

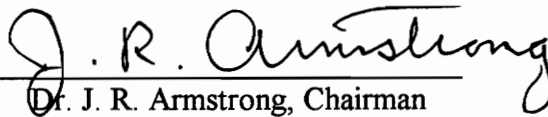
# Automatic Back Annotation of Timing into VHDL Behavioral Models

by

Gayatri P. Mahadevan

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. F. G. Gray

  
Dr. W.R. Cyre

June 1995

Blacksburg, Virginia

C.2

LD  
5655  
V855  
1995  
M343  
C.2

# **Automatic Back Annotation of Timing into VHDL Behavioral Models**

By

Gayatri Mahadevan

Dr. J.R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

This thesis presents a design system that significantly speeds up development of VHDL behavioral models with back annotated timing. The behavioral model is developed using the CAD tool called Modeler's Assistant by inputting the model in the form of a Process Model Graph. Then using the built-in primitive process library and user responses the Modeler's Assistant generates a complete VHDL source description of the model. The models developed can be classified into four classes. The first class of circuits are combinational fanout free circuits in which the fanout of each process in the Process Model Graph is one. Combinational circuits in which outputs of the processes are fed in as input to more than one process are classified as Class 2 circuits. Sequential register circuits are classified as class 3 circuits. Class 4 circuits are highly sequential circuits which have either feedback loops or irregular register or flip-flop structures. The principle for back annotating the generic delay values is discussed for the first three classes of circuits. The back annotation tool Backann2 uses the VHDL description from the Modeler's Assistant, the CLSI-VTIP CAD tool and the Synopsys Design Compiler to calculate the timing delays and to back annotate the delays into the behavioral model. The CLSI-VTIP tool is used to extract the details from the VHDL model and store it in the form of data structures. These details are used for computing the paths traversed by the signals associated with the generics. The behavioral model is synthesized into a gate level design

and the end to end delays in the model are obtained using Synopsys Design Compiler. With the end to end delays and the different paths traversed by the signals an algorithm to find realistic and accurate delays has been found. Thus a system is available to designers which builds behavioral models with accurate timing information.

*To  
Ma, Pa, Paati, Karthik,  
Anand, Prasad and Sathya  
Thanks.*

## **Acknowledgments**

I wish to express my thanks to my advisor Dr. James R. Armstrong whose guidance and encouragement was invaluable. It has been an honor and a great pleasure to study and work under him.

Also, I would like to thank Dr. F. Gail Gray and Dr. Walling R. Cyre for serving as members on my graduate committee.

I would also like to thank my family and friends for having stood behind me every step of the way. I appreciate the backing and assistance of all my friends who have been with me through thick and thin. My special thanks to Sravasti, for listening patiently to all my views while programming.

**Table of Contents**

**Chapter 1. Introduction ..... 1**

1.1 Modeling in VHDL ..... 3

1.2 Task Definition ..... 4

1.3 Contributions ..... 5

1.4 Contents ..... 6

**Chapter 2. Background and Literature Review ..... 7**

2.1 Structural Modeling ..... 7

2.2 Behavior Modeling ..... 8

2.3 Process Model Graph ..... 9

2.4 Timing Analysis in Digital Circuits ..... 13

2.5 Timing in Behavioral VHDL Models ..... 14

**Chapter 3. Algorithm Development ..... 16**

3.1 Modeler's Assistant .....	16
3.2 Path Enumeration .....	17
3.2.1 CLSI VTIP, DLS and SPI.....	20
3.2.2 Adjacency Matrix and Adjacency List representation.....	27
3.2.3 Depth First Search Algorithm.....	29
3.3 BACKANN .....	33
3.4 Synthesis of the Model using Synopsys Design Compiler.....	35
3.5 Circuits and Their Types .....	36
3.5.1 Calculation of delays .....	39
3.6 Interface with MATLAB.....	41
<b>Chapter 4. Results .....</b>	<b>46</b>
4.1 CLASS 1 Circuits .....	46
4.1.1 Majority Function Detector.....	46
4.1.2 ALARM Circuit.....	52
4.2 CLASS 2 Circuits .....	58
4.2.1 AND-OR Circuit.....	58
4.2.2 AND-OR-INVERT Circuit .....	65
4.2.3 ADDER-MUX Circuit .....	70
4.3 CLASS 3 Circuits .....	80
4.3.1 A Simple Load Unit .....	80
4.4 Summary and Interpretation of Results.....	87
<b>Chapter 5. Suggestions and Future Work.....</b>	<b>89</b>
5.1 Redundant Synthesis .....	89

5.2 Calculation of delays .....	90
5.3 Calculation of Generics for Class 4 Circuits.....	90
<b>Chapter 6. Conclusion.....</b>	<b>92</b>
<b>References.....</b>	<b>93</b>
<b>Appendix-A: User commands for backann2 .....</b>	<b>98</b>
<b>Vita.....</b>	<b>101</b>

## List of Illustrations

Figure 1 VHDL description and pictorial description of a 4x1 MUX gate.....	10
Figure 2 Process Model Graph - Functional Partitioning .....	12
Figure 3 Process Model Graph - Physical Partitioning .....	12
Figure 4 Process Model Graph and its paths in terms of generic delays.....	18
Figure 5 VHDL description for PMG in Figure 4 .....	19
Figure 6 Structure of DLS Data Definition.....	22
Figure 7 A portion of 'C' code to illustrate the use of SPI functions .....	23
Figure 8 VTIP Design Library System (DLS) .....	24
Figure 9 Algorithm for obtaining data from the DLS library .....	26
Figure 10 Algorithm for obtaining adjacency matrix and adjacency list .....	29
Figure 11 DFS Algorithm to find all paths between inputs and outputs .....	30
Figure 12 Algorithm for represnting the paths in terms of generic delays .....	31
Figure 13 The log file of user specified commands to the synthesizer .....	32
Figure 14 A part of the timing report file from Synopsys Design Compiler .....	34

Figure 15 Algorithm for extracting the delay values from the report file .....	37
Figure 16 Algorithm for calculating the delays .....	39
Figure 17 Flowchart of Backann2 .....	42
Figure 18 Process Model Graph of Majority function.....	47
Figure 19 VHDL description of MAJ3 .....	48
Figure 20 Paths obtained from input to output ports for MAJ3.....	50
Figure 21 Equations used for obtaining delays for MAJ3.....	50
Figure 22 New delay values obtained from MATLAB .....	51
Figure 23 Part of VHDL description after back annotation of generic delays.....	51
Figure 24 Process Model Graph of ALARM system.....	53
Figure 25 VHDL description of ALARM system .....	54
Figure 26 Different paths in the ALARM system.....	57
Figure 27 Input to MATLAB for ALARM system .....	57
Figure 28 The delays obtained from backann2 for ALARM system .....	58
Figure 29 Process Model Graph of AND-OR circuit .....	59
Figure 30 VHDL description of AND-OR circuit.....	60
Figure 31 The different paths in the AND-OR circuit .....	61
Figure 32 The input equations to MATLAB for AND-OR circuit.....	62
Figure 33 The outputs obtained from backann2 for AND-OR circuit.....	62
Figure 34 VHDL description of the back-annotated AND-OR circuit.....	62
Figure 35 Process Model Graph of AND-OR-INVERT circuit.....	64
Figure 36 VHDL description of AND-OR-INVERT circuit .....	65
Figure 37 Different paths in the AND-OR-INVERT circuit.....	68
Figure 38 Input to MATLAB for AND-OR-INVERT circuit .....	68
Figure 39 Output of MATLAB for AND-OR-INVERT circuit.....	69

Figure 40 VHDL description of back-annotated AND-OR-INVERT model .....	69
Figure 41 Process Model Graph of ADDER-MUX circuit.....	71
Figure 42 VHDL description of ADDER-MUX circuit .....	72
Figure 43 The different paths in the ADDER-MUX circuit.....	76
Figure 44 The input equations to MATLAB for ADDER-MUX circuit .....	76
Figure 45 Output of MATLAB for ADDER-MUX circuit.....	78
Figure 46 A part of VHDL description of back-annotated ADDER-MUX.....	48
Figure 47 Process Model Graph of SIMPLE .....	81
Figure 48 VHDL description of SIMPLE.....	82
Figure 49 The different paths in SIMPLE.....	84
Figure 50 The equations for calculation of delays for SIMPLE.....	85
Figure 51 The outputs of backann2 for SIMPLE.....	85
Figure 52 VHDL description of SIMPLE (back-annotated model) .....	86

## **List of Tables**

Table 1 Comparison of delays for MAJ3 .....	52
Table 2 Comparison of delays for ALARM system.....	58
Table 3 Comparison of delays for AND-OR circuit .....	63
Table 4 Comparison of delays for AND-OR-INVERT circuit.....	70
Table 5 Comparison of delays for ADDER-MUX circuit.....	80
Table 6 Comparison of delays for SIMPLE .....	86

## Chapter 1

### Introduction

The field of VLSI design has undergone rapid technology changes. The complexity of the circuitry that can be built on a single chip has been growing at an enormous rate. This increase in complexity has rendered chip design at the transistor level virtually impossible [1]. Also, the critical importance of the time to market is especially recognized in high technology computer products [2]. New products, from global positioning system-based computers for navigation to multimedia educational environments, demand that microelectronics system designers find new approaches to the concept-to-deliverable system design cycle. In turn, designers require ever-improving capabilities from their computer-aided design (CAD) tools, which frequently require new developments in theory. Advances in theory facilitate such needs as a provably correct design methodology and formal languages for specifications.

The fast growth of *rapid system prototyping*, which signifies the need to develop systems in significantly less time or with significantly less effort, provides the context for the driving problem in the design community in the present and for years to come [2].

Development of systems in significantly less time needs a high level language with hardware constructs, which can be easily mapped onto readily available commercial gate arrays. The significant advantage in using this design flow is: it is faster and hence costs less in terms of time. ASIC designs are now routinely used for microelectronic systems, and entire systems are synthesized from models in hardware description languages (HDLs) such as VHDL.

HDLs allow designers to create models of chips at various levels in the design hierarchy ranging from the system level to the switch level. HDLs can model control flow, data flow and timing relationships between the various blocks while reducing the quantity and detail of information that a designer needs to manage [3].

The *VHSIC Hardware Description Language* (VHDL) [4] is becoming very popular in the electronic industry as a modeling tool. It has a large set of constructs for describing circuit behavior and the circuits can be designed in top-down or bottom-up fashion through varying levels of abstraction. It can be used by the designer to describe the chip at the behavioral level, which is at a much higher level in the design hierarchy than the transistor level and thus makes design quicker and easier. These models can, and should be created so that they accurately represent the timing and functionality of the design. In order to study timing relationships between modules, it is important to have accurate delay information for high level models of those modules. These delays should be taken into account while simulating the design at any level of abstraction.

The creation of accurate VHDL models for any design was one of the primary objectives for developing this software tool, backann2, that back annotates realistic timing

delays into the VHDL behavioral models. This work describes the design implementation of backann2.

## **1.1 Modeling in VHDL**

The behavioral VHDL model created by a designer is simulated and tested for correct results. This VHDL representation is then synthesized and verified again before being transferred to silicon [5]. In a VHDL model the interface to and from the design is defined in the entity declaration. This specifies the name of the entity, the port clause and generic clause. In the behavioral VHDL model, the architecture body specifies the actual behavior of the system in terms of inputs and outputs and the relationship between them. This is made up of procedures, functions and one or more processes which have signal assignments. These signal assignment statements change the value of the signal at an instant in the future. This instant can be specified by appending a delay to the signal assignment in terms of a number, a generic, or an equation [6,7]. Generics are constants that are declared in the entity declaration or in the component declaration [4], and can be assigned values through component instantiations and configuration specification. Usually, delays for signal assignments are specified in terms of generics and are of type TIME. With this modeling style, the chip can be implemented using different technologies and the values of these generics can be changed to reflect the change in technologies without changing the functionality of the model. When the design is simulated these generics are assigned values by the designer, depending upon the technology of the chip.

## 1.2 Task Definition

A simulation that performs an accurate timing analysis can give the designer the leeway to test the timing tolerance of the design by simulating it for different conditions [9]. With early HDL's the behavioral models were simulated to verify the behavior and timing was not considered. From the discussion above, it is seen that the accuracy of the simulation of a VHDL behavioral model created using generics depends upon the accuracy of the values specified for the generics. The design of a complex system becomes a time consuming and labor intensive task if the designer has to calculate these delays manually. It is even more difficult to calculate these values without a physical representation of the chip. Also, the actual delay values in the chip may vary from what the designer calculated and incorporated for the model's simulation. This might lead to race problems and timing hazards [10,11] at a later stage in the design process. With the synthesis tools available now the behavioral model can be synthesized into a gate level circuit and the delays can be obtained.

In a behavioral model that is comprised of processes, each processes is associated with delays. Generics are used to represent these delays associated with each process. Thus the problem is to supply the model with accurate generic values for it to simulate correctly [12,13]. This is one of the obstacles facing every designer.

### 1.3 Contributions

The primary objective of developing this software tool, backann2, was to calculate accurate generic delays associated with each process and back-annotate these values into the VHDL behavioral models. It has the following characteristics:

1. It was developed on a SUN SPARC station platform and written in 'C'.
2. It is interfaced with an X-window based graphical tool, the Modeler's Assistant [14,15].
3. It uses the CLSI VHDL tool integration Platform and Design Library System to extract information from the VHDL code.
4. With the information obtained from CLSI-VTIP all the paths from the input ports to the output ports are enumerated. This capability is of immense importance since it is possible to verify the delays along all the paths which help in calculating accurate generic values.
5. The software also uses another software routine backann [16]. The routine backann calculates the individual process delays, using extract [17] developed for the hierarchical behavioral test generator which interfaces with Modeler's Assistant.
6. The Synopsys Design Compiler is used to realize the behavioral model into a gate level circuit and obtain timing details from it.

Complex data structures were defined to store the information extracted from VHDL source file (signal declarations, generic values, input and output ports) and delay values obtained from the gate level design created by Synopsys Design Compiler. The software program consists of 2500 lines of Sun-C code developed independently and about 4500 lines of modified Sun-C code from the original backann routine. The software is well documented and commented to facilitate further development.

## 1.4 Contents

Chapter 2 discusses the background and literature review behind the development of backann2.

Chapter 3 discusses and overviews the interfaces used by backann2 and the development of the algorithm. It also provides a detailed description of the path enumeration algorithm used in obtaining the exact generic delays.

The results obtained by applying the software to different classes of VHDL models is presented in Chapter 4.

The future developments possible and certain inadequacies are discussed in Chapter 5.

The Conclusions drawn from the work done are discussed in Chapter 6.

## **Chapter 2**

### **Background and Literature Review**

One of the most widely used hardware description languages is the VHSIC Hardware Description Language (VHDL) [3]. VHDL was standardized by IEEE in 1987 as its 1076 standard [4]. The growing popularity of VHDL is because of its powerful sets of constructs for modeling. Work with VHDL has successfully been done in the areas of behavioral modeling, structural level modeling and modeling for logic synthesis. The VHDL models are generally defined in two domains in the abstraction hierarchy: the structural domain and the behavioral domain. In the structural domain the model is represented as an interconnection of lower level primitives. In the behavioral domain, the model is described by defining its input and output response [3].

#### **2.1 Structural Modeling**

A structural form of the design hierarchy implies a design decomposition process [3]. This is because, at any chosen level, the system model is composed of an

interconnection of the primitives defined for that level. These primitives are frequently defined with more primitives in the next lower level of the hierarchy. At the lowest level of this structure is the primitive that has been specified in terms of its behavior. Thus, a structural design can be compared to a tree, with the different levels of the tree corresponding to the different levels of the hierarchy, the leaves of the tree (lowest level) being equivalent to the behavior of the lowest-level design components.

## **2.2 Behavior Modeling**

The behavioral model differs from the structural model in the manner in which the architectural body is represented. In this domain the component is described by defining its input/output response. It is not structural and the responses are defined in the form of procedures and functions. The behavioral descriptions of systems are frequently divided into two types: algorithmic and data flow.

**Algorithmic:** A behavioral description in which the procedure defining the input/output response is not meant to imply any particular physical implementation [3].

**Data flow:** A behavioral description in which the data dependencies in the description match those in a real implementation [3].

When the designer creates a behavioral model of the system, the inputs and outputs to the system, called ports [4] are defined along with the generics in the entity declaration. The architectural body for a behavioral model consists of one or more

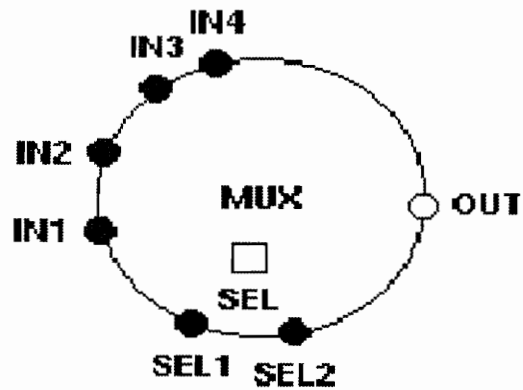
processes that run concurrently. These processes contain signal assignments, loops and other constructs that characterize the input-output relationship of the system.

For synthesis, a behavioral model is preferred when compared to structural model because, with the behavioral model, the scope of the synthesizer can be used extensively. The synthesizer is given the freedom to design the VHDL model which is optimized in terms of all the constraints given by the designer. In case of structural model the whole decomposition process is predefined and only the lowermost level can be optimized.

The VHDL description of a system is a textual representation. Development of a VHDL behavioral model using text alone is a time consuming process. The model development may be speeded up by using pictorial representations combined with text. The pictorial representations make it easier to specify the interconnections between the different components in the system. The following section will look into one such pictorial representation of the system that aids in the quick development of VHDL behavioral models [3].

## **2.3 Process Model Graph**

The architectural body in a behavioral model is used to define the behavior of the device. Each architectural body is a set of concurrently running processes. The processes are either process blocks or various forms of the signal assignment statement [4]. The architectural body can be represented pictorially as a Process Model Graph (PMG). In a PMG each node represents a process in the architectural body and arcs represent the signals passed between these process nodes [3].




---

*-- Process Name: MUX*

---

```

MUX_1: process (IN4,IN3,IN2,IN1,SEL2,SEL1)
  variable SEL: BIT_VECTOR(1 downto 0);
begin
  SEL := SEL1 & SEL0;
  case SEL is
    when "00" => OUT <= IN1 after MUX_DEL;
    when "01" => OUT <= IN2 after MUX_DEL;
    when "10" => OUT <= IN3 after MUX_DEL;
    when "11" => OUT <= IN4 after MUX_DEL;
  end process MUX_1;

```

**Figure 1 VHDL description and pictorial representation of a 4x1 MUX gate**

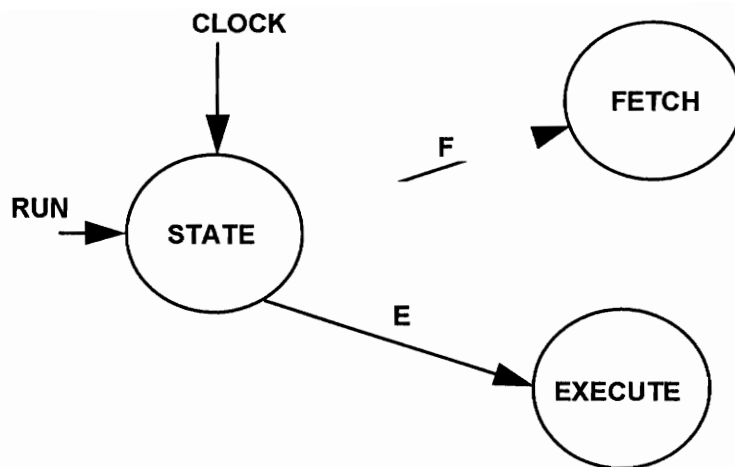
There are various ways to represent a Process Model Graph. One of the representations considers every process in the behavioral model to be a node and represents it pictorially as a circle. The signals going into and coming out of each process are represented as bubbles on the circumference of this circle. These process inputs and outputs are referred to as *ports* in the rest of this thesis. The signals that are in the

sensitivity list are shown as filled bubbles [3]. The variables declared inside each process are denoted in the form of a square. A process and its pictorial representation is shown in Figure 1.

