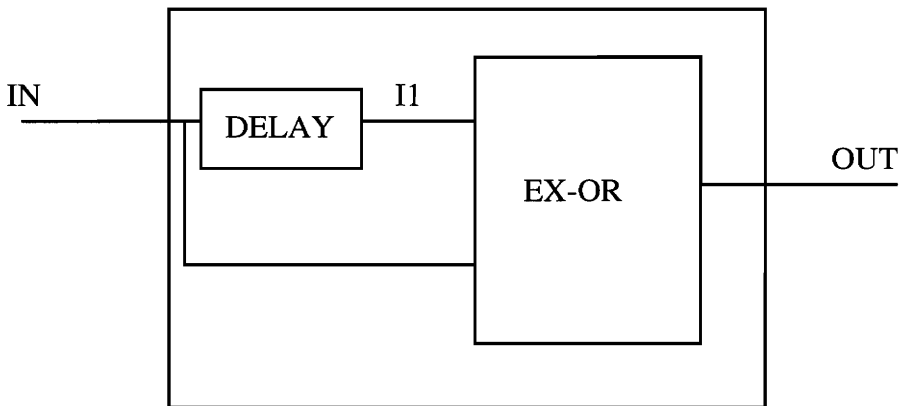


Figure 27: Pulse generator

Example 8:

architecture PULSE_ARCH of PULSE is

signal I1: BIT;

begin

```
I1<= IN'delayed (5 ns);  
OUT <= IN XOR I1;  
end PULSE_ARCH;
```

The attribute 'delayed represents the delay through the delay element. Thus any fault in the attribute can be mapped to a fault in the delay element.

4.3 Pure Behavioral Faults: Another Consideration

As mentioned in Chapter 3, there is a separate class of behavioral level delay faults called pure behavioral faults. These faults modify the timing behavior of the model and do not change the point to point delay.

Consider a monostable shown in Figure 28. Whenever $I1 = I2$, the exor gate output is '0', which enables the monostable and a pulse is generated at the output. The VHDL model for this behavior can be written as shown in Example 9.

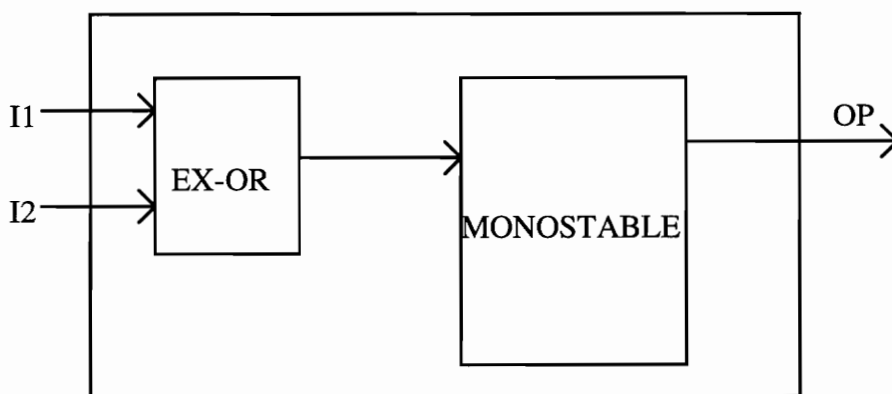


Figure 28: A monostable multivibrator

Example 9:

```
architecture MONO_ARCH of MONO is  
begin  
process  
begin  
s1:    wait until I1 = I2;  
        OP<= transport '1';  
        OP<= transport '0' after 5 ns;  
s2:    wait until I1 /= I2;  
end process;  
end MONO_ARCH;
```

It is possible to inject fault in statements s1 and s2 by changing the until clause. For example, s1 can be modified to *wait until I1 /= I2* and s2 can be modified to *wait until I1 = I2*. The resulting wave forms in the fault free and faulty case is shown in Figure 29. The pulse is generated whenever $I1 = I2$ in the fault free case whereas the pulse is generated whenever $I1 \neq I2$ in the faulty case. This implies an ex-nor gate instead of ex-or gate at the enabling circuitry. This can not be justified in terms of physical failures.

The gate level fault model considers physical failures in the circuit and accordingly modifies the behavior at the gate level. But no gate level fault can change the behavior of the ex-or gate to ex-nor gate. This is a pure behavioral fault and modifies the timing

behavior of the circuit. A separate study of such faults may lead to some interesting facts. These faults are not considered in the present research.

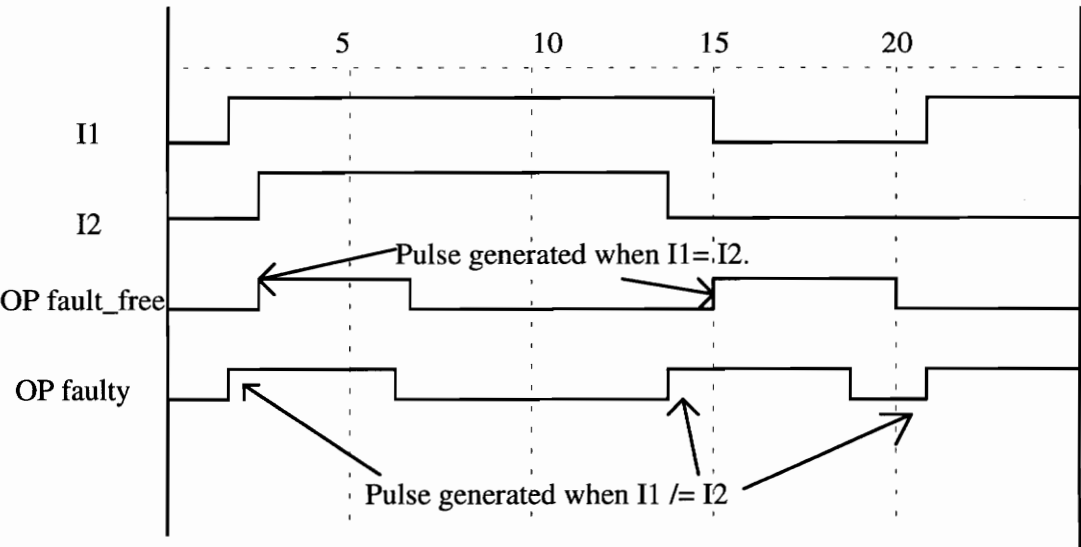


Figure 29: A pure behavioral fault

Chapter 5

BEHAVIORAL LEVEL INPUT-OUTPUT PATHS

As mentioned in section 4.2, it is possible to construct a Data Flow Graph (DFG) of a behavioral model to generate a list of behavioral i/o paths. In this chapter, a systematic way to generate a list of all behavioral level i/o paths is presented.

In section 5.1, a standard notation used to represent behavioral statements graphically is explained. In section 5.2, the use of a data flow graph to construct a list of all behavioral input-output paths implied by the model is presented. In the present research, a data flow description of a behavioral model is used to generate a data flow graph.

There are different graphical representations proposed [39, 40, 41] to represent a behavioral model in terms of a DFG. The representation scheme in the present research is adapted from [40, 41]. This representation gives necessary information regarding the i/o paths in a behavioral model.

Since we are only interested in input-output paths implied by a behavioral model, **control paths are not considered separately**. The representation adapted here gives enough information to generate a list of all the **i/o paths** implied by a behavioral model.

5.1 Analysis of Statements

5.1.1 A simple example of a DFG

Consider an expression $f = (u + v) * (v + w)$. The DFG corresponding to the expression is shown in Figure 30. Inputs to this expression are variables u , v and w . These define starting nodes of the DFG. A node is defined by each operator. f is the output of the expression.

From the DFG it is possible to identify paths implied by the expression. For example, between u and f there is one path and between v and f there are two paths.

If this expression represents an architectural body of an entity in VHDL description where u , v and w are inputs and f is output, then it is possible to identify that v is a fanout stem.

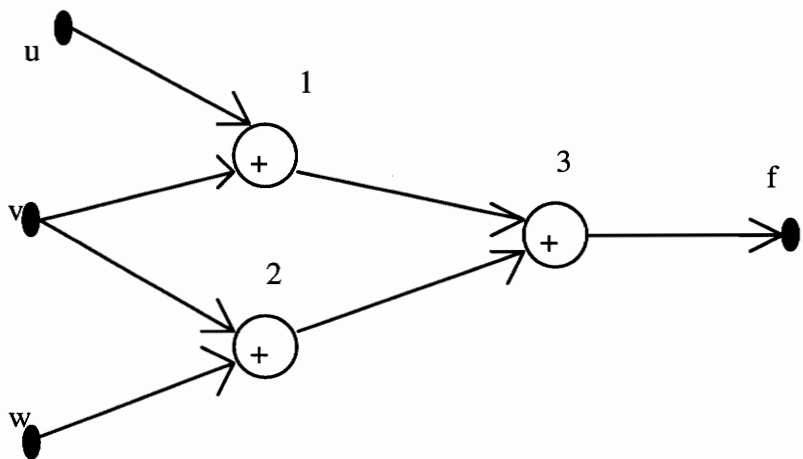


Figure 30: Data Flow Graph of the statement $f = (u + v) * (v + w)$

5.2.2 If-then-else statement/case statement

A typical if-then-else statement has the following format:

```
if condition then
    ----if part
else
    ----then part
end if;
```

A conditional node is defined corresponding to the *condition* in the statement. (Figure 31). The condition node has two outputs: true and false. The *if part* and *then part* is represented by a node. The input to the *if part* and *then part* nodes are any data signals or variables appearing in these parts respectively. If the condition is true, then *if part* is executed and if the condition is false, then *then part* is executed.

It can be noticed that from the control signal input, two paths are originating corresponding to the edge T and F. Thus this control signal is a behavioral fanout stem.

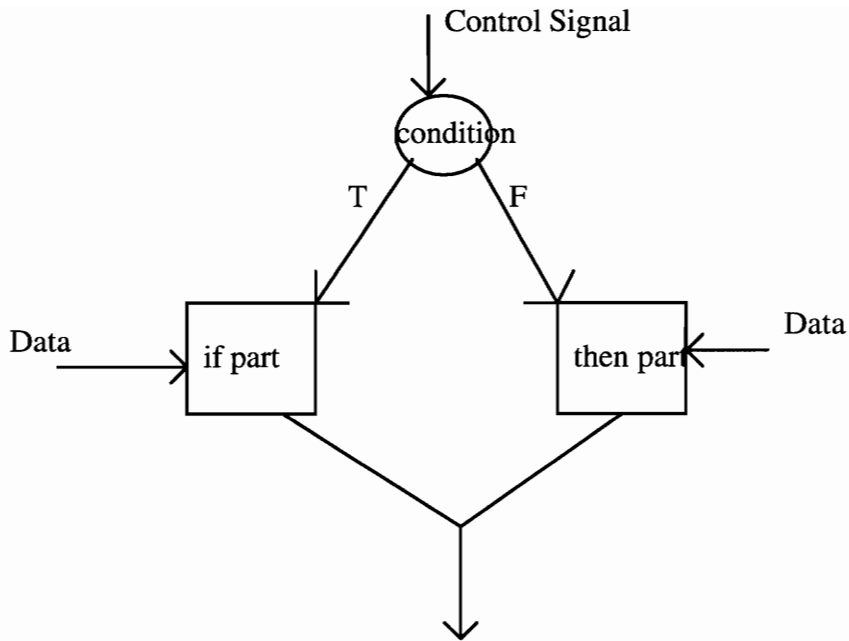


Figure 31: if-then-else statement

A case statement consists of multiple if-then-else statements. Thus using the same methodology, DFG for a case statement can be constructed as shown in Figure 32. Again there is more than one path originating from the control input and thus the control input is a behavioral fanout stem.

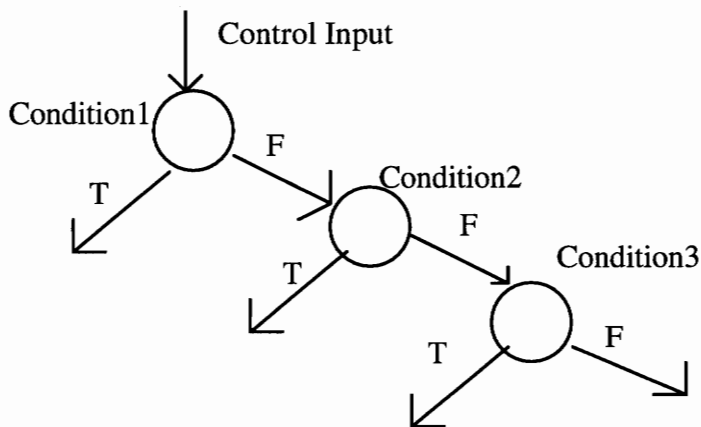


Figure 32: Case statement

5.1.3 Graphical Representation of the wait statement

A wait statement with the for clause inserts a delay in the behavioral block. For example in the statement,

wait for 10 ns;

the simulator delays the execution of all the statements following the wait statement by 10 ns.

In order to represent this graphically, a node with the delay defined by the wait clause is inserted in the DFG as shown in Figure 33.

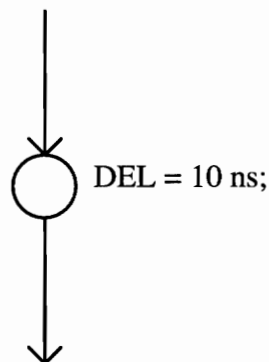


Figure 33: Representation of *Wait for* statement

5.1.4 Representation of Attributes

An attribute can be represented by a node shown in Figure 34. The attributes used in this research to define behavioral delay faults use a parameter of type TIME in its argument. The node represents the delay implied by this parameter.

For example:

A'delayed (5 ns) or A'stable(5 ns) can be represented as shown in Figure 34.

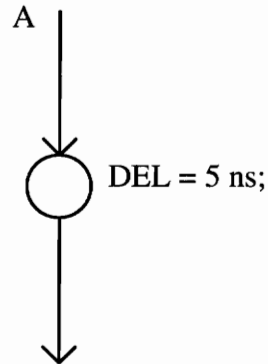


Figure 34: Attribute representation

The attribute A'last_event, does not involve any parameter of type TIME as one of its arguments. But if it appears in a conditional statement (such as *if A'last_event > 5 ns then.....*), then it can be represented by the node shown in Figure 34.

Since we are interested in the delay implied by the node, attributes such as S'delayed'stable(5 ns)), can also be represented by the node shown in Figure 34.

5.2 Construction of a DFG: Examples

In order to construct a DFG, the following convention is followed:

1. Nodes in a DFG are numbered serially.
2. If the node corresponds to a condition, then there are two outputs to this node, TRUE and FALSE. The TRUE output points to the line number that is executed when the

condition is TRUE and the FALSE output points to the line number that is executed when the condition is FALSE.

3. The control nodes are shadowed, since there is no delay involved at that node.

Consider the behavioral model for a two-to-one multiplexer shown below. When SEL = '0', A[0] is connected to the output and when SEL = '1', A[1] is connected to the output. This behavior can be written using an if-then-else statement as shown below. A DFG corresponding to the model can be constructed as shown in Figure 35.

Example 1.

entity TWO_ONE_MUX is

port(A: BIT_VECTOR(1 downto 0); SEL: in BIT; O: out BIT);

end TWO_ONE_MUX;

architecture MUX_ARCH of TWO_ONE_MUX is

begin

process (A, SEL)

begin

1: if SEL='0' then

2: O <= A[0] after 5 ns;

3: else

4: O <= A[1] after 5 ns;

end if;

end process;

end MUX_ARCH;

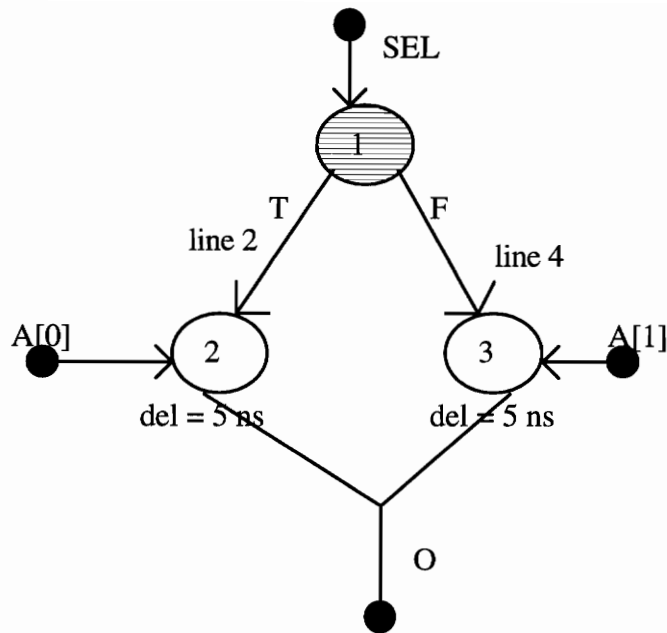


Figure 35: A DFG of a 2-to-1 mux

Node 1 is a conditional node and implies the condition defined by the statement. The input to this node is the control signal SEL. If the condition is true, then it points to line 2 and if the condition is false, then it points to line 4. The nodes 2 and 3 represent signal assignments. Inputs to nodes 2 and 3 are the signals A[0] and A[1] respectively.

From this DFG, the following list of behavioral level i/o paths can be easily constructed:

SEL-1-2-O

SEL-1-3-O

A[0]-2-O

A[1]-3-O

The total delay along a path is the sum of delay at each node along the path. For example delay along the path SEL-1-2-O is 5 ns¹.

Example 2: Two-to-one mux with enable

Consider a two to one mux with enable signal shown below (Example 2). This example illustrates the DFG for a case statement. The mux is enabled when EN1 and EN2 both are equal to one. The code for this model is shown below.

entity EN_MUX is

port(A: BIT_VECTOR(1 downto 0); EN1, EN2, SEL: in BIT; O: out BIT);

end TWO_ONE_MUX;

architecture MUX_ARCH of EN_MUX is

begin

process (EN1, EN2, A, SEL)

begin

1. if EN1 and EN2= '1' then

2: case SEL is

3: when '0' =>

4: O<= A[0];

5: when '1' =>

¹ This is an unrealistic representation of the delay through the path. A more realistic representation, which considers the delay at node 1 is presented in chapter 6.

```

        6:          O<= A[1];
                end case;
        end if;
end process;
end MUX_ARCH;

```

The corresponding DFG can be constructed as shown in Figure 36.

Node 1 corresponds to the if statement in line 1, and inputs to this node are control signals EN1 and EN2. If the condition is true, then it points to line 3. If the condition is false, no action is taken. Therefore, the output of this node has only one output, pointing to line 3.

Nodes 2 and 4 correspond to the case statement. If the first condition in case statement is true, then the DFG points to line 4 and if the condition is false, it points to the next condition on line 5. These nodes 3 and 5 represent signal assignment statements and thus input to these nodes are signals A[0] and A[1] respectively.

