Development of VHDL Behavioral Models

with Back Annotated Timing

by

Sathyanarayanan Narayanaswamy

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

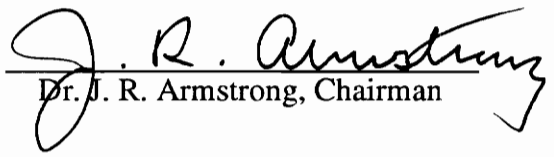
in partial fulfillment of the requirements for the degree of

Master of Science

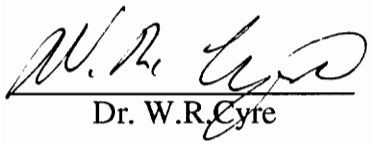
in

Electrical Engineering

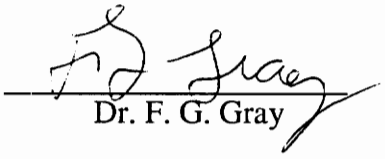
APPROVED:



Dr. J. R. Armstrong, Chairman



Dr. W.R.Cyre



Dr. F. G. Gray

February 1994
Blacksburg, Virginia

Development of VHDL Behavioral Models

with Back Annotated Timing

Sathyanarayanan Narayanaswamy

Dr. James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

This thesis describes the development of BACKANN, a tool for the back annotation of timing delays into VHDL models. BACKANN uses the Process Model Graph and the VHDL behavioral model generated by the Modeler's Assistant as the base for back-annotation. BACKANN determines the delay values that are required for the signal assignments in the behavioral model. It generates a 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. BACKANN is thus a design automation tool that helps the development of VHDL behavioral models with realistic timing and thus quickens the design cycle.

To Amma

Acknowledgements

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

I would also like to thank Dr. Walling R. Cyre and Dr. F. Gail Gray for serving as members on my commitee.

I would also like to thank my family and all my friends who have stood behind me every step of the way.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

Introduction

The field of VLSI design has undergone rapid technological changes and the complexity of circuitry that can be built on a single chip has been growing at an enormous rate. The number of transistors that can be built on a chip has reached the millions. This increase in complexity has resulted in the design of a chip at the transistor level being virtually impossible [1]. The duration of the product development cycle, from forming the concept, to fabricating the chip on silicon is also important and has to be kept short to maintain competitiveness. Architectural level iterations which are time-consuming have become unacceptable [2]. So, designers have had to follow various other approaches to design chips, and their needs have been matched by advancements in design automation, resulting in the availability of a number of software tools which make the design cycle quicker and easier. These software tools make it possible to simulate and synthesize designs at abstraction levels that are closer to human reasoning [3]. Hardware description languages have played an important role in this transformation. HDLs like the VHSIC (Very High Speed Integrated Circuit)

Hardware Description Language (VHDL) allow designers to create models of chips at various levels of the design hierarchy ranging from the system level to the switch level [4].

VHDL models can be created by the designer to describe the chip at the behavioral level, which is at a much higher plane in the design hierarchy than the transistor level and thus makes design quicker and easier. These models can, and should, be created such that they accurately represent the timing and functionality of the design. The models accurately represent the design and the simulation of them closely parallels the working of the physical implementation of the design. Thus the creation of accurate VHDL models for any design was one of the primary objectives for developing the software tool BACKANN, that back-annotates realistic timing delays into VHDL behavioral models. This work describes and discusses the design and implementation of the tool.

1.1 VHDL Modeling

The designer first creates the behavioral models using VHDL. Then they are simulated and tested by the designer to see if they produce the correct results and satisfy the constraints specified. Then the VHDL representation is synthesized and verified again before being transferred to silicon [5]. When a behavioral description is constructed by the designer using VHDL, the architecture body specifies the behavior of the system in terms of inputs and outputs and the relationships between them. This architecture body may consist of one or more processes, procedures or functions. An entity declaration is used to define the interface to and from a design. The

entity declaration specifies the name of the entity, the port clause lists the ports and the generic clause lists the generics. The actual behavior of the system is defined in the architecture body. The architecture body may consist of process blocks and signal assignment statements. In VHDL a signal is assigned a value using a signal assignment statement. 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. If a delay is not specified, the signal is assumed to be assigned a new value an infinitesimal amount of time later, called delta delay. If a delay value is specified, it can be in terms of a number or a generic or an equation [6,7]. Generics are constants that can be declared in the component declaration or in the entity declaration [8], and can be assigned values through component instantiations and configuration specifications. The generics used in signal assignment statements are of type TIME. The modeling style favored by many designers is to specify the delays for the signal assignments in terms of these generics. 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 technology without having to change the functionality of the model. When the design is simulated these generics are supplied values by the designer, depending upon the technology and implementation of the chip.

1.2 Task Definition

The accuracy of the simulation of a VHDL behavioral model created using generics to specify timing depends upon the accuracy of the values specified for the generics. 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]. The design of a complex system becomes a time consuming and labor intensive task if the designer has to calculate the different delays by hand. It is also very difficult to calculate these values without a view of the physical representation of the design on the chip, since the behavioral description is at a more abstract point of view. Thus the delay in the real chip might be different from the initial estimate. If the model is simulated for timing verification using the wrong values, it might work, but when the chip is fabricated, unanticipated timing hazards and conflicts may cause incorrect operationa. The problem thus is to supply the high level abstract model with accurate generic values so that the high level simulation is as accurate as possible [10,11].

1.3 Contributions

The software tool BACKANN was written to back-annotate realistic timing delays into VHDL behavioral models. It was developed on a Sun SPARCstation platform and written in Sun C. It uses other software like the graphical CAD tool Modeler's Assistant, the Synopsys Design Compiler and the CLSI VHDL Tool Integration Platform and Design Library System. It also uses a software routine called **extract** developed for the Hierarchical Behavioral Test Generator [12,13], to interface with the Modeler's Assistant. It uses the CLSI software tools to parse the VHDL description. BACKANN uses the Synopsys Design Compiler to synthesize the VHDL description to the gate level.

Complex data structures were defined to store the information extracted from the VHDL source file, like the timing values required and the delay values obtained from the gate level design created by the Synopsys Design Compiler. The data structures

have been defined using programmer defined header files and are explained in Appendix B.

BACKANN consists of about 4000 lines of Sun-C code developed independently and about 500 lines of modified Sun-C code from the **extract** routine of the Hierarchical Behavioral Test Generator. The software is well documented and commented to facilitate further development.