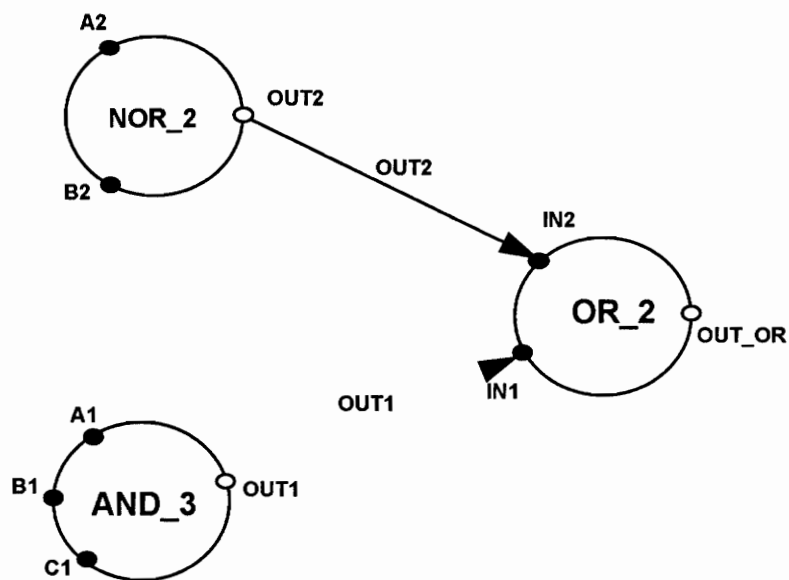
A typical Process Model Graph consists of a number of similar representations (circles) as shown in Figure 1, connected by arcs. Each process circle in the graph has a behavioral description associated with it. The arcs represent the interconnection between the processes. These interconnections represent the signals between the different processes.

The Process Model Graph represents a partitioning of the model [3]. This partitioning can either be physical or functional. Both types of partitioning are shown in Figure 2 and Figure 3. With physical partitioning, the delay values on the arcs represent the propagation delay across specific blocks of real logic gates and flip-flops. Thus the nodes may represent hardware components like multiplexers, logic gates and flip-flops. Functional partitioning, as the name suggests represents the major functions of the system. These functions do not correspond to any particular block of hardware [3]. In fact, two functional nodes may share a block of logic in some physical implementation. For example both the fetch and execute nodes in Figure 2 could access the same "real" memory.

We thus see that a Process Model Graph pictorially represents a VHDL behavioral model, clearly illustrating the various relationships between parts of the design. There are other graphical representations for behavioral models [30,31,32] but none of them is as suitable as the Process Model Graph for the task at hand.



**Figure 2 Process Model Graph- Functional partitioning**



**Figure 3 Physical partitioning in a PMG**

## 2.4 Timing Analysis in Digital Circuits

For designers, at any of the levels of hierarchy, one of the tasks facing them is timing. In the past timing analysis was performed only at the gate level. This is because the design was not modeled at higher levels. With the rising growth of HDL's it is possible to model the designs in the abstract level and perform timing analysis. It is an advantage to do timing at higher levels to verify the interaction of the different modules. Also, if timing is not considered at any level, erratic behavior of the system, resulting in hazards may occur during actual implementation of the system. Various algorithms are being investigated to overcome this problem at different levels of the design hierarchy [3].

Computation of exact circuit delays, under both bounded and unbounded delay models, has been formulated using Timed Boolean Functions by William K.C. Lam, et.al., [18]. Exact delays of combinational circuits for transition delay and delay as a sequence of vectors was also computed by proposing a general circuit delay model that unifies all delay models.

A "Path-Oriented Approach for Reducing Hazards in Asynchronous Designs" was proposed by Meng-lin Yu and P.A. Subrahmanyam [19], from a specification in the form of a signal transition graph. This approach also suggests a method to avoid hazards by adjusting the delays in the circuit.

The fundamental timing analysis problem in the verification and synthesis of interface logic circuitry is the determination of allowable time separations or skews between interface events, given timing constraints and circuit propagation delays. An

algorithm that provides tighter skew bounds and how this can be applied to synthesis tasks is discussed by Elizabeth A. Walkup and Gaetano Borriello [20].

## **2.5 Timing in Behavioral VHDL Models**

As discussed in an earlier section, the delay values for the processes in a PMG represent the propagation delay across specific blocks of gates and other components in the gate level model. These can be represented in the behavioral model in the form of generics. When simulating these behavioral models, it is imperative that these models should correspond to the equivalent gate level design in behavior. This means that the timing of the model should be identical to that of the gate level design. Some of the factors affecting the timing of gate level design are the wire load, the input delay, external delay and operating power. The behavioral model or process represented by this gate should take into account all the above factors.

A VHDL subset for high-level synthesis allowing flexible timing specification of the circuit interfaces such that the optimization potential of classical scheduling and allocation techniques can be used had been presented by A.Stoll, J.Biessenack and S. Rumler [21]. This work mainly concentrates on the description style of VHDL model, so that the algorithmic specification can be validated by a conventional VHDL simulator.

A. Gadagkar and J.R. Armstrong [13] developed an algorithm for timing distribution inside each process for a given end to end timing. The methodology detects and corrects inconsistencies in the timing specifications and allocates block delays to

meet the specifications. This has been incorporated in a tool called TIMESPEC. A weakness of this tool is that the delays calculated may not be realistic.

S. Narayanaswamy and J.R. Armstrong [16] developed an algorithm Backann to find the individual process and signal assignment delays. Backann determines the delay values that are required for the signal assignments in the behavioral model by generating the gate-level design of the model using the Synopsys Design Compiler. It extracts the values for the delays required from the gate-level design. It then back annotates these values into the VHDL behavioral model. The VHDL model is in the form of a PMG. Backann extracts each process from the PMG, converts it into a separate entity, synthesizes it, extracts the signal assignment delays and back annotates them into the VHDL model. Though it calculates the signal assignment delays for each signal accurately, it does not take into fact that when the whole model is synthesized, the design obtained will not be the same as the one when each process is synthesized individually. This is because when synthesizing a gate-level design, logic from different processes is frequently merged and optimized.

The approach used in backann can be modified and refined to calculate accurate delays for the whole synthesized models. The following chapter describes this modified approach in detail.

## **Chapter 3**

### **Algorithm Development**

Earlier work done to calculate timing delays in behavioral models and back annotating them, was not accurate for the reasons discussed in the previous chapter [16]. In this chapter we discuss a methodology for calculating accurate process delays and the various tools used to achieve the same.

#### **3.1 Modeler's Assistant**

The Modeler's Assistant provides an X-windows environment for the input of the information required for generating VHDL models. The tool uses the Process Model Graph described in Chapter 2 as the basis for the creation of VHDL behavioral models. It makes the creation of behavioral models easy and error free.

The development of a VHDL behavioral model starts with the creation of the processes required. The designer creates the processes that are not available as a part of the built-in library. This creation, is done interactively by clicking and selecting the menus

to create a process, naming it and then adding different ports to it. The designer can also add the generics, constants and variables used in the same manner the ports were added. Once this is done the behavioral description of the process is entered as text by the designer. The VHDL language was developed primarily for simulation and not all constructs of VHDL are synthesizable [22,23]. Thus the description written should be synthesizable by the Synopsys Design Compiler.

After creating the processes required, the designer use these processes (user-defined and built-in) to create the whole entity. These processes are then interconnected by adding signals that are represented by the lines between the circles in the PMG [14,15]. A Process Model Graph developed from Modeler's Assistant and its VHDL description is shown in Figure 4 and Figure 5.

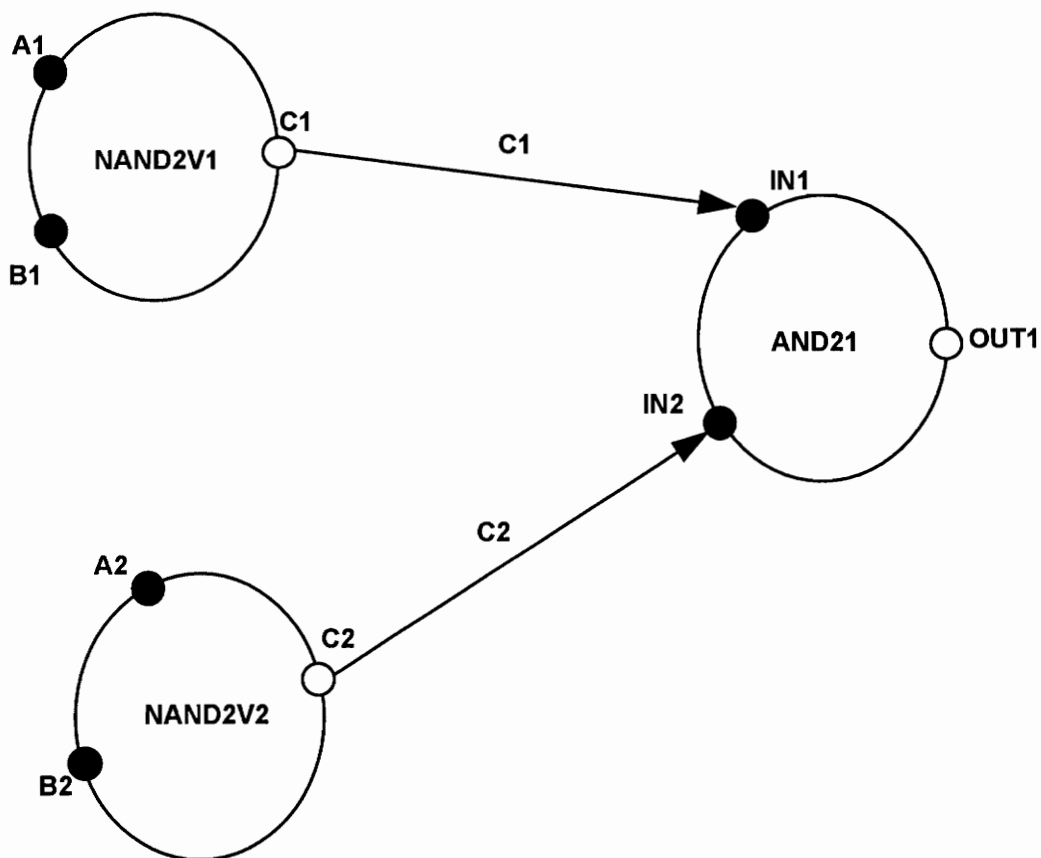
### 3.2 Path Enumeration

When assigning exact delay values to the generics that are provided as a part of the PMG description, one must consider the path traversed by the signals associated with the generics. This can be achieved by identifying all the paths from the primary inputs to the primary outputs in the PMG and enumerating them in terms of the generic delays along the paths traversed. For example, the paths from the primary inputs A1 (B1) and the inputs A2 (B2) to the primary output OUT1, for the PMG shown in Figure 4, are defined in terms of the processes generics as follows:

$$\text{PATH1} = \text{NAND2\_DEL2} + \text{AND\_DEL}$$

$$\text{PATH2} = \text{NAND2\_DEL1} + \text{AND\_DEL}$$

where NAND2\_DEL2, NAND2\_DEL1 and AND\_DEL are the generic delays associated



**PATHS FROM PRIMARY INPUTS TO PRIMARY OUTPUTS**

**PATH1 = NAND2\_DEL2 + AND\_DEL**

**PATH2 = NAND2\_DEL1 + AND\_DEL**

**Figure 4 A Process Model Graph and its paths in terms of generic delays**

```

use WORK.VHDLCAD.all,WORK.USER_TYPES.all;
-- *****

entity EG1 is
  generic (
    AND_DEL: TIME ;
    NAND2_DEL2: TIME ;
    NAND2_DEL1: TIME
  );
  port (OUT1: out BIT;
        B2: in BIT;
        A2: in BIT;
        B1: in BIT;
        A1: in BIT);
end EG1;
-- *****

architecture BEHAVIORAL of EG1 is
  signal C2: BIT;
  signal C1: BIT;
begin
  -----
  -- Process Name: AND21
  -----
  AND21_4: process (C2,C1)
  begin
    OUT1 <= C1 and C2 after AND_DEL ;
  end process AND21_4;
  -----
  -- Process Name: NAND2V2
  -----
  NAND2V2_10: process (B2,A2)
  begin
    C2 <= not(A2 and B2) after NAND2_DEL2 ;
  end process NAND2V2_10;
  -----
  -- Process Name: NAND2V1
  -----
  NAND2V1_16: process (B1,A1)
  begin
    C1 <= not(A1 and B1) after NAND2_DEL1 ;
  end process NAND2V1_16;
end BEHAVIORAL;

```

**Figure 5 VHDL Description for the PMG in Figure 4**

with processes NAND2V2, NAND2V1 and AND21 respectively.

To extract the details from the VHDL code of the model for path enumeration CLSI-VTIP CAD tool is being used.

### 3.2.1 CLSI VTIP, DLS and SPI

As mentioned above, the CLS-VTIP cad tool is used to extract the details from the VHDL model and store it in the form of data structures. In this application, the input to the CLSI VTIP is the VHDL code obtained from Modeler's Assistant. The analyzer processes the VHDL description, checks for syntactic and semantic errors and stores it in an internal format in the Design Library System (DLS) [25,26].

The DLS consists of two fundamental components - one abstract, the other concrete. The abstract component is the DLS Data Definition, which specifies the data elements, objects and structures that are supported by the DLS, together with the abstract operations that can be performed upon them. The concrete component is the Software Procedural Interface (SPI), which is a software implementation of the DLS Data Definition that supports the development of applications based on the DLS Data Definition.

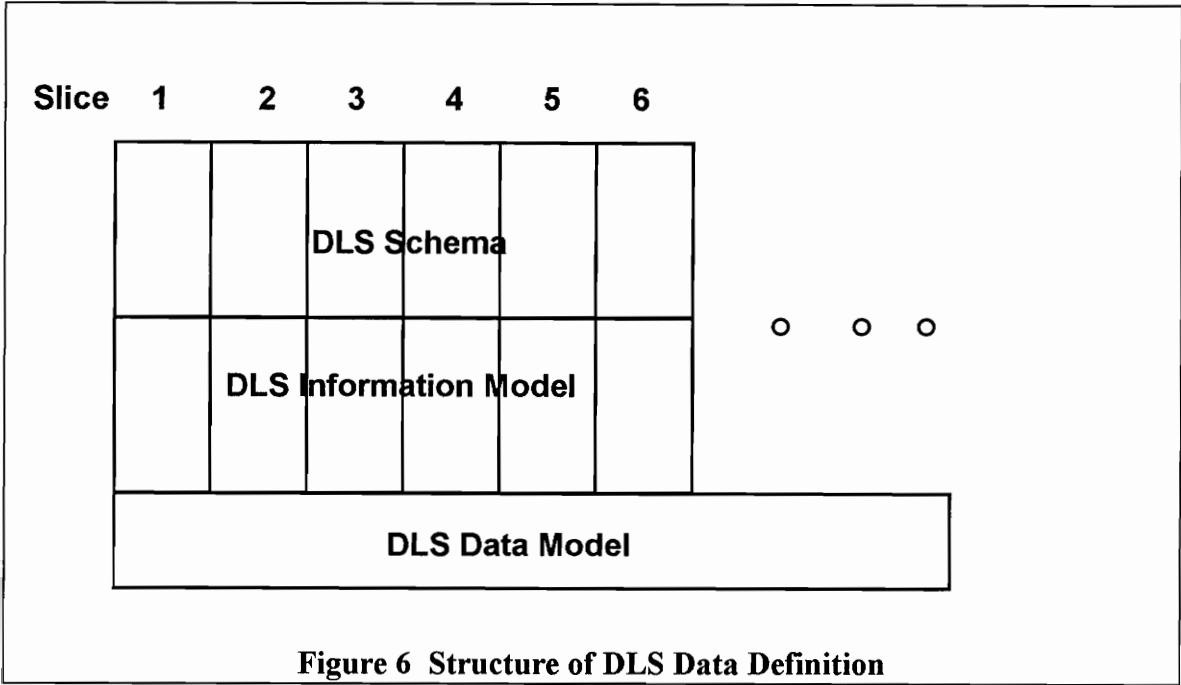
The DLS Data Definition consists of three layers: the Data Model, the Schema and the Information Model. The Data Model defines the basic data elements (e.g., BOOLEAN, INTEGER, etc.) that can be used to represent the simplest concepts in any design data (such as numbers, strings, lists, etc.). Some of these elements are *generics*\*

---

\* as defined by CLSI-VTIP CAD tool

(like NODES, ATTRIBUTES, LISTS, ITEMS). The Schema defines specific versions of such generic objects, and in doing so specifies their interpretation. For example *ObjectType* is defined as an instance of the generic type Node. Various attributes are defined for each *ObjectType* and constraints are defined for each attribute. The Schema defines four categories of *ObjectTypes*: *General ObjectTypes*, *Expression ObjectTypes*, *Miscellaneous ObjectTypes* and *Library ObjectTypes*. The top-level organization of the design data information stored by DLS consists of LIBRARIES that contain LIBRARY UNITS. A library is modeled by ObjectType *LibraryRegion*.

The Information Model defines how the specific objects can be assembled into a complex structure. It mainly describes how libraries and library units store design data information in a domain with a hierarchy of region nodes forming the backbone of the domain. This representation of the DLS can be better understood from the Figure 6 [25,26]. Each domain or view modeled within the DLS Data Definition involves a separate segment, or slice, of the Schema or the Information Model, but each slice is based upon the common Data Model [25,26]. The Information Model defines the type of each signal, variable and generic along with its initial value. For the VHDL model in Figure 4 the three process statements and the signal assignment declarations are defined in the DLS Data Model and the information in each process and details of each signal assignment is defined in either the Schema or the Information Model. The signal type (BIT) is also stored in the Information Model. The Schema contains data like which library the model is in and contains entity, architecture and procedure declarations.