From this the following set of behavioral i/o paths can be constructed:

EN1-1-2-3-O

EN1-1-2-4-5-O

EN2-1-2-3-O

EN2-1-2-4-5-O

SEL-2-3-O

SEL-2-4-5-O

A[0]-3-O

A[1]-5-O

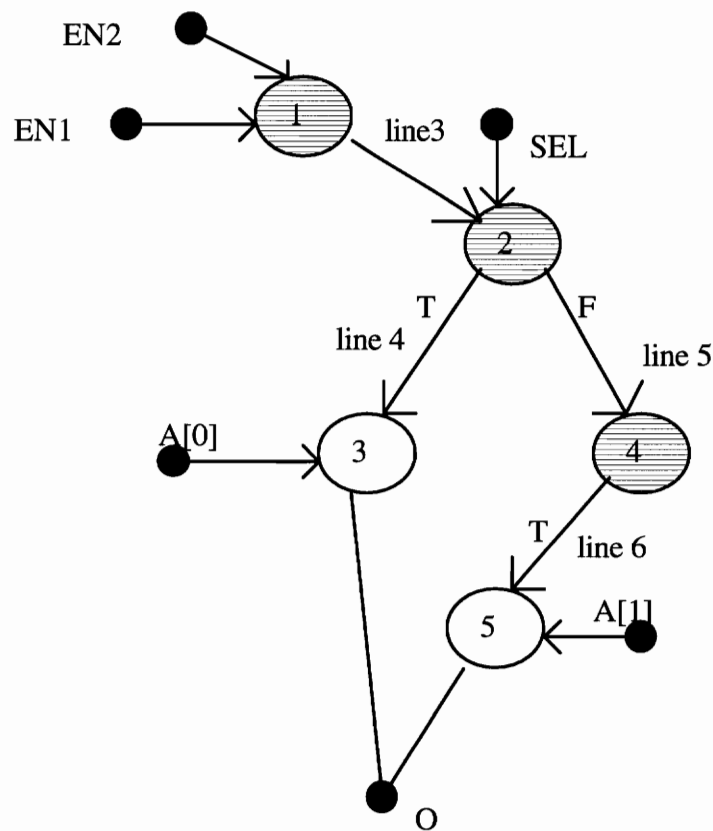


Figure 36: DFG corresponding to a MUX with Enable

Example 3: An Arithmetic Logic Unit:

Consider an arithmetic logic unit shown in Figure 37. This ALU has two multiplexers at the input. The MUX1 is selected by EN1 and EN2 signals whereas MUX2 is selected by EN3 and EN4 signals. The operations performed within the ALU (NOT, AND and OR) are controlled by the control signal CON.

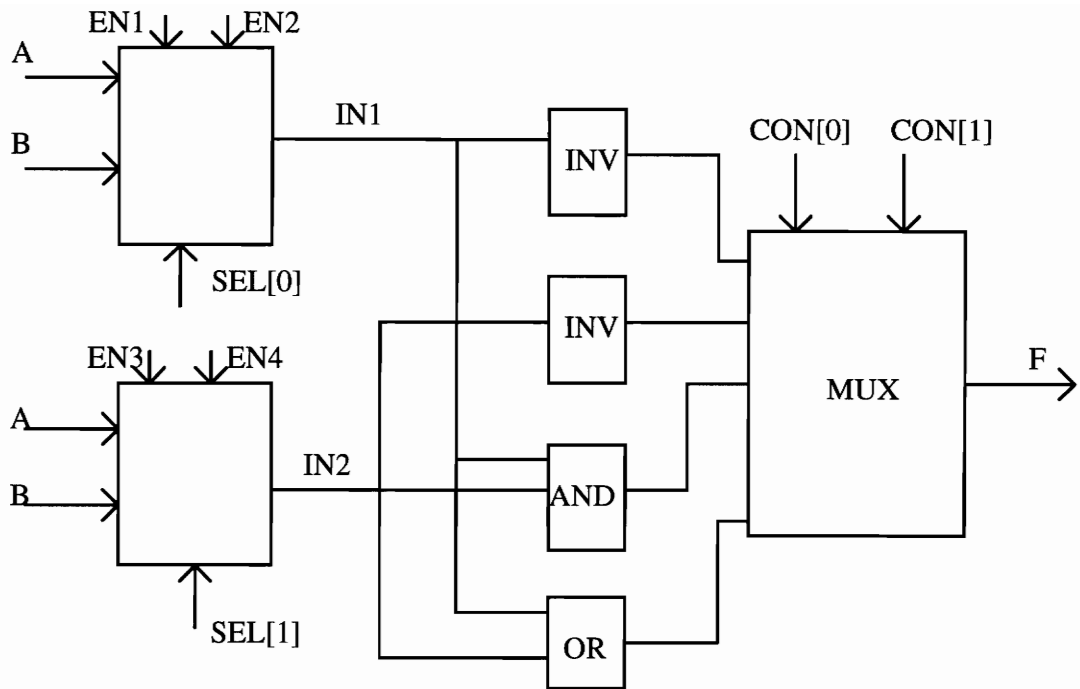


Figure 37: An Arithmetic Logic unit

The code corresponding this behavior is shown below:

entity ALU is

port(A, B, EN1, EN2: in BIT; SEL, CON: in BIT_VECTOR(1 downto 0); F: out BIT);

end ALU;

architecture ALU_ARCH of ALU is

signal IN1, IN2: BIT;

begin

MUX1: process(A, B, EN1, EN2, SEL[0])

begin

1. if EN1 and EN2= '1' then

```

2:    case SEL is
3:        when '0' =>
4:            IN1<= A;
5:        when '1' =>
6:            IN1<= B;
        end case;

```

end process MUX1;

MUX2: process(A, B, EN3, EN4, SEL[1])

begin

```

7. if EN3 and EN4= '1' then
8:    case SEL is
9:        when '0' =>
10:           IN2<= A;
11:       when '1' =>
12:           IN2<= B;
        end case;

```

end process MUX2;

ALU: process (IN1, IN2, CON)

begin

case CON is

```

13: when "00"=>
14:     F:<= not IN1;
15: when "01"=>

```

```

16:          F<= not IN2;
17: when "10"=>
18:          F<= IN1 and IN2;
19: when "11"=>
20:          F<= IN1 or IN2;
    end case;
end process ALU;
end ALU_ARCH;

```

The DFG corresponding to the model is shown in Figure 38. The DFG of processes MUX1 and MUX2 is constructed as explained in previous examples.

The nodes numbered 11, 13, 15 and 17 define conditional nodes implied by the case statement for the process ALU. The nodes numbered 12, 14, 16 and 18 define the signal assignments.

From this DFG, it is possible to construct complete list of all the behavioral i/o paths implied by the model.

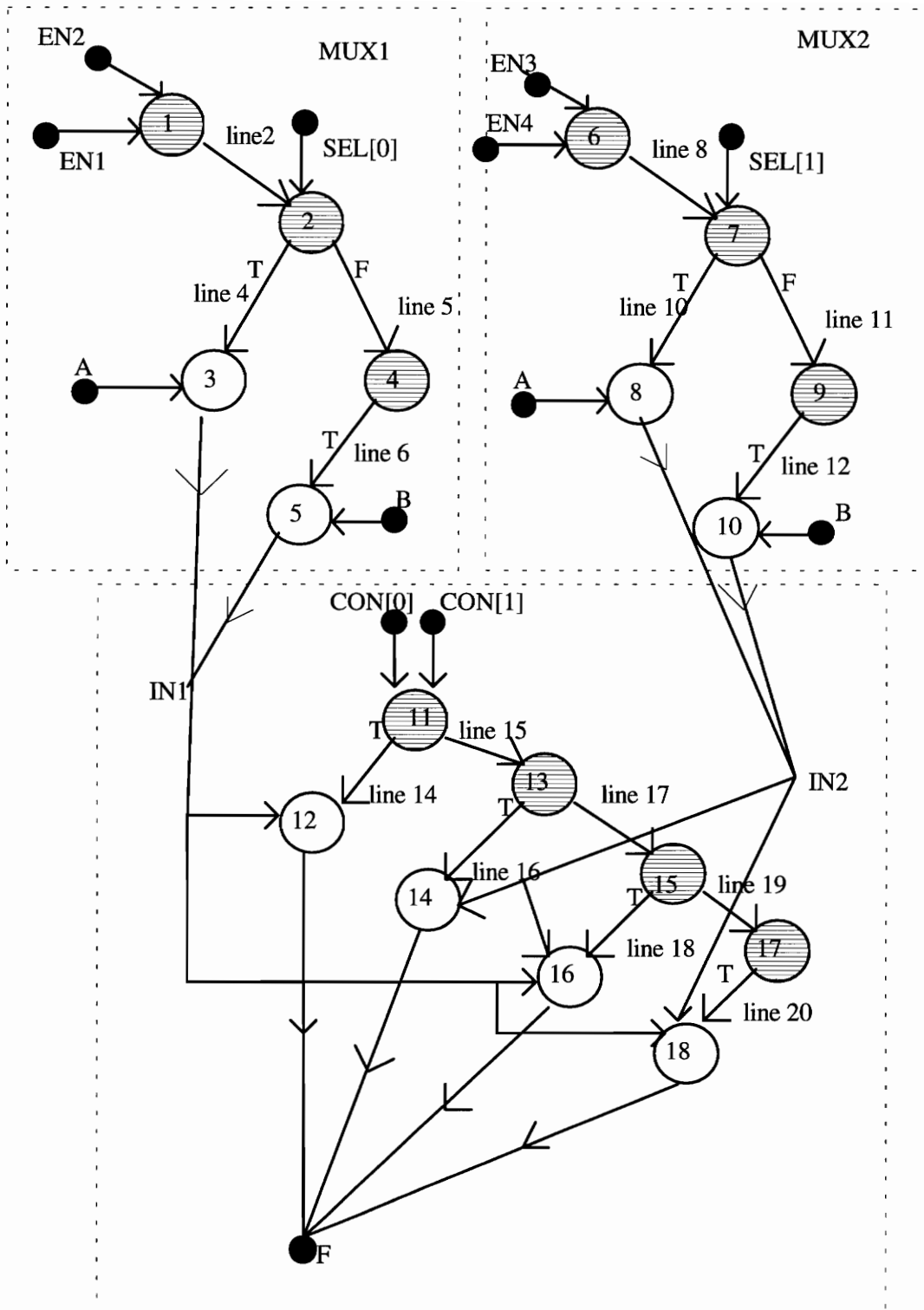


Figure 38: DFG of an ALU

Example 4: Controller for a counter

This example illustrates a simple case of representation of a wait statement.

In Example 4, a control unit for an up-down counter is shown. If CON ='00' then the controller loads the count in the counter. The controller waits till the count is loaded in the counter, and then enables the counter to count in up direction. When CON='11', the controller loads the count in the counter, again waits till the data is loaded on counter and sets the counter in the down direction. When CON = "01", it sends a reset signal to the counter and when CON="10", it sends a set signal to the counter. A partial DFG for the lines numbered 1-19 is shown in Figure 39.

entity CONTR is

```
port ( CON , INCNT : in BIT_VECTOR(0 to 1); OPCNT: out BIT_VECTOR(0 to 1);  
LOAD, UPCNT, DNCNT,SET,RESET: out BIT);  
end CONTR;
```

architecture CONTR_ARCH of CONTR is

```
signal COND: BIT_VECTOR(0 to 1);
```

```
begin
```

```
process
```

```
begin
```

```
1      wait on CON, INCNT;
```

```
2      COND<= CON ;
```

```

3      wait for 1 ns;
4      if COND="00" then
5          DNCNT<='0';
6          SET<='0';
7          RESET<='0';
8          LOAD<= '1' after 1 ns;
9          OPCNT <= INCNT after 1 ns;
10     wait for 5 ns;
11     LOAD<='0';
12     UPCNT <='1';
13     elsif COND = "11" then
14         UPCNT<='0';
15         SET<='0';
16         RESET<='0';
17         LOAD<='1' after 1 ns;
18         OPCNT <= INCNT after 1 ns;
19         wait for 5 ns;
20         LOAD<='0';
21         DNCNT <= '1';
22     elsif COND="01" then
23         RESET<='1';
24         LOAD<='0';
25         UPCNT<='0';
26         DNCNT<='0';
27         SET<='0';

```

elsif COND="10" then

RESET<='0';

SET<='1';

LOAD<='0';

UPCNT<='0';

DNCNT<='0';

end if;

end process;

end CONTR_ARCH;

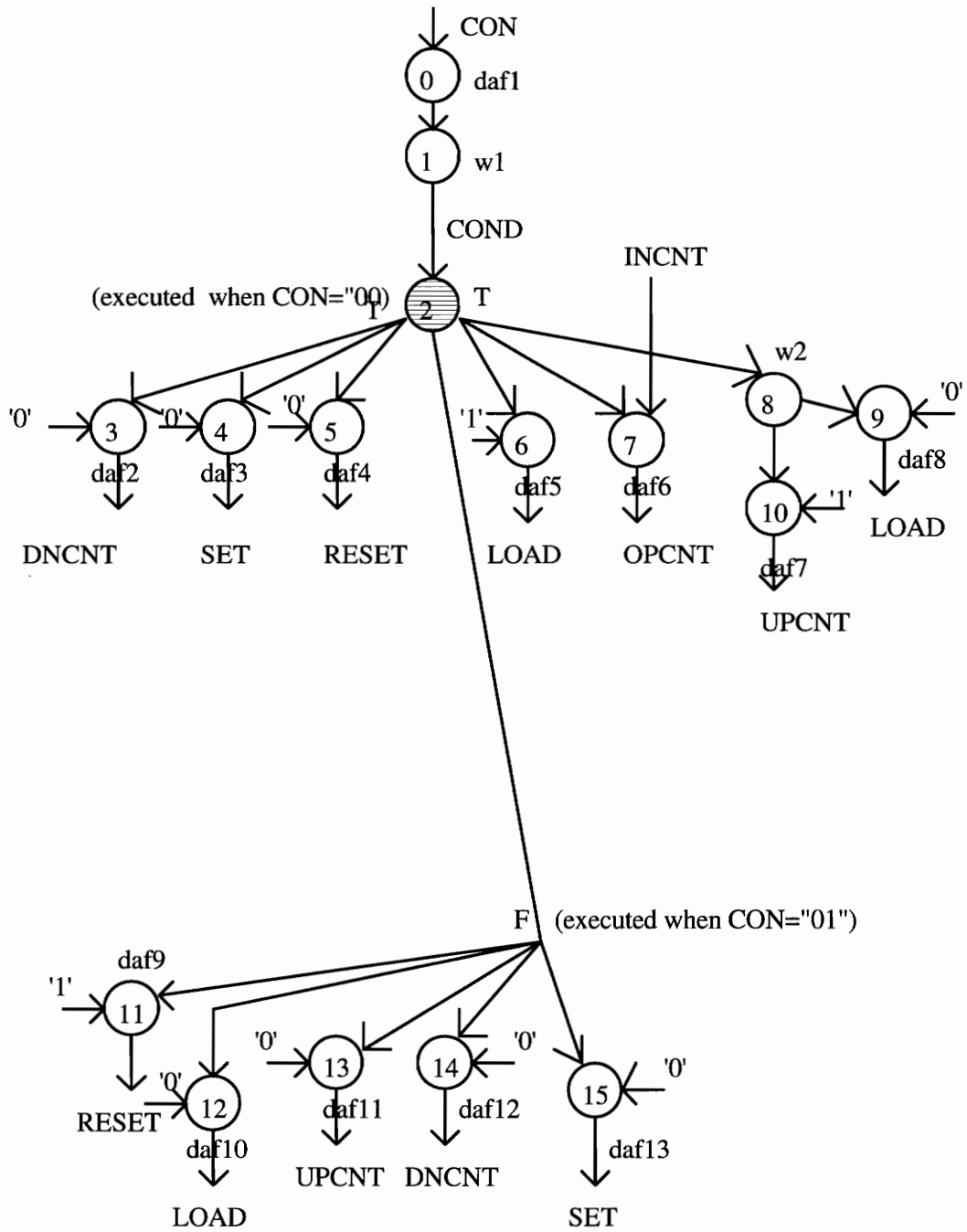


Figure 39: Partial DFG of example 1

The nodes numbered 1 and 8 correspond to the delay implied by the wait statement. The delayed after faults are denoted by daf1-daf13 and wait statement faults are denoted by w1 and w2. Other statements are represented graphically as explained in section 5.2.

Example 5:

This example illustrates a more complex case of representation of the wait statement.

Consider a peripheral communicator shown in Figure 40. This communicator is enabled by the signal EN. It communicates with two peripheral devices. When DAV1 is '1', it indicates that the data is available on peripheral device 1, and when DAV2 is '1', it indicates that the data is available on peripheral device 2. The data is read by the communicator, temporarily latched and then placed on the data bus. The delay involved in this action is represented by the wait statement. After that the buffer sends the DATARD signal indicating that the data has been read and has been placed on the main data bus.

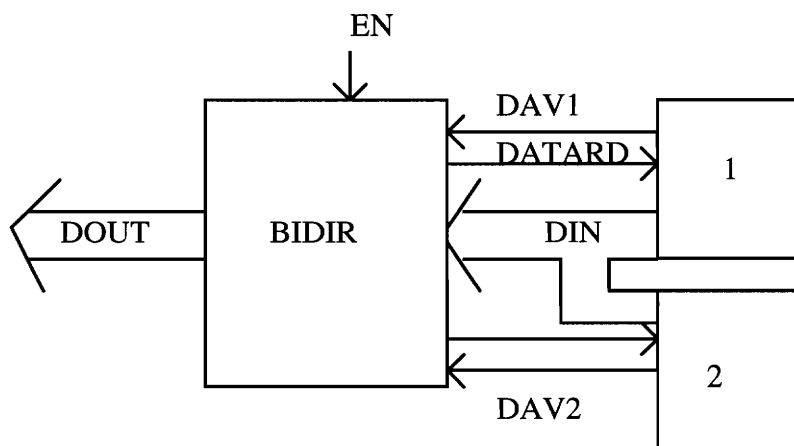


Figure 40: Example 5

The VHDL code for the model is shown below:

```
use work.all;

entity BIDIR is
port (DIN : in BIT_VECTOR(0 to 1); EN, DAV1, DAV2: in BIT; DOUT: out
BIT_VECTOR(0 to 1); DATARD: out BIT);
end BIDIR;

architecture BIDIR_ARCH of BIDIR is
signal BUF: BIT_VECTOR(0 to 1);
signal DAV1D, DAV2D, ENDA: BIT;
begin
process
begin
1      wait on EN, DAV1, DIN, DAV2;
2          ENDA<=EN;
3          DAV1D<= DAV1;
4          DAV2D<= DAV2;
5          wait for 1 ns;
6      if ENDA='1' then
7          if DAV1D='1' then
8              DATARD<='0' after 1 ns;
9              BUF<= DIN;
```

```

10          wait for 5 ns;
11          DATARD<='1';
           elsif DAV2D='1' then
           DATARD<='0' after 1 ns;
           BUF<= DIN;
           wait for 5 ns;
           DATARD<='1';
           end if;
12  else
13      DOUT<="00";
14      DATARD<='0';
15  end if;
16  DOUT<= BUF;
end process;
end BIDIR_ARCH;

```

A partial DFG for the lines numbered 1-16 can be drawn as shown in Figure 41. The nodes 1 and 2 correspond to the dummy signal assignment statements (line 2, 3). The nodes 3 and 4 represent the delay implied by the wait statement (line 5). The node representing wait statement appears more than once in the DFG. This is because all the statements in the behavioral block after the line 5 are delayed by the delay implied by the statement. This means that if EN or DAV1 or DAV2 changes, then there is always a delay associated with that signal before the change can reach the behavioral block after line 5. Thus the delay node is placed in all the signal lines: EN, DAV1 and DAV2 (only EN and DAV1 are shown in the Figure 41).