1.4 Contents

Chapter 2 discusses the Process Model Graph (PMG) and VHDL modeling using the PMG.

Chapter 3 discusses the graphical CAD Tool Modeler's Assistant and model development using it.

Chapter 4 discusses the process of developing the algorithm for BACKANN. It discusses the different options available and also the different methods by which the exact timing delays are calculated. It also discusses the nature of BACKANN.

Chapter 5 discusses the results obtained from BACKANN.

Chapter 6 discusses future developments possible and discusses ceratin inadequacies that arise because BACKANN has to work with other software designed for different tasks.

Chapter 7 states the conclusions drawn from the work done.

Chapter 2

VHDL Modeling and the Process Model Graph

VHDL, a language for hardware description, has been standardized by the IEEE [8]. It has made the textual representation of any design an easy task. It can be used to describe and simulate a system at most levels of the abstraction hierarchy. This abstraction can be either in the structural domain or in the behavioral domain. In the structural domain the system is described in terms of lower-level components and the interconnections between them. In the behavioral domain the system is described by defining the inputs and outputs and the relationships between them [4].

2.1 Behavioral Modeling

When the designer creates a behavioral model of the system, the inputs and outputs to the system are defined in the interface description in the entity declaration. The inputs and outputs defined in the entity declaration are called *ports* [8].

Generics, which were discussed before, are also declared as part of the entity declaration. The actual behavior of the system, specifying the relationships between the inputs and the outputs, is a part of the architecture body. The architecture body consists of one or more processes which run concurrently. These maybe either process blocks or various forms of the signal assignment statement[8]. These process blocks in turn contain signal assignments, loops and other constructs that characterize the input-output relationship in the system.

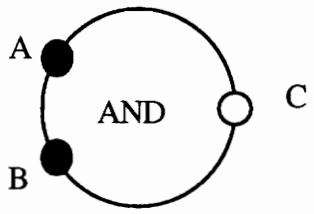
The behavioral description of a system may itself be of either the algorithmic type or the data flow type. In an algorithmic behavioral description the manner in which the relationships between the inputs and outputs are characterized does not correspond to any particular physical implementation. In a dataflow description the relationships match those in the actual physical implementation. It shows the transfer of data across the different components, as the name implies [4].

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 process may be speeded up by using pictorial representations combined with text. The pictorial representations make it easier to specify the interconnections between different components in the system. In a following section we shall look at the Process Model Graph which is a pictorial representation of a system and which aids in the quick development of VHDL behavioral models [4].

2.2 Process Model Graph

As discussed above, the architecture body of a VHDL description consists of a number of concurrently running processes. The architecture body can be represented pictorially as a Process Model Graph. A Process Model Graph is a graph whose nodes represent the various processes in an architectural body and whose arcs represent the various signals that are transferred between the processes [4].

Every process in the architectural body is defined as a node in the graph and is represented 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 signals will be referred to as process ports in the rest of the work. The signals which are in the sensitivity list of the process are shown as filled bubbles [15]. The VHDL description of a simple process and its pictorial representation are shown in Figure 1.




```
process(A,B)
begin
        C <= A and B after AND_DEL;
end process;
```

Figure 1 VHDL description and pictorial representation of an AND gate

A typical Process Model Graph consists of a number of circles similar to the one shown in Figure 1 in which each circle represents a process. Each process circle in the graph has a VHDL behavioral description associated with it describing the behavior of the corresponding process in the architecture body. The interconnections between the different processes represent the transfer of signals between the different processes and are specified by connecting the process ports on different processes by the arcs of the graph [15].

A Process Model Graph can be of two types - functional or physical. A functional type Process Model Graph is shown in Figure 2. In this type of graph, the nodes do not represent any particular piece of hardware but instead represent the major functions of the system [4].

A Process Model Graph that represents a model with a physical partitioning is shown in Figure 3. In this type of graph, the nodes represent hardware components such as multiplexers or logic gates , as shown in Figure 3.

We thus see that a Process Model Graph can pictorially represent a VHDL behavioral model, clearly illustrating the various relationships between the signals in the design. There are other graphical representations for behavioral models [16,17,18] but none is as suitable as the Process Model Graph for the task of back-annotation of timing delays.