**Figure 6 Structure of DLS Data Definition**

The SPI is the fundamental part of the DLS [27]. The SPI consists of data types and callable routines that implement the data types and operations of the DLS. The SPI provides layers of data types and routines to support the layers (primitive and generic data defined by the DLS model) of the DLS Data Definition. The SPI is implemented in the C programming language. These routines provide high level functions that can be implemented in terms of lower-level functions. They support addition of extra information to nodes, evaluation of expressions, symbol searches and generalized queries of DLS structures. These SPI routines are made accessible by compiling the 'C' source files with the SPI library functions. The functions used by SPI have names that are the same as the data structures used by the DLS structures (these can be viewed using *dlsbrowse*) and are easy to use. Figure 7 is a part of 'C' source code that uses some of the SPI functions. Most of the routines used to access different ObjectTypes are named as the ObjectType names themselves. The first routine that should be called before any other routine is called

is the *OpenDLS*. This initializes the Library System and the SPI. A variable "Liba" is then given a value by calling an SPI routine *NewLibrarySymbol*, with the LibNamea supplied

```

/*****OpenDLS*****/

OpenDLS();
SetReportLevel(Warning);
SetAbortLevel(Error);
strcpy(LibNamea, "../CLSI/clsi_work");

/*****Open library "LibName"*****/

Liba = NewLibrarySymbol(LibNamea);
OpenLibrary(Liba);

/*****Open the Unit *****/

Unit = OpenUnit(Liba, entity_name, arch_name, VHDLView, AugmentMode);

/** Locate the region corresponding to the design unit declaration **/

region = qRegion(qView(Unit)); /* get the library-level region(1) */
Decl = Value(LastItem(qDecls(region))); /* get its last declaration & Val */
Decl1 = Value(LastItem(qDecls(qExtends(region)))); /* get into the entity region */
region = qRegion(Decl); /* get the corresponding region(2) */
Entity_region = qRegion(Decl1); /* this is the region inside entity */

```

**Figure 7 A portion of 'C' code to illustrate the use of SPI functions**

as parameter. This function creates a Library symbol node that serves as a parameter to procedure *OpenLib*, which makes available the library units within it. An appropriate library unit is opened using the function *OpenUnit*. Using the function call *qView*, the VHDLView node is obtained and from here *qRegion* is used to get into the library-level RegionNode.

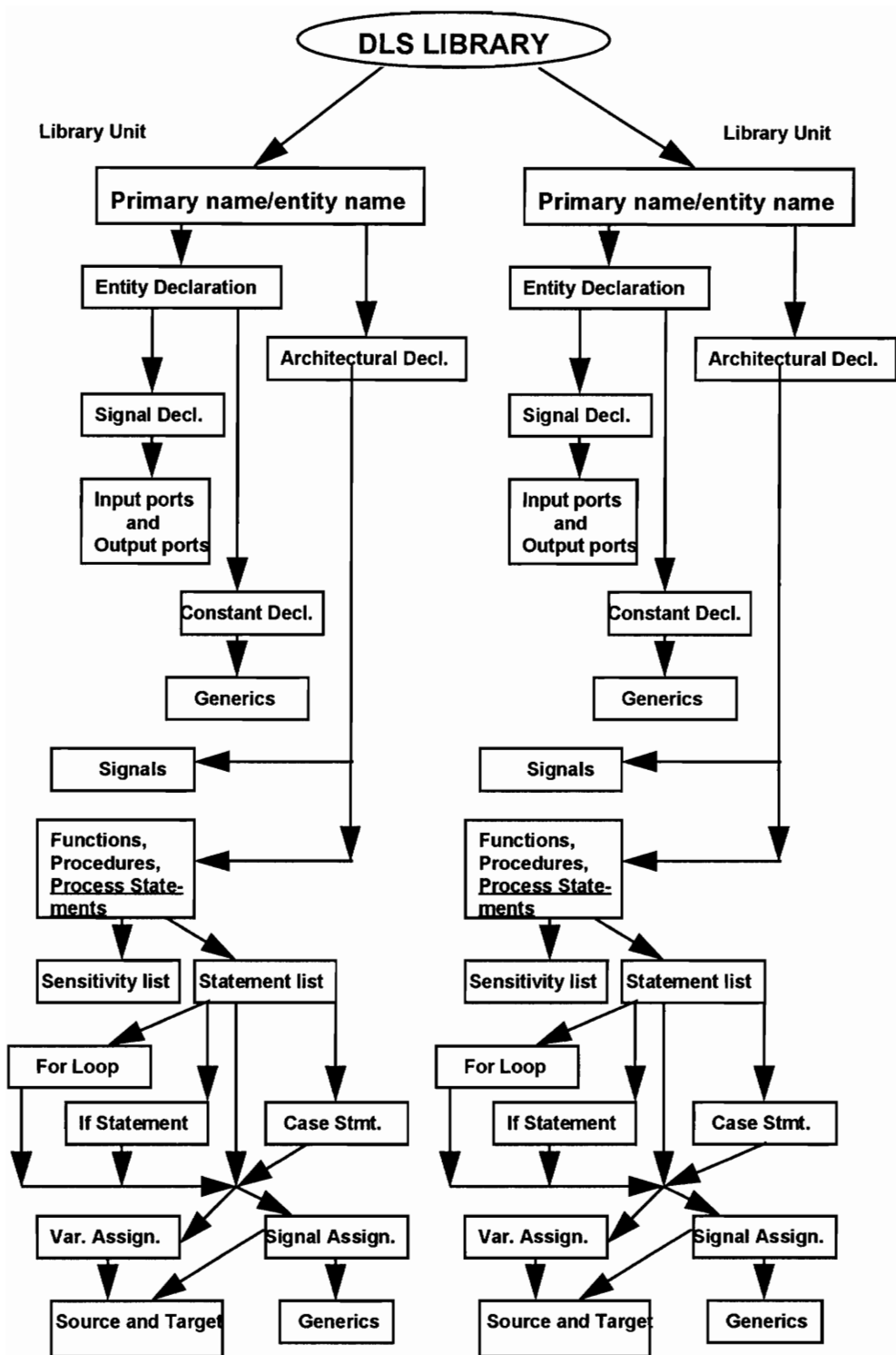


Figure 8 The VTIP Design Library System (DLS)

The DLS Browser, as mentioned above, is a screen-oriented utility that allows the user to examine the DLS library units. The Browser enables a user to open library units, traverse and examine the nodes, the lists and data structures within a library unit.

With the help of the SPI functions the paths from the VHDL model's inputs to the outputs are obtained. It is easier to visualize the VHDL code that was parsed and stored in the DLS as a tree of data structures whose top starts with a string that specifies the name of the file in which the description is stored. This is called the main unit or the library unit. The tree then moves to what is called the body where the entity and the architecture declarations are made. In the entity declaration branch the input and output ports and the generic values are all stored. In the architecture declaration branch the different processes, procedures, function declarations or statements are stored if it is a behavioral model, and component instantiations are stored in case of structural models. The branches keep on growing till all the subtypes (statements) are covered. This is shown in Figure 8 (for behavioral models).

For example, if a Statement list is considered, it is classified into different types like signal assignment, variable assignment, case statement, for loop and if loop statements. If it is a signal assignment statement it has attributes like *source* and *target* that may consist of a signal or a combination of signals, the generic associated with the signal assignment, if any, and the line number it occurred in. Similarly, if it is any other type of statement then it would branch off into a sub region that consists of statement lists, which in turn would contain a number of statements.

```

Open the DLS library unit
Access the architectural body (secondary name) of the unit
Obtain the different process statements.
For Each Process,
    Assign a unique number to it (process-number),
    To get all the inputs scan the sensitivity list of the process.
    Obtain all the inputs to the process
        for each input,
            Get the mode ( IN -> primary input,
                        INOUT-> an intermediate signal)
    To get each output in the process scan all the signal assignment
    statements.
    Get all the outputs of the process
        for each output,
            Get the mode (OUT-> primary output,
                        INOUT -> an intermediate signal)
            Get the delay associated with each output.
    Store these in the form of data structures
Close the DLS unit

```

**Figure 9 Algorithm for obtaining the data from DLS library**

SPI functions are called by the program to extract information. The information extracted using these functions consists of process names, the inputs to each process, the outputs of each process, the mode of the input and output signals and the generic delay associated with each signal assignment inside each process. Each process is given a unique number for identification. According to CLSI VTIP, a signal can be classified into one of the three modes. If a signal is an output or input to the whole model then it is of

mode OUT or IN respectively, else the signal is of mode INOUT\* [25,26]. These are stored with other details, as shown in Figure 8, in the form of data structures.

### 3.2.2 Adjacency Matrix and Adjacency List representation

With the data available from the SPI functions, the connectivity of each process is established in the form of a square matrix. The rows and columns of the matrix will be the processes that are represented by the unique number assigned. If the output of one process is connected to the input of another process then the entry [process\_output][process\_input] is assigned *one*. The connectivity details for each input signal and output signal, is obtained as described in the previous section. This square matrix thus obtained is the *adjacency matrix*. The edge from each vertex to itself is also assigned zero if there is no feedback. For example a matrix A of order 4x4 with entries,

0	1	1	0
0	0	1	0
0	0	0	1
0	0	0	0

imply there are 4 processes in the system. The output(s) of process 1 is(are) connected to input(s) of process 2 and process 3. Similarly, process 2 is connected to process 3 and process 3 to process 4.

---

\* as defined by CLSI-VTIP CAD tool

From the adjacency matrix an adjacency list for each process can be obtained. This is nothing but a linked list that stores the adjacency between each process. The adjacency list is used for easier access. For the above example the adjacency list is shown below.

for process1 : 1    -> 3   -> 2

for process2 : 2    -> 3

for process3 : 3    -> 4

for process4 : 4

The order of the adjacency list is determined when each element in each row of the matrix is scanned for a value one. When a one appears it will store it in the linked list and the algorithm goes on to the next column in the same row. If it finds a one it pushes the previous value, like a stack, and stores the recent value. Once the row is finished, it scans the other rows and stores it in the next list in the same manner.

The algorithm to obtain the adjacency matrix and hence the adjacency list is shown in Figure 10. The adjacency matrix and adjacency list for the model in Figure 4 is shown below. Numbers 1, 2 and 3 are assigned to the processes AND21, NAND2V2 and NAND2V1 respectively as their unique numbers.

Adjacency matrix :

0	0	0
1	0	0
1	0	0

Adjacency list :

for process 1 : 1

for process 2 : 2    --> 1

for process 3 : 3    --> 1

Create an  $N \times N$  matrix ( $N$  -- number of processes in the model). Make all its entries 0

For all the processes,

    Compare the outputs of each process with the inputs of every other process.

    If the two signals match (and the mode of these signals is INOUT) then the

    Output of the former process is connected to the input of the later process.

    If the above condition is true then set the matrix entry to 1.

The matrix thus formed is the Adjacency matrix.

In the adjacency matrix,

    For each row in the matrix

        for each column in that row,

            Check if the entry is 1,

            if it is 1 then push the previous node value and store the new value in the list

            else continue till all the columns are scanned.

Perform the same operation for each row and obtain the list for each row

**Figure 10 Algorithm for obtaining the adjacency matrix and the adjacency list**

### 3.2.3 Depth First Search Algorithm

The signal paths in the VHDL model are obtained by implementing the Depth First Search (DFS) Algorithm [8]. As mentioned earlier, each of the processes in the PMG is considered to be a node and the sensitivity list to each process is considered as inputs to the node. To obtain a path the algorithm visits every node in the graph, the first node and the last node being the process that contains the primary input and the process that contains the primary output respectively, and checks every edge in the graph systematically. The edges for the processes are checked by inspecting the adjacency list.

The program is written in such a way that an array is filled (mark\_val[]) as it visits every node of the graph. The array is initially set to all zeros, so that the entry for mark\_val for the i th node is a one when visited. To visit a node, we check all the edges to and from the node to see if they lead to nodes that haven't yet been visited (as indicated by zero values); if so visit them. The above mentioned recursive function, which visits all the intermediate nodes from the input to the output, is called and all paths for a particular combination of input to output is obtained. The output of each process connects to the input of another process. Once the output node is reached it traces back to the input node and obtains all the paths. The same procedure is used to trace all the paths from different inputs to

1. Obtain the start and end node (this is the process number of the nodes with primary inputs and primary outputs)
2. Create an array and initialize all its elements to zero (val\_mark[]). This array is to denote whether the node has been visited.
3. Create an array which contains the latest path ( que[]) and a variable (last) which gives the number of entries in the array.
4. If the start and end node is the same, then a path has been obtained else store the start node in a temp variable.
5. Assign the the link list to a variable of the same type.(say p)
6. If the node, in the linked list points to another node say x and the value of val\_mark[x] is equal to 0, add x to que and perform 4 to 6 again till the end node is reached.
7. If the node, in the linked list points to another node say x and the value of val\_mark[x] is equal to 1, p = linked field of p.

*Perform steps 1 to 7 for all start-end node combinations.*

**Figure 11 DFS Algorithm to find all the paths between inputs and outputs**

outputs. With the data obtained from the DLS and the adjacency list the paths are obtained using this algorithm. The algorithm is explained in Figure 11. This gives the paths in the form of node numbers. For example consider Figure 4, its adjacency matrix and adjacency list. The inputs to the model are in Process 2 and Process 3 and the output of the model is in Process 1. Using Depth First Search algorithm recursively, we obtain

2 1 and 3 1

as the paths. Each path is stored in a linked list. The paths obtained in the form of node numbers are transformed to the form of generics. The algorithm for doing this is given in Figure 12.

1. Obtain the initial node (number) and store it in  $i$
  2. Obtain the next node number from the linked list and store it in  $j$
  3. Find the processes corresponding to  $i$  and  $j$
  4. If  $i$  and  $j$  are equal then it is the final node, and so obtain the output of the process and the generic delay associated with it. To obtain the output the signal will be of mode OUT.
  5. If  $i$  is not equal to  $j$  compare the output signal of process  $i$  and input signal of process  $j$  and if they are equal and the mode is INOUT then find the generic associated with it. Store the generic delays associated with each path in a file.
  6. Now  $i = j$  and  $j = \text{next number in the path}$ .
- Perform steps 4 -6 for all nodes in the path and steps 1 to 6 for all paths

**Figure 12 Algorithm for representing the paths in terms of generic delays**

```

search_path = ". /project/rassp1/synopsys3.2b/libraries/syn
/project/rassp1/synopsys3.2b/libraries /project/rassp1/synopsys3.2b/sparc/packages "
link_library = "lsi_10k.db "
target_library = "lsi_10k.db "
symbol_library = "lsi_10k.sdb "
default_schematic_options = "-size infinite"
hdlin_source_to_gates_mode = "off"
create_schematic -size infinite -gen_database
set_operating_conditions -library "lsi_10k" "BCCOM"
set_wire_load "20x20" -library "lsi_10k"
set_max_fanout 5 " /home/mahadev/scode/EG1.db:EG1"
set_min_fault_coverage 95 -area_critical -timing_critical
set_register_type -flip_flop "FJK2SP"
derive_timing_constraints -no_max_period -min_delay -fix_hold -max_delay_scale 1.00 -
min_delay_scale 1.00 -period_scale 1.00
set_boundary_optimization " /home/mahadev/scode/EG1.db:EG1"
set_flatten true -design{" /home/mahadev/scode/EG1.db: EG1"} -effort high -minimize
none -phase true
set_structure true -design{" /home/mahadev/scode/EG1.db: EG1"}-boolean true -timing
true
link -all
compile -map_effort high -verify -verify_effort high -boundary_optimization
report_cell >> rep.out
report_area >> rep.out
report_constraints -all_violators >>rep.out
report_timing -from A1 -to OUT1 -max_paths 2 >>rep.out
report_timing -from B1 -to OUT1 -max_paths 2 >>rep.out
report_timing -from A2 -to OUT1 -max_paths 2 >>rep.out
report_timing -from B2 -to OUT1 -max_paths 2 >>rep.out

```

**Figure 13 The logfile of user specified commands to the synthesizer for Figure 4**

Thus using the CLSI VTIP tool and Depth First Search algorithm all the paths from the input ports to the output ports can be found. These paths are used for calculating the exact delays. This is explained in the sections below.

### **3.3 BACKANN**

Backann is a software tool which uses the Process Model Graph and the VHDL model generated by the Modeler's Assistant [14,15]. It synthesizes each process in the PMG individually. The gate level design is synthesized for each process and the delays across each process are calculated accurately. It extracts [17] each process, stores it as a separate entity and uses the Synopsys Design Compiler to synthesize the gate level design and calculate the end to end delays of each process, by calculating the delay values for signal assignments in the process. Backann thus calculates each process delay and then stores it back into the VHDL model as generics. The designer can specify whether a minimum delay model (minimum rise and minimum fall) or a maximum delay model (maximum rise and maximum fall) is required. Depending on the type of delay model required, (maximum or minimum) the generic delays are calculated as an average of rise time and fall time. The commands for executing the Design Compiler is obtained from a script file. The script file is obtained from a log-file given by the user. A sample log\_file is shown in Figure 13. This file contains details like the design to be read, the constraints for the model, the operating conditions, etc. The back annotated generic delays in the VHDL model are important for analyzing the timing in all the paths, which was obtained in the previous section by path enumeration.

Performing report\_timing on port 'A1'.  
 Performing report\_timing on port 'OUT1'.

\*\*\*\*\*

Report : timing  
 -path full  
 -delay max  
 -max\_paths 2

Design : EG1

Version: v3.2b

Date : Sun May 7 15:45:46 1995

\*\*\*\*\*

Operating Conditions: BCCOM Library: lsi\_10k  
 Wire Loading Model Mode: top

Design	Wire Loading Model	Library
EG1	20x20	lsi_10k

Startpoint: A1 (input port)  
 Endpoint: OUT1 (output port)  
 Path Group: default  
 Path Type: max

Point	Incr	Path
<hr/>		
input external delay	0.00	0.00 f
A1 (in)	0.00	0.00 f
U4/Z (AO2P)	0.47	0.47 r
OUT1 (out)	0.00	0.47 r
data arrival time		0.47
max_delay	0.00	0.00
output external delay	0.00	0.00
data required time		0.00
<hr/>		
data required time		0.00
data arrival time		-0.47
<hr/>		
slack (VIOLATED)		-0.47

**Figure 14 A part of the timing report from the Synopsys Design Compiler**

### 3.4 Synthesis of the Model using Synopsys Design Compiler

The Synopsys Design Compiler is invoked from the program through the UNIX Bourne shell and after synthesizing the design gives an optimized gate level circuit. The level of optimization and the constraints for the design should be given before optimizing. Since the circuit is optimized the intermediate signals cannot be identified unless the design and the reports for the design are analyzed manually. Only the input and output ports are visible to the designer. The set of commands for reading in and then optimizing the design is done by creating a script file that will perform all the functions required by the user. This file is specified in the command line to the Synopsys Design Compiler and the required tasks are performed [29].