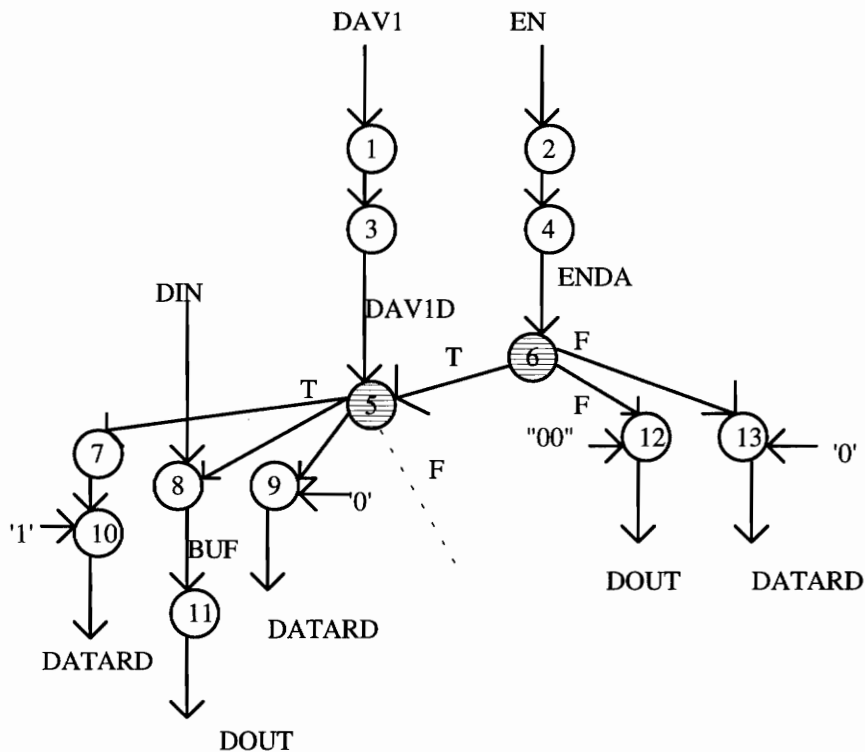


Figure 41: Partial DFG of example 5

Example 6:

This example illustrates representation of an attribute, in which delayed attribute fault can occur.

Consider a 2 bit data buffer. The data is latched on the rising edge of the signal STRB, and is available at the output when DS1='1' and NDS2='0'. The VHDL model for this behavior is shown below.

use work.all;

entity BUFF is

port(DI: in BIT_VECTOR(0 to 1); STRB, DS1, NDS2: in BIT; DO: out BIT_VECTOR(0 to 1));

end BUFF ;

architecture BUFF_ARCH of BUFF is

begin

A: block (STRB='1' and not STRB'stable)

signal REG: BIT_VECTOR(0 to 1);

signal ENBL: BIT;

begin

REG<= guarded DI after 3 ns;

ENBL <= DS1 and not NDS2 after 2 ns;

DO<= REG after 3 ns when ENBL='1'

else "11" after 2 ns;

end block A;

end BUFF_ARCH;

The DFG is shown in Figure 42. Node 2 is a dummy node and is not shown in the code. This node may represent the delayed attribute fault (STRB='1' and STRB'stable changed to STR'delayed(T)='1' and not STRB'delayed(T)'stable) which maps to a clock skew.

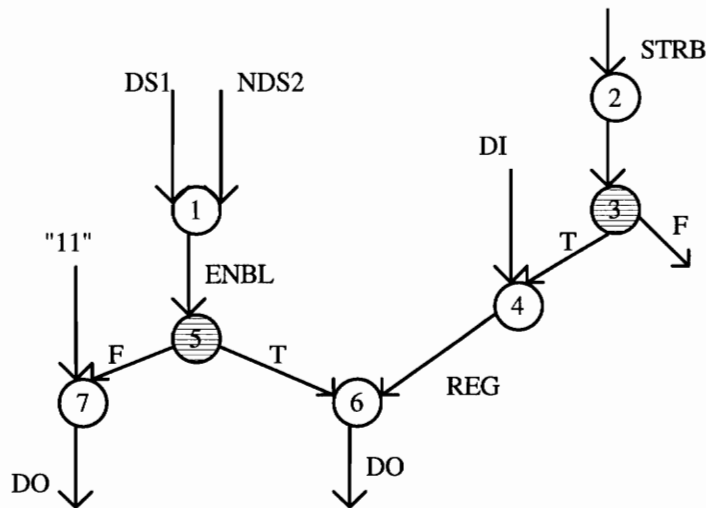


Figure 42: DFG of a buffer

Chapter 6

BEHAVIORAL DELAY TEST GENERATION

In this chapter, examples are presented that illustrate how a delay test can be generated for a behavioral model. It is also shown that these tests developed from the behavioral model can be used as a test for a path at the gate level.

6.1 Dummy Nodes

Consider a simple 2-1 multiplexer. The mux has two inputs A and B. When SEL = '0', A is selected and when SEL = '1', B is selected. The behavioral model for this mux is shown in Example 1. In this description, two delays are defined: ADEL and BDEL. If the signal A or B changes and SEL remains steady, then the change in the value of signal A or B is propagated to the output with a delay defined by the generics ADEL or BDEL. If SEL changes and A and B remain steady, then this change in the value of SEL is propagated to the output with the delay defined by the generics ADEL or BDEL.

Example 1

entity TWO_ONE_MUX is

```

port( A, B: BIT ; SEL: in BIT; O: out BIT);
generic(ADEL, BDEL: TIME);
end TWO_ONE_MUX;

architecture MUX_ARCH of TWO_ONE_MUX is
begin
process (A,B, SEL)
begin
    1: if SEL='0' then
    2:      O<= A after ADEL;
    3: else
    4:      O<= B after BDEL;
    end if;
end process;
end MUX_ARCH;

```

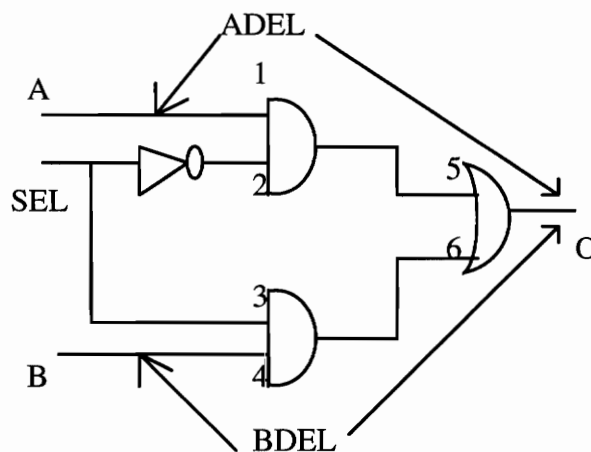


Figure 43: 2-1 MUX: Gate level equivalent

The gate level equivalent of the 2-1 mux is shown in Figure 43. From this the following statements can be made:

1. If SEL is steady, and A or B changes then the effect is propagated to the output with ADEL or BDEL.
2. IF A and B are steady, and SEL changes, the effect is propagated to the output with a delay different from ADEL or BDEL. This delay value is independent of the value of the delay ADEL or BDEL. Also one path between SEL to O has an inverter, the other does not.

The first statement is true for the behavioral model, but second statement is not true for a behavioral model. This is a fundamental problem with modeling at the behavioral level. It is not possible to map all the delay parameters from gate level accurately to the behavioral level. There will be some loss of information while going from gate to behavioral level.

If in a gate level circuit, there is a fault at pin SEL, then the delay through both the paths(SEL-2-5-O, SEL-3-6-O) to the output is affected. Although it is not possible to map the exact delay at the behavioral level, it is possible to model this faulty behavior at the behavioral level. An additional statement can be inserted in a behavioral model before the control statement with a delay CDEL as shown in example 1(a):

Example 1(a)

architecture MUX_ARCH of TWO_ONE_MUX is
signal SELD: BIT;

```

begin
process (A,B, SEL,SELD)
begin
    1:SELD<= SEL after CDEL;
    2: if SELD='0' then
    3:     O<= A after ADEL;
    4: else
    5:     O<= B after BDEL;
    end if;
end process;
end MUX_ARCH;

```

Any delayed after fault in this statement(statement 1) can affect the delay through the both if and else clauses. This fault represents a delay fault at the behavioral fanout stem. Considering this fault would enable us to consider a fault at the fanout stem using the behavioral level description and would help in achieving a better gate level equivalent fault coverage.

In this work a dummy signal is inserted before each conditional expression to simulate the effect of the fault mentioned above.

6.2 Test Generation

6.2.1 General Considerations:

A data flow graph of the multiplexer (with additional signal assignment statement) described in Example 1(a) is shown in Figure 44.

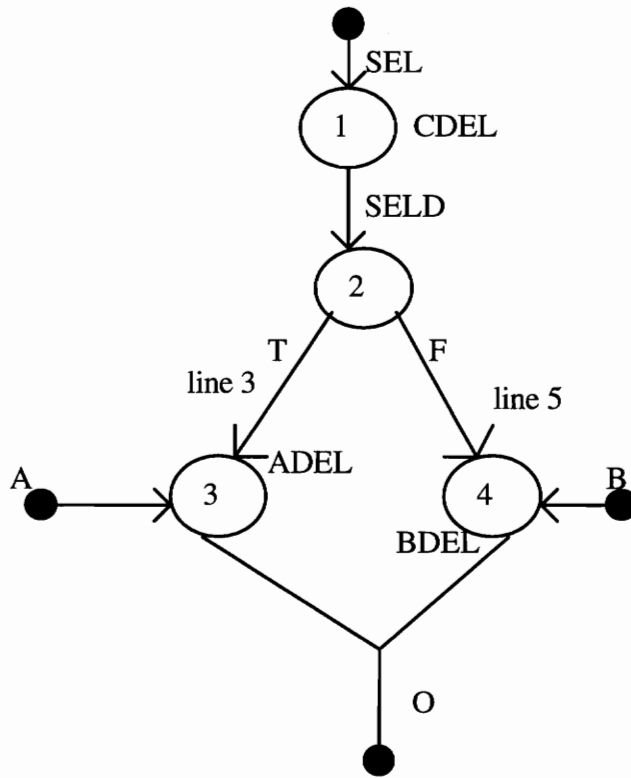


Figure 44: A DFG of a 2-to-1 mux

Case I: First assume that only one fault can occur at a time. Consider a delay fault at the node 1. In the presence of the fault, all the signals passing through this node to output will be delayed by the size of this fault. Thus in order to test this fault, it will be necessary to

sensitize the fault and propagate the effect along any path from the fault site to a primary output. This case gives rise to the first test generation strategy.

Case II: If the delay along the path SEL-1-2-3-O exceeds the normal value, then this could be attributed to a fault at node 1 or 3. Thus if we want to test faults which might affect delay through a behavioral path, both the faults should be sensitized and the effect should be propagated to the output. This gives rise to strategy II. In strategy II, a test sensitizes all the delay faults on a behavioral i/o path. Thus this strategy involves multiple fault sensitization. A test for a behavioral i/o path should sensitize all the faults on the path and propagate the effect to the output on the path.

The two strategies that should be considered in order to generate a test for the fault are explained below.

6.2.2 Strategy I

Test generation for this fault will involve the following steps:

1. Generate a fault list. This fault list will contain different defined behavioral level delay faults.
2. Pick one fault from the fault list and sensitize it.
3. Propagate the transition to a primary output.
4. Justify any assigned values.
5. If any fault is not tested, go back to step 2.

6.2.3 Strategy II

Test generation for this fault will involve the following steps:

1. Generate a behavioral i/o path list. A fault list will contain a list of all the behavioral delay faults along each path in a behavioral model. This is explained in the next section.
2. Pick one path and create a transition to sensitize the first delay fault on the path.
3. This transition should also sensitize the next delay fault on the path.
4. Repeat step 3 until the transition reaches the primary output on the path and all the delay faults on the path are sensitized.
4. Justify any assigned values.
5. If any paths are not tested, go back to step 2.

6.3 An Example

Consider a 2-1 mux with enable(example 2). The mux is enabled when both A and B are '1'. Depending on the value of SEL, either I1 or I2 is selected. The behavioral description for the model is shown in Example 2. As mentioned in section 6.1, dummy nodes are inserted before control statements. The DFG corresponding to the model is shown in Figure 45.

Example 2:

entity MUXEN is

port (A, B, I1, I2, SEL: in BIT; O: out BIT);

end MUXEN;

architecture MUXEN_ARCH of MUXEN is

signal ENB : BIT;

signal SELD, ENDA: BIT;

begin

process(A,B,I1, I2, A'delayed(3 ns), B'delayed(3 ns),ENB, ENDA, SELD, SEL)

begin

1: ENB<= A'delayed(3 ns) and B'delayed(3 ns) after 2 ns;

2: ENDA<= ENB;

3: SELD<=SEL;

4:if ENDA = '1' then

5: case SELD is

6: when '0' =>

7: O <= I1 after 5 ns;

8: when '1' =>

9: O <= I2 after 5 ns;

10: end case;

11:end if;

end process;

end MUXEN_ARCH;

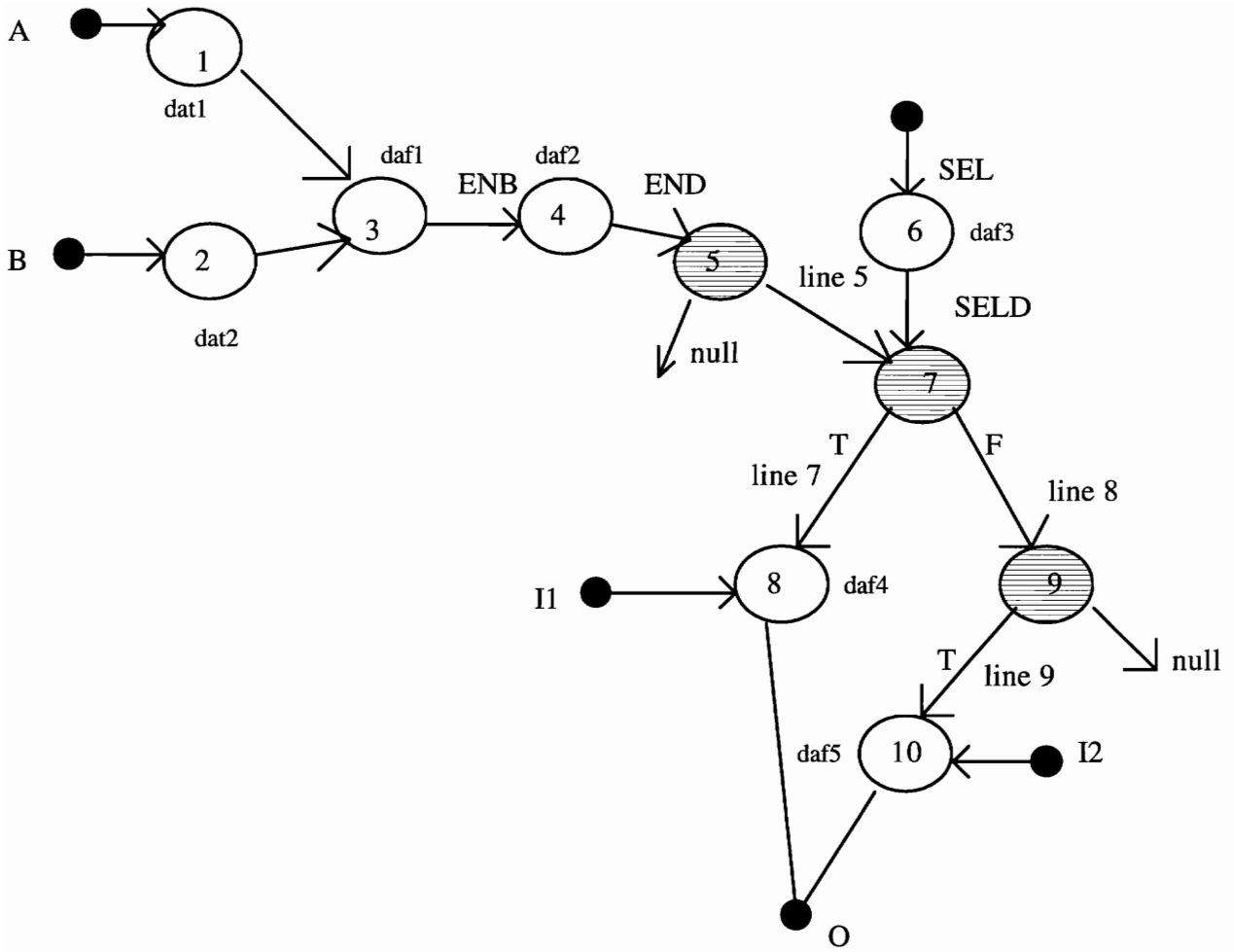


Figure 45: DFG of mux with enable

The shaded nodes indicate that these are control nodes and there is no delay clause involved.

6.3.1 STRATEGY I:

6.3.1.1 Fault list generation:

A fault list consists of a list of faults of all the types that may exist in a model. The following convention is followed:

Delayed after fault is denoted by daf*, where * is the fault number.

Fault in attribute is denoted by dat*, where * is the fault number.

Each fault list entry contains the following information:

1. Fault Number (fn)
2. Fault Type (ft). The fault types are: delayed after faults, faults in wait statements, faults in generics and faults in attributes. These are abbreviated at the respective nodes in Figure 45.
3. Statement Number (sn)
4. Node Number (nn)

Complete fault list for the Example 2 is shown below. The fault list has a format (fn, ft, sn, nn).

(1, dat1, 1, 1)

(2, dat2, 1, 2)

(3, daf1, 1, 3)

(4, daf2, 2, 4)

(5, daf3, 3, 6)

(6, daf4, 7, 8)

(7, daf5, 9, 10)

6.3.1.2 Test generation:

Test generation consists of three steps: sensitization, propagation and justification.

Fault dat1:

Consider a fault dat1. In order to sensitize this fault, A should be changed from an initial value Vi to a final value Vf. Thus A can be changed from '0' to '1' to sensitize the fault.

This transition should be propagated to a primary output along any path. If B is maintained at '1', then the transition can reach node 7. At this point, signal SELD should be considered. If SELD is assigned '0' value, and I1 is assigned '1' value, then for A= '0' output will be '0' and for A='1', output will be '1'. Thus the transition can reach the output.

Thus the following values can be assigned to signals:

B='1'

SELD='0'

I1= '1'

Any assigned values can now be justified. SEL can be set equal to '0' and I2 can be assigned any value, say '0'.

Thus complete test sequence can be constructed as follows:

	A	B	I1	I2	SEL	O	O faulty
t= 0	0	1	1	0	0	0	0
t=t1	1	1	1	0	0	0	0
t= t1 + max_del	1	1	1	0	0	1	0

The wave forms corresponding to the faulty and fault free case are shown in Figure 46.

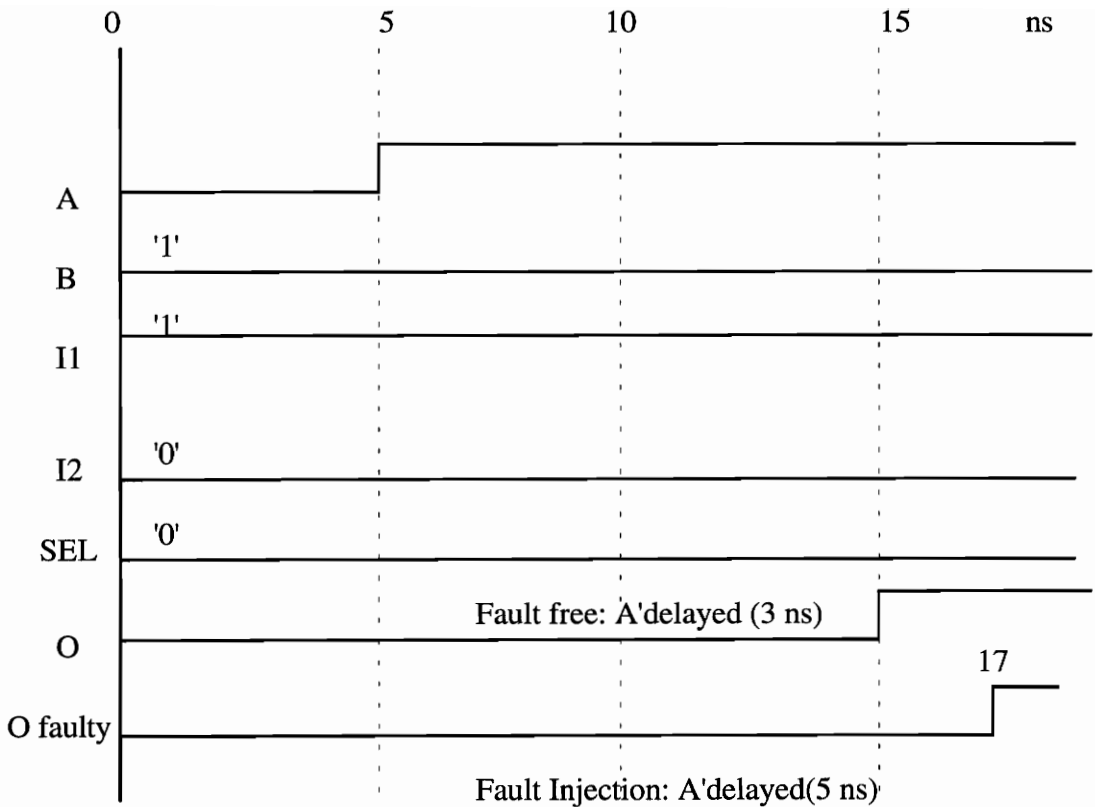


Figure 46: Fault dat1

Fault 'daf3':

Consider a fault daf3. This fault can be sensitized by changing the value of SEL signal from '0' to '1'.