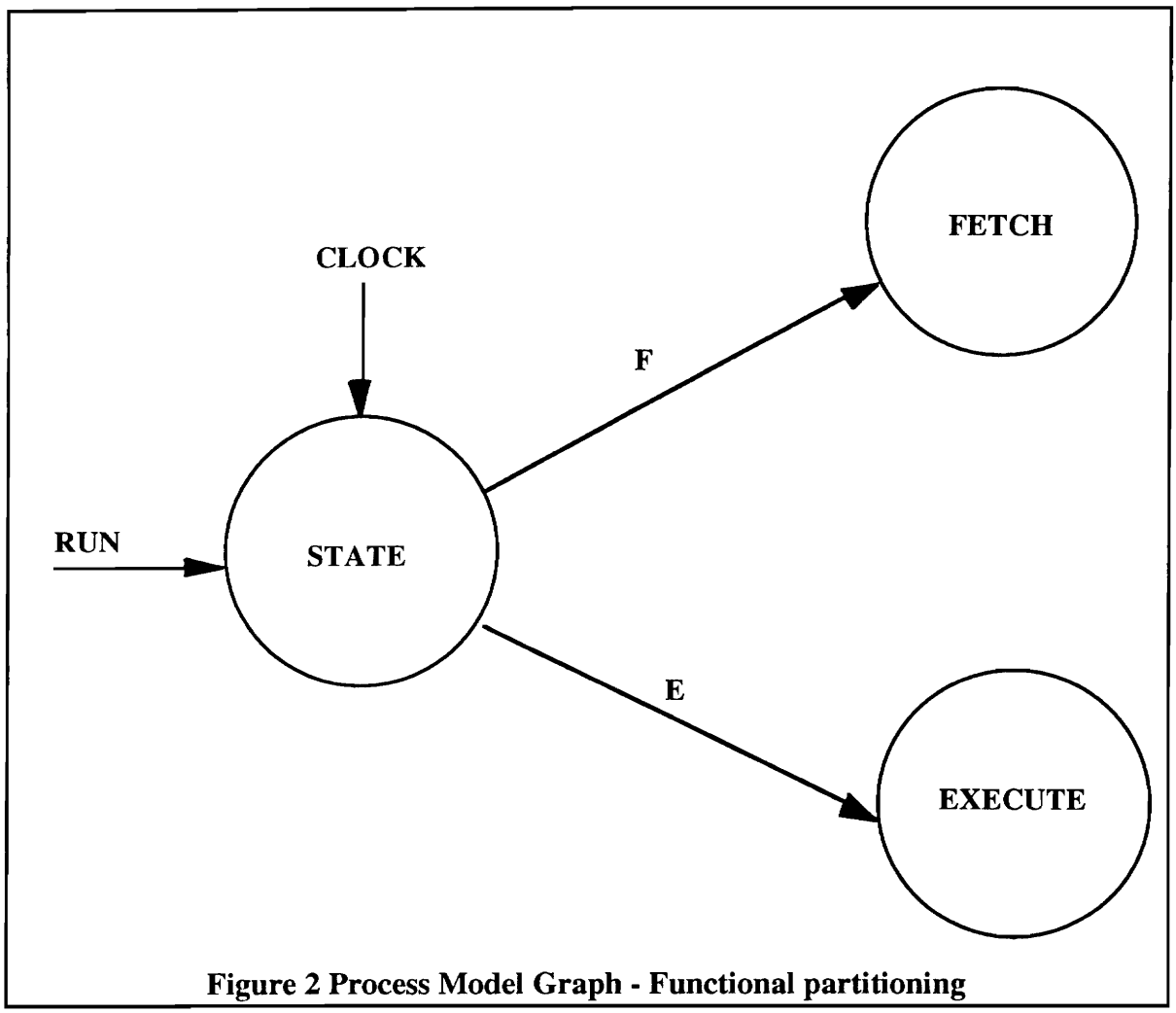


Figure 2 Process Model Graph - Functional partitioning

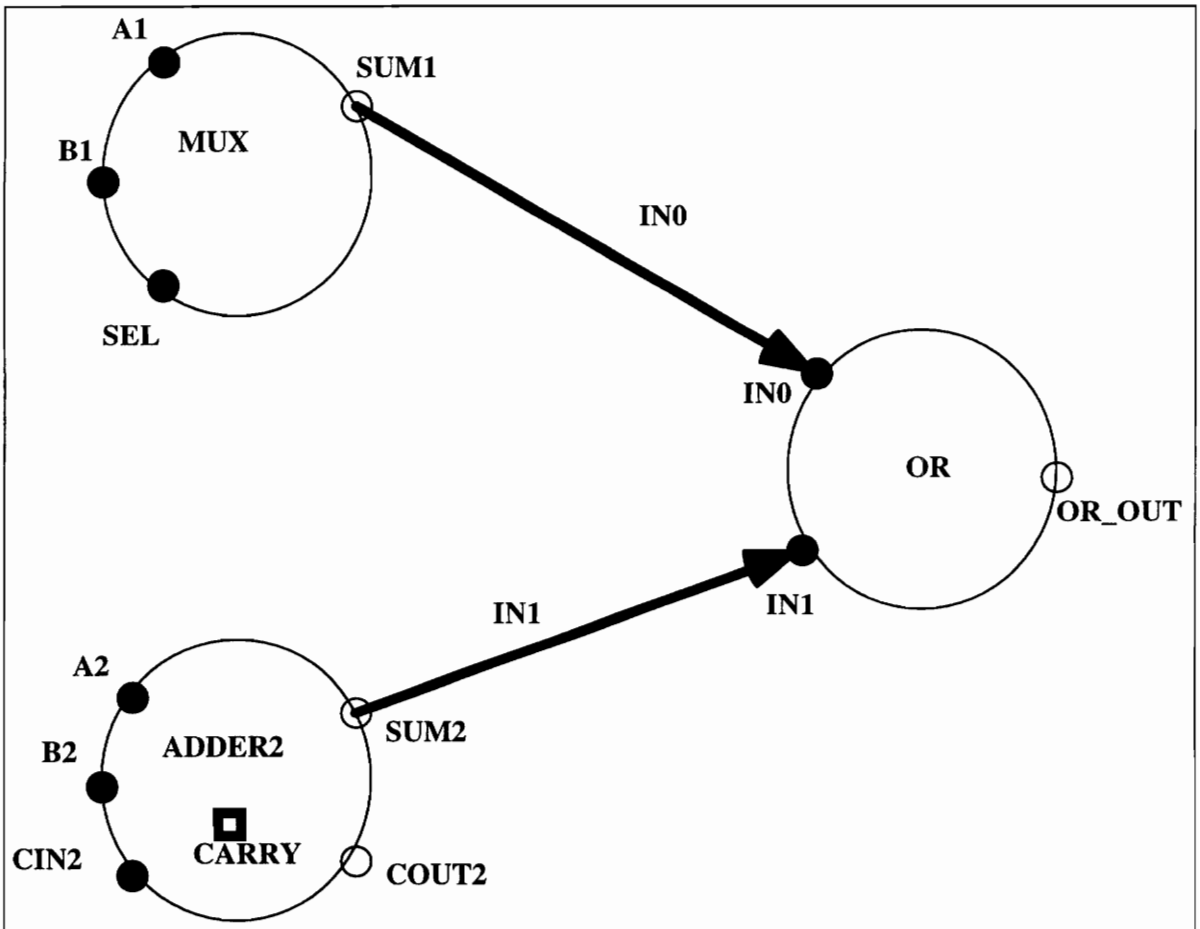


Figure 3 Process Model Graph - Physical partitioning

Chapter 3

Modeler's Assistant

The Modeler's Assistant is an X-Windows based graphical CAD tool that aids in the development of VHDL behavioral models. The tool uses the Process Model Graph described in Chapter 2 as the basis for the creation of VHDL behavioral models.

3.1 Model Creation

The Modeler's Assistant is a tool that makes the creation of behavioral models easy and error-free. It presents the designer with a series of point-and-click menus which prompt for user interaction. The responses from the designer combined with textual input from him or her are used by the Modeler's Assistant to create a VHDL behavioral model [19,20].

The development of a VHDL behavioral model starts with the designer first creating the processes required. These processes are the ones that are not a part of the Modeler's Assistant's builtin primitive library [21,15]. The creation of these user-defined processes is done by clicking and selecting the menus to create a process, naming it and then adding the different ports to it. The designer can also add variables, generics to the process. Then the behavioral description of the process is entered as text by the designer. VHDL as a language was developed primarily for simulation and not all constructs of VHDL are synthesizable [22,23]. The description thus written for the process should be synthesizable, for BACKANN to be able to obtain a gate level model using the Synopsys Design Compiler.