The specifications to the designer include the design libraries to be used, the different components like specific flip-flop and latch types, maximum and minimum fanout and fanin. Constraints like area, speed, setup time, hold time can also be specified. It also includes the level of optimization and verification. These specifications are written by the designer and stored as a log file. One such log file is shown in Figure 13. Since the Design Compiler is evoked to compile each process in the design by backann, to synthesize each process, and then to compile the whole design, two log-files are required. The first log file is used to compile each process (using backann) and the second log file is used to compile the whole model. The same log file cannot be used for both because while compiling the whole design additional information for the timing report to be generated is required. To differentiate these log-files the one used to store the whole design is stored with a suffix 1 (e.g. : *EG1.log1* and *EG1.log*). The main constraint used for calculating the generics for the given model was minimum area. If registers are used in the circuit, the D flip-flop was

used. The maximum fanout of each gate in the circuit was restricted to five and Boolean optimization was used. The Design Compiler is invoked by a script file created by the software. The first line of the script file is

```
read -f <vhdl_file_name_to be synthesized>
```

which when executed would read in the VHDL file to be synthesized. To open the Design Compiler the command used is

```
dc_shell -f <script_file_name>
```

The contents of the log\_file created by the user is appended to the script file. The log\_file also includes commands to extract delays in Synopsys Design Compiler. It is done by using *report\_timing* [28]. The *report\_timing* command not only reports the delay values required but also other information. The other information includes the technology used, the wire delay model, which determines the propagation of a signal through a wire between two cells, the start and end points of the signal, type of delay (maximum or minimum), the intermediate cells through which the signal is propagated and delays through each of the cells. These are all stored in the report file *rep.out*. A part of a report\_file generated for the VHDL model shown in Figure 4 is shown in Figure 14.

The program then scans the report file and obtains the delays and stores them in a file. The algorithm for scanning the report file *rep.out* is given in Figure 15. It scans for the string *data arrival time* in the report file and obtains the time taken for the signal. The output is stored in a file *TMPREP.OUT*.

*Open the report file*

*Open a new file to store the extracted data (TMPREP.OUT)*

*If string "Startpoint: " is found in the report file and*

*If string "Endpoint: " occurs in the next line,*

*then scan the next few lines for Arrival Time.*

*If arrival time is found then copy the value along with Startpoint and*

*Endpoint into the file TMPREP.OUT*

*Continue the procedure till all arrival times are obtained.*

**Figure 15 Algorithm for extracting the delay values from the report file**

### 3.5 Circuits and Their Types

Circuits can be broadly classified as combinational and sequential. A combinational circuit is one whose outputs depend only on its current inputs. The outputs of a sequential circuit depend not only on the current inputs but also on the past sequence of inputs, possibly far back in time. A feedback loop is a signal path of a circuit that allows the output of a gate to propagate back to the input of the same or another gate; such a loop generally creates a sequential circuit behavior.

In our application which uses Process Model Graphs, the above mentioned circuits can be classified as :

Class 1 : Combinational fanout free circuits.

Class 2 : Combinational circuits with fanout.

Class 3 : Register Sequential Circuits

#### Class 4 : Highly Sequential Circuits

Fanout free combinational circuits are circuits in which the fanout of the outputs of each process is one. In graphical terms this is like a binary tree, where the root of the tree is the output and the leaves are the inputs. Circuits in which the outputs of the processes are fed in as input to more than one process are classified as Class 2. Class 3 circuits are sequential circuits that alternate blocks of combinational logic with registers. Class 4 circuits are sequential circuits which either have feedback loops or irregular register or flip-flop structures. All the above circuits can either have asymmetrical nodes (e.g. : ADDER's, MUX's) or symmetrical nodes (e.g. : AND gates, NOR gates etc.).

The principle for calculating the generic delays for the first three classes is discussed. The circuits are subdivided into these classes for better understanding of the algorithm used to obtain the delays.

##### 3.5.1 Calculation of Delays

The VHDL model illustrated in Figure 4 is a Class 1 circuit. Using the result obtained from Depth First Search Algorithm and the algorithm mentioned in Figure 12 the paths obtained from the primary inputs to the primary outputs are,

$$\text{PATH1} = \text{NAND2\_DEL2} + \text{AND\_DEL} ;$$

$$\text{PATH2} = \text{NAND2\_DEL1} + \text{AND\_DEL} ;$$

When this model is synthesized as a whole, the functional paths that are considered are PATH1 and PATH2. When the timing is reported we can find the time taken for a signal to traverse through PATH1 and PATH2.

1. Consider all the paths from input to output
2. Obtain the values of each path from the Synopsys Design Compiler
3. From backann obtain the individual generic delay.
4. If there are paths with only one generic, i.e. if it traverses only one process obtain the generic and store it. Also, overwrite the value earlier obtained for the generic.

$$\text{genericX} = \text{total delay of the path( single process delay)}$$

5. If in any of the paths, if any of the above generic value occurs, subtract the generic from the path and obtain the new path delay. This will be equal to the sum of the remaining generic delays, which are to be obtained.
6. If after performing step 5 any singular generic delay is obtained, steps 4 and 5 are repeated until all delays are exhausted.
7. If either all the singular delays are exhausted and the paths are equivalent to sum of individual generic delays or steps 4,5 and 6 were never performed, consider the path with minimum number of generics. For each generic in that path,

$$\text{genericY\_new} = \text{genericY} / \text{<new\_path\_delay>} * \text{path\_delay (1)}$$

$$\text{genericY} = \text{genericY\_new (2)}$$

where genericY in (1) is the value calculated from backann, path delay - is the sum of individual delays obtained from Synopsys Design Compiler or from step 5 and new\_path\_delay is the sum of the generics in that path obtained from backann. The path maybe either the entire path from input to output or a portion of the path if Step 5 is performed.

8. After step 7 check if any of the remaining paths has a generic just calculated. If it does perform step 5 and 6 and if necessary steps 7 and 8 till all delays are exhausted. If it does not then perform steps 7 and 8.

**Figure 16 Algorithm for calculating the delays**

When each of these processes is converted into a separate entity as in Backann, the optimization would be only for that process, but when the whole model is being optimized the gate level circuit of the design can no longer be identified as individual processes. The whole model would be optimized and redundancy in the code would also be checked for. The end to end delay will no more be the sum of individual process delays. The new end to end delay after optimizing the whole model will in general be different. It may be more or less than the original values of PATH1 or PATH2 depending on the constraints applied to it.

The motivation is to find individual process delays which would satisfy the given constraints when the model is synthesized. It is clear that when the end to end path delay changes, the logic corresponding to the processes in the original model also changes accordingly. This means that an increase or decrease in the end to end delay signifies a similar change in the original process delays. The word original is used because the gate level circuit of the VHDL model will perform the same function as the behavioral model, but the similarity ends there.

Now the problem of finding the generic delays reduces to a linear programming problem (ratio problem). The algorithm for finding the individual process delays is shown in Figure 16. Consider the above mentioned example. The new generic delays from the algorithm (step 4 and step 5) will be of the following form,

$$\text{NAND2\_DEL2\_new} = \text{NAND2\_DEL2}/\text{PATH1} * < \text{path delay from design comp} > \dots (1)$$

$$\text{NAND2\_DEL2} = \text{NAND2\_DEL2\_new} \dots (2)$$

$$\text{AND\_DEL\_new} = \text{AND\_DEL}/\text{PATH1} * < \text{path delay from design comp} > \dots (3)$$

$$\text{AND\_DEL} = \text{AND\_DEL\_new} \dots (5)$$

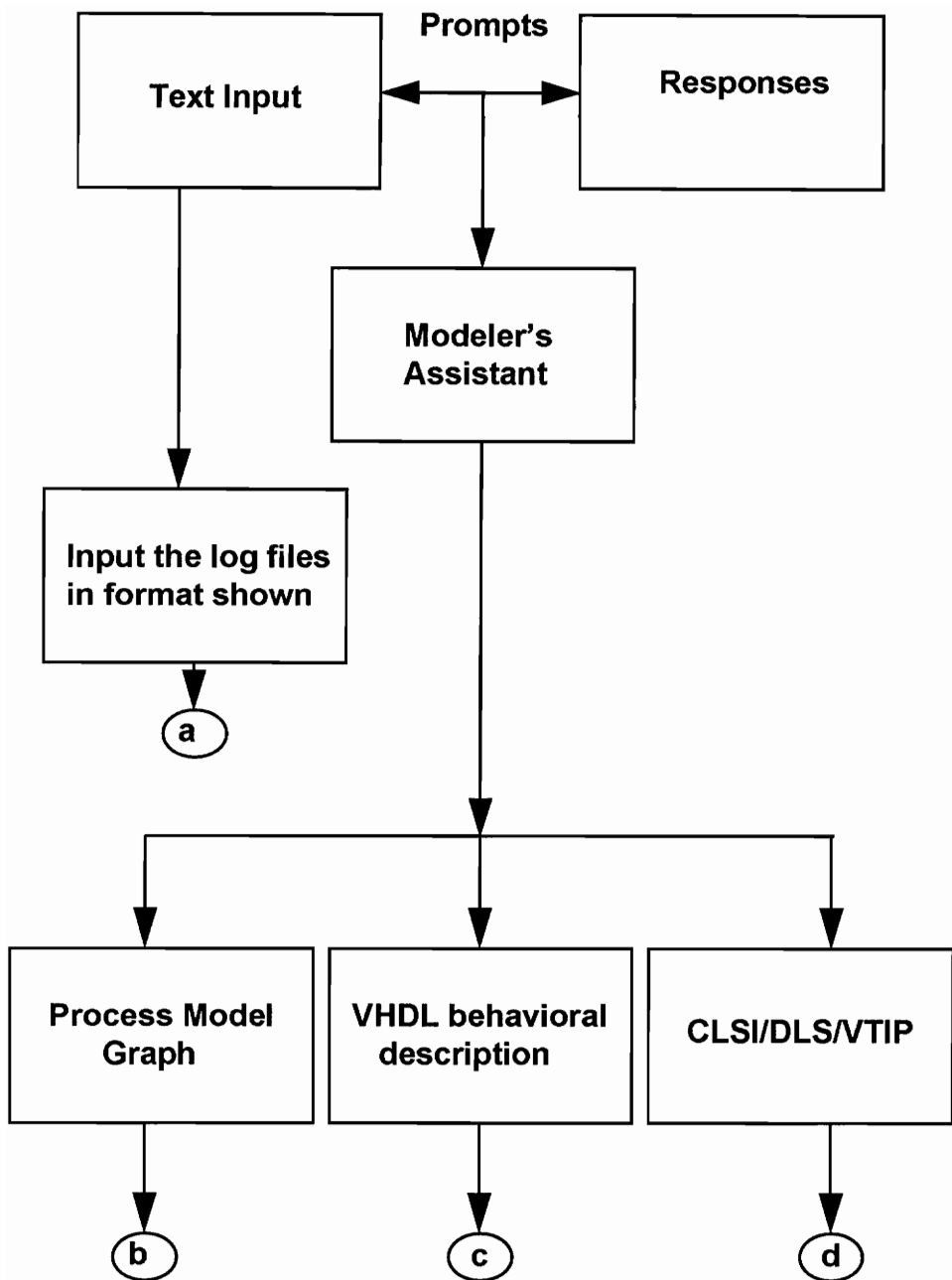
$$\text{NAND2\_DEL1\_new\_2} = \langle \text{path delay from design comp} \rangle - \text{AND\_DEL}....(6)$$

The values for PATH1, NAND2\_DEL2 and AND\_DEL in equations 1 and 3 are assigned the values obtained in backann.

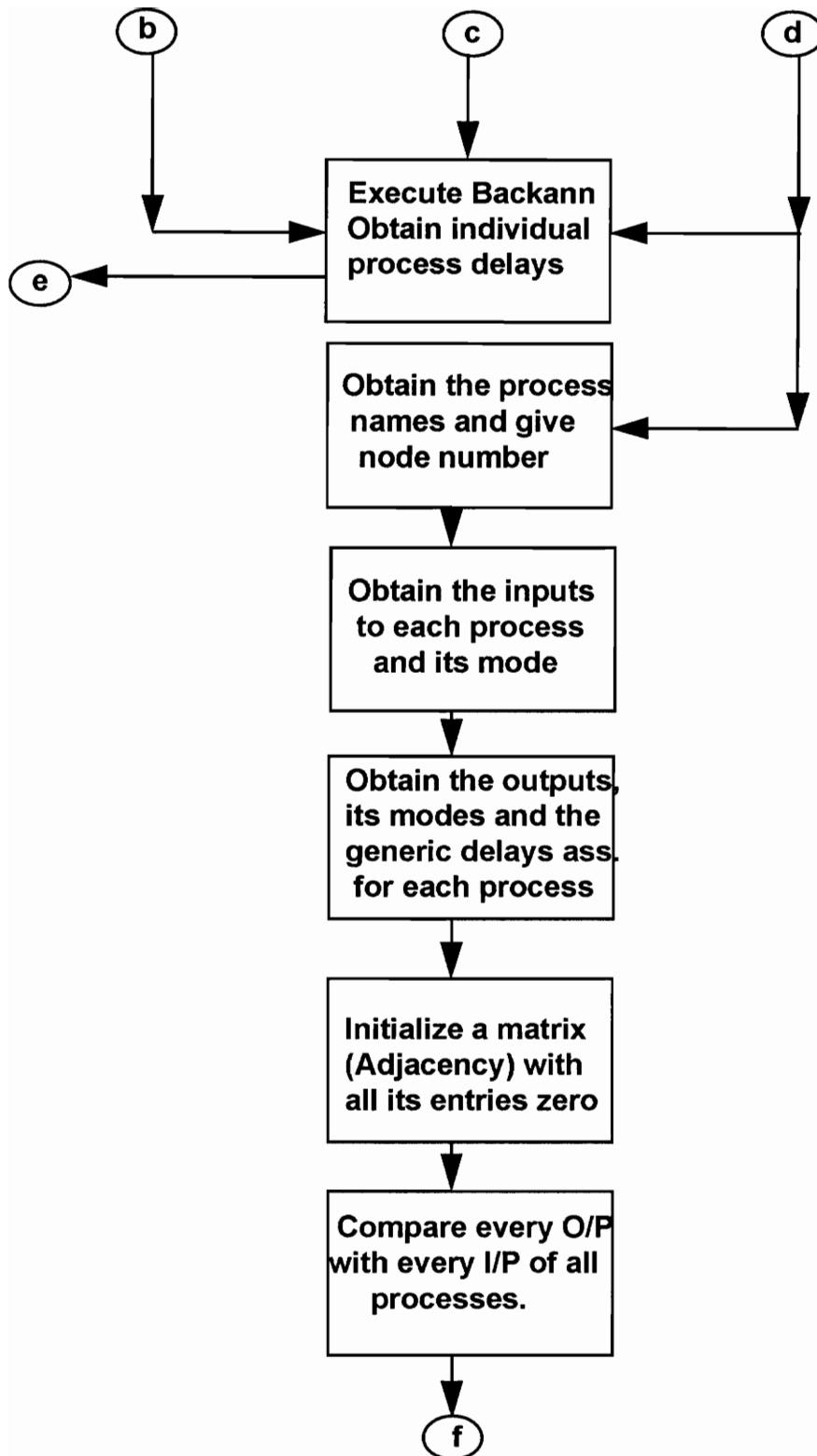
### 3.6 Interface with MATLAB

The new values were obtained using MATLAB which is a complete, self-contained environment for programming and working with data and which often interacts with data and programs external to it. The results obtained from the previous sections are stored in a single file. MATLAB is invoked using system calls in 'C' and the output is stored in a file. The file is then scanned and the delay values are obtained. These values are then back annotated into the VHDL model.

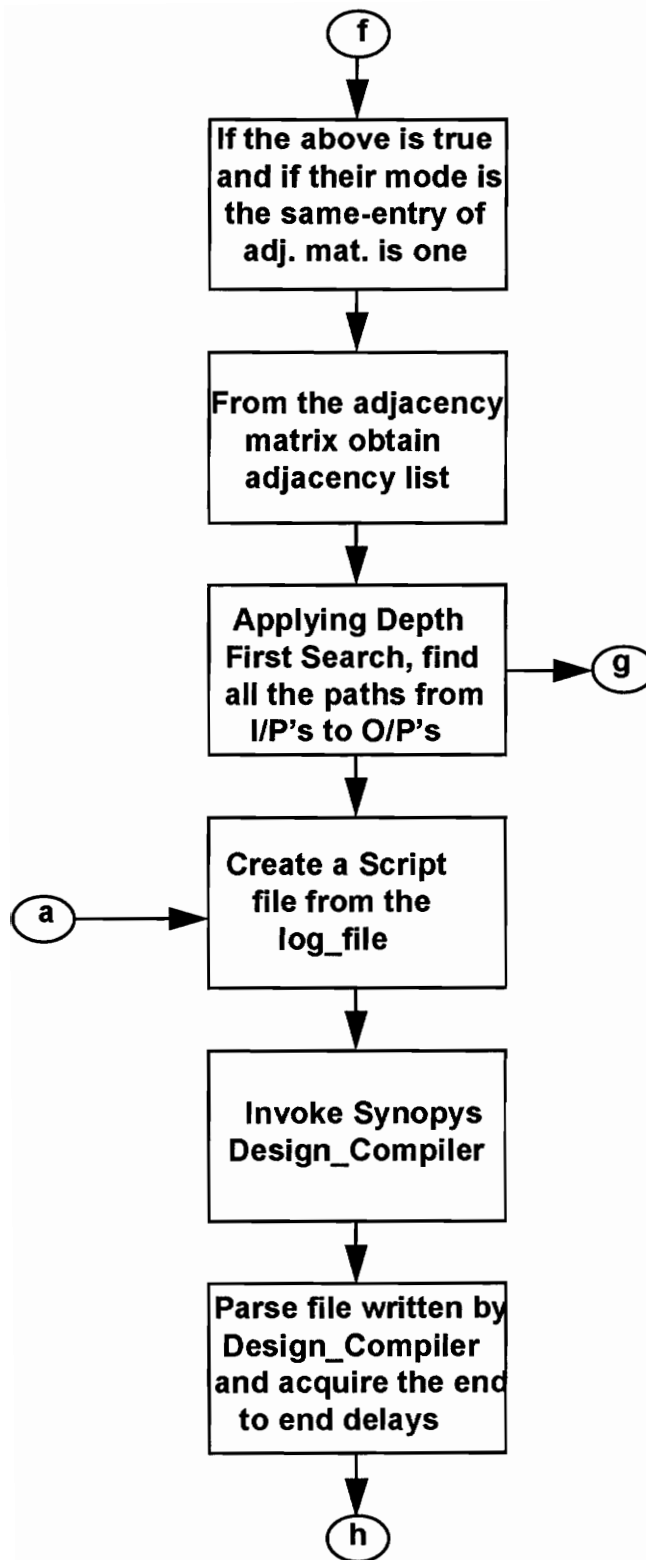
This chapter thus explains the method by which individual process delays are calculated and back annotated to the VHDL description. The entire flow of the process is shown in Figure 17.