To propagate this fault, values should be assigned to the different inputs. The transition at SEL can propagate to output only if END is maintained at '1'. Next, it can be observed that when SEL='0', line 7 is executed and when SEL='1', line 8 is executed. Thus if I1 is maintained at '0' and I2 is maintained at '1', corresponding to SEL='0', output will be '0' and corresponding to SEL='1' output will be '1'. Thus the following values can be assigned to signals:

I1='0'

I2='1'

END='1'

During the justification phase, we need to justify the assignment END='1'. A and B are assigned the following values to maintain END ='1':

A='1'

B='1'

Thus the following test sequence can be constructed:

	A	B	I1	I2	SEL	O	O faulty
t= 0	1	1	0	1	0	0	0
t=t1	1	1	0	1	1	0	0
t= t1 + max_del	1	1	0	1	1	1	0

The fault free and faulty wave forms are shown in Figure 47.

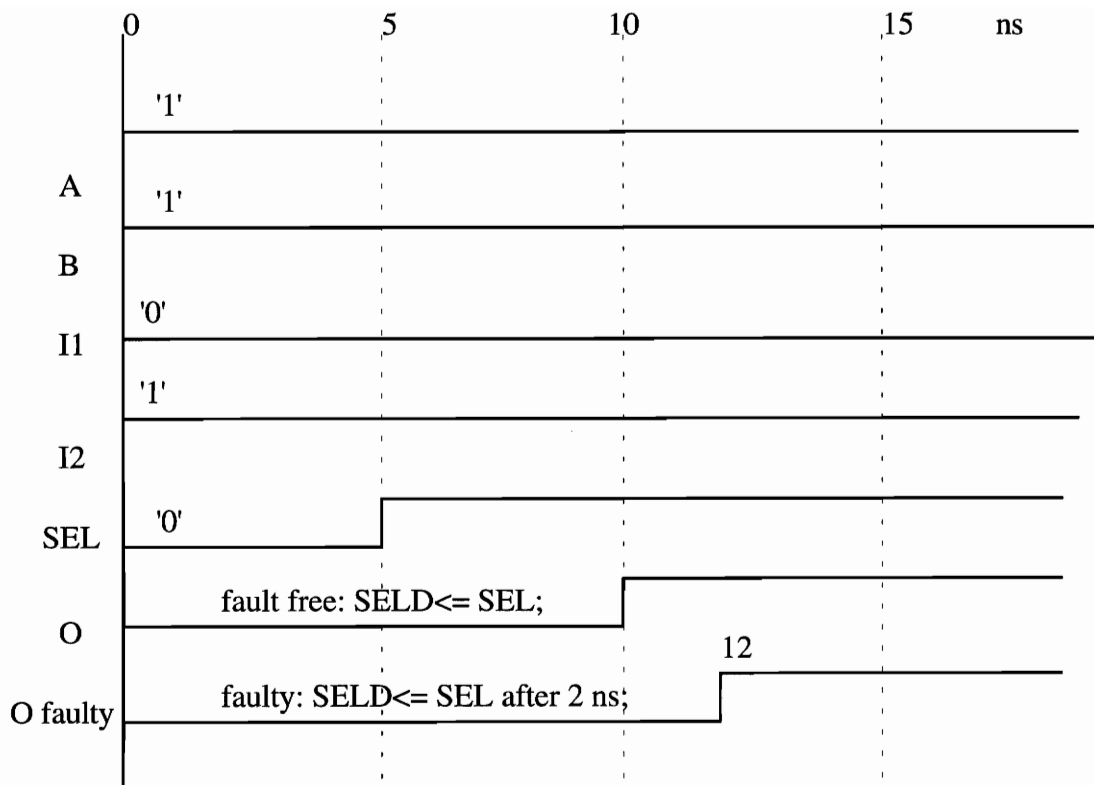


Figure 47: Fault daf4

Fault daf4:

Consider the delayed after fault daf4. In order to sensitize this fault, signal I1 should be changed from '0' to '1'.

This value can propagate to the output if two conditions are satisfied: SELD is maintained at '0' and ENDA is maintained at '1'. Thus during the propagation phase the following assignments can be made:

SELD='0'

ENDA='1'

To justify ENDA='1' and SELD='0', the following values can be assigned:

A='1'

B='1'

SEL='0'

The input I2 can be assigned any value say '0'.

This following test sequence can be constructed:

	A	B	I1	I2	SEL	O	O faulty
t= 0	1	1	0	0	0	0	0
t=t1	1	1	1	0	0	0	0
t= t1 + max_del	1	1	1	0	0	1	0

The wave form for the faulty and fault free case is shown in Figure 48.

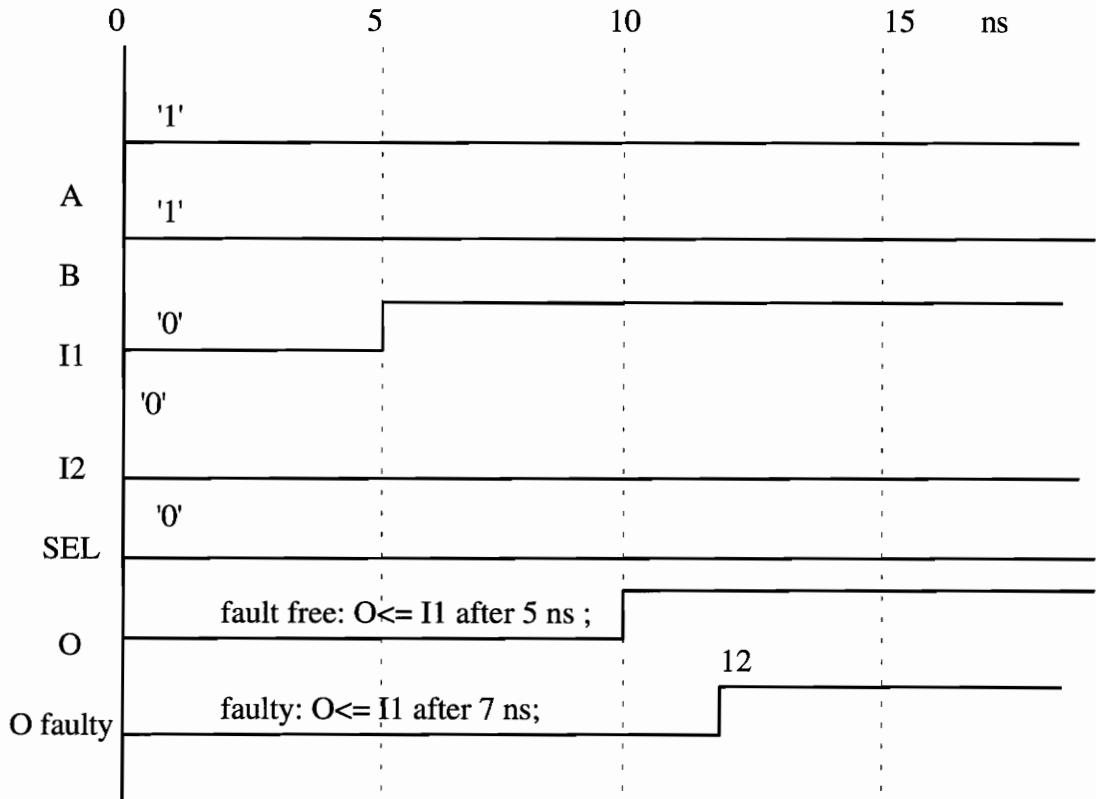


Figure 48: Fault daf4

6.3.2 Strategy II:

Consider the same example again. The fault list is now a list of possible delay faults along each path. Thus for a path 2-3-4-5-7-8, the fault list may appear as follows:

$\{(1, \text{dat2}, 1, 2)/(1, \text{daf1}, 1, 3)/(1, \text{daf2}, 2, 4)/(1, \text{daf4}, 7, 8)\}$

This means that along the path 2-3-4-5-7-8, all the faults dat2, daf1, daf2 and daf4 can exist **or** any one of the faults dat2 or daf1 or daf2 or daf4 can exist. Thus in order to test the fault, all the delay clauses along the path must be sensitized.

The following fault list can be generated for all paths:

```
{(1, dat1, 1, 1)/(1, daf1, 1, 3)/(1, daf2, 2, 4)/(1, daf4, 7, 8)}  
{(2, dat2, 1, 2)/(2, daf1, 1, 3)/(2, daf2, 2, 4)/(2, daf4, 7, 8)}  
{(3, dat1, 1, 1)/(3, daf1, 1, 3)/(3, daf2, 2, 4)/(3, daf5, 7, 10)}  
{(4, dat2, 1, 2)/(4, daf1, 1, 3)/(4, daf2, 2, 4)/(4, daf5, 7, 10)}  
{(5, daf3, 3, 6)/(5, daf4, 7, 8)}  
{(6, daf3, 3, 6)/(6, daf5, 7, 10)}  
{(7, daf4, 7, 8)}  
{(8, daf5, 7, 10)}
```

Note that the first number appearing with each fault here denotes the path number.

6.3.2.1 Constructing a test

Path A-1-3-4-5-7-8-O

Consider a path A-1-3-4-5-7-8-O. The fault list along this path is :

```
{(1, dat1, 1, 1)/(1, daf1, 1, 3)/(1, daf2, 2, 4)/(1, daf4, 7, 8)}
```

The first step is to sensitize the first delay fault along the path. Thus A can be changed from '0' to '1' to sensitize the fault in attribute at node 1.

In this strategy, we have to sensitize each fault listed in the fault list. The next fault in the fault list is daf1(at node 3). The transition from node 1 is input to node 3. We have to make sure that this transition also sensitizes the fault daf1. If input B is assigned a value

'1', then the transition can sensitize the fault daf1. The same transition will also sensitize the fault daf2.

Next step of sensitizing the delayed after fault at node 8 is more difficult. Here it is desired that the transition input to node 8 from node 4 should sensitize the fault daf4. We have to make a clever choice of values at inputs I1, I2 and SEL such that the transition sensitizes the fault daf4. This is best illustrated with the help of an example.

Consider a 2-1 mux in Example 1 again. The code is repeated here for convenience. Let ADEL=BDEL = 5 ns.

architecture MUX_ARCH of TWO_ONE_MUX is

signal SELD: BIT;

begin

process (A,B, SEL)

begin;

1: if SELD='0' then

2: O<= A after ADEL;

3: else

4: O<= B after BDEL;

end if;

end process;

end MUX_ARCH;

If $A=0$, $B=1$ and SEL changes from '0' to '1', then output follows the transition after 5 ns (Figure 49). This means that in a behavioral model, since only two delays are defined, it is possible to propagate the transition at the control input (SEL) to output with the delays defined.

A delayed after fault can be injected in the statement 4. If the output response is observed for the same inputs (Figure 49), it can be noticed that the fault is sensitized by the transition on SEL input. In the presence of these test vectors, if the output signal is sampled after $t_1 + 5$ ns, it is possible to detect the delayed after fault. Thus the assignment of values ($A=0$, $B=1$, SEL changing from '0' to '1') will detect the fault.

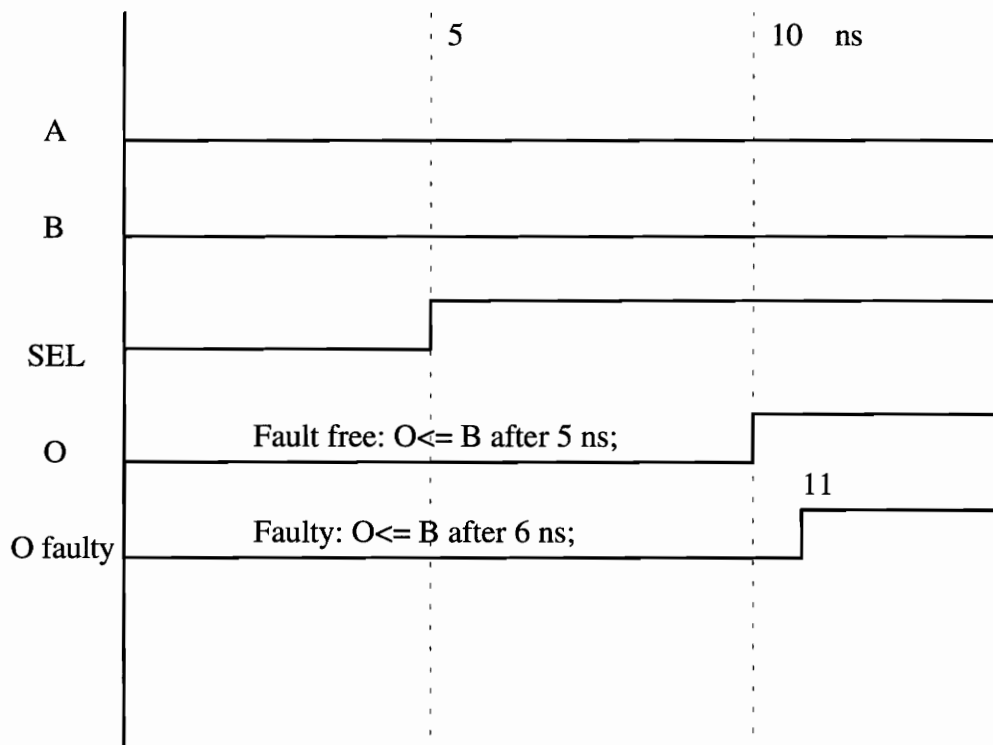


Figure 49: 2-1 mux: delayed after fault

The situation presented above is exactly encountered while sensitizing node 8 in our present example. Here there are two control inputs: SEL and ENB. ENB is changing from '0' to '1' and it is desired that this transition should sensitize the daf4 fault. This is possible only if the following assignments are made:

$$SEL = '0'$$

$$I1 = '1'$$

SEL = '0' ensures that when ENB = '1', the line 7 on the path between daf2 and daf4 is executed and I1 = '1' ensures that the value at the output is '1' when ENB = '1'. When ENB = '0', the default output value is '0'.

Thus when ENB changes from '0' to '1', with these assignments, all the faults in the fault list are sensitized(Figure 50)

I2 can be assigned any value , say '0'.

Thus the following test sequence can be constructed:

	A	B	I1	I2	SEL	O	O faulty
t= 0	0	1	1	0	0	0	0
t=t1	1	1	1	0	0	0	0
t= t1 + max_del	1	1	1	0	0	1	0

The resulting wave forms for faulty and fault free case are shown in Figure 50.

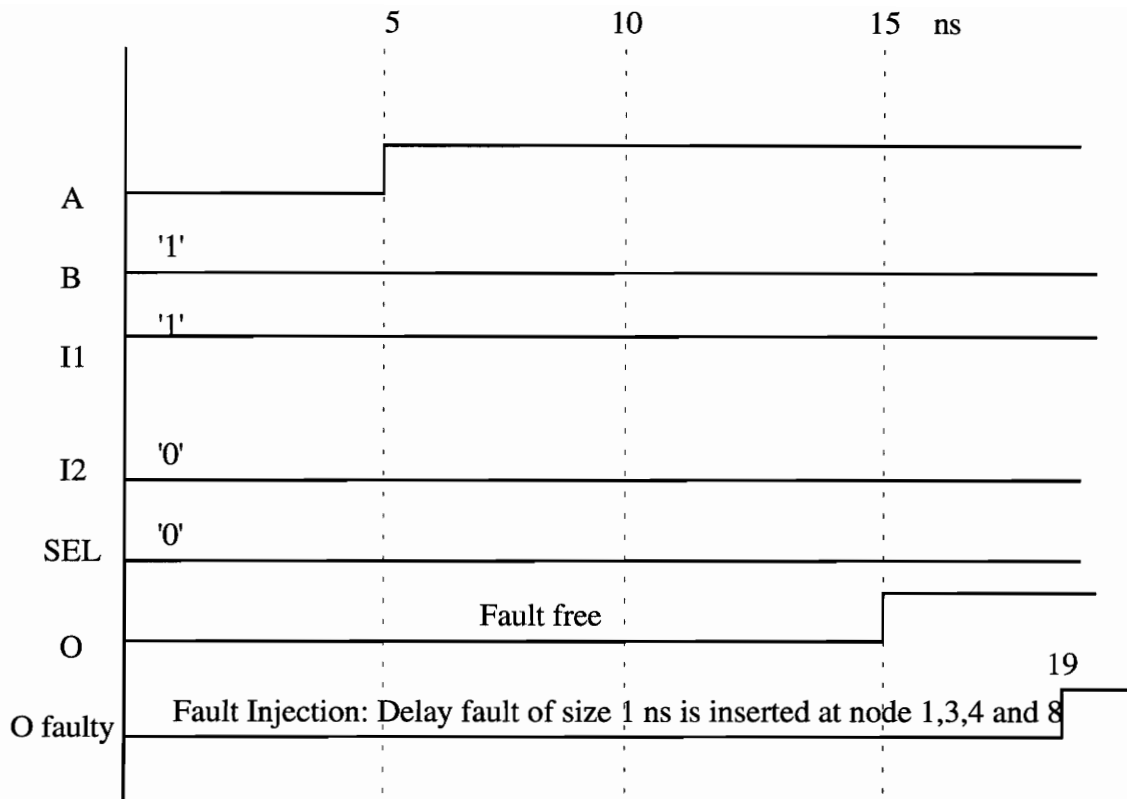


Figure 50: Path A-1-3-4-5-7-8-O

Path SEL-6-7-9-10-O

Consider the path SEL-6-7-9-10-O.

The fault list for the path is as follows:

$\{(6, \text{daf3}, 3, 6)/ (6, \text{daf5}, 7, 10)\}$

Again here value of the control signal should be changed to sensitize the fault. The first fault is daf3. In order to sensitize the fault, SEL should be changed from an initial value Vi

to a final value Vf. This assignment should be such that it should also sensitize the fault daf5.

The final value of the control signal should be such that whenever the control signal takes this final value, the line number which appears on that path is executed. Thus if SEL is changed from '0' to '1', then when SEL ='1', the lines 8 and 9 along the path are executed. Thus we can set final value as '1' and initial value as '0'.

This transition should also sensitize the fault daf5. For this proper value assignment to the signals I1 and I2 is necessary. Thus if I1 is maintained at '0' and I2 at '1', then it is possible to sensitize the fault.

This transition can reach output only if ENDA is maintained at '1'. Thus propagation phase involves assigning value to ENDA ='1'.

Justification phase involves assigning values to inputs A and B to maintain ENDA ='1'. Thus the following values can be assigned:

A='1'

B='1'.

The following test sequence can be constructed:

	A	B	I1	I2	SEL	O	O faulty
t= 0	1	1	0	1	0	0	0
t=t1	1	1	0	1	1	0	0
t= t1 + max_del	1	1	0	1	1	1	0

The fault free and faulty wave forms are shown in Figure 51.