After creating all the processes required for the system, the designer creates the entity or unit which represents the whole system. This is done by adding processes (user-defined and built-in) to build the entity. The processes are then interconnected by adding signals which are represented by the lines between the circles in the Process Model Graph [19,20].

In creating a unit or entity the designer may use any combination of user-defined processes and primitives from the built-in library. These processes are placed on the screen as shown in Figure 4. The Modeler's Assistant thus helps the designer create a design which is divided into its major subcomponents and where the functionality of each is defined using VHDL [32].

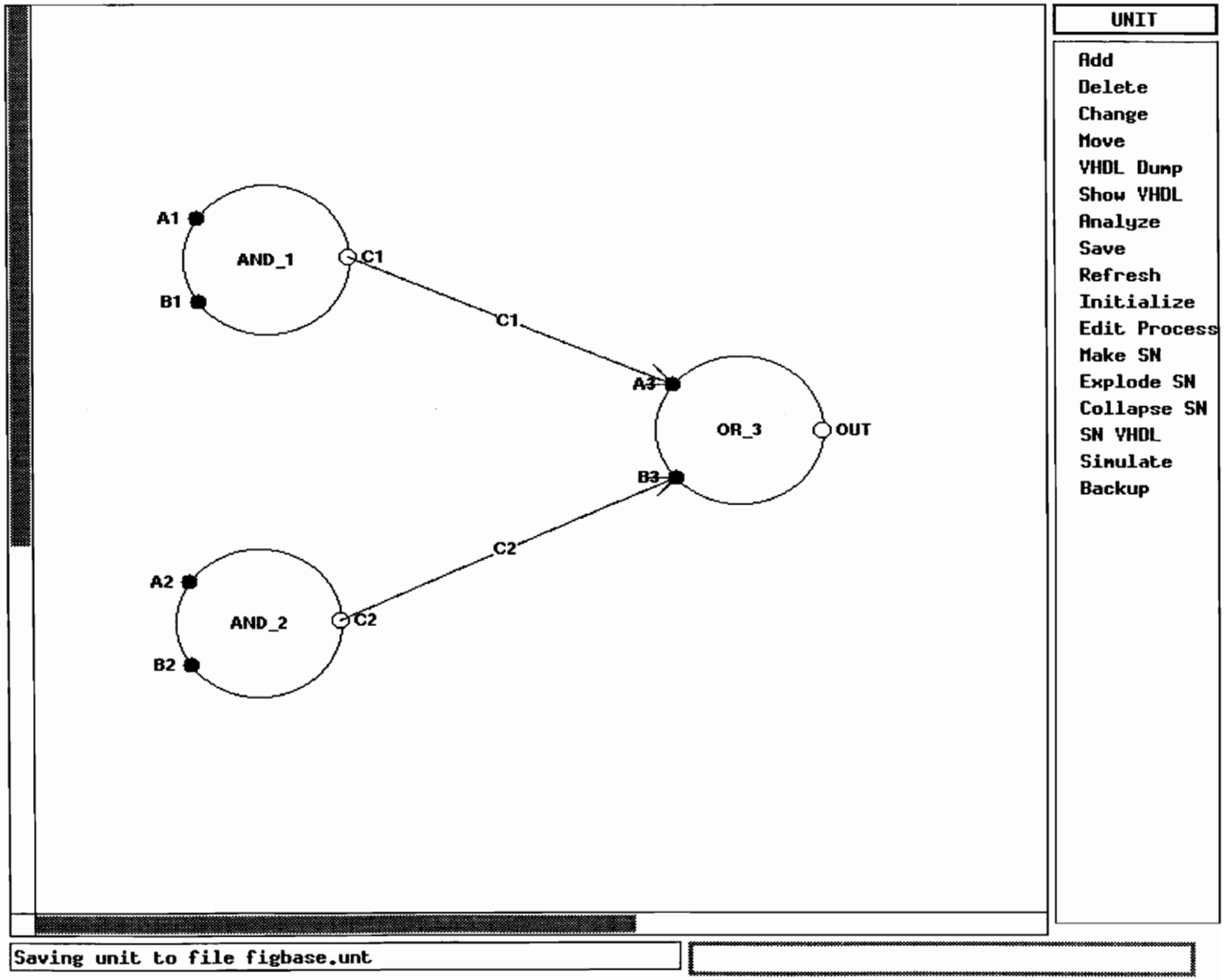


Figure 4 Modeler's Assistant - Process Model Graph

3.2 The Modeler's Assistant database

The Modeler's Assistant stores the structure of the PMG and its elements in the form of a linked list [24,25]. Each element in the linked list is defined by a data-struture ***a_node*** which is defined as

```
typedef struct
        {
                char name[32];  /* Name of the object */
                int type;       /* Type of the object such as process, port etc.,
                int ptr[6];     /* Points to structures defining object elements
                                   depending on type of object */
                rect R;         /* Position of the object in the display */
        } a_node;
```

For the Process Model Graph shown in Figure 4 the Modeler's Assistant database would be as shown in Figure 5.

The linked list of information about the Process Model Graph is accessed using a software routine called **extract** written for the Hierarchical Behavioral Test Generation software [12,13]. This software was modified to suit BACKANN and is used to extract individual processes from the Process Model Graph and then to construct the entity declaration for each of these processes. To construct the entity declaration, the modified **extract** routine returns the names of the process ports, and if any of the process ports are connected to signals from other processes, the name of that signal is also returned.