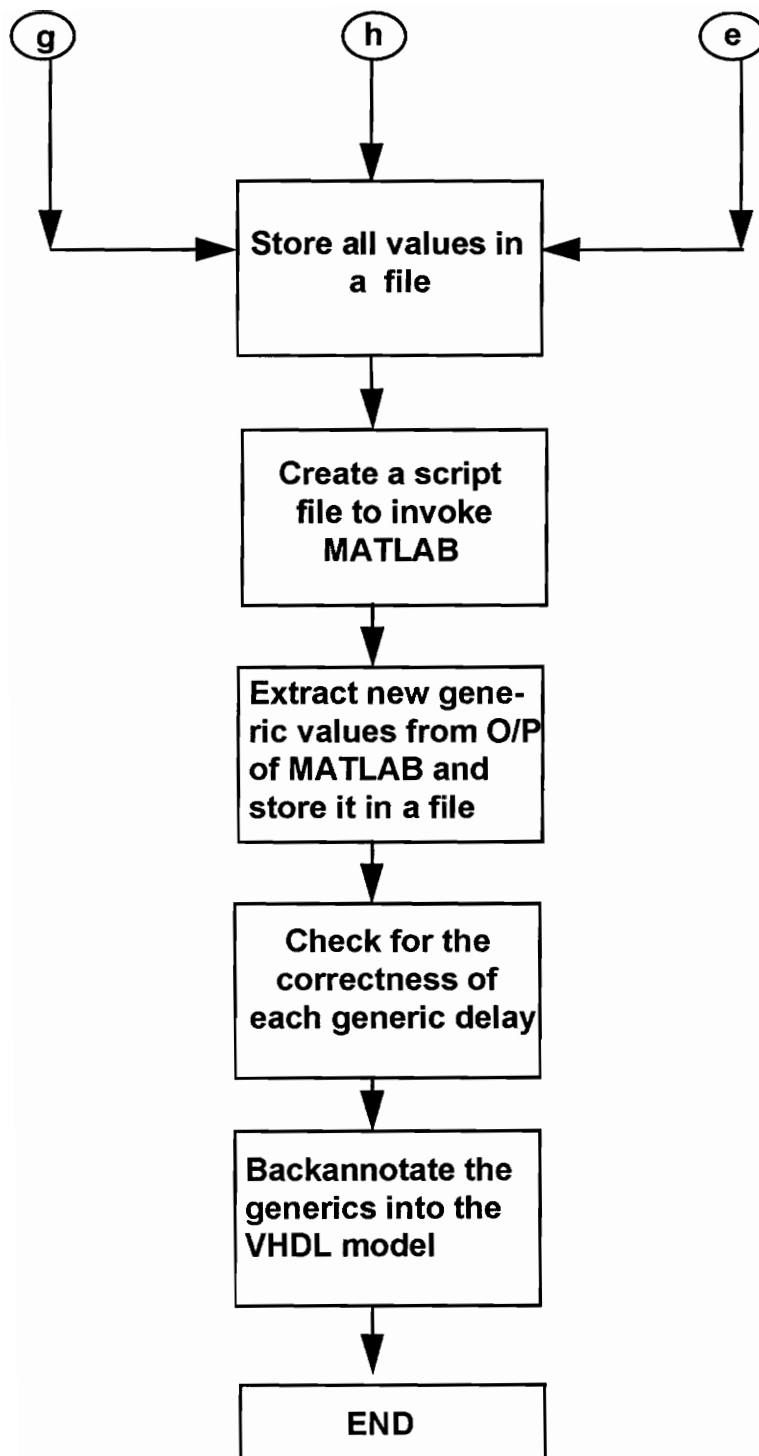
**Figure 17 Flow chart for Backann2**



**Figure 17 Flow chart for Backann2**



**Figure 17 Flow chart for Backann2**



**Figure 17 Flow chart for Backann2**

## Chapter 4

### Results

In this chapter the back-annotated models produced by the software for three classes of circuits will be discussed.

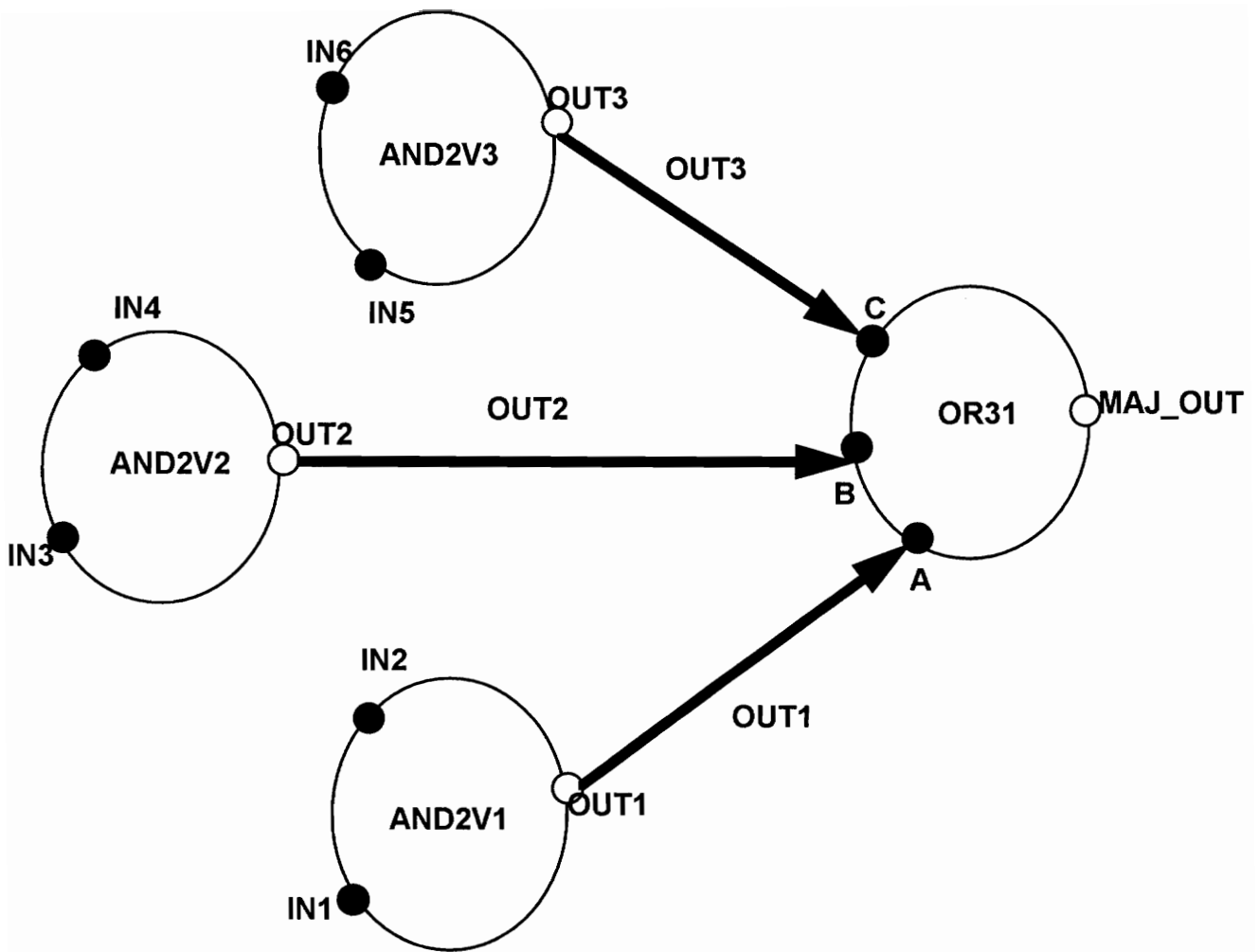
#### 4.1 CLASS 1 Circuits

##### 4.1.1 Majority Function Detector

This is a simple combinational circuit which performs the function

$$\text{MAJ3} = \text{IN1} \cdot \text{IN2} + \text{IN3} \cdot \text{IN4} + \text{IN5} \cdot \text{IN6}$$

The inputs to the AND gates are bit\_vectors of length 5 and the output is of the same length. The three AND gates and the OR gate form the four processes in the Process Model Graph as shown in Figure 18. The VHDL description of the combinational circuit generated by Modeler's Assistant is shown in Figure 19. This type of classification can be classified as Class 1 type since output of each process is fanout free.



**Figure 18 Process Model Graph of Majority Function**

```

use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *****

entity MAJ3 is
  generic (
    OR3_DEL: TIME;
    AND2_DEL3: TIME;
    AND2_DEL2: TIME;
    AND2_DEL1: TIME
  );
  port (MAJ_OUT: out BIT_VECTOR(4 downto 0);
        IN6: in BIT_VECTOR(4 downto 0);
        IN5: in BIT_VECTOR(4 downto 0);
        IN4: in BIT_VECTOR(4 downto 0);
        IN3: in BIT_VECTOR(4 downto 0);
        IN2: in BIT_VECTOR(4 downto 0);
        IN1: in BIT_VECTOR(4 downto 0));
end MAJ3;
-- *****

architecture BEHAVIORAL of MAJ3 is

  signal OUT3: BIT_VECTOR(4 downto 0);
  signal OUT2: BIT_VECTOR(4 downto 0);
  signal OUT1: BIT_VECTOR(4 downto 0);
begin

  -----
  -- Process Name: OR31
  -----

  OR31_2: process (OUT3,OUT2,OUT1)
  begin

    MAJ_OUT <= OUT1 or OUT2 or OUT3 after OR3_DEL;

  end process OR31_2;

  -----
  -- Process Name: AND2V3
  -----

```

**Figure 19 VHDL Description of MAJ3**

```

AND2V3_20: process (IN6,IN5)
begin

    OUT3 <= IN5 and IN6 after AND2_DEL3;

end process AND2V3_20;

-----
-- Process Name: AND2V2
-----

AND2V2_14: process (IN4,IN3)
begin

    OUT2 <= IN3 and IN4 after AND2_DEL2;

end process AND2V2_14;

-----
-- Process Name: AND2V1
-----

AND2V1_8: process (IN2,IN1)
begin

    OUT1 <= IN1 and IN2 after AND2_DEL1;

end process AND2V1_8;
end BEHAVIORAL;

```

**Figure 19 VHDL description of MAJ3**

The input to output paths are shown in Figure 20 and the delays obtained from backann and the equations used for calculating the new delays are shown in Figure 21.

*TP1 = AND2\_DEL3+OR3\_DEL ; IN6 MAJ\_OUT*  
*TP2 = AND2\_DEL2+OR3\_DEL ; IN4 MAJ\_OUT*  
*TP3 = AND2\_DEL1+OR3\_DEL ; IN2 MAJ\_OUT*

**Figure 20 The paths obtained from input to output ports for MAJ3**

In Figure 21 the first three equations are obtained from the report generated by the Synopsys Design Compiler. The individual process delays are the outputs from backann.

*TP1 = 0.545*  
*TP2 = 0.545*  
*TP3 = 0.545*  
  
*OR3\_DEL = 0.485000*  
*AND2\_DEL3 = 0.360000*  
*AND2\_DEL2 = 0.360000*  
*AND2\_DEL1 = 0.360000*  
  
*TP1\_newx = AND2\_DEL3+OR3\_DEL ;*  
  
*AND2\_DEL3\_new = AND2\_DEL3/TP1\_newx\*TP1*  
*AND2\_DEL3 = AND2\_DEL3\_new*  
*OR3\_DEL\_new = OR3\_DEL/TP1\_newx\*TP1*  
*OR3\_DEL = OR3\_DEL\_new*  
*AND2\_DEL2 = TP2-OR3\_DEL*  
*AND2\_DEL1 = TP3-OR3\_DEL*

**Figure 21 The equations used for obtaining the delay values for MAJ3**

The equations following the individual delays are obtained by manipulating the equations

in Figure 20. The delay values obtained are shown in Figure 22. These values are backannotated into the VHDL model. A part of the back annotated model is shown in Figure 23. A comparison of the actual value obtained from Synopsys and the calculated value using backann2 is shown in Table 1. It is seen that there is an individual error of

<i>AND2_DEL3</i>	=	0.2322
<i>OR3_DEL</i>	=	0.3128
<i>AND2_DEL2</i>	=	0.2322
<i>AND2_DEL1</i>	=	0.2322

**Figure 22 The new delay values obtained from MATLAB**

```

use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *****
entity MAJ3 is
  generic (
    OR3_DEL: TIME := 0.3128 ns;
    AND2_DEL3: TIME := 0.2322 ns;
    AND2_DEL2: TIME := 0.2322 ns;
    AND2_DEL1: TIME := 0.2322 ns
  );
  port (MAJ_OUT: out BIT_VECTOR(4 downto 0);
        IN6: in BIT_VECTOR(4 downto 0);
        IN5: in BIT_VECTOR(4 downto 0);
        IN4: in BIT_VECTOR(4 downto 0);
        IN3: in BIT_VECTOR(4 downto 0);
        IN2: in BIT_VECTOR(4 downto 0);
        IN1: in BIT_VECTOR(4 downto 0));
end MAJ3;
-- *****
architecture BEHAVIORAL of MAJ3 is

  signal OUT3: BIT_VECTOR(4 downto 0);
  signal OUT2: BIT_VECTOR(4 downto 0);
  signal OUT1: BIT_VECTOR(4 downto 0);
begin
Figure 23 A part of the VHDL description after back annotation of generic delays

```

**Table 1 Comparison of delay values for MAJ3**

<b>Generic Delays in the Model</b>	<b>Values Obtained from the report</b>	<b>Values obtained from backann2</b>
<b>AND2_DEL3</b>	0.22	0.2322
<b>AND2_DEL2</b>	0.22	0.2322
<b>AND2_DEL1</b>	0.22	0.2322
<b>OR3_DEL</b>	0.32	0.3128

5.54% and 2.25 % for the AND\_DEL and OR\_DEL respectively. The average error is found to be 4.86 %.

#### **4.1.2 ALARM CIRCUIT**

In this circuit [33], the output alarm of a house is triggered if the panic input is one or if the enable input is one, the exiting input is zero, and the house is not secure; the house is secure if window, door and garage inputs are all one. This circuit is a fanout free combinational circuit where nodes are symmetrical but the circuit is not symmetrical. The Process Model Graph and the VHDL description of the model are shown in Figure 24 and Figure 25 respectively. The different paths and the equations involved are shown in Figure 26 and Figure 27 respectively.

This example is different from the previous example because there are paths which traverse only one node and so all the steps mentioned in the algorithm are performed. These nodes are first considered. Here, OR\_DEL2 is a path from PANIC to ALARM.

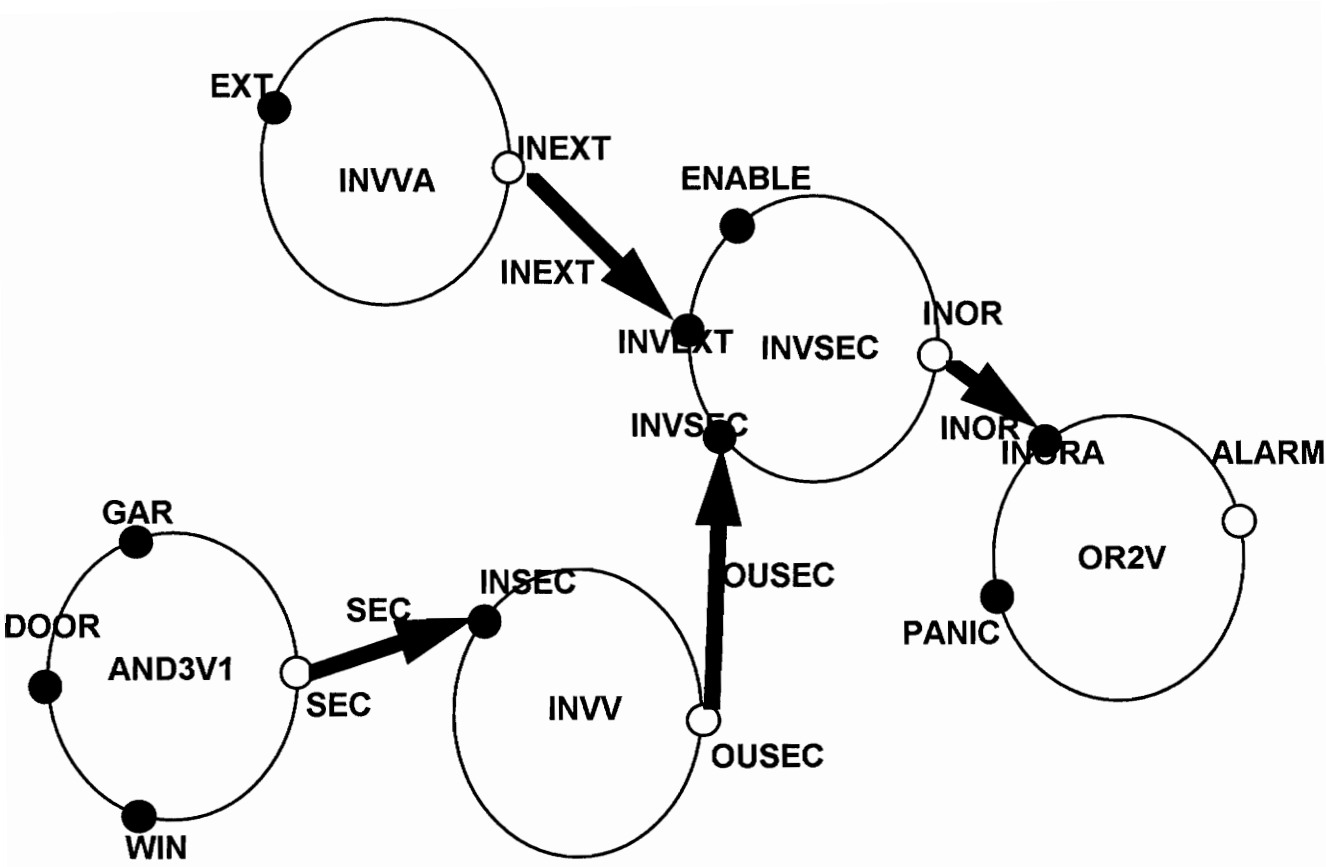


Figure 24 Process Model Graph of ALARM System

The gate\_delay is obtained from the Synopsys Design Compiler. This delay is then subtracted from the other paths as shown. Since in PATH3 the two generics INVDEL1 and AND3DEL1 cannot be obtained like the other paths, it is obtained in the form of proportions. The same principle can be used for all circuits.