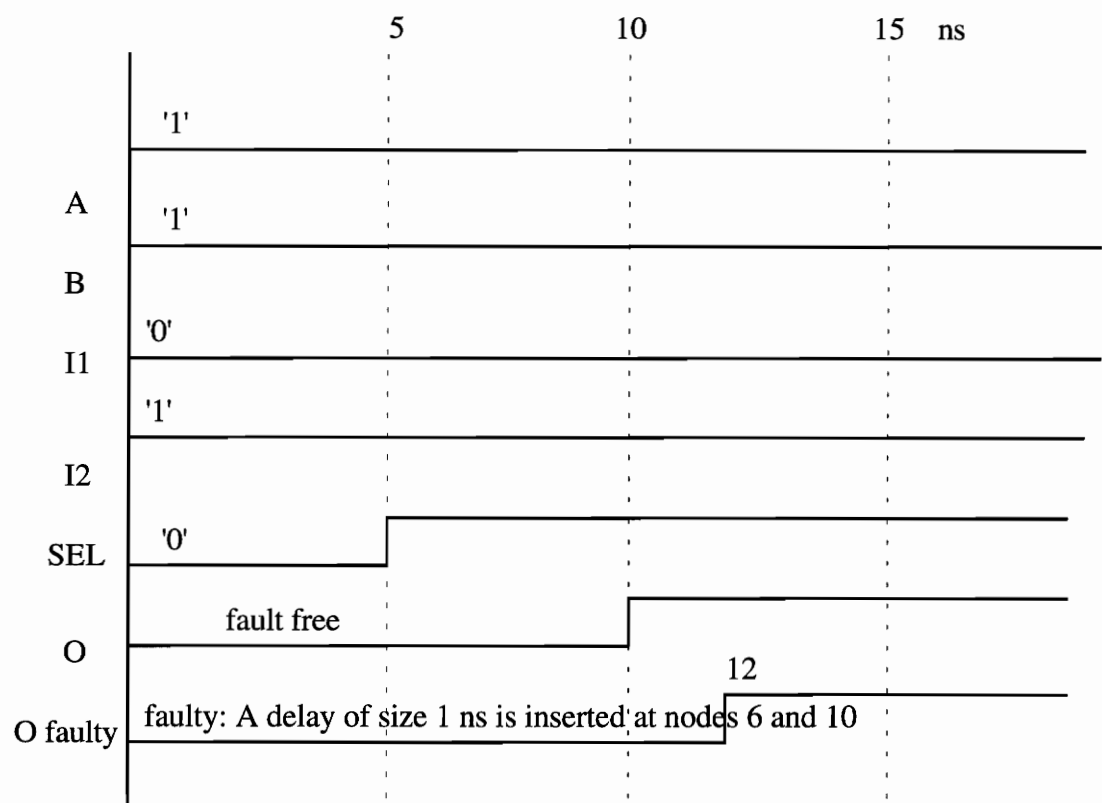


Figure 51: Path SEL-7-8-9-10-O

Path I-8-O

Consider the path I1-8-O. Test generation for the path is quite simple since there is only one delay clause along the path.

The following is the fault list:

{(7, daf4, 7, 8)}

In order to sensitize the fault, I1 can be changed from '0' to '1'.

To propagate the transition to the output, ENDA should be maintained at '1' and SELD should be maintained at '1'. Thus the following assignments can be made:

SELD='0'

ENDA ='1'

Justification phase involves assigning values to A, B, SEL and I2. The following values can be assigned:

A='1'

B='1'

SEL='0'

I2='0'

Thus the following test sequence can be constructed:

	A	B	I1	I2	SEL	O	O faulty
t= 0	1	1	0	0	0	0	0
t=t1	1	1	1	0	0	0	0
t= t1 + max_del	1	1	1	0	0	1	0

The wave forms in this case are the same as shown in Figure 48 and are not repeated.

6.4 Validation of the Test at the Gate Level

In order to validate tests generated from the behavioral model, the 2-1 mux with enable was synthesized using Synopsys Synthesis tool and the tests generated were applied to the gate level equivalent circuit. As can be seen in Figure 52 through 54, each test sensitizes a path and propagates the transition to the output. Thus it can be seen that the tests developed from the behavioral level model can indeed be used as a test for a path at the gate level.

In Figure 52, the test developed for fault dat1 or path A-1-3-4-5-8-O using the behavioral model tests the path A-3-5-O in gate level equivalent.

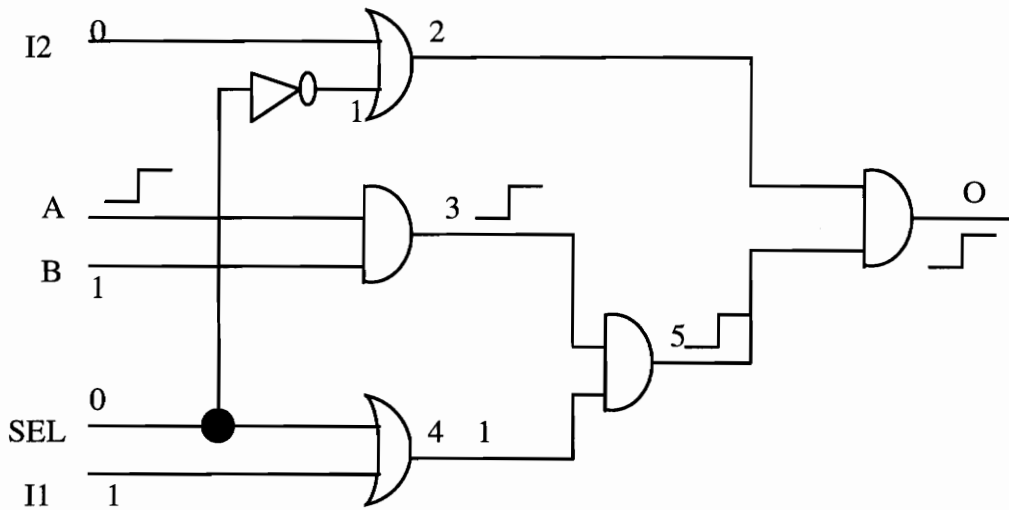


Figure 52: Path A-3-5-O

In Figure 53, the test developed for fault daf3 or path SEL-6-7-8-O using the behavioral model tests the path SEL-4-5-O in gate level equivalent.

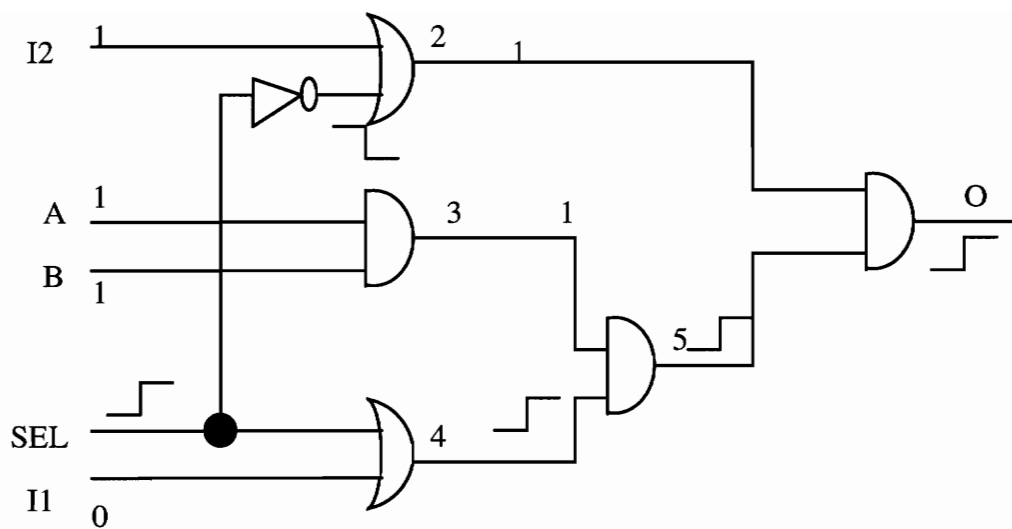


Figure 53: Path SEL-4-5-O

The test developed for fault daf1 or path I-8-O using the behavioral model tests the path I1-4-5-O in gate level equivalent(Figure 54).

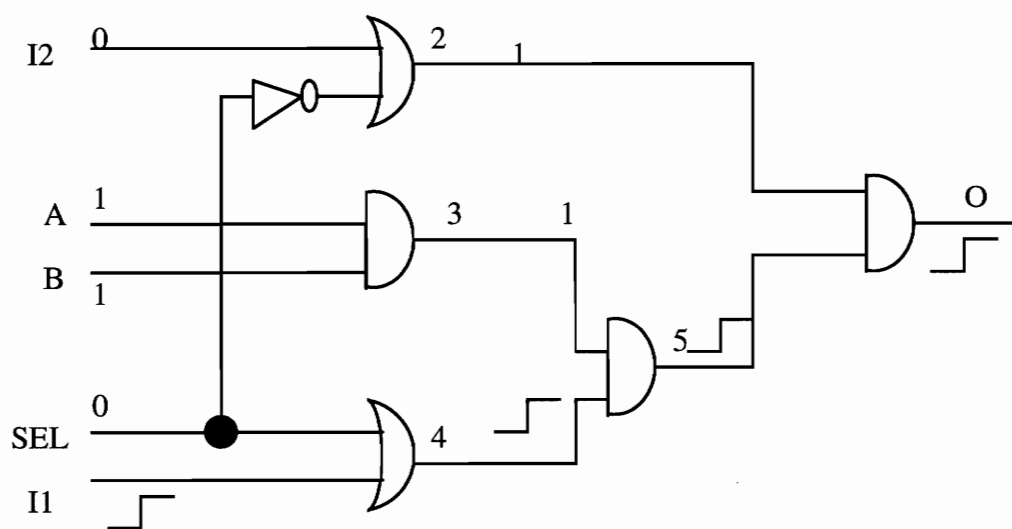


Figure 54: Path SEL-4-5-O

6.4.1 Discussion

As shown in Figure 52-54, it is possible to generate a delay test from the behavioral model and apply the test to a gate level equivalent circuit to test a delay fault. In this particular example tests developed from a gate level model will test all the gate level paths. But as the complexity of the model will increase, tests developed from a behavioral model may not test all the paths at the gate level.

It is mentioned in [13] that it is not necessary to test every path in a gate level circuit, and tests developed for a certain number of paths should be able to give 100% delay fault coverage. This raises an interesting question: As a pure mathematical exercise, can it be proved that the paths dictated by a behavioral model are the only paths to be tested in a gate level equivalent circuit? Here it can be argued that the effect of the change at the input can propagate to output only through these paths (as represented by the behavioral model). Can this mean that these paths are more important than other paths ?

Also the tests developed from a behavioral fault model can be hazard prone when applied to a gate level equivalent model. As mentioned in section 2.4.2, a test for a path can be non robust, i.e. , it may sensitize more than one path. Such tests may generate hazard at the output in the gate level circuit and invalidate the test. There is no mechanism to map hazards from a gate level model to a behavioral model. A further research of hazards at the gate level and their mapping at the behavioral level can throw light on some interesting facts of test generation at the behavioral level.

6.5 Generalization of the Test Procedure

The test generation procedure for a behavioral i/o path can be generalized. This section presents a general approach to generate a test for a behavioral path.

Different terms used in this section are defined below. A **control node** is the node implied by a conditional statement. This node does not have a delay value associated with it and has output edges, corresponding to true and false condition. A **delay node** is the node which has a finite delay value associated with it. **Control input signals** are the input signals to a control node, and **data input signals** are the signals which are not input to a control node.

In the following discussion, a sensitization strategy for two consecutive nodes on a path is considered. Along a path a delay node can be followed by a control node or delay node and a control node can be followed by a delay node or control node. The input to the first node has to be assigned a value such that it will also sensitize the next node. This section discusses those aspects. In the next section, a behavioral model is treated as a set of delay and control nodes, and a procedure to generate delay test is given.

6.5.1 Sensitization of two consecutive nodes

Case I: A delay node followed by a delay node (Figure 55).

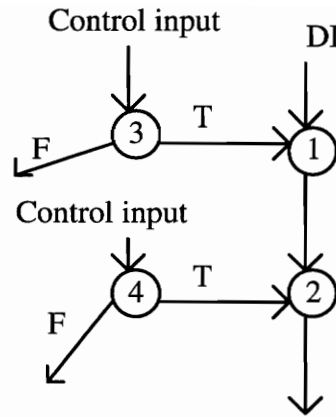


Figure 55. Sensitization of two delay nodes

Let DI be the data input signal to the node 1. A transition at the input DI can sensitize both the data nodes if all the control signals input to the data nodes on this path are maintained at such a value that this value enables the execution of statements implied by nodes. Any other data inputs to these delay nodes should be maintained at such a value that the transition at input DI can propagate to the output.

Proof: A control node controls the execution of different statements in a behavioral model. Thus if the control signals are assigned such values that the line numbers on the path are always executed, then any change at the input will propagate to output. This change will undergo all the operations represented by different statements which are executed and will thus sensitize any delay fault along the path.

Case II: A control node followed by a delay node (Figure 56).

A transition at the input CI can also sensitize a delay fault at the following delay node if (a) the signal CI is changed from an initial value to a final value such that this final value executes the line number implied by the delay node and

(a) the delay nodes, which are executed when the condition takes two different values (true or false) are input with two different data input signal values (so that the primary output on the path takes two different values corresponding to initial and final control input value).

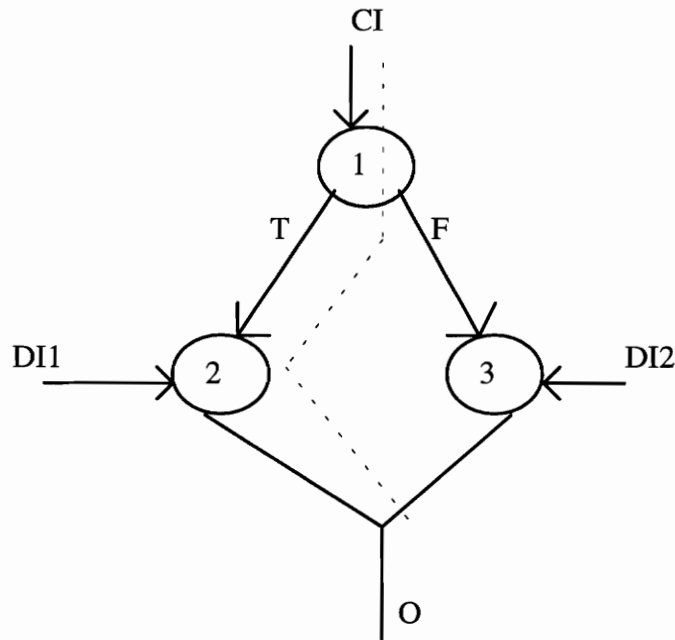


Figure 56: A control node followed by a delay node

For example in Figure 56, if a test for path 1-2-O is to be developed, then following steps can be taken:

1. CI can be changed from an initial value to a final value. This final value should be such that it executes the line number implied by the node 2.
2. The initial value of CI can be the value that sets condition false. The data inputs DI1 and DI2 should be assigned different values such that the output O is different for initial and final value.

Proof: A control statement controls the execution of the statements in a behavioral model. When the control node is assigned an initial value, the lines which are not on the path under test are executed and a certain value V_i appears at the output. This value is controlled by the delay node following the control node, which is not on the path under test. When the control signal takes the final value, the statement implied by the delay node on the path under test is executed. Now if this data input signal value(input to the delay node on the path under test) is assigned a different value from V_i , the output value will change from V_i to a new value, when the control signal takes the final value. Thus the change in the control signal value is propagated to the output with delay implied by the delay node, which sensitizes the delay node.

This leads to an important conclusion. If a control node is the starting node of a behavioral i/o path, then there must be at least two paths to the same output to generate a test for the path: one to propagate the effect of initial value of the control signal and other to propagate the effect of final value of the control signal.

Case III: A control node followed by a control node (Figure 57)

If along a path, a control node is followed by a control node, then following assignments will sensitize both the nodes:

1. CI can be changed from an initial value to a final value. This final value should be such that it executes the line number(s) on the path.
2. Other control inputs should be such that the execution of the lines on the path is enabled.

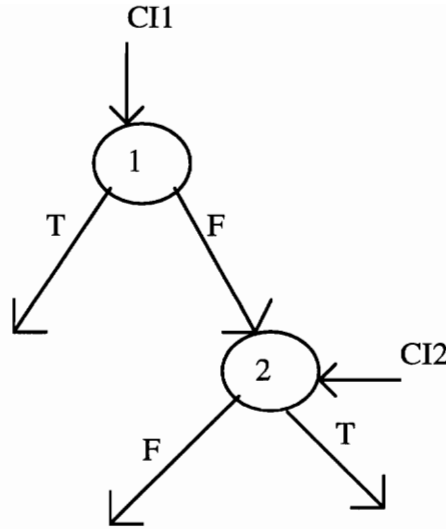


Figure 57: A control node followed by a control node

In Figure 57, if CI1 1-2-T path is to be sensitized, then CI1 can be changed from an initial value to a final value, this final value should enable the execution of the statement implied by node 2. CI2 should be assigned value such that the node 2 condition is always true.

If CI2-T path is to be sensitized, CI2 can be changed from an initial value to a final value, this final value should execute the statement implied by the edge T. CI1 should be assigned a value such that the condition implied by node 1 is always false.

The proof is similar to the case II.

Case IV: A delay node followed by a control node (Figure 58).

The data input can be changed from an initial value to a final value such that:

1. it sensitizes the data node

2. the final value of this assignment is such that the condition implied by control node following the data node always executes the statements on the path.

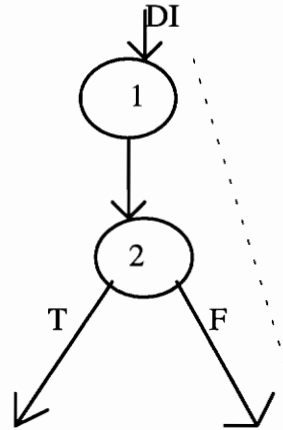


Figure 58: A delay node followed by a control node

In Figure 58, to sensitize the path DI-1-2-F, DI should be changed from an initial value to a final value. This final value should be such that the condition implied by node 2 becomes false when DI takes this final value. Any other inputs to nodes along the path under test should be maintained at a value that enables the propagation of the transition along the path to the output. The proof is the same as case II.

6.5.2 A Behavioral Model

A typical behavioral model can be represented by a collection of delay and control nodes as shown in Figure 59.