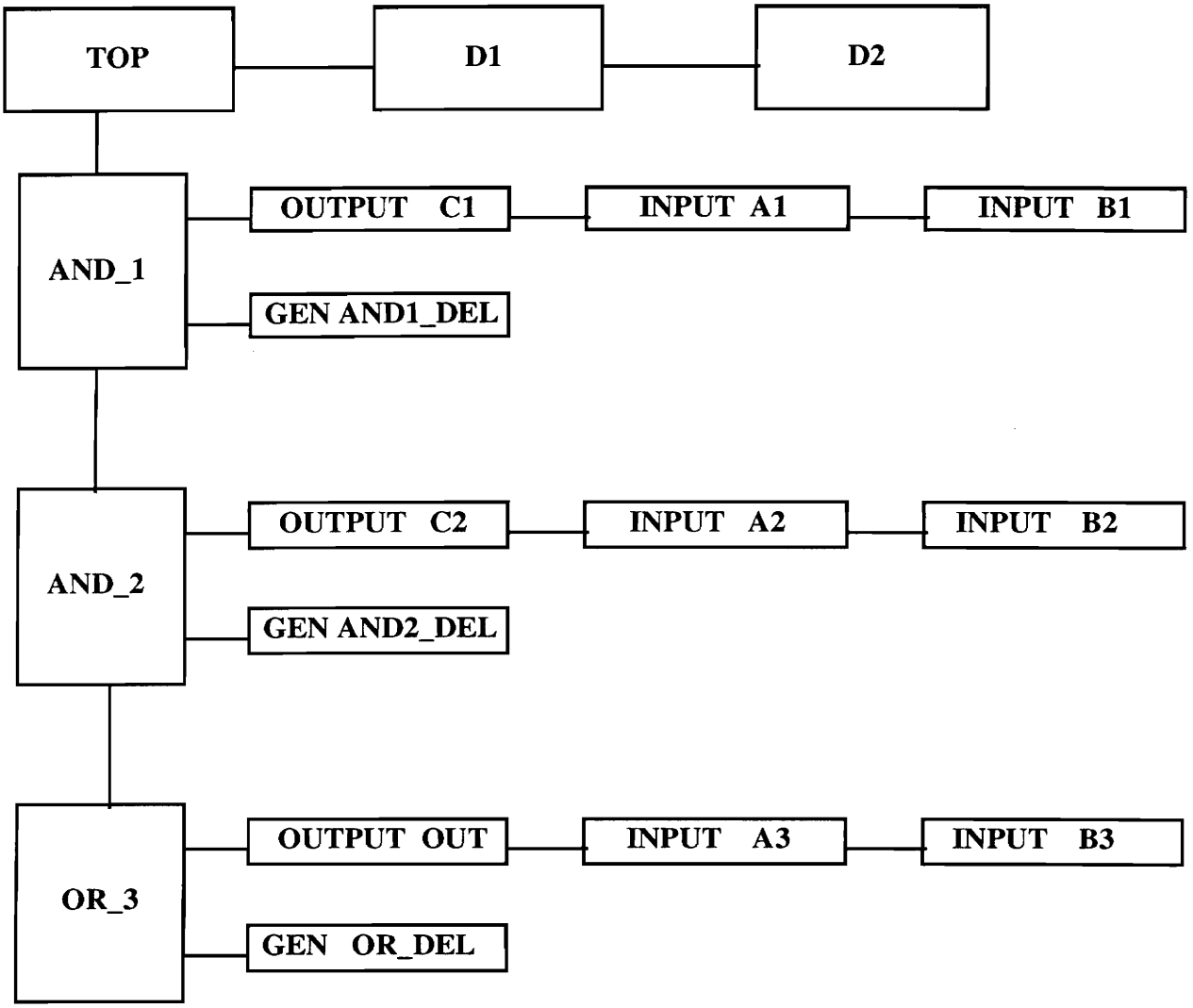


Figure 5 Database of a PMG in the Modeler's Assistant

Chapter 4

Algorithm Development

As discussed in Chapter 1 the task is to calculate accurate timing delays for the different signal assignment statements in a behavioral VHDL model. In this chapter we discuss the development of the algorithm used to achieve this purpose.

4.1 Synthesizer- Synopsys Design Compiler

The exact delays that occur in a design can be calculated only by obtaining a physical realization of the design. Thus the behavioral description has to be transformed to a physical level design. Synthesis is the process by which we transform a representation from one level in the abstraction hierarchy to another [4]. In this case, we transform the design from a VHDL behavioral description to a gate-level design [4,26]. This is achieved by using a synthesizer. A synthesizer is a tool that takes a high-level

language description of a system and after going through a number of stages, like scheduling, allocation etc., produces a gate-level design [26].

BACKANN uses a commercially available synthesizer called the Synopsys Design Compiler. The Synopsys Design Compiler is a set of CAD tools that can transform a VHDL description to a gate-level design, that conforms to the design goals specified by the designer [14,27].

4.2 Synthesis of a Process Model Graph

The basis of our VHDL behavioral model is the Process Model Graph explained in Chapter 2. Let us consider the Process Model Graph shown in Figure 6. It consists of a simple circuit with two adders whose SUM outputs go to a 2-to-1 multiplexer. Each component (the two adders and the 2-to-1 multiplexer) is a process and is represented as a node in the Process Model Graph. The VHDL description for this model is given in Figure 7. In this example, all the signal assignments have delta delays. Signal assignments with numerical and generic delays are handled differently and are explained later. In this model we need the delays from A1, B1 and CIN1 to SUM1 and COUT1, A2, B2 and CIN2 to SUM2 and COUT2 and from SUM1 (or IN0) and SUM2 (or IN1) to Q. If these delays are back-annotated into the VHDL behavioral description, we get a model with accurate timing information built into it.

This VHDL description was input to the Synopsys Design Compiler and the design goals were specified. The gate level design generated by the Synopsys Design Compiler is shown in Figure 8. If we invoke the Synopsys Design Compiler to analyze the timing in the circuit and then return the delay from A1 to SUM1,

analyze the timing in the circuit and then return the delay from A1 to SUM1, the Synopsys Design Compiler reports back that signal SUM1 cannot be found in the gate level design as shown in Figure 8.