The new generic values thus obtained, shown in Figure 28, are then back

```
-- *****
entity ASYM is
  generic (
    INVVDELA: TIME;
    INVVDEL1: TIME;
    OR2_DEL: TIME;
    INVSECDEL2: TIME;
    AND3DEL1: TIME
  );
  port (EXT: in BIT_VECTOR(8 downto 0);
        ALARM: out BIT_VECTOR(8 downto 0);
        PANIC: in BIT_VECTOR(8 downto 0);
        ENABLE: in BIT_VECTOR(8 downto 0);
        GAR: in BIT_VECTOR(8 downto 0);
        DOOR: in BIT_VECTOR(8 downto 0);
        WIN: in BIT_VECTOR(8 downto 0));
end ASYM;
-- *****

architecture BEHAVIORAL of ASYM is

  signal INEXT: BIT_VECTOR(8 downto 0);
  signal OUSEC: BIT_VECTOR(8 downto 0);
  signal SEC: BIT_VECTOR(8 downto 0);
  signal INOR: BIT_VECTOR(8 downto 0);
begin

  -----
  -- Process Name: INVVA
  -----
```

**Figure 25 VHDL description of ALARM system**

```

INVVA_4: process (EXT)
begin
  for I in 0 to 8 loop
    INEXT(I) <= not EXT(I) after INVVDELA;
  end loop;

end process INVVA_4;

-----
-- Process Name: INVV
-----

INVV_9: process (SEC)
begin
  for I in 0 to 8 loop
    OUSEC(I) <= not SEC(I) after INVVDEL1;
  end loop;

end process INVV_9;

-----
-- Process Name: OR2V
-----

OR2V_14: process (PANIC,INOR)
begin
  for I in 0 to 8 loop
    ALARM(I) <= INOR(I) or PANIC(I) after OR2_DEL;
  end loop;

end process OR2V_14;

```

**Figure 25 VHDL description of ALARM system**

annotated to the VHDL behavioral model. The comparison of the delays obtained from backann2 and Synopsys Design Compiler report is shown in Table 2.

```
-- -----  
-- Process Name: INVSEC  
-- -----  
  
INVSEC_20: process (ENABLE,INEXT,OUSEC)  
begin  
  for I in 0 to 8 loop  
    INOR(I) <= OUSEC(I) and INEXT(I) and ENABLE(I) after INVSECDEL2;  
  end loop;  
  
end process INVSEC_20;  
  
-- -----  
-- Process Name: AND3V1  
-- -----  
  
AND3V1_27: process (GAR,DOOR,WIN)  
begin  
  for I in 0 to 8 loop  
    SEC(I) <= WIN(I) and DOOR(I) and GAR(I) after AND3DEL1;  
  end loop;  
  
end process AND3V1_27;  
  
end BEHAVIORAL;
```

**Figure 25 VHDL description of ALARM system**

It can be seen from Table 2, that AND3DEL1 and INVVDEL1 have an error percentage of 1.18 and 1.05 respectively. The other generics have zero error. The average error percentage is found to be 1.12 %.

$$\begin{aligned}
 TP1 &= INVVDELA + INVSECDEL2 + OR2\_DEL \\
 TP2 &= INVSECDEL2 + OR2\_DEL \\
 TP3 &= AND3DEL1 + INVVDEL1 + INVSECDEL2 + OR2\_DEL \\
 TP4 &= OR2\_DEL
 \end{aligned}$$

**Figure 26 The different paths in the ALARM system**

$$\begin{aligned}
 TP1 &= 1.035 \\
 TP2 &= 0.85 \\
 TP3 &= 1.565 \\
 TP4 &= 0.36 \\
 \\ 
 INVVDELA &= 0.150000 \\
 INVVDEL1 &= 0.150000 \\
 OR2\_DEL &= 0.360000 \\
 INVSECDEL2 &= 0.440000 \\
 AND3DEL1 &= 0.440000 \\
 \\ 
 OR2\_DEL &= TP4 \\
 INVSECDEL2 &= TP2 - OR2\_DEL \\
 INVVDELA &= TP1 - OR2\_DEL - INVSECDEL2 \\
 TP3\_new &= TP3 - OR2\_DEL - INVSECDEL2 \\
 TP3\_newx &= AND3DEL1 + INVVDEL1 \\
 AND3DEL1\_new &= AND3DEL1 / TP3\_newx * TP3\_new \\
 AND3DEL1 &= AND3DEL1\_new \\
 INVVDEL1\_new &= INVVDEL1 / TP3\_newx * TP3\_new \\
 INVVDEL1 &= INVVDEL1\_new
 \end{aligned}$$

**Figure 27 Input to MATLAB for ALARM system**

<i>OR2_DEL</i>	=	<i>0.3600</i>
<i>INVSECDEL2</i>	=	<i>0.4900</i>
<i>INVVDELA</i>	=	<i>0.1850</i>
<i>AND3DEL1</i>	=	<i>0.5336</i>
<i>INVVDEL1</i>	=	<i>0.1819</i>

**Figure 28 The delays obtained from backann2 for ALARM system**

**Table 2 Comparison of the delays for ALARM system**

<b>Generic delays of the model</b>	<b>Values Obtained from the report</b>	<b>Values Obtained from Backann2</b>
<b>OR2_DEL</b>	0.3600	0.3600
<b>INVSECDEL2</b>	0.4900	0.4900
<b>INVVDELA</b>	0.1850	0.1850
<b>AND3DEL1</b>	0.5400	0.5336
<b>INVVDEL1</b>	0.1800	0.1819

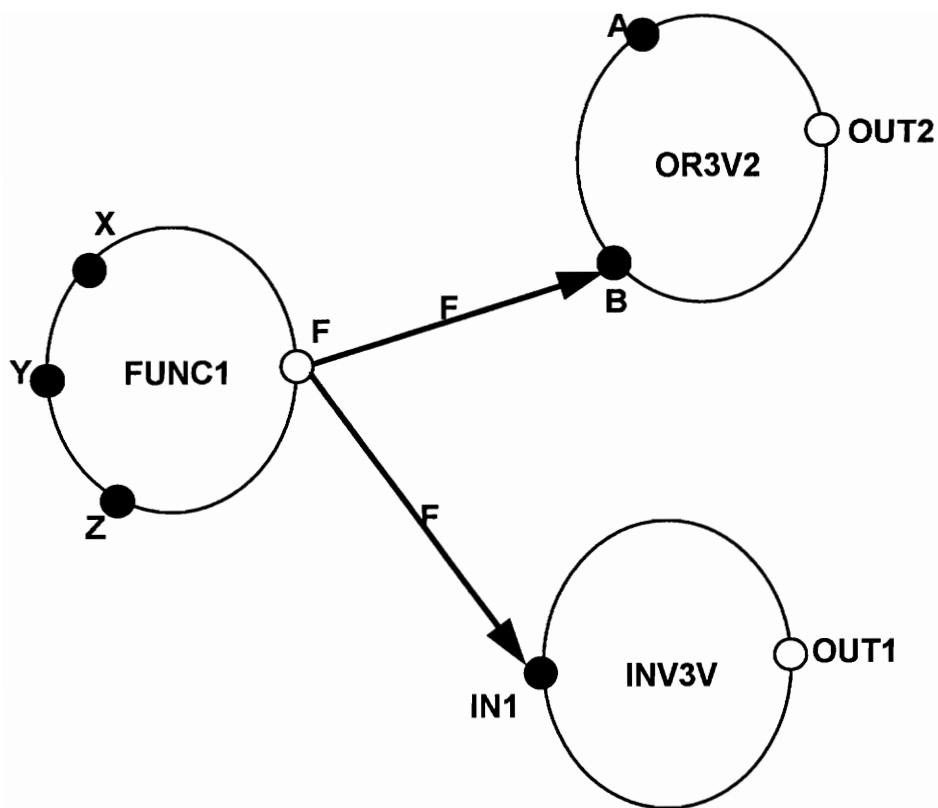
## 4.2 CLASS 2 CIRCUITS

### 4.2.1 AND-OR CIRCUIT

This is a simple combinational circuit with inputs X,Y,Z and A bit\_vectors and outputs OUT1 and OUT2. The operation performed by one of the processes, FUNC1,

$$F = X.Y + X'.Y.Z$$

is given in as input to 2 other processes INV3V and OR3V2 and the outputs are obtained. The Process Model Graph representing the above combinational logic is shown in Figure 29 . The outputs OUT1 and OUT2 are of the form



**Figure 29 Process Model Graph of AND-OR circuit**

$$\text{OUT1} = F' = (X \cdot Y + X' \cdot Y \cdot Z)'$$

$$\text{OUT2} = A + F = A + X \cdot Y + X' \cdot Y \cdot Z$$

```

use WORK..VHDL.CAD.all,WORK.USER_TYPES.all;
-- *****
entity FOUT_EG1 is
  generic (
    OR_DEL: TIME;
    INV_DEL: TIME;
    F_DEL: TIME
  );
  port (OUT2: out BIT_VECTOR(3 downto 0);
        A: in BIT_VECTOR(3 downto 0);
        OUT1: out BIT_VECTOR(3 downto 0);
        Z: in BIT_VECTOR(3 downto 0);
        Y: in BIT_VECTOR(3 downto 0);
        X: in BIT_VECTOR(3 downto 0));
end FOUT_EG1;
-- *****

architecture BEHAVIORAL of FOUT_EG1 is

  signal F: BIT_VECTOR(3 downto 0);
begin

  -- -----
  -- Process Name: OR3V2
  -- -----

  OR3V2_4: process (F,A)
  begin

    OUT2 <= A or F after OR_DEL;

  end loop;

end process OR3V2_4;

```

**Figure 30 VHDL description of the AND-OR circuit**

```

-----
-- Process Name: INV3V
-----

INV3V_10: process (F)
begin

    OUT1(I) <= not F after INV_DEL;

end process INV3V_10;

-----
-- Process Name: FUNC1
-----

FUNC1_15: process (Z,Y,X)
begin

    F <= ((X and Y) or ((not X) and Y and Z)) after F_DEL;

    end loop;

end process FUNC1_15;

end BEHAVIORAL;

```

**Figure 30 VHDL description of the AND-OR circuit**

```

TP1 = OR_DEL ;
TP2 = F_DEL+OR_DEL ;
TP3 = F_DEL+INV_DEL ;

```

**Figure 31 The different paths in the AND-OR circuit**

```

TP1 = 0.36
TP2 = 1.0250
TP3 = 0.8200

OR_DEL = 0.360000
INV_DEL = 0.150000
F_DEL = 0.565000

OR_DEL = TP1
F_DEL = TP2-OR_DEL
INV_DEL = TP3-F_DEL

```

**Figure 32 The input equations to MATLAB for AND-OR circuit**

```

OR_DEL    =    0.3600
F_DEL     =    0.6650
INV_DEL   =    0.1550

```

**Figure 33 The outputs obtained from backann2 for AND-OR circuit**

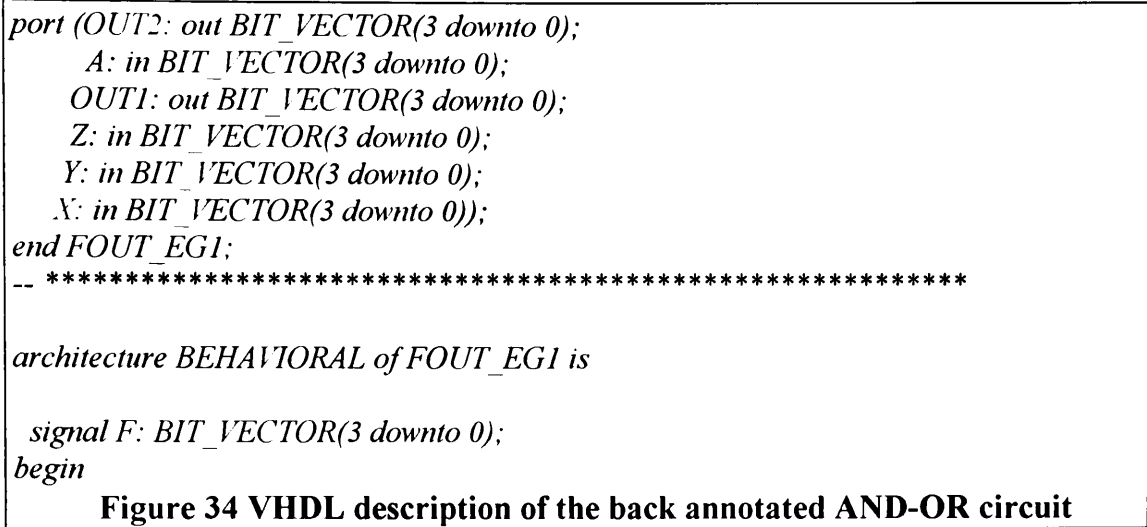
Since the output of the process FUNC1 is fed in as input to two other processes this circuit is classified under Class 2 circuits. The VHDL representation of this circuit is shown in Figure 30. The different paths from the primary input to the primary outputs are shown in Figure 31 and the equations involved to obtain the generic delays are shown in

```

-- *****
entity FOUT_EG1 is
  generic (
    OR_DEL: TIME := 0.3600 ns;
    INV_DEL: TIME := 0.1550 ns;
    F_DEL: TIME := 0.6650 ns
  );

```

**Figure 34 A part of the back annotated VHDL description (AND-OR circuit)**



shown in Figure 32 The values obtained from MATLAB are then back annotated into the VHDL description. These are shown in Figure 33 and Figure 34 respectively.

The values obtained from backann2 and the Synopsys design compiler are compared and is shown in Table 3. It is seen that the values obtained by both are equal.

**Table 3 Comparison of delay values for AND-OR circuit**

Generic delays of the model	Values Obtained from the report	Values Obtained from Backann2
<b>OR_DEL</b>	0.3600	0.3600
<b>INV_DEL</b>	0.1550	0.1550
<b>F_DEL</b>	0.6650	0.6650

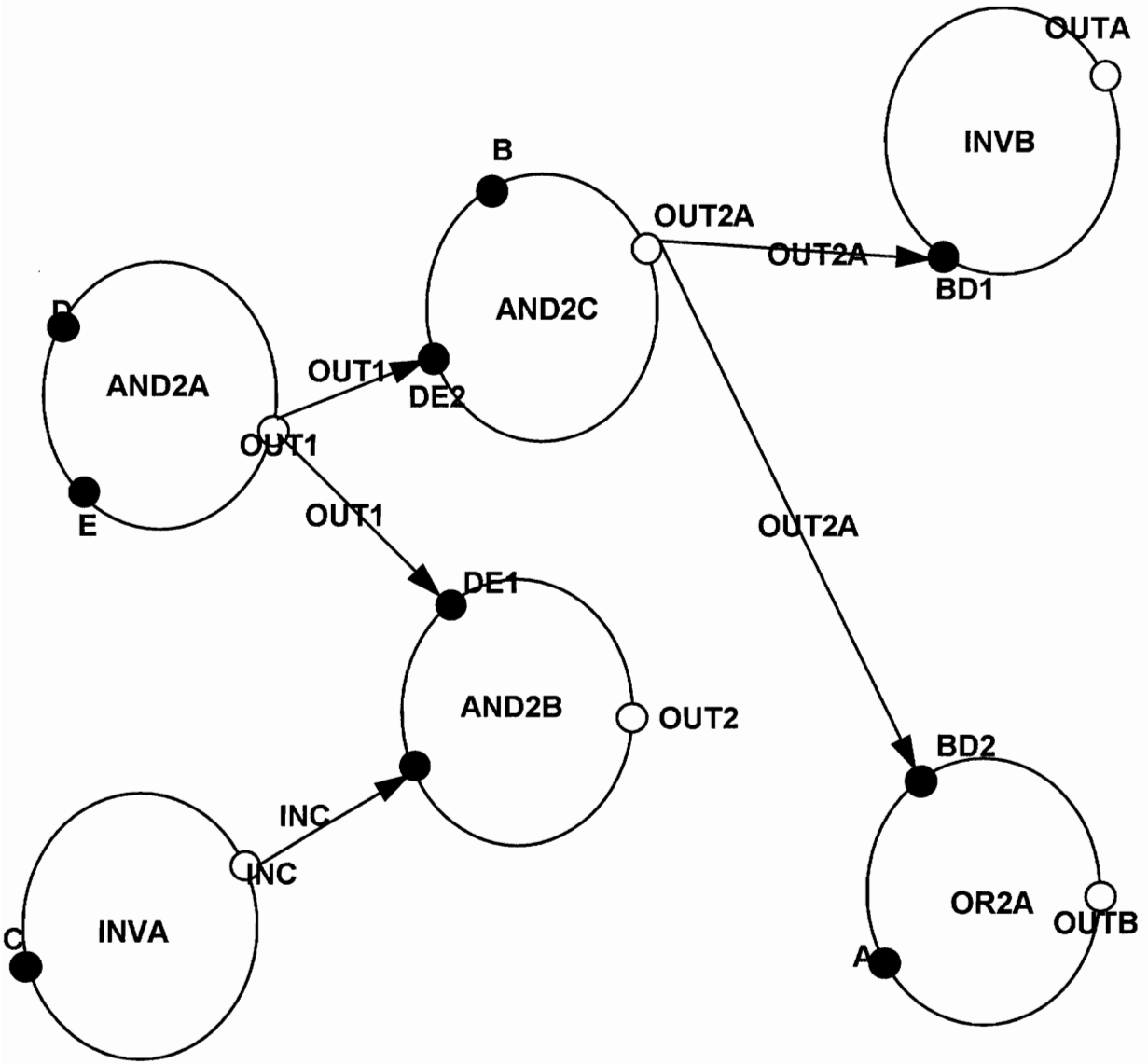


Figure 35 Process Model Graph of AND-OR-INVERT circuit

## 4.2.2 AND-OR-INVERT CIRCUIT

This circuit is another example of a fan-out ( Class 2) circuit. It has more than one fanout node. The outputs of this circuit are OUTA, OUTB and OUT2 and are expressed of the form,

$$\text{OUTA} = (\text{B} \cdot \text{D} \cdot \text{E})'$$

$$\text{OUTB} = \text{A} + \text{B} \cdot \text{D} \cdot \text{E}$$

$$\text{OUT2} = \text{C}' \cdot \text{D} \cdot \text{E}$$

where A, B, C, D and E are inputs to the model. The Process Model Graph and its VHDL description for the above circuit is shown in Figure 35 and Figure 36 respectively. The different paths from inputs to outputs is shown in Figure 37. Figure 38 gives the input file to MATLAB to calculate the generic delays and Figure 39 gives the corresponding outputs.

```
use WORK.VHDL.CAD.all,WORK.USER_TYPES.all;
-- *****
entity FOUT_EG2 is
  generic (
    OR_DEL: TIME;
    INV_DELA: TIME;
    INV_DELB: TIME;
    AND_DELC: TIME;
    AND_DELB: TIME;
    AND_DELA: TIME
  );
  port (OUTB: out BIT;
        A: in BIT;
        C: in BIT;
        OUTA: out BIT;
        B: in BIT;
        OUT2: out BIT;
```

**Figure 36 VHDL Description of AND-OR-INVERT circuit**

```

E: in BIT;
D: in BIT);
end FOUT_EG2;
-- *****

architecture BEHAVIORAL of FOUT_EG2 is

    signal OUT2A: BIT;
    signal INC: BIT;
    signal OUT1: BIT;
begin

    -----
    -- Process Name: OR2A
    -----

    OR2A_2: process (A,OUT2A)
    begin
        OUTB <= OUT2A or A after OR_DEL;
    end process OR2A_2;

    -----
    -- Process Name: INVA
    -----

    INVA_10: process (C)
    begin
        INC <= not C after INV_DELA;
    end process INVA_10;

    -----
    -- Process Name: INVB
    -----

    INVB_15: process (OUT2A)

```

**Figure 36 VHDL Description of AND-OR-INVERT circuit**

```

begin
    OUTA <= not OUT2A after INV_DELB;

end process INVB_15;

-----
-- Process Name: AND2C
-----

AND2C_20: process (OUT1,B)
begin
    OUT2A <= B and OUT1 after AND_DELC;

end process AND2C_20;

-----
-- Process Name: AND2B
-----

AND2B_26: process (INC,OUT1)
begin
    OUT2 <= OUT1 and INC after AND_DELB;

end process AND2B_26;

-----
-- Process Name: AND2A
-----

AND2A_32: process (E,D)
begin
    OUT1 <= D and E after AND_DELA;

end process AND2A_32;

end BEHAVIORAL;

```

**Figure 36 VHDL Description of AND-OR-INVERT circuit**

$TP1 = OR\_DEL ;$   
 $TP2 = AND\_DELC + OR\_DEL ;$   
 $TP3 = AND\_DELA + AND\_DELC + OR\_DEL ;$   
 $TP4 = AND\_DELC + INV\_DELB ;$   
 $TP5 = AND\_DELA + AND\_DELC + INV\_DELB ;$   
 $TP6 = AND\_DELA + AND\_DELB ;$   
 $TP7 = INV\_DELA + AND\_DELB ;$