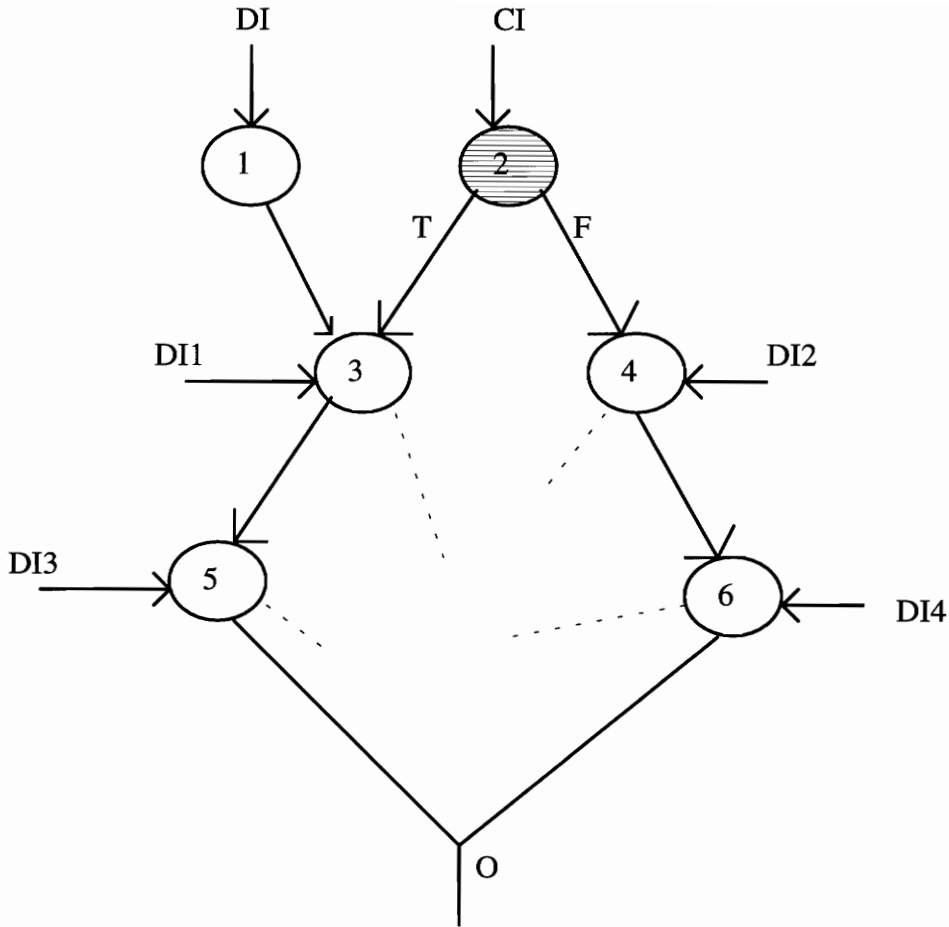


Figure 59: DFG of a behavioral model

A behavioral i/o path may start at the delay node and there could be delay or control nodes along the path. To sensitize such a path following procedure can be followed:

- (a) The data input signal should be changed from an initial value to a final value that will sensitize the first two nodes. This procedure can be repeated for other nodes on the path.
- (b) If there are control signal inputs to different nodes along the path, they should be maintained to a value such that the lines implied by the delay nodes are always executed.
- (c) Any other data inputs to the delay and control nodes should be maintained at such a value that enables the transition to propagate to the output along the path.

A behavioral i/o path may start at the control node and there could be delay or control nodes along the path. In this case, there should be at least one path each to the primary output on the path from the control node, when the condition is true and false. This is because we have to propagate two different values to the same output starting from the control node.

To sensitize such a path following procedure can be followed:

- (a) The control signal should be changed from an initial value to a final value. This final value should be such that it executes the line numbers along the path.
- (b) A path to the same primary output, when control signal is in initial value can be found. Now it is desired to initialize the output to a certain value V_i through this path. The data and control input signals along this path (note that this is not the path under test. This path initializes the output) should be assigned values to maintain output at a certain value V_i .
- (c) If there are control nodes along the path under test, then the control inputs should be maintained at such a value that will enable execution of the statements on the path.
- (d) Now the data and control nodes on the path under test can be assigned values, such that the output value is different that the initial value V_i .

6.6 Test Generation for a Behavioral Path: Examples

Several examples of test generation for a behavioral i/o path are presented below. Because of the complexity of the DFG, in some examples only a partial DFG is shown.

EXAMPLE I:

This example illustrates a simple case of a test generation for a wait statement fault on a path. The construction of DFG is already explained in Chapter 5. The code and DFG are repeated here for convenience.

In example I, a control unit for an up-down counter is shown. If CON = '00' then the controller loads the count in the counter. The controller waits till the count is loaded in the counter, and then enables the counter to count in up direction. When CON='11', the controller loads the count in the counter, again waits till the data is loaded on counter and sets the counter in the down direction. When CON = "01", it sends reset signal to the counter and when CON="10", it sends set signal to the counter. A partial DFG for the lines numbered 1-19 is shown in Figure 60.

entity CONTR is

```
port ( CON , INCNT : in BIT_VECTOR(0 to 1); OPCNT: out BIT_VECTOR(0 to 1);  
LOAD, UPCNT, DNCNT,SET,RESET: out BIT);  
end CONTR;
```

architecture CONTR_ARCH of CONTR is

```

signal COND: BIT_VECTOR(0 to 1);

begin

process

begin

1      wait on CON, INCNT;
2      COND<= CON ;
3      wait for 1 ns;
4      if COND="00" then
5          DNCNT<='0';
6          SET<='0';
7          RESET<='0';
8          LOAD<= '1' after 1 ns;
9          OPCNT <= INCNT after 1 ns;
10         wait for 5 ns;
11         LOAD<='0';
12         UPCNT <='1';
13     elsif COND = "11" then
           UPCNT<='0';
           SET<='0';
           RESET<='0';
           LOAD<='1' after 1 ns;
           OPCNT <= INCNT after 1 ns;
           wait for 5 ns;
           LOAD<='0';

```

```

                                DNCNT <= '1';
14    elsif COND="01" then
15        RESET<='1';
16        LOAD<='0';
17        UPCNT<='0';
18        DNCNT<='0';
19        SET<='0';

                                RESET<='0';
                                SET<='1';
                                LOAD<='0';
                                UPCNT<='0';
                                DNCNT<='0';

    end if;

end process;
end CONTR_ARCH;

```

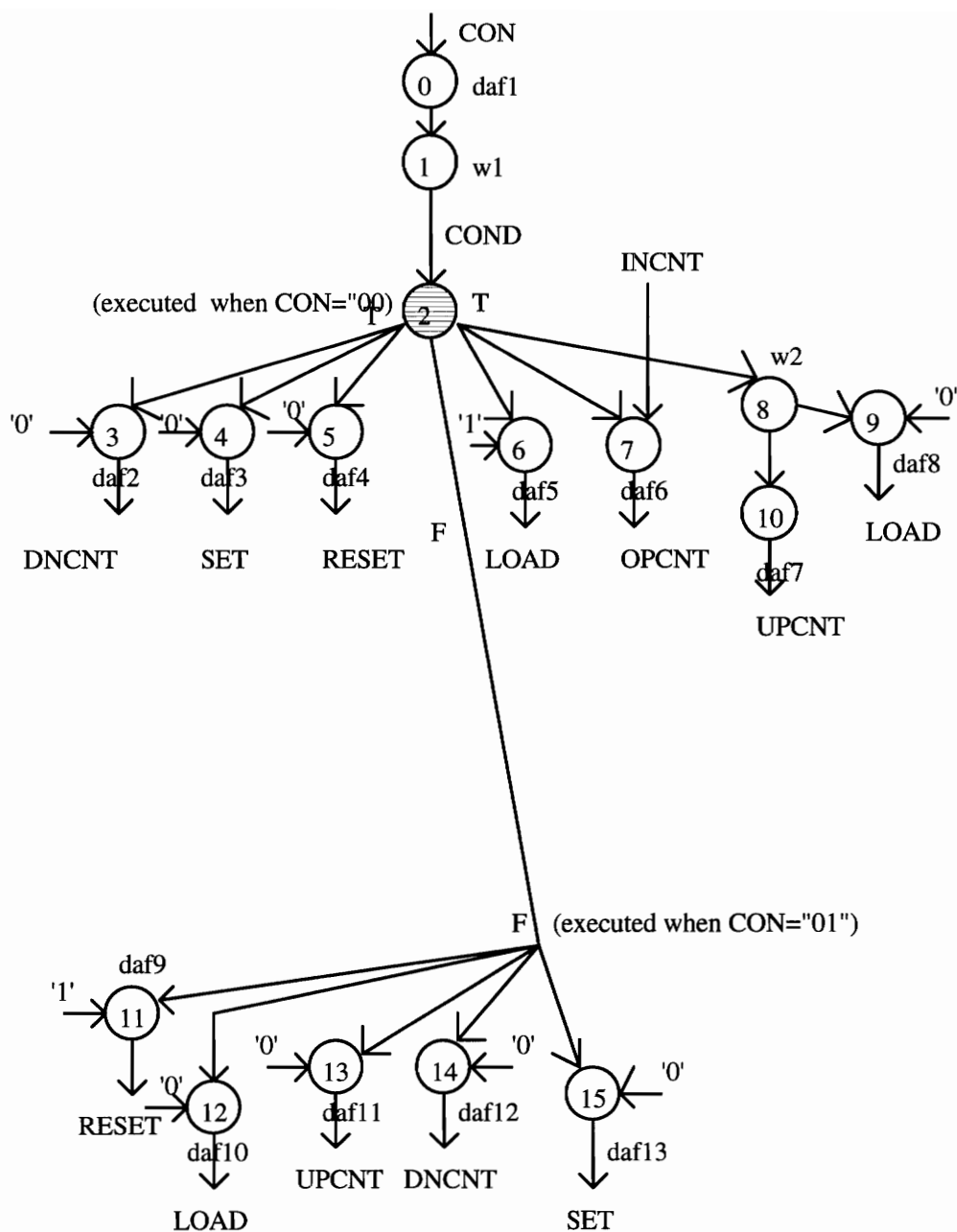


Figure 60: Partial DFG of example 1

Path CON-0-1-2-6-LOAD

Consider the path CON-0-1-2-6-LOAD. The delay fault along this path could exist at nodes 1(fault in wait statement) or 0, 6 (delayed after faults). Thus the test generated should be able to sensitize all the faults.

Sensitization: CON should be changed from an initial value to a final value to sensitize the faults at node 0 and 1. This final value should be such that it sensitizes (or executes) the line number on the path. (In figure 60, line numbers are not shown. This final value should be that when CON equals this value, it should execute the statement implied by the node 6) Thus if CON is changed from "01" to "00", it will sensitize the fault at node 0 and 1 and also execute the line implied by the node 6. Now it is desired that this should also sensitize the fault at node 6. This involves proper value assignment to input signals.

It can be observed that for CON ="01", the value of LOAD is '0' and for CON ="00", the value of LOAD is '1'. Thus the assignment will also sensitize the delayed after fault at node 6. Here no more assignments are necessary, and all the delay faults along the path can be sensitized by changing CON from "01" to "01".

Propagation and justification: Any value can be assigned to the input INCNT, say "00". Thus following test sequence can be constructed:

	CON	INCNT	LOAD	LOAD (faulty)
t=0	01	00	0	0
t=t1	00	00	0	0
t=t1 + max_del	00	00	1	0

The wave forms are shown in Figure 61.

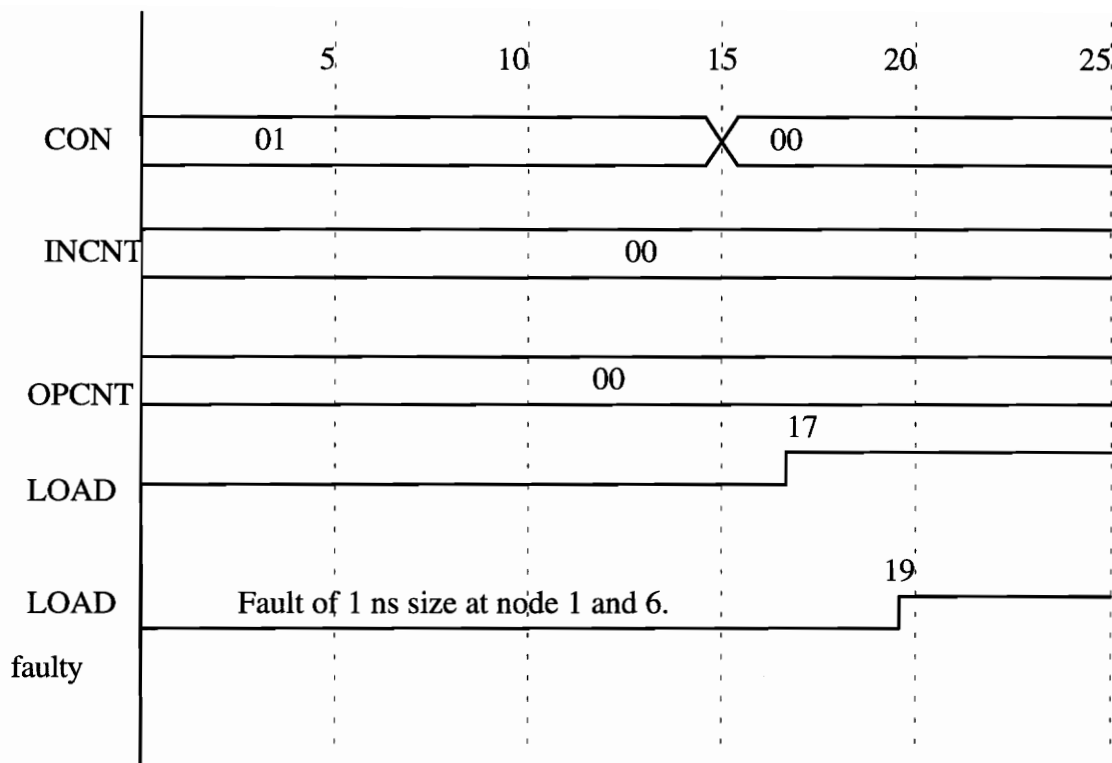


Figure 61: Path CON-1-6-LOAD

Path INCNT-7-OPCNT

Consider another path INCNT-7-OPCNT. The delayed after fault can exist at node 7.

Sensitization: The input INCNT can be changed from an initial value V_i "00" to a final value V_f "01" to sensitize the fault at node 7.

Propagation: To propagate this, CON should be maintained at value "00". All the inputs are assigned values and thus no justification is necessary. Thus following test sequence can be constructed:

	CON	INCNT	OPCNT	OPCNT (faulty)
t=0	00	00	00	00
t=t1	00	01	00	00
t=t1 + max_del	00	01	01	00

The waveforms for the faulty and fault free case are shown in Figure 62.

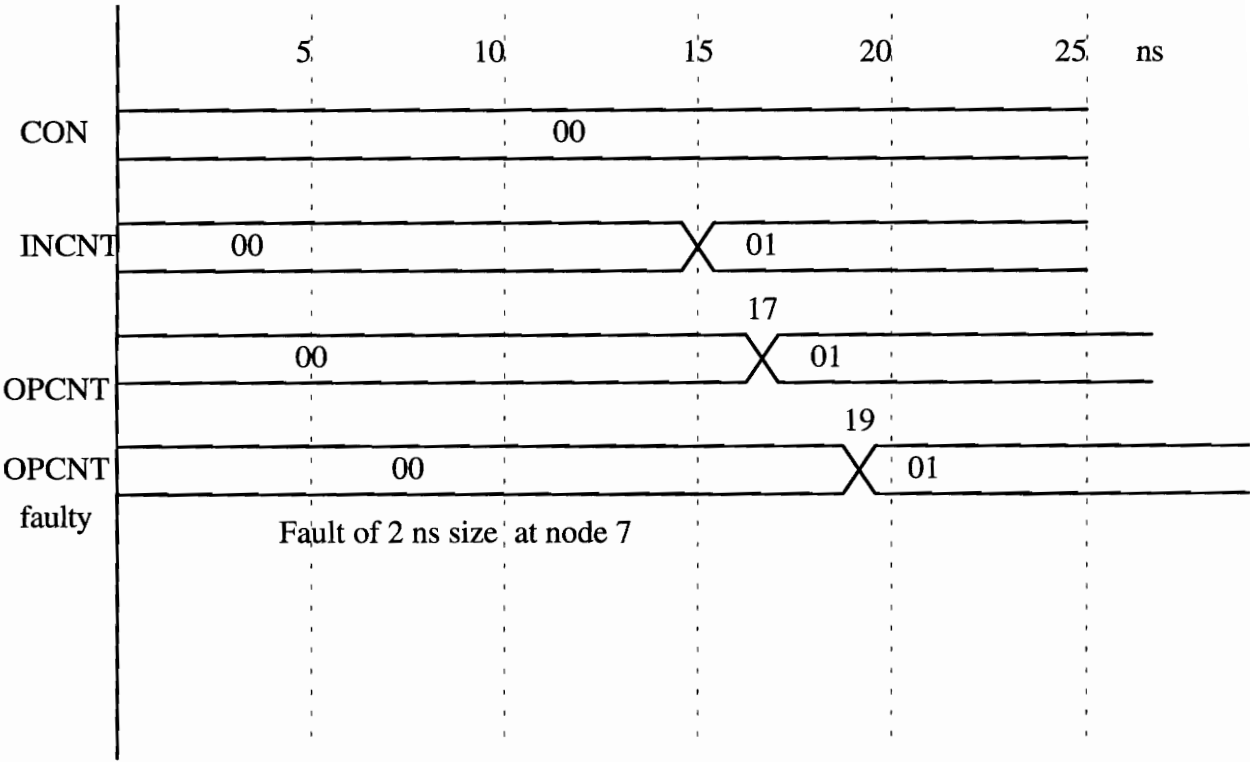


Figure 62: Path INCNT-7-OPCNT

EXAMPLE 2:

This example illustrates a test generation for a sequential element.

Consider a 2 bit data buffer. The data is latched on the rising edge of the signal STRB, and is available at the output when DS1='1' and NDS2='0'. The VHDL model for this behavior is shown below.

entity BUFF is

port(DI: in BIT_VECTOR(0 to 1); STRB, DS1, NDS2: in BIT; DO: out BIT_VECTOR(0 to 1));

end BUFF ;

architecture BUFF_ARCH of BUFF is

begin

A: block (STRB='1' and not STRB'stable)

signal REG: BIT_VECTOR(0 to 1);

signal ENBL: BIT;

begin

REG<= guarded DI after 3 ns;

ENBL <= DS1 and not NDS2 after 2 ns;

DO<= REG after 3 ns when ENBL='1'

else "11" after 2 ns;

end block A;

end BUFF_ARCH;

The DFG is shown in Figure 63. Node 2 is a dummy node and is not shown in the code. This node may represent the delayed attribute fault (STRB='1' and STRB'stable changed to STR'delayed(T)='1' and STRB'delayed(T)'stable) which maps to a clock skew.

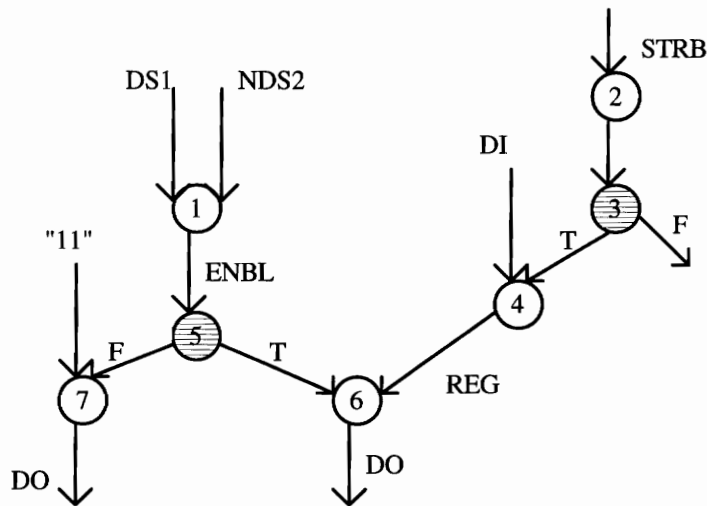


Figure 63: DFG of a buffer

Path DS1-1-5-6-DO

Consider the path DS1-1-5-6-DO. The delay fault can exist at node 1 or 6. Following steps can be taken:

The fault at node 1 can be sensitized by changing DS1 from an initial value '0' to a final value '1'. The other input to node 1, NDS2 should be maintained at '0'. It is desired that the same transition should also sensitize the fault at node 6.

This model represents the behavior of a sequential element and we have to store proper value in the signal REG to sensitize the fault at node 6.

For this purpose, during the first time frame (at $t=0$), any value other than "11" should be assigned to the signal REG. Thus if $DI="00"$ and STRB is changed from '0' to '1' (at $t=0$), it is possible to load the value "00" in REG. This assignment to the signal REG will sensitize the fault at node 6.