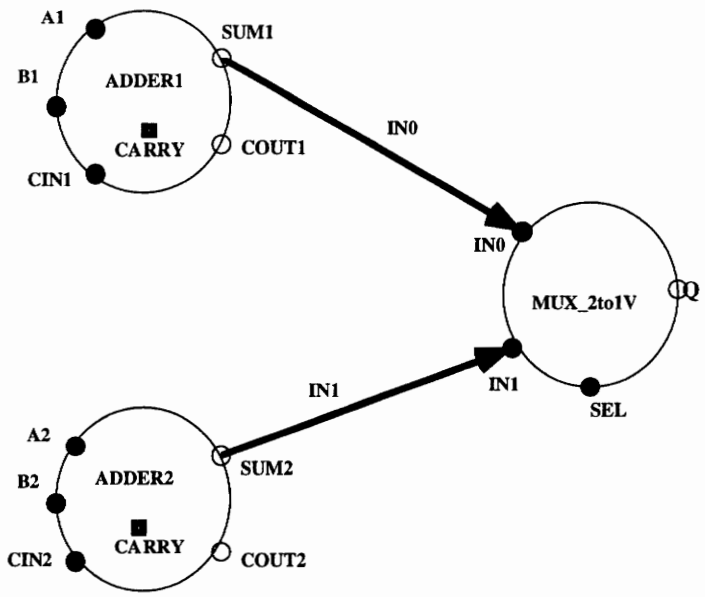


Figure 6 Process Model Graph - two adders and a multiplexer

```
entity SIGNAL_MISS is
        port (Q: out BIT_VECTOR(7 downto 0);
        SEL: in BIT;
        COUT2: out BIT;
        CIN2: in BIT;
        B2: in BIT_VECTOR(7 downto 0);
        A2: in BIT_VECTOR(7 downto 0);
        COUT1: out BIT;
        CIN1: in BIT;
        B1: in BIT_VECTOR(7 downto 0);
        A1: in BIT_VECTOR(7 downto 0));
end SIGNAL_MISS;
-- ********************************************************

architecture BEHAVIORAL of SIGNAL_MISS is

        signal SUM2: BIT_VECTOR(7 downto 0);
        signal SUM1: BIT_VECTOR(7 downto 0);
begin

 -- ------------------------------------------------
 -- Process Name: MUX1
 -- ------------------------------------------------
        MUX1_4: process (SEL,SUM2,SUM1)
        begin
                case SEL is
                        when '0' => Q <= SUM1 ;
                        when '1' => Q <= SUM2 ;
                 end case;

        end process MUX1_4;

  -- ------------------------------------------------
 -- Process Name: ADDER2
 -- ------------------------------------------------
        ADDER2_11: process (CIN2,B2,A2)
        variable CARRY: BIT;
        begin
                CARRY := CIN2;
                for I in 0 to 7 loop
                        SUM2 ( I ) <= A2(I) xor B2(I) xor CARRY ;
                        CARRY := ((A2(I) and B2(I)) or (A2(I) and CARRY) or (B2(I) and
                                        CARRY));
                 end loop;

                COUT2 <= CARRY ;

        end process ADDER2_11;
```

Figure 7 VHDL description of an entity with two adders and a multiplexer

```
-------------------------------------------- --
Process Name: ADDER1
 -- ------------------------------------------------
        ADDER1_20: process (CIN1,B1,A1)
       variable CARRY: BIT;
       begin
              CARRY := CIN1;
              for I in 0 to 7 loop
                     SUM1 ( I ) <= A1(I) xor B1(I) xor CARRY ;
                     CARRY := ((A1(I) and B1(I)) or (A1(I) and CARRY) or (B1(I) and
                                   CARRY));
              end loop;
              COUT1 <= CARRY ;
       end process ADDER1_20;
end BEHAVIORAL;
```

Figure 7 VHDL description of an entity with two adders and a multiplexer (continued)

This disappearance of intermediate signals is due the optimization performed by the Synopsys Design Compiler. The Synopsys Design Compiler tries to minimize the gates required in the circuit and makes the different processes share the same hardware, wherever possible, and this results in a number of intermediate signals specified in the Process Model Graph being eliminated in the gate level design. The delays the Synopsys Design Compiler returns are only from the inputs like A1, B1, A2, B2 etc., to the outputs COUT1, COUT2, Q etc., i.e., only end-to-end delays can be analyzed by the Synopsys Design Compiler. However, the delays required are the delays across each process in the Process Model Graph as shown in Figure 9, whereas the delays returned would be as shown in Figure 10.