**Figure 37 The different paths in the AND-OR-INVERT circuit**

$TP1 = 0.36$   
 $TP2 = 0.845$   
 $TP3 = 1.285$   
 $TP4 = 0.67$   
 $TP5 = 1.105$   
 $TP6 = 0.81$   
 $TP7 = 0.555$

$OR\_DEL = 0.360000$   
 $INV\_DELA = 0.150000$   
 $INV\_DELB = 0.150000$   
 $AND\_DELC = 0.360000$   
 $AND\_DELB = 0.360000$   
 $AND\_DELA = 0.360000$

$OR\_DEL = TP1$   
 $AND\_DELC = TP2 - OR\_DEL$   
 $AND\_DELA = TP3 - OR\_DEL - AND\_DELC$   
 $INV\_DELB = TP4 - AND\_DELC$   
 $AND\_DELB = TP6 - AND\_DELA$   
 $INV\_DELA = TP7 - AND\_DELB$

**Figure 38 The input to MATLAB for AND-OR-INVERT circuit**

```

OR_DEL = 0.3600
AND_DELC = 0.4850
AND_DELA = 0.4400
INV_DELB = 0.1800
AND_DELB = 0.3700
INV_DELA = 0.1850

```

**Figure 39 The output of MATLAB for AND-OR-INVERT circuit**

```

use WORK.VHDL.CAD.all,WORK.USER_TYPES.all;
-- *****
entity FOUT_EG2 is
  generic (
    OR_DEL: TIME := 0.3600 ns;
    INV_DELA: TIME := 0.1850 ns;
    INV_DELB: TIME := 0.1800 ns;
    AND_DELC: TIME := 0.4850 ns;
    AND_DELB: TIME := 0.3700 ns;
    AND_DELA: TIME := 0.4400 ns
  );
  port (OUTB: out BIT;
    A: in BIT;
    C: in BIT;
    OUTA: out BIT;
    B: in BIT;
    OUT2: out BIT;
    E: in BIT;
    D: in BIT);
end FOUT_EG2;
-- *****

architecture BEHAVIORAL of FOUT_EG2 is

  signal OUT2A: BIT;
  signal INC: BIT;

```

**Figure 40 VHDL description of the back annotated AND-OR-INVERT model**

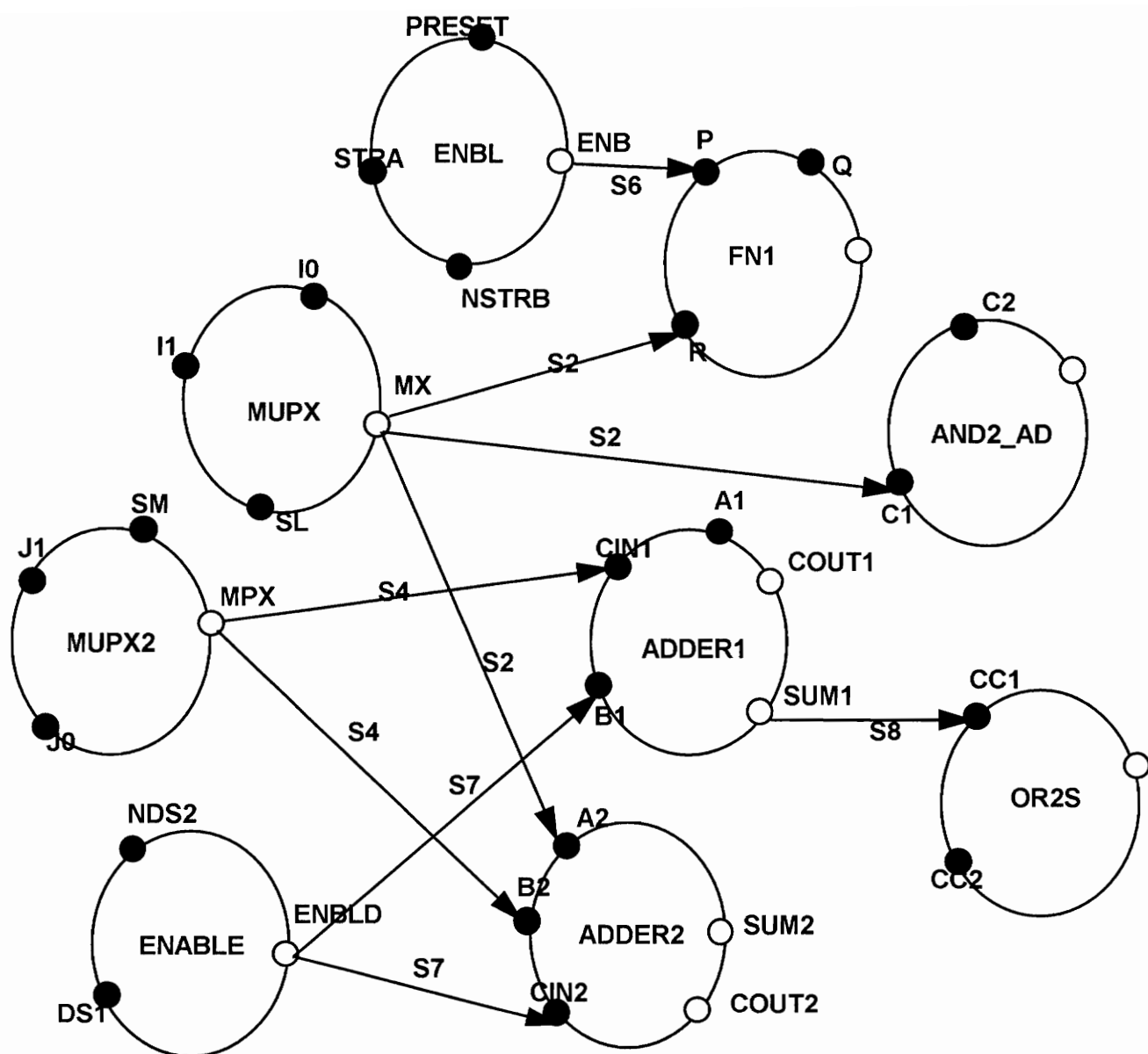
These values are back annotated into the VHDL model and is shown in Figure 40. The values obtained from backann2 is compared with those obtained by the Synopsys Design Compiler and the results are shown in Table 4. It is seen that both are almost equal.

**Table 4 Comparison of delays for AND-OR \_INVERT circuit**

<b>Generic delays of the model</b>	<b>Values Obtained from the report</b>	<b>Values Obtained from Backann2</b>
<b>OR_DEL</b>	0.3600	0.3600
<b>AND_DELC</b>	0.4850	0.4850
<b>AND_DEL1</b>	0.4400	0.4400
<b>INV_DELB</b>	0.1800	0.1800
<b>AND_DELB</b>	0.3750	0.3700
<b>INV_DELA</b>	0.1800	0.1850

### 4.2.3 ADDER-MUX CIRCUIT

This circuit is another example of a fanout circuit. The nodes in this circuit are asymmetrical. The Process Model Graph in Figure 41 shows two multiplexers, two adders and five processes with simple combinational logic. The output of the first multiplexer is fed into an adder (ADDER2), a two input AND gate and the process FN1 which performs a simple function. The output of the second multiplexer is fed in as input to the two adders ( ADDER1 and ADDER2 ). The output of the process ENABLE is fed in as input to the adders. The output of ADDER1 is fed into a two input OR gate. The VHDL description



**Figure 41 Process Model Graph of the ADDER-MUX circuit**

```

use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *****
entity ADMX is
  generic (
    FN1_DEL: TIME ;
    ENBL_DEL: TIME ;
    MUPX2_DEL: TIME ;
    MUPX_DEL: TIME ;
    CARRY2_DEL: TIME ;
    SUM2_DEL: TIME ;
    CARRY1_DEL: TIME ;
    SUM1_DEL: TIME;
    ENABLE_DEL: TIME ;
    AND_DEL: TIME;
    OR2S_DEL: TIME
  );
  port (Y: out BIT;
        Q: in BIT;
        STRA: in BIT;
        NSTRB: in BIT;
        PRESET: in BIT;
        J1: in BIT;
        J0: in BIT;
        SM: in BIT;
        I1: in BIT;
        I0: in BIT;
        SL: in BIT;
        COUT1: out BIT;
        A1: in BIT;
        COUT2: out BIT;
        SUM2: out BIT;
        DS1: in BIT;
        NDS2: in BIT;
        CON: out BIT;
        C2: in BIT;
        CCOP: out BIT;
        CC2: in BIT);
end ADMX;
-- *****

architecture BEHAVIORAL of ADMX is

```

**Figure 42 VHDL description of ADDER-MUX circuit**

```

signal S6: BIT;
signal S2: BIT;
signal S4: BIT;
signal S8: BIT;
signal S7: BIT;
begin

-----
-- Process Name: FNI
-----

FNI_4: process (S6,Q,S2)
begin
    Y <= S6 and Q and not S2 after FNI_DEL ;

end process FNI_4;

-----
-- Process Name: ENBL
-----

ENBL_11: process (STRA,NSTRB,PRESET)
begin
    if (PRESET='1') then
        S6 <= STRA after ENBL_DEL ;
    else
        S6 <= STRA and not NSTRB after ENBL_DEL ;
    end if;

end process ENBL_11;

-----
-- Process Name: MUPX2
-----

MUPX2_18: process (J1,J0,SM)
begin

```

**Figure 42 VHDL Description of ADDER-MUX circuit**

```

if (SM='0') then
    S4 <= J0 after MUPX2_DEL ;
else
    S4 <= J1 after MUPX2_DEL ;
end if;

end process MUPX2_18;

-----
-- Process Name: MUPX
-----

MUPX_25: process (I1,I0,SL)
begin
    if (SL='0') then
        S2 <= I0 after MUPX_DEL ;
    else
        S2 <= I1 after MUPX_DEL ;
    end if;

end process MUPX_25;

-----
-- Process Name: ADDER2
-----

ADDER2_32: process (A1,S7,S4)
begin
    S8 <= A1 xor S7 xor S4 after SUM2_DEL ;
    COUT1 <= (A1 and S7) or (A1 and S4) or (S7 and S4) after CARRY2_DEL ;

end process ADDER2_32;

-----
-- Process Name: ADDER1
-----

```

**Figure 42 VHDL Description of ADDER-MUX circuit**

```

ADDER1_41: process (S2,S7,S4)
begin
    SUM2 <= S2 xor S7 xor S4 after SUM1_DEL ;
    COUT2 <= (S2 and S7) or (S2 and S4) or (S7 and S4) after CARRY1_DEL ;

end process ADDER1_41;

-----
-- Process Name: ENABLE
-----

ENABLE_50: process (DS1,NDS2)
begin
    S7 <= DS1 and not NDS2 after ENABLE_DEL ;

end process ENABLE_50;

-----
-- Process Name: AND2_AD
-----

AND2_AD_56: process (S2,C2)
begin
    CON <= C2 and S2 after AND_DEL ;

end process AND2_AD_56;

-----
-- Process Name: OR2S
-----

OR2S_62: process (S8,CC2)
begin
    CCOP <= CC2 or S8 after OR2S_DEL ;

end process OR2S_62;
end BEHAVIORAL;

```

**Figure 42 VHDL Description of ADDER-MUX circuit**

$TP1 = MUPX2\_DEL + SUM2\_DEL + OR2S\_DEL$   
 $TP2 = SUM2\_DEL + OR2S\_DEL$   
 $TP3 = ENABLE\_DEL + SUM2\_DEL + OR2S\_DEL$   
 $TP4 = OR2S\_DEL$   
 $TP5 = FN1\_DEL$   
 $TP6 = ENBL\_DEL + FN1\_DEL$   
 $TP7 = MUPX\_DEL + FN1\_DEL$   
 $TP8 = MUPX2\_DEL + CARRY2\_DEL$   
 $TP9 = ENABLE\_DEL + CARRY2\_DEL$   
 $TP10 = MUPX2\_DEL + SUM1\_DEL$   
 $TP11 = MUPX2\_DEL + CARRY1\_DEL$   
 $TP12 = MUPX\_DEL + SUM1\_DEL$   
 $TP13 = MUPX\_DEL + CARRY1\_DEL$   
 $TP14 = ENABLE\_DEL + SUM1\_DEL$   
 $TP15 = ENABLE\_DEL + CARRY1\_DEL$   
 $TP16 = MUPX\_DEL + AND\_DEL$   
 $TP17 = AND\_DEL$

**Figure 43 The different paths in ADDER-MUX circuit**

of the model is shown in Figure 42. The different paths in the circuit are illustrated in Figure 43. Figure 44 and Figure 45 show the equations for calculating the generic delays and the results obtained from MATLAB. In Figure 44 it is seen that the same generic delay has been calculated more than once. This is because these generics can be calculated in more than one way and which ever path is considered, it can be verified that the delay obtained is the same.

$TP1 = 4.31$   
 $TP2 = 2.99$   
 $TP3 = 4.81$   
 $TP4 = 0.615$   
 $TP5 = 0.77$   
 $TP6 = 1.755$   
 $TP7 = 2.055$   
 $TP8 = 2.725$   
 $TP9 = 3.225$   
 $TP10 = 3.59$

**Figure 44 The input equations to MATLAB for ADDER-MUX circuit**

$$TP11 = 2.725$$

$$TP12 = 3.555$$

$$TP13 = 2.69$$

$$TP14 = 4.085$$

$$TP15 = 3.225$$

$$TP16 = 1.91$$

$$TP17 = 0.625$$

$$FN1\_DEL = 0.625000$$

$$ENBL\_DEL = 0.755000$$

$$MUX2\_DEL = 0.690000$$

$$MUX\_DEL = 0.690000$$

$$CARRY2\_DEL = 0.855000$$

$$SUM2\_DEL = 1.315000$$

$$CARRY1\_DEL = 0.855000$$

$$SUM1\_DEL = 1.315000$$

$$ENABLE\_DEL = 0.460000$$

$$AND\_DEL = 0.360000$$

$$OR2S\_DEL = 0.360000$$

$$OR2S\_DEL = TP4$$

$$FN1\_DEL = TP5$$

$$AND\_DEL = TP17$$

$$SUM2\_DEL = TP2-OR2S\_DEL$$

$$ENBL\_DEL = TP6-FN1\_DEL$$

$$MUX\_DEL = TP7-FN1\_DEL$$

$$MUX\_DEL = TP16-AND\_DEL$$

$$MUX2\_DEL = TP1-OR2S\_DEL-SUM2\_DEL$$

$$ENABLE\_DEL = TP3-OR2S\_DEL-SUM2\_DEL$$

$$SUM1\_DEL = TP12-MUX\_DEL$$

$$CARRY1\_DEL = TP13-MUX\_DEL$$

$$CARRY2\_DEL = TP8-MUX2\_DEL$$

$$CARRY2\_DEL = TP9-ENABLE\_DEL$$

$$SUM1\_DEL = TP10-MUX2\_DEL$$

$$CARRY1\_DEL = TP11-MUX2\_DEL$$

$$SUM1\_DEL = TP14-ENABLE\_DEL$$

$$CARRY1\_DEL = TP15-ENABLE\_DEL$$

**Figure 44 The input equations to MATLAB for ADDER-MUX circuit**

```

OR2S_DEL    =    0.6150
FN1_DEL     =    0.7700
AND_DEL     =    0.6250
SUM2_DEL    =    2.3750
ENBL_DEL    =    0.9850
MUPX_DEL    =    1.2850
MUPX_DEL    =    1.2850
MUPX2_DEL   =    1.3200
ENABLE_DEL  =    1.8200
SUM1_DEL    =    2.2700
CARRY1_DEL  =    1.4050
CARRY2_DEL  =    1.4050
CARRY2_DEL  =    1.4050
SUM1_DEL    =    2.2700
CARRY1_DEL  =    1.4050
SUM1_DEL    =    2.2650
CARRY1_DEL  =    1.4050

```

**Figure 45 The outputs from MATLAB for ADDER-MUX circuit**

A part of the VHDL model with the delays back annotated is shown in Figure 46 and a comparison of the delays obtained from backann2 and the report generated by Synopsys Design Compiler is shown in Table 5. It is seen that there is no error.

```

use WORK.VHDLCAD.all, WORK.USER_TYPES.all;
-- *****
entity ADMX is
  generic (
    FN1_DEL: TIME := 0.7700 ns ;
    ENBL_DEL: TIME := 0.9850 ns ;
    MUPX2_DEL: TIME := 1.1100 ns ;
    MUPX_DEL: TIME := 1.3200 ns ;
    CARRY2_DEL: TIME := 1.4050 ns ;
    SUM2_DEL: TIME := 2.3750 ns ;
    CARRY1_DEL: TIME := 1.4050 ns ;
    SUM1_DEL: TIME := 2.2650 ns ;

```

**Figure 46 A part of the VHDL description of back annotated ADDER-MUX**

```

ENABLE_DEL: TIME := 1.8200 ns ;
AND_DEL: TIME := 0.6250 ns ;
OR2S_DEL: TIME := 0.6150 ns
);
port (Y: out BIT;
      Q: in BIT;
      STRA: in BIT;
      NSTRB: in BIT;
      PRESET: in BIT;
      J1: in BIT;
      J0: in BIT;
      SM: in BIT;
      I1: in BIT;
      I0: in BIT;
      SL: in BIT;
      COUT1: out BIT;
      A1: in BIT;
      COUT2: out BIT;
      SUM2: out BIT;
      DS1: in BIT;
      NDS2: in BIT;
      CON: out BIT;
      C2: in BIT;
      CCOP: out BIT;
      CC2: in BIT);
end ADMX;
-- *****

architecture BEHAVIORAL of ADMX is
  signal S2: BIT;
  signal S6: BIT;
  signal S4: BIT;
  signal S8: BIT;
  signal S7: BIT;
begin

  -----
  -- Process Name: FN1
  -----

  FN1_4: process (S6,Q,S2)
  begin