Thus during first time frame ($t=0$) following assignments are necessary:

$DS1='0'$, $NDS2='0'$.

$STRB= R(\text{rising})$

$DI= "00"$.

During second time frame, DS1 should be changed to 1. STRB can be maintained at any value, say '1'. Thus following test sequence can be constructed:

	DS1	NDS2	DI	STRB	DO	DOfaulty
$t=0$	0	0	00	R	U	U
$t= t1$	1	0	00	1	11	11
$t=t1+ \text{max_del}$	1	0	00	1	00	11

U implies unknown value.

The waveforms for faulty and fault free case are shown in Figure 64.

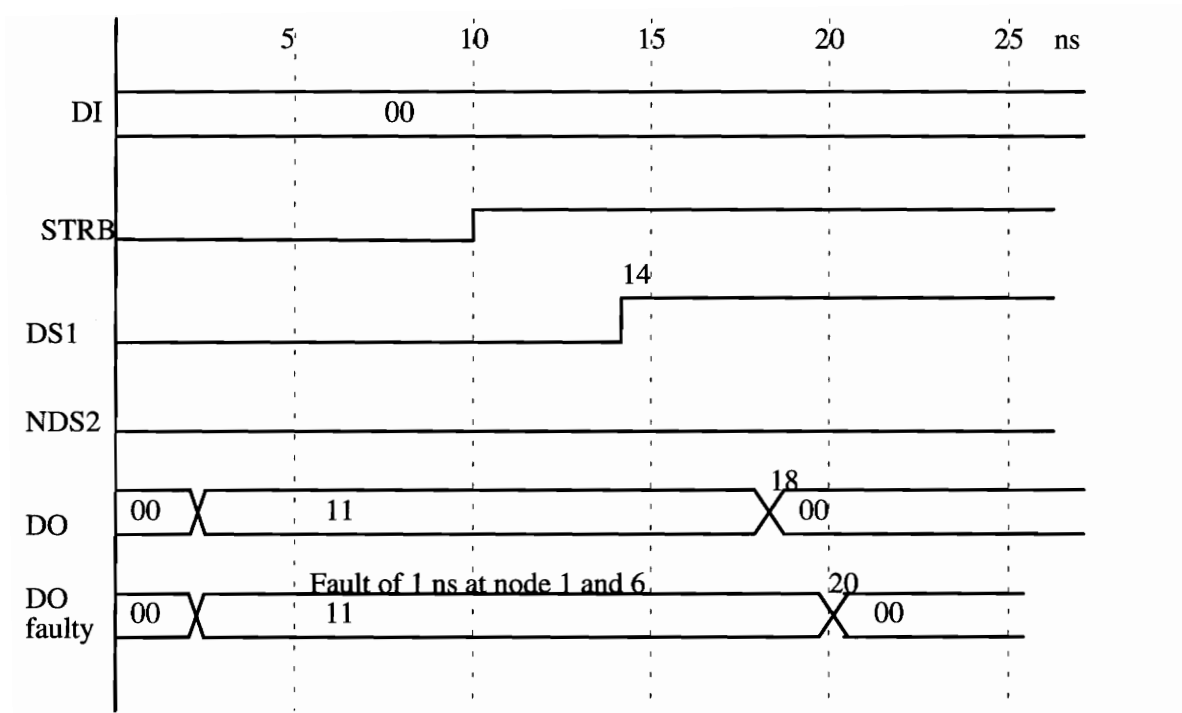


Figure 64: Path DS1-1-5-6-DO

Path DI-4-6-DO

Consider the path DI-4-6-DO. The delay fault can exist at node 4 or 6. In order to generate the test, following procedure can be followed:

Sensitization: Change DI from an initial value "00" to a final value "01". This will sensitize the fault at node 4 and node 6.

Propagation: This change can propagate to output only if DS1 = '1' and NDS2 = '0'. STRB should be changed from '0' to '1' and during first ($t=0$) and second ($t=t_1$) time frame.

Thus complete test sequence can be constructed as follows:

	DS1	NDS2	DI	STRB	DO	DOfaulty
t=0	1	0	00	R	U	U
t= t1	1	0	01	R	00	00
t=t1+ max_del	1	0	01	1	01	00

U: unknown value.

R: rising signal

The wave forms for the faulty and fault free cases are shown in Figure 65.

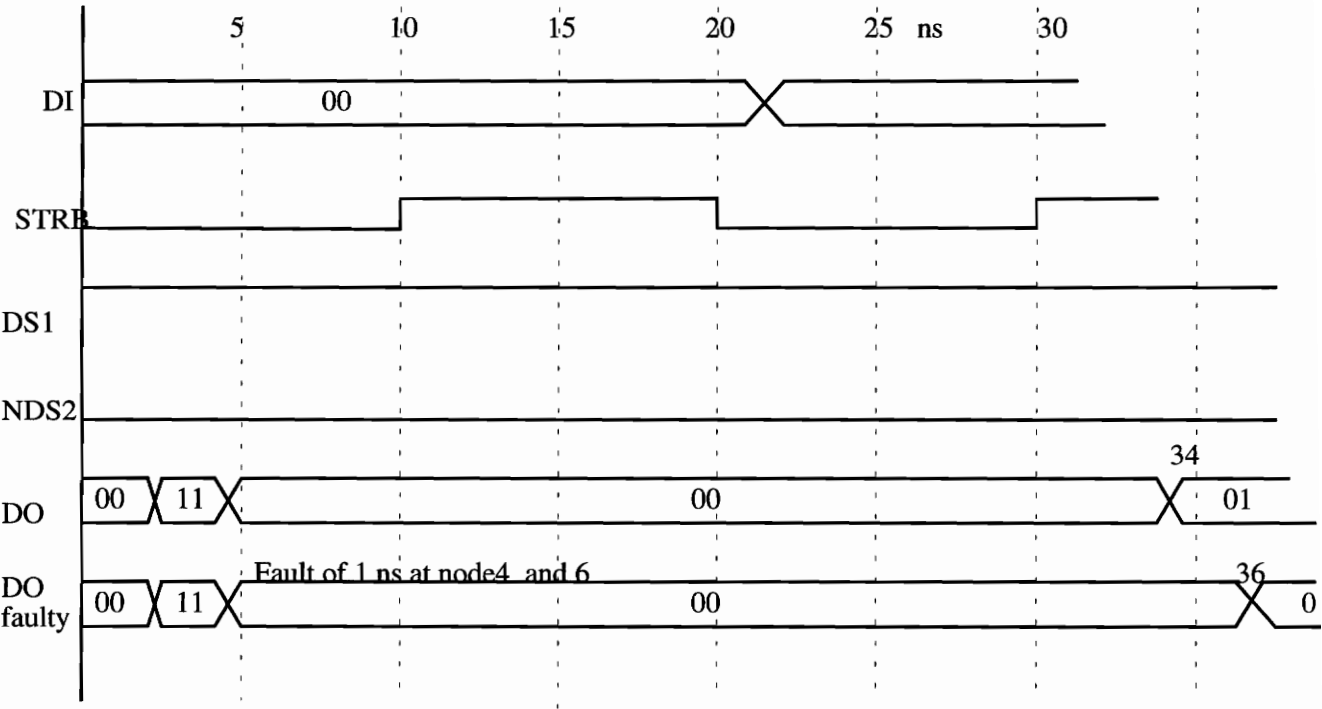


Figure 65: Path DI-4-6-O

EXAMPLE 5: FAULTS IN GENERIC

In the following example, a test for a delay fault in a generic is generated. A behavioral model for a memory unit(Fig 66) is shown below. When (EN1 and EN2) ='1', the data is temporarily latched in the buffer and when (EN3 and EN4)='1', the data is stored in the memory.

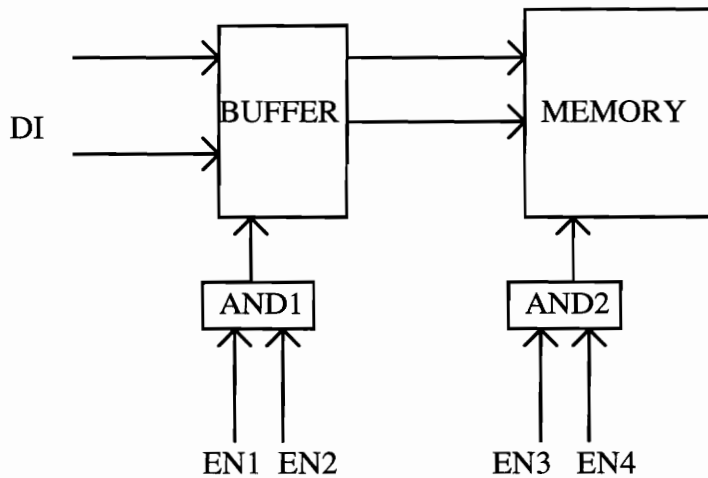


Figure 66: Fault in generic

```

entity GEN is
  generic(ADEL, TDEL, MDEL: TIME);
  port(EN1, EN2, EN3, EN4: in BIT; DI: in BIT_VECTOR(0 to 1); MO: out
    BIT_VECTOR(0 to 1));
end GEN;

architecture GEN_ARCH of GEN is
  signal TEMP: BIT_VECTOR(0 to 1);
  signal S1, S2: BIT;

```

begin

process(EN1, EN2, EN3, EN4, S1, S2, TEMP)

begin

S1<= EN1 and EN2 after ADEL;

S2<= EN3 and EN4 after ADEL;

if S1 = '1' then

TEMP<= DI after TDEL;

end if;

if S2='1' then

MO<= TEMP after MDEL;

end if;

end process;

end GEN_ARCH;

The DFG can be drawn as shown in Figure 67.

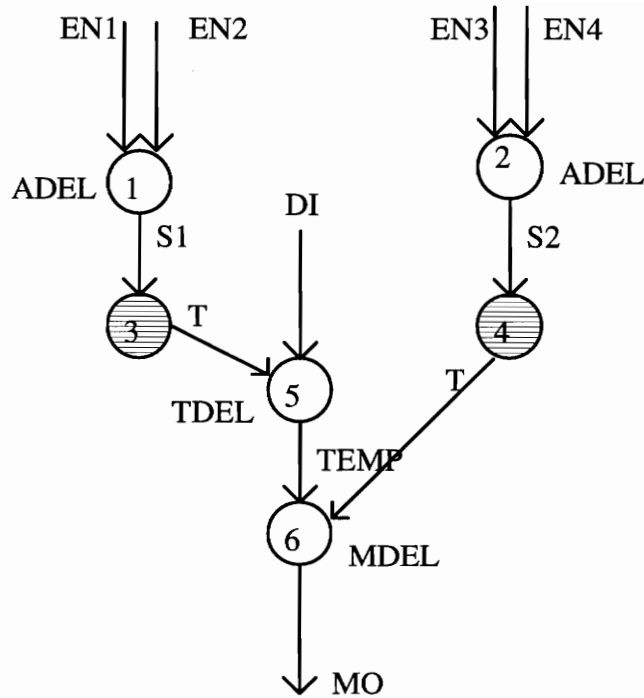


Figure 67: Fault in generics

Consider a fault in generic ADEL. This fault will alter the delay through nodes 1 and 2. Thus in order to sensitize this fault, both the statements representing node 1 and 2 should be sensitized. This is different than generating test for a behavioral path. This test will sensitize the nodes which correspond to a delay implied by a generic.

If EN1 and EN3 are changed from '0' to '1' while maintaining both EN2 and EN4 at '1', then it is possible to sensitize both the faults.

Any value can be assigned to DI, say "01".

Thus following test sequence can be constructed:

	EN1	EN2	EN3	EN4	DI	MO	MOfaulty
t=0	0	1	0	1	01	00	00
t=t1	1	1	1	1	01	00	00
t=t1 + maxdel	1	1	1	1	01	01	00

Note that this test involves multiple sensitization, i.e. both after clauses are sensitized by the test. The wave forms for the faulty and fault free case are shown in Figure 68.

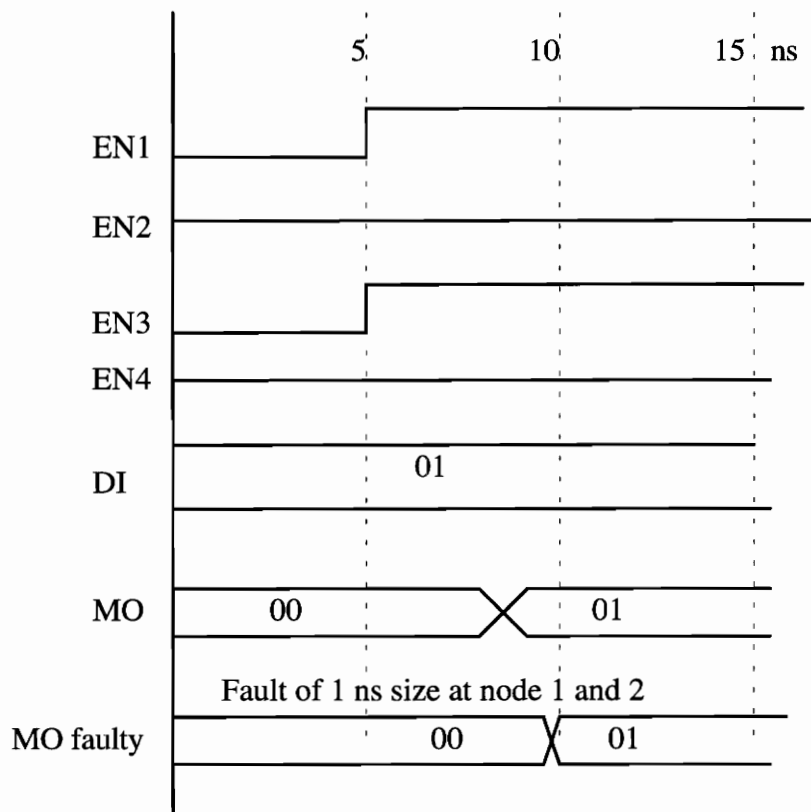


Figure 68: Fault in generics

EXAMPLE 6:

This example illustrates test generation when attribute 'last_delay' is encountered.

Consider the following D-flip-flop. The data is loaded in the flip-flop only if there is no setup or hold time fault.

```
use work.all;

entity DF is
generic (ST, HT: TIME);
port (CLK, D: in BIT; O: out BIT);
end DF;

architecture DF_ARCH of DF is
signal CLKD, DD, OD,CLKF,DF: BIT;
begin

B:process(CLK'delayed(HT), D)
begin

if CLK'delayed(HT)'event and CLK='1' then
    if (D'last_event >= ST + HT ) then
        O<= D;
```

end if;

end if;

end process B;

end DF_ARCH;

The DFG can be drawn as shown in Figure 69.

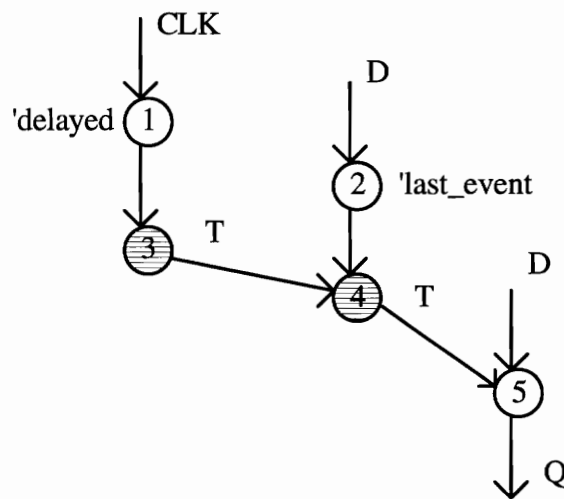


Figure 69: DFG of a DFF

Consider the path D-2-4-5-Q. The delay fault can exist at node 2 or 5. The fault at node 2 can be sensitized by changing D from '0' to '1'. The same transition will also sensitize the fault at node 5.

In order that this should propagate, at $t=0$, first clock pulse should be applied, maintaining $D=0$. This will make output '0' at time $t=t_1$. At $t = t_1$, D can be changed to

'1' and second clock pulse can be applied. This will propagate the transition along the path to the output Q. Thus following test sequence can be constructed:

	CLK	D	Q	Q faulty
t=0	R	0	U	U
t=t1	R	1	0	0
t=t1+max_del -		1	1	0

U: unknown

R: rising

The wave forms are shown in Figure 70.

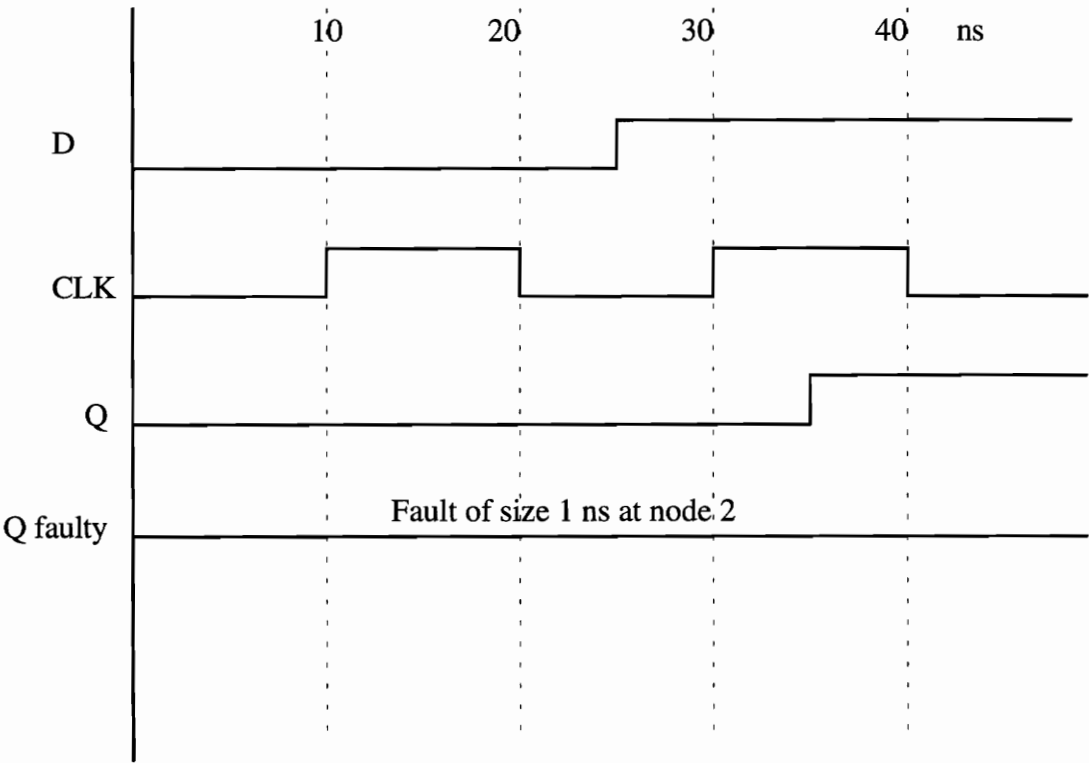


Figure 70: Fault along the path D-2-4-5-Q

6.7 Hierarchical Delay test Generation

The examples illustrated in the previous section correspond to a primitive, i.e. a VHDL model with only one process. Tests for primitives can be generated using a DFG of a behavioral model. In a multiprocess behavioral model, it is possible to construct a test sequence using hierarchy inherent in VHDL behavioral models.

To exploit hierarchy, a process within an entity can be viewed as a black box, with a set of input ports I_i and output ports O_j (Figure 71).