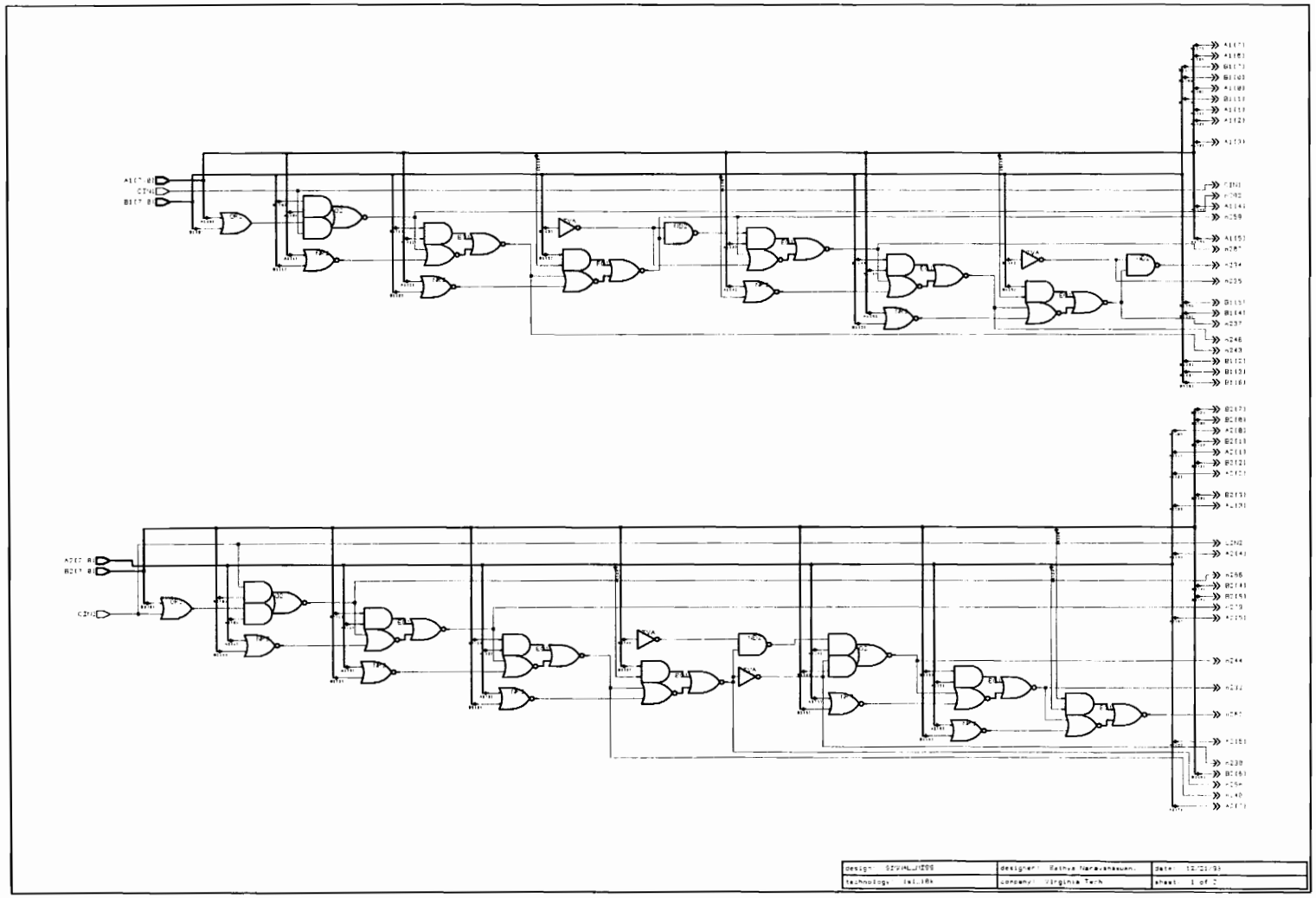


Figure 8 Gate level design of an entity with two adders and a multiplexer

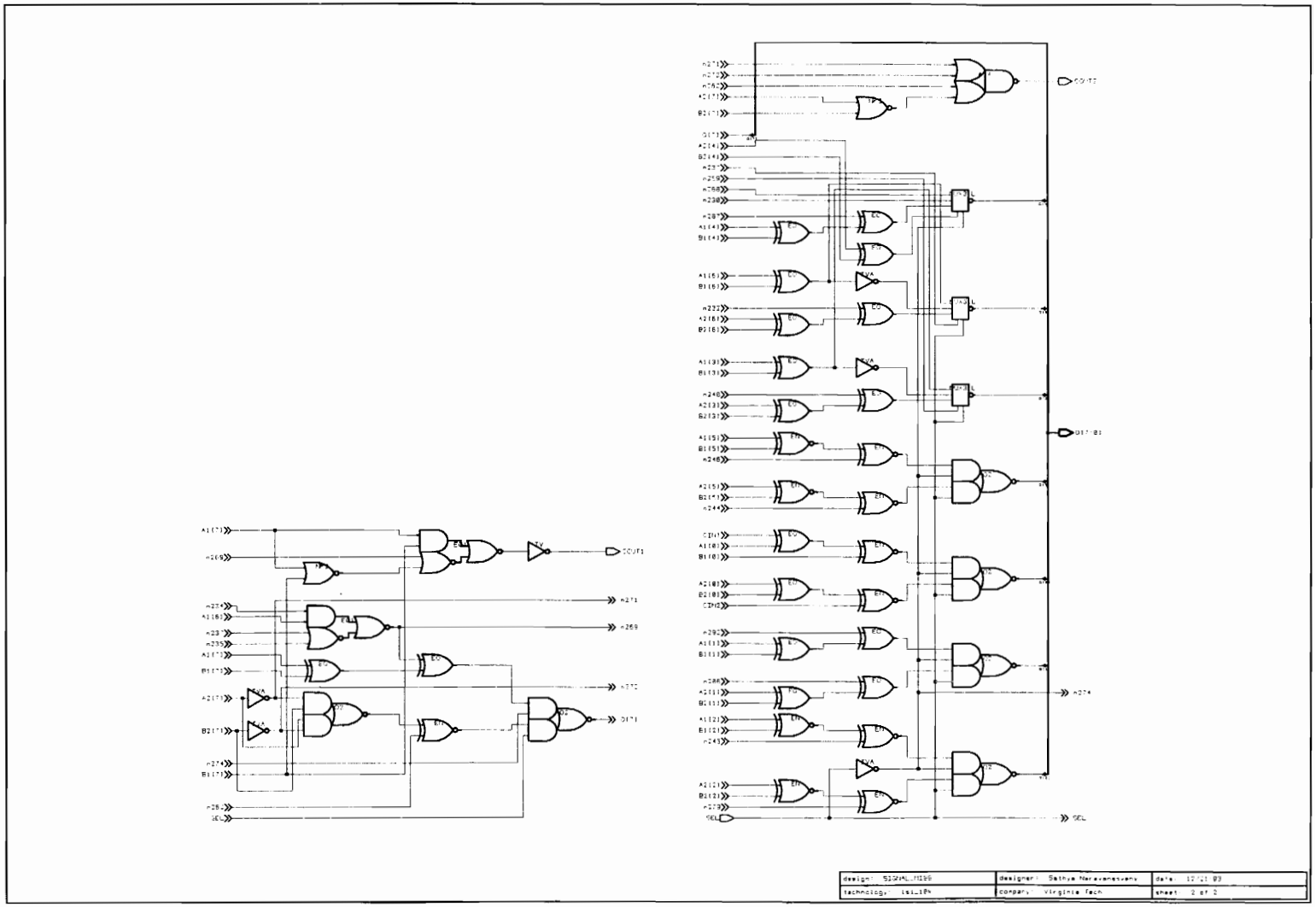


Figure 8 Gate level design of an entity with two adders and a multiplexer (continued)