```

**Figure 46 A part of the VHDL description of back annotated ADDER-MUX**

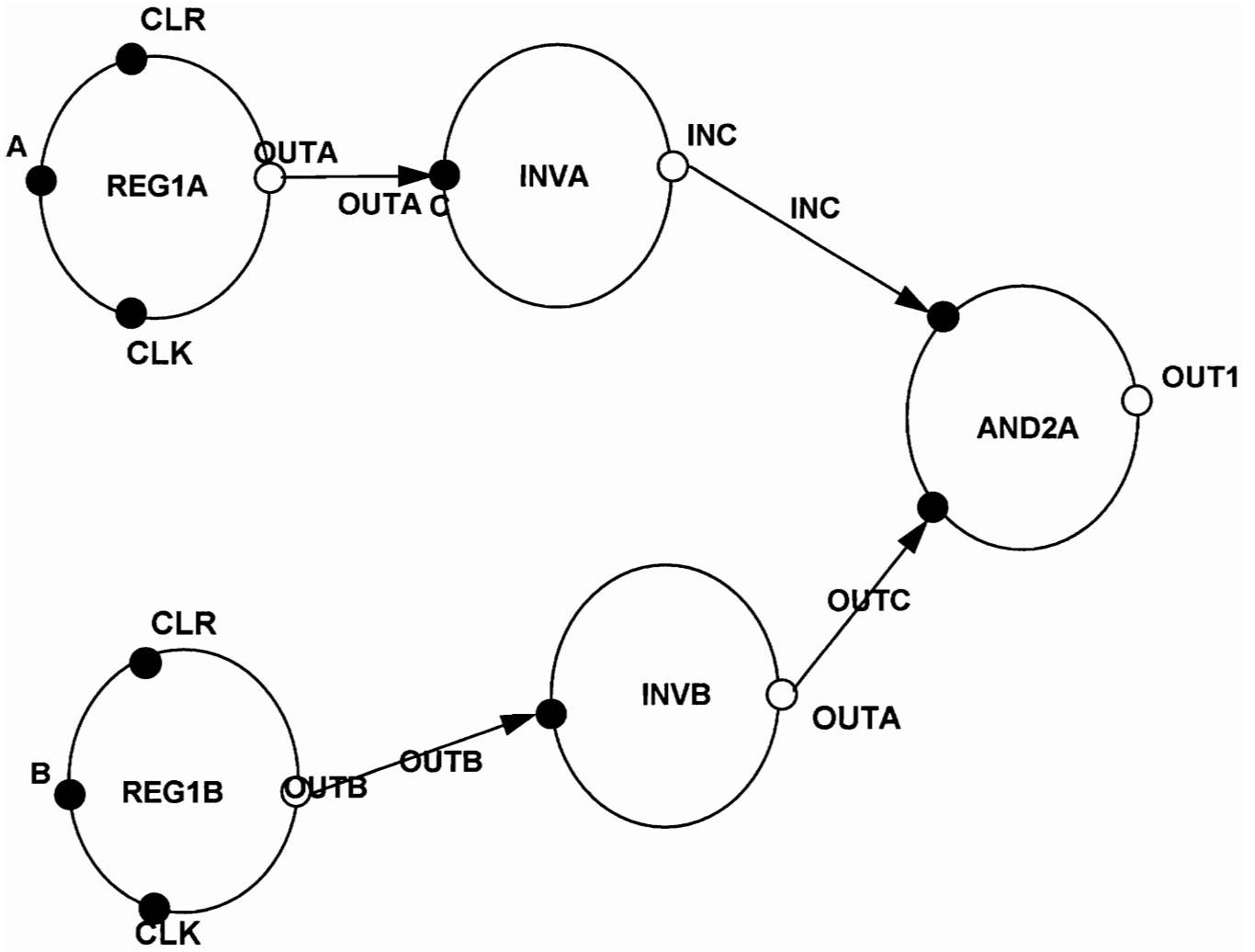
**Table 5 Comparison of delays for ADDER-MUX circuit**

<b>Generic delays of the model</b>	<b>Values Obtained from the report</b>	<b>Values Obtained from Backann2</b>
<b>OR2S_DEL</b>	0.6150	0.6150
<b>FN1_DEL</b>	0.7700	0.7700
<b>AND_DEL</b>	0.6250	0.6250
<b>SUM2_DEL</b>	2.3750	2.3750
<b>ENBL_DEL</b>	0.9850	0.9850
<b>MUPX_DEL</b>	1.2850	1.2850
<b>MUPX2_DEL</b>	1.3200	1.3200
<b>ENABLE_DEL</b>	1.8200	1.8200
<b>SUM1_DEL</b>	2.2650	2.2650
<b>CARRY1_DEL</b>	1.4050	1.4050
<b>CARRY2_DEL</b>	1.4050	1.4050

### **4.3 CLASS 3 CIRCUITS**

#### **4.3.1 A Simple Load Unit**

This unit consists of a combinational logic, whose inputs are single bits of data stored in a single bit register (flip-flop) and are available at the rising edge of the clock. The principle used here can be applied for more than one register. The difference between this sequential circuit and a purely combinational circuit is the way in which the log\_file



**Figure 47 Process Model Graph of SIMPLE**

is given. Since registers are involved, a clock should be specified in the log\_file.

The Process Model Graph and the VHDL generated for it are shown in Figure 47 and Figure 48 respectively. The paths and the calculations involved is shown in Figure 49 and Figure 50 respectively. The delay values calculated by MATLAB is shown in Figure 51. The back annotated VHDL model is shown in Figure 52.

The delay values obtained are compared with the delays obtained from the Synopsys Design compiler and are shown in Table 6.

```
-- *****
entity SIMPLE is
  generic (
    AND_DEL: TIME;
    INV_DELB: TIME;
    INVA_DEL: TIME;
    REG1B_DEL: TIME;
    REG1A_DEL: TIME
  );
  port (AND_OUT: out BIT;
        CLK: in BIT;
        CLR: in BIT;
        B: in BIT;
        A: in BIT);
end SIMPLE;
-- *****

architecture BEHAVIORAL of SIMPLE is

  signal OUT2: BIT;
  signal OUT1: BIT;
  signal OUTB: BIT;
  signal OUTA: BIT;
begin
```

**Figure 48 VHDL description of SIMPLE**

-----  
-- Process Name: AND2A  
-----

```
AND2A_4: process (OUT2,OUT1)
begin
    AND_OUT <= OUT1 and OUT2 after AND_DEL;

end process AND2A_4;
```

-----  
-- Process Name: INVB  
-----

```
INVB_10: process (OUTB)
begin
    OUT2 <= not OUTB after INV_DELB;

end process INVB_10;
```

-----  
-- Process Name: INVA  
-----

```
INVA_15: process (OUTA)
begin
    OUT1 <= not OUTA after INVA_DEL;

end process INVA_15;
```

-----  
-- Process Name: REG1B  
-----

```
REG1B_20: process (CLK,CLR,B)
begin
    if (CLR = '1') then
```

**Figure 48 VHDL description of SIMPLE**

```

OUTB <= '0' after REG1B_DEL;

elsif (CLK'event and CLK = '1') then
    OUTB <= B after REG1B_DEL;

end if;

end process REG1B_20;

-----
-- Process Name: REG1A
-----

REG1A_27: process (CLK,CLR,A)
begin
    if (CLR = '1') then
        OUTA <= '0' after REG1A_DEL;

    elsif (CLK'event and CLK = '1') then
        OUTA <= A after REG1A_DEL;

    end if;

end process REG1A_27;

end BEHAVIORAL;

```

**Figure 48 VHDL description of SIMPLE**

```

TP1 = REG1B_DEL+INV_DELB+AND_DEL
TP2 = REG1A_DEL+INVA_DEL+AND_DEL

```

**Figure 49 The different paths in SIMPLE**

```

TP1 = 2.41
TP2 = 2.41

AND_DEL1 = 0.290000
INV_DELB = 0.150000
INV_DELA = 0.150000
REG1B_DEL = 0.680000
REG1A_DEL = 0.680000

TP1_newx = REG1B_DEL+INV_DELB+AND_DEL1 ;
REG1B_DEL_new = REG1B_DEL/TP1_newx*TP1
REG1B_DEL = REG1B_DEL_new
INV_DELB_new = INV_DELB/TP1_newx*TP1
INV_DELB = INV_DELB_new
AND_DEL1_new = AND_DEL1/TP1_newx*TP1
AND_DEL1 = AND_DEL1_new

TP2 = TP2-AND_DEL1
TP2_newx = REG1A_DEL+INV_DELA ;
REG1A_DEL_new = REG1A_DEL/TP2_newx*TP2
REG1A_DEL = REG1A_DEL_new
INV_DELA_new = INV_DELA/TP2_newx*TP2
INV_DELA = INV_DELA_new

```

**Figure 50 The equations for the calculation of delays for SIMPLE**

```

REG1B_DEL = 1.4632
INV_DELB = 0.3228
AND_DEL1 = 0.6240
REG1A_DEL = 1.4632
INV_DELA = 0.3228

```

**Figure 51 The output of backann2 for SIMPLE**

```

-- *****
entity SIMPLE is
  generic (
    AND_DEL: TIME := 0.6240 ns;
    INV_DELB: TIME := 0.3228 ns;
    INVA_DEL: TIME := 0.3228 ns;
    REG1B_DEL: TIME := 1.4632 ns;
    REG1A_DEL: TIME := 1.4632 ns;
  );
  port (AND_OUT: out BIT;
        CLK: in BIT;
        CLR: in BIT;
        B: in BIT;
        A: in BIT);
end SIMPLE;
-- *****

architecture BEHAVIORAL of SIMPLE is

  signal OUT2: BIT;
  signal OUT1: BIT;
  signal OUTB: BIT;
  signal OUTA: BIT;
begin

```

**Figure 52 VHDL description of SIMPLE (back annotated model)**

**Table 6 Comparison of delays for SIMPLE**

Generic delays of the model	Values Obtained from the report	Values Obtained from Backann2
<b>AND_DEL</b>	0.6250	0.6240
<b>INV_DELB</b>	0.3100	0.3228
<b>INVA_DEL</b>	0.3100	0.3228
<b>REG1B_DEL</b>	1.4750	1.4632
<b>REG1A_DEL</b>	1.4750	1.4632

From Figure 58 it is seen that the average error percentage is 2.69% and the individual error percentage for AND\_DEL, INV\_DEL and REG1\_DEL are 0.16%, 4.12% and 0.8% respectively.

The example shown above can be extended to multiple registers and registers at the output.

## 4.4 Summary and Interpretation of Results

Three classes of circuits have been discussed in this chapter and the generic delay values for each model was found and compared with those obtained from the Synopsys Design Compiler. The following conclusions can be reached by interpretation of the above results :

1. The accuracy of backann2 does not depend on the size of the circuit but on the way each of the processes are connected.
2. When in the model there is at least one process whose inputs and outputs are the primary inputs and the primary outputs, the generic delay calculated is more accurate than when there is no such process.
3. The delay values backannotated are 100 % error free when there is one or more process which has a path from input to output and these processes when connected to other processes in the model, lead to equations such that these delays can be calculated without using the proportionality method. This is shown in the ADDER-MUX circuit.
4. The worst case error obtained in the results is close to 5%. It is seen that it occurs in models which there is no path in the circuit which has its primary input and primary

outputs in the same process.

## **Chapter 5**

### **Suggestions and Future Work**

The software developed is a powerful tool, but certain inadequacies do exist in it. One of the reasons these inadequacies arise is because it has to work with another software like Backann even though it was designed with the same philosophy of calculation of delay values.

#### **5.1 Redundant Synthesis**

Synthesis using Synopsys Design Compiler is being done  $n+2$  times where  $n$  is the number of processes in the model. The whole model is being synthesized twice and each process is synthesized once. Synthesizing the whole model twice is redundant. The first time, it is synthesized by the earlier version of backann to verify whether the model is synthesizable and if the processes can be synthesized individually. The second time the whole model is synthesized to find the end to end delays of the model. The source code can be modified in such a way that synthesis of the whole model is done only once and performs both the functions. This will reduce the execution time of backann2 and hence

speeden up the design cycle as it is the synthesizer that uses up most of the time required to produce a working design.

## **5.2 Calculation of delays**

The end to end path delays which are obtained from the report generated by the Synopsys Design Compiler, is the average of the maximum rise and maximum fall times for a maximum delay model. The software backann2, though, obtains both the maximum and minimum values and uses only one of them. The average of the values obtained from the report is calculated manually and fed in into the file and the remaining software is executed. The calculation of the average path values can be automated.

It is seen that, in a model, if any of the processes has primary inputs and primary outputs, as its inputs and outputs, the generic delays obtained have lesser error than the generic delays obtained for models which does not have such inputs and outputs. This is because the initial value of the generics, for the later case, are taken to be the individual process delays obtained from backann. Instead if linear programming methods are used such that these values are limiting values, better results may be obtained. One such method is the simplex method [34].

## **5.3 Calculation of Generics for Class 4 Circuits**

The algorithm discussed will work only for the first three classes of circuits. This will not work for circuits with feedback loops or irregular register structures. The principle for calculating the generic delay values is different. Other linear programming

techniques like the above mentioned Simplex method have to be used to calculate these delays. However the path enumeration part of backann2 will work for Class 4 circuits also.

## **Chapter 6**

### **Conclusion**

This thesis has presented a back-annotation tool - Backann2. The working of Backann2 to back annotate timing delays has been explained. The results obtained have been presented and were analyzed to show that the tool achieves its purpose with reasonable accuracy. This tool is an important aid in the development of behavioral models with back-annotated timing delays. This tool thus helps quicken the design cycle from concept to silicon.

## REFERENCES

- [1] D. D. Gajski, N. D. Dutt, A. C-H Wu, S. Y-L Lin, *HIGH-LEVEL SYNTHESIS - Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.
- [2] Apostolo Dollas and Nick Kanopoulos, "Reducing time to market through Rapid Prototyping", *IEEE Computer*
- [3] J. R. Armstrong and F. G. Gray, *Structured Logic Design with VHDL*, Prentice Hall, 1993.
- [4] *IEEE Standard VHDL Language Reference Manual*, 1988.
- [5] L. Maliniak, "Synthesis Tools Move Into the Mainstream," *Electronic Design*, August 1991.

- [6] J. R. Armstrong and D. G. Burnette, "A Systematic Approach to Chip Level Modeling with VHDL," *WESCON 89*, pp.333-338, Nov 1989.
- [7] J. R. Armstrong, " ", *SIGDA Newsletter*, vol. 18, pp.72-75, Dec 1988.
- [8] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, New York : North Holland, 1976.
- [9] D. Giles, C. Berking, K. Wacks, "Integrated Functional/Structural Timing for Digital Simulation," *IEEE Test Conference*, pp.153-160, April 1982.
- [10] N.D.Dutt,D.D.Gajski,"Design Controlled Behavioral Synthesis," *26th ACM/IEEE Design Automation Conference*, 1989, pp754-757,
- [11] N.D.Dutt, T.Hadley, D.D.Gajski, "An Intermediate Representation for Behavioral Synthesis," *27th ACM/IEEE Design Automation Conference*,1990,pp-14-19
- [12] A. Gadagkar and J.R.Armstrong, "Timing Distribution in VHDL behavioral Models," *Proceedings of ICCAD 1992*, pp. 82-89.
- [13] A. Gadagkar, "Timing Distribution in VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.
- [14] B. Singh, "A Parametrized CAD tool for VHDL Model Development with X-Windows," Master's Thesis, Virginia Polytechnic Institute and State University,

1990.

- [15] P. A. Wright, "Rapid Development of VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.
- [16] S. Narayanaswamy, "Development of VHDL behavioral Models with Back Annotated timing," Master's Thesis, Virginia Polytechnic Institute and State University, 1993.
- [17] S. R. Rao, "A Hierarchical Approach to Effective Test Generation for VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.
- [18] William K.C. Lam, Robert K. Brayton, Alberto L. Sangiovanni-Vincentelli, "Circuit Delay Models and Their Exact Computation Using Timed Boolean functions," *DAC 93*, pp128-134.
- [19] Meng-Lin Yu and P.A.Subrahmanyam, "A Path-Oriented Approach for Reducing Hazards in Asynchronous Designs," *DAC 92* pp239-244.
- [20] Elizabeth A. Walkup, Gaetano Borriello, "Interface Timing Verification with Application to Synthesis," *DAC 94*, pp106-112
- [21] A.Stoll,J.Biesenack, S.Rumler, "Flexible Timing Specification in a VHDL Synthesis Subset,"*European Design Automation Conference 92*, pp610-615

- [22] Paul D. Lindemann, "Top-Down Design Synthesis Using VHDL," *Wescon 1990*, pp.382-383.
- [23] E. Meyer, "VHDL strives to cover both synthesis and modeling," *Computer Design*, Oct 1989, pp.42-45.
- [24] E. A. Rundensteiner and D Gajski, "Functional Synthesis using Area and Delay Optimization," *29th ACM/IEEE Design Automation Conference*, pp.291-296.
- [25] CAD Language Systems, *VHDL Analyzer Designer's Manual*, April 1993.
- [26] CAD Language Systems, *Design Library System*, April 1991.
- [27] CAD Language Systems, *DLS Application Development : The Software Procedural Interface*, March 1993.
- [28] Synopsys Inc., *Command Reference* , Dec 1992.
- [29] Synopsys Inc., *Design Compiler Reference Manual*, Dec 1992.
- [30] F. Vahid, S. Narayan, D. D. Gajski,, "SpecCharts: A Language for System Level Synthesis," *Proceedings of IS* 1991, pp. 165-174.
- [31] J. Lahti, J. Kivela, "Logic Compilation from Graphical Dependency Notation,"

*ICCAD 90*, pp.474-477, Nov 1990.

[32] i-Logix, Inc., *Statemate User Reference Manual - Volume 1*, June 1993.

[33] John F. Wakerly, *Digital Design Principles and Practices*, Prentice Hall, 1990

[34] Cooper and Steinberg, *Methods and Applications of Linear Programming*, W.B. Saunders Company

## Appendix A

This User Guide details procedures that have to be performed by a designer to obtain the back-annotated model using Backann2.

### Behavioral Model Creation

The designer creates a behavioral model using the Modeler's Assistant . After creating the model the designer has to save the VHDL code generated by the Modeler's Assistant in a file, using the VHDLDump option in the Unit menu.

### CLSI-VTIP\_VHDL Analyzer

The behavioral model has to be analyzed using the CLSI VHDL analyzer. The analyzer stores the VHDL model in the form of a data tree which is accessed by backann2. The command used to analyze the model is

```
vhdl -preserve <VHDL-filename>
```

Before invoking the CLSI-VTIP analyzer, the work library in which the data is stored has to be created.

## Creating the log\_file

The user has to create two log\_file which contains the specifications and constraints to be adhered to by the Synopsys Design Compiler. These files are named

`<entity_name>.log`

`<entity_name>.log1`

The additional details in the second file that should be mentioned are

1. group -hdl\_all\_blocks which groups all the logic inside a process and optimizes it
2. The report\_timing command for maximum rise and maximum fall from all input ports to all output ports.

The group -hdl\_all\_blocks should be written before the compile line. The compile line is the last line of the first log file. The report\_timing commands should be written after the compile line in the second log file. A typical log\_file of the second type is shown in Figure 13.

## Invoking backann2

Backann2 is invoked with the name of a VHDL file as input. The format for invoking backann2 is

`backann2 <VHDL_filename>`

Once backann2 is executed, files TMPREP.OUT and PATH\_CAL.OUT will be obtained. The TMPREP.OUT file contains the end-to-end delays obtained from the Synopsys Design Compiler. The delays obtained are for maximum rise and maximum fall times for each path. The average of the maximum rise and maximum fall has to be calculated manually and should be written into PATH\_CAL.OUT. A typical PATH\_CAL.OUT file

will be like the inputs to MATLAB for different models shown. The first set of equations in the PATH\_CAL file is the maximum rise time for each path. These values should be replaced by the average values calculated manually.

Once PATH\_CAL.OUT is edited, the remaining part of backann2 is run using the command *delays*. The output of this is the back-annotated VHDL file. This file is named

*<VHDL\_filename>.new2*

This is stored in the directory from which backann2 is invoked.

## **Vita**

Gayatri Mahadevan was born on the 7th of August, 1971 in India. She graduated with a Bachelor of Engineering degree in Electronics and Communication Engineering from the Government College of Technology, Coimbatore, India in May 1992. She attended graduate school at Virginia Polytechnic Institute and State University and received a Master of Science degree in Electrical Engineering in June 1995. She has been employed with LSI Logic Lo, Milpitas, California since June 1995.