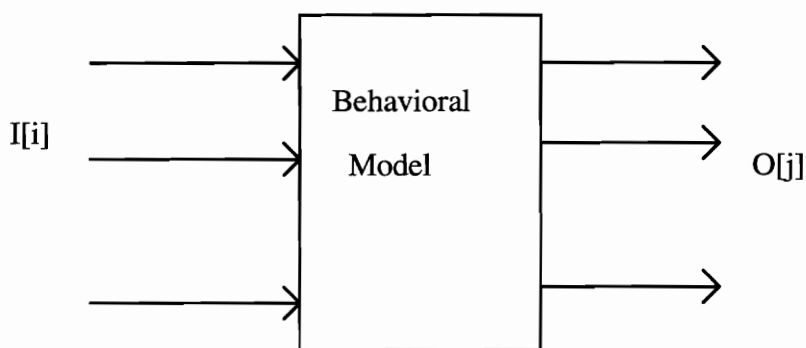


Figure 71. A process at the behavioral level

Each input line undergoes some operation in the behavioral block. The behavior of the model is defined within the process. There is a finite delay involved in the propagation of a signal from input to output. This behavioral process, as explained in chapter 5 can be represented by a DFG.

A behavioral level model may consist of many black boxes connected to perform a particular function. Figure 72 shows three different blocks, P1, P2 and P3 connected to

form a module A. This representation is equivalent to a Process Model Graph (PMG) in VHDL and each block P1, P2 or P3 represents a process.

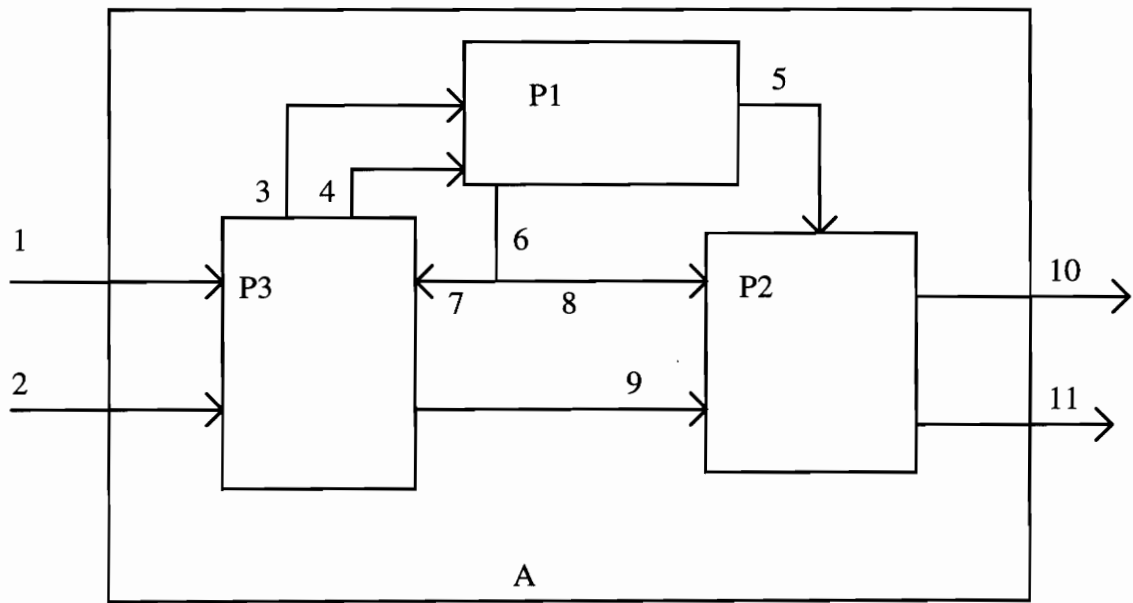


Figure 72. Process Model Graph

This can be used to exploit hierarchy in the test generation. It is possible to consider each process separately using a DFG and construct test as explained in section 6.1. If the information regarding test for each process is stored in a library then it is possible to exploit this information to generate test for a complete model.

Previous work done at Virginia Tech [41] on hierarchical test generation can be referred to get better understanding of the hierarchical test generation procedures. Exploiting hierarchy in the test generation for delay faults could be an interesting extension of the present work and is not discussed here.

6.8 Test Generation for an MCM Interconnect

For a MCM, two considerations are important: generating tests for the complete module and generating tests for interconnects. As explained in section 1.1, it may be necessary to generate a test for a MCM with the mixed level description available. (Figure 73)

The hierarchical test generation scheme proposed in the previous section can be used to generate test for an IC described behaviorally.

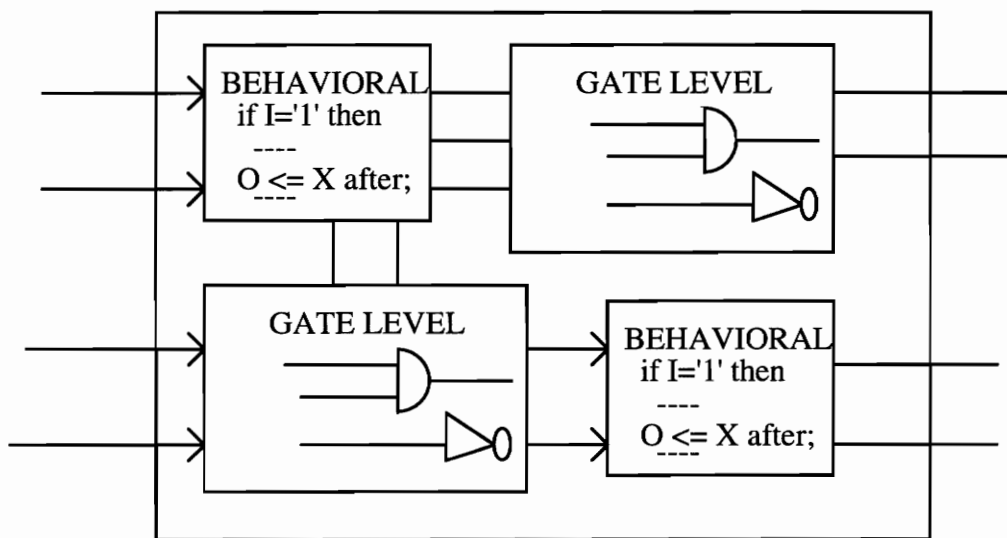


Figure 73: Mixed level description of a MCM

To generate test for an interconnect, a process model graph of the MCM can be used. Figure 74 shows three different blocks, P0, P1 and P2 connected to form a module A. At MCM level, a behavioral model can be written for each IC.

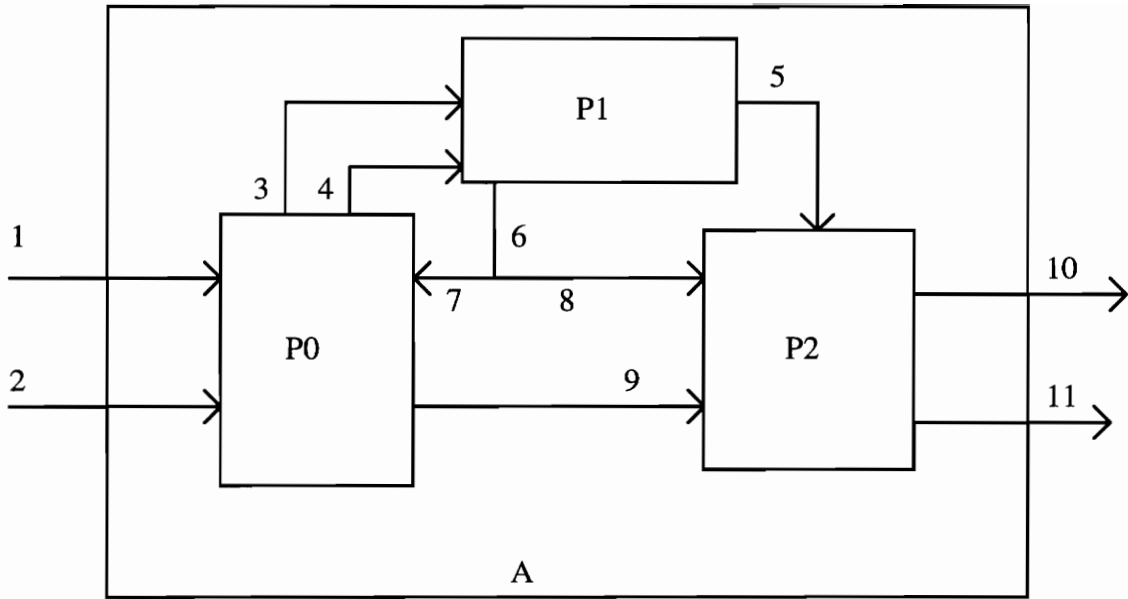


Figure 74. Process Model Graph

To test an interconnect for a delay fault, it is desired to generate a transition at each interconnect. This transition should be propagated to the nearest primary output.

In order to generate a fault list of interconnect faults, we define input and output faults.

Definition: We define input delay faults as ones that exist at the input pin of a block. The effect of these faults is propagated to other blocks only through the output of the faulty model. Thus input faults define faults at the input of the unit. For example, faults at pins 1, 2, 3, 4, 5, 7, 8, 9 represent input faults.

Definition: We define output delay faults as ones which (a) exist at the output pin of the block and (b) their effect is propagated to more than block immediately following the

faulty pin (if the pin is a fanout stem) or is not observed inside the model (if the pin is a primary output). The faults at pin 6 (fanout stem), 10 and 11 (primary outputs) can be classified as output faults. Thus the output faults represent faults at the output of the complete model or at fanout stems.

Note that at any pin only one fault can exist. For example, although pin 5 is output of block A_[1], the fault at pin 5 will be regarded as input fault since the pin is neither a fanout stem nor a primary output.

If any of the block A_[i] considered separately, the input and output faults will be different (Figure 75). For example, for block A_[0], pins 1, 2 and 7 define input faults, whereas pins 3, 4 and 9 define output faults.

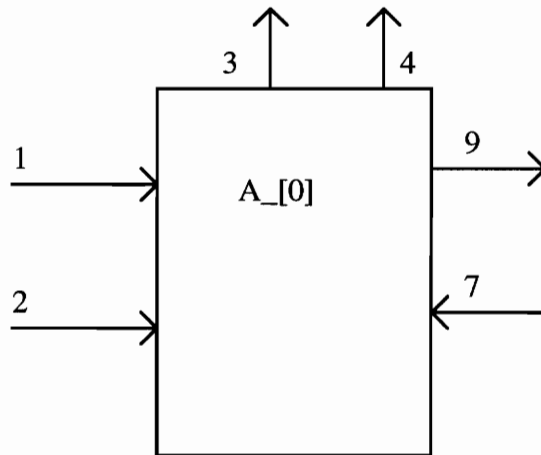


Figure 75. Delay faults within a process

Input and output faults can be used to generate a fault list. An input fault itself (if there are no branches along that line) or an output-input fault pair (if the line is a fanout stem)

can represent an interconnect. A possible methodology to generate a test for an interconnect can be written as follows:

1. Generate an interconnect fault list using the strategy mentioned above.
2. Select an interconnect not yet tested. Generate a transition at that point to sensitize the fault.
3. Propagate the value to a closest primary output.
4. Justify the assigned values backwards until primary inputs are reached.
5. If all the faults are not tested, go to step 1.

Chapter 7

CONCLUSIONS AND FUTURE RESEARCH

In the presented work the following contributions were made:

1. Defined a new delay fault model at the behavioral level of abstraction. A hardware description language, VHDL, was used for this purpose. Different behavioral timing constructs in VHDL were perturbed to define delay faults.
2. Mapped these faults to equivalent physical failures in actual circuit layout. Several examples were given to support claims.
3. Defined a behavioral level io path in a behavioral model using a Data Flow Graph of the behavioral model. Proved that this path maps to an equivalent path in a gate level equivalent circuit representing the same behavior.
4. Developed a systematic strategy to generate a path list of a data flow style behavioral model.

5. Showed how the model can be used in behavioral delay test generation for digital circuits. Proposed a possible test generation scheme. By synthesizing gate level circuit from behavioral description, it was shown that the behavioral delay tests can test a path at the gate level.

6. Showed how the model can be used for the interconnect test generation of a multi chip module.

One of the most important extensions to the present work would be in the area of delay test generation using the behavioral fault model. The first step in this direction involves study of graphical representation of the behavioral model using a DFG. As explained in section 5.1, this representation can give necessary information to generate a delay test.

The second step would involve a systematic implementation of the test generation algorithm for a primitive. This will involve (a) generation of a list of all behavioral level io paths in the graphical representation, (b) sensitization of the path and (c) propagation and justification of the assigned values.

Next step would involve test generation for a model using the process model graph. The Modeler's Assistant tool developed at Virginia Tech can be used for this purpose. An approach similar to [16] can be taken to generate tests. In [16], test generation involves assigning a value to all primary inputs to generate a test. The test generation in this case would be different since it is desired to propagate a transition along the path to a primary output and thus involves assigning two values (initial and final) to the primary inputs. Multivalued logic can be used to simplify the test generation problem.

Automation of the complete procedure outlined above would take care of the test generation process of a model. CAD tool CLSI VTIP, which converts the VHDL description into equivalent C data structures can be used to parse VHDL and generate equivalent graphical representation.

It will be necessary to apply tests developed from the behavioral level model to a gate level equivalent circuit and check the delay fault coverage. This will show effectiveness of the test to test physical failures. It is mentioned in [13] that it is not necessary to test every path in the gate level circuit, and the test developed for a certain number of faults should be able to give 100% delay fault coverage. This raises an important question: As a pure mathematical exercise, can it be proved that the paths dictated by a behavioral model are the only paths to be tested in a gate level equivalent circuit? Here it can be argued that the effect of the change at the input can propagate to output only through these paths (as defined by the behavioral model). Can this mean that these paths are more important than other paths ?

The testing for delay faults in the microprocessor is another area where the behavioral level delay fault model can be applied. It is necessary to further study timing diagrams in a microprocessor to show that it is important to test delay faults in a microprocessor. The test generation for delay faults in a microprocessor would be another challenging problem.

In summary, the defined behavioral level delay fault model has many applications and the present research can be extended in different directions.

REFERENCES

1. Linda Geppart, "The New Contenders", IEEE Spectrum, December 1993
2. T. Storey, "A test methodology for VLSI chips on silicon", International Test Conference, 1993, pp 359-368
3. D.K. Pradhan (ed), "Fault Tolerent Computing: Theory and Techniques", Printice Hall, 1986
4. J. P. hayes, "Fault Modeling", IEEE Design and Test of Computers, April 1985, pp 88-95
- 5.W. Maly, "Tutorial: Realistic Fault Models for VLSI Testing", 24th Design Automation Conference, June 1987, pp173-180
- 6.Roth J. , "Diagnosis of Automata Failures,: A calculus and a Method", IBM Journal of Research and Development, Vol 10, No 4, pp 278-291, July 1966
7. Goel P., "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", IEEE Transaction on Computers, Vol C-30, No. 3, pp 215-222, March 1981
8. Fujiwara H. and Shimono T., "On the acceleration of the Test Generation Algorithm", IEEE Transaction on Computers, Vol C-32, No 12, pp 1137-1144, December 1993

9. E. P. Heish, "Delay Test Generation", 14th Design Automation Conference, 1977, pp 486-491
10. Lin C., Reddy S., "On the Delay Fault Testing in Logic Circuits", IEEE Transaction on CAD, pp 694-704, Nov, 1985
11. Li W., Reddy S., Sahni S., "On Path Selection in Combinational Logic Circuit", 25th Design Automation Conference, pp 142-148, 1988
13. W. Lam, A. Saldhana, R. Bryton, A. Sangiovanni-Vincentelli, "Delay Fault Coverage and Performance Tradeoffs", DAC 1993, pp 446-451
14. Breuer M., Friedman A., "Functional Level Primitives in Test Generation", IEEE Transaction on Computers, Vol C-29, pp 223-235, March 1980
15. O'Neill M., Jani D., Cho C., Armstrong J., "BTG: Behavioral Test Generator", CHDL 1990, pp 347-361
16. Rao S., Armstrong J., "Hierarchical Test generation for VHDL Behavioral Model", EURO ASIC, pp 214-226, 1993
17. C. Timoc et.al., "Logical Model for Physical Failures", Proc. Int'l. Test Conference, 1983, pp 546-553

18. Bannerjee P., Abraham J., "Generating Test for Physical Failures in MOS logic circuits", Proceedings of International test Conference, 1983, pp 554-559
19. T. Hayashi et. al., "A Delay Test Generator for Logic LSI", Proc. of 14th International Fault Tolerant Computing, June 1984, pp 146-149
20. Smith G., "Model for Delay Faults based on Paths" , Proceedings of International test Conference, 1985, pp342-349
21. IEEE VHDL Language Reference Manual, Std 1076, 1987
22. Armstrong J., Chip Level Modeling with VHDL, Prentice Hall, 1988
23. Perry D. , VHDL, McGraw Hill, 1991
24. Navabi Z., VHDL: Analysis and Modeling of Digital Systems, McGraw Hill, 1993
25. Lipsett R., VHDL, Hardware description and Design, Kluwer Academic, 1989
26. Coelho D., The VHDL Handbook, Kluwer Academic, 1989
27. Carlson S., Introduction to HDL-based design methodology using VHDL, Synopsys Inc., 1991
28. The VHDL System Simulator Manual, Synopsys Inc., 1990

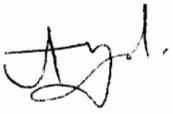
29. Kuck D., The Structure of Computers and Computations, Vol I, J. Wiley and Sons, 1978
30. Armstrong J., Gray F., Structural Level Design using VHDL, Prentice Hall, 1993
31. Thatte S., Abraham J., "Test Generation for Microprocessors", IEEE Transactions on Computers, Vol C-29, No.6, pp 429-441, June 1980
32. Thatte S., Abraham J., "User Testing of Microprocessors", 1979 IEEE test Symposium, Cherry Hill, NJ, pp 108-115, October 1979
33. Storey T., Barry J., "Delay Test Simulation", 14th DAC, pp 492-494, June 1977
34. Davidson S., "Fault Simulation at the Architectural level", International test Conference, pp 669-675, 1984
35. Malaiya Y., Narayanswami R., "Testing for Timing Faults in Synchronous Sequential Circuits", International Test Conference, pp 560-566, 1983
36. Lin C., Reddy S., "On Delay Fault Testing in Logic Circuits", IEEE Transaction on CAD, Vol CAD-6, No 5, pp 694-703, September 1987
37. Singh B, "A Parametrized CAD Tool for VHDL model development with X-windows", Master's Thesis, Virginia Tech, 1990

38. Singh B., Wicks J., Wright P., Armstrong J., "The Modeler's Assistant: A CAD Tool for Behavioral Model Development", CHDL 1993, pp 347-354
39. Hummer H., Veit H., Topfer H., "Functional tests for hardware derived from VHDL", 10th International Symposium on CHDL and their Applications, pp 433-445, April 1991
40. Broy M.(Ed), Control Flow and Data Flow: Concepts of Distributed Programming, Springer-Verlag, 1985
41. Sharp J., Data Flow Computing, Theory and COmputing, Ablex Publishing Co., 1992
42. Rao S., "Hierarchical Approach to Effective Test genaration for VHDL Behavioral Model", Master's Thesis, Virginia Polytechnic Institute and State University, 1993
43. Cho C., "An Improved Algorithm for the Test Generation of VHDL Behavioral Model", PhD. Dissertation, Virginia Polytechnic Institute and State University, Blacksburg, 1994
44. Navabi Z., "Behavioral Model of Loading Effect on a Gate", CHDL, 1991, pp 135-144

Vita

Anand Joshi was born on August 8, 1970 at Khamgaon, India. He entered Amravati University in July 1987 and passed B.E. in August 1991. He graduated with M.S. degree from Virginia Tech in May 1994.

Anand is now working for Synopsys Corporation in Long Beach, California.

A handwritten signature in black ink, appearing to read 'Anand Joshi'.