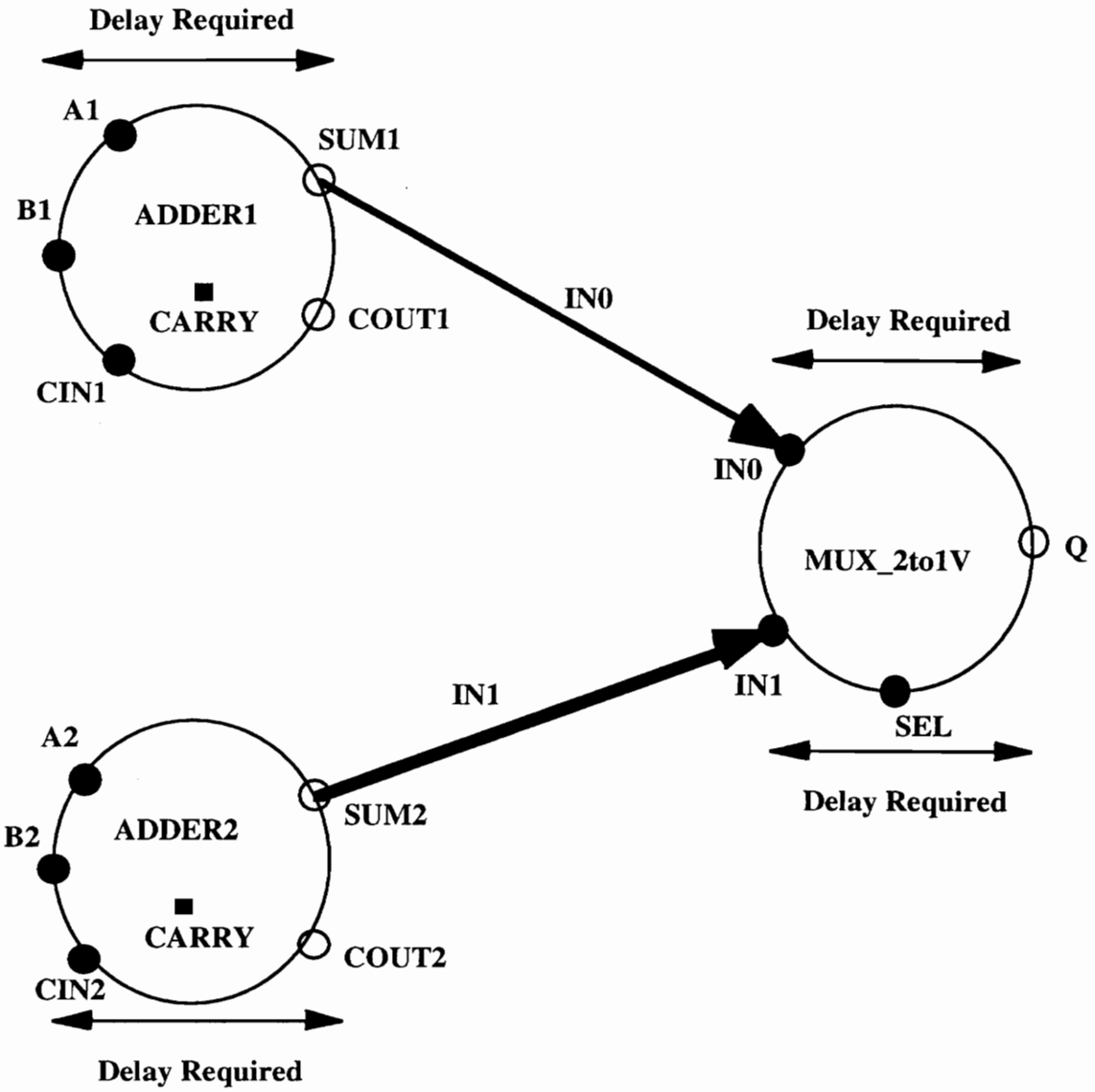


Figure 9 Process Model Graph - Delays required

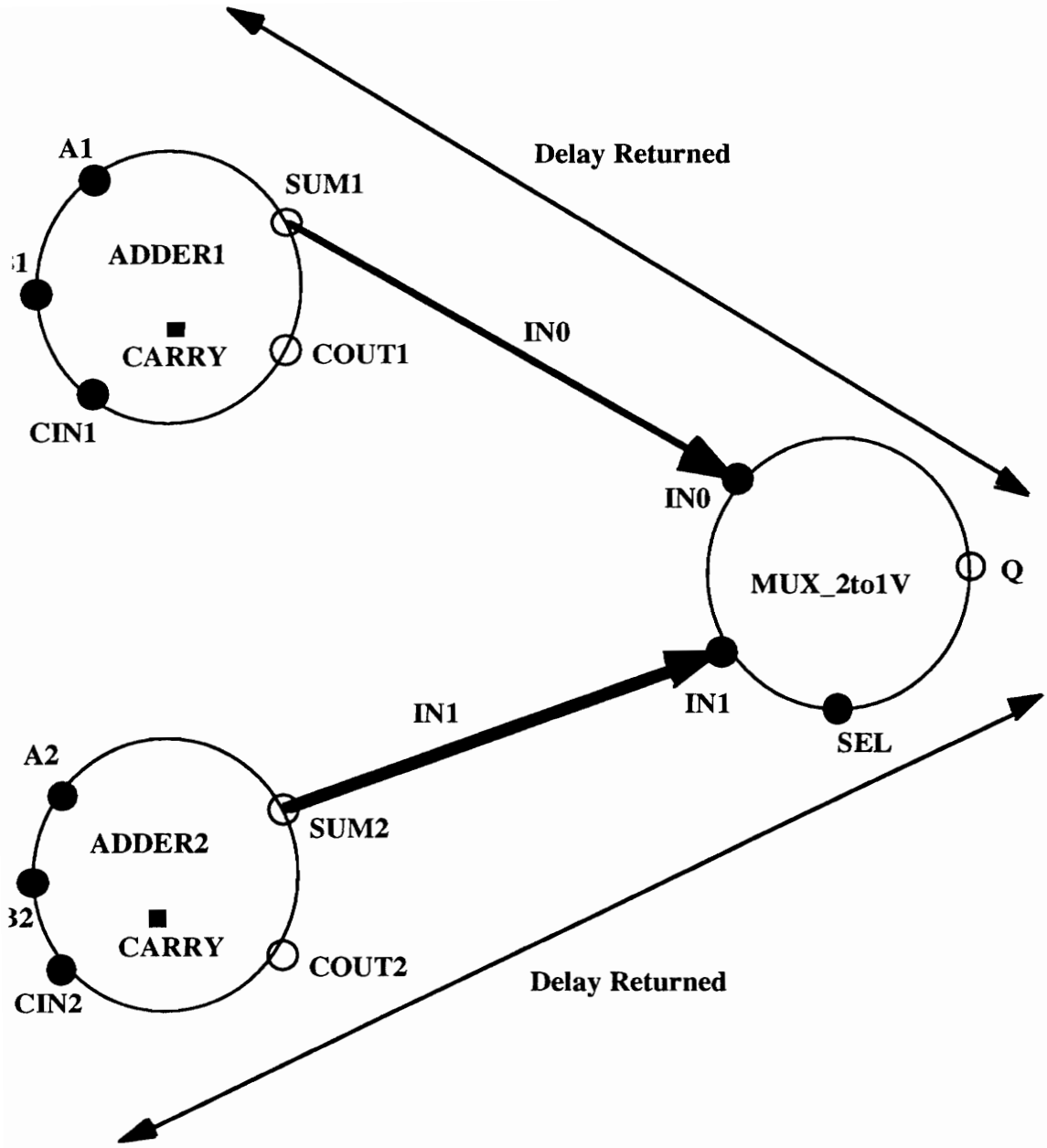


Figure 10 Process Model Graph - Delays returned

4.3 Process Extraction from the PMG

This problem of the delays returned not being the delays required, as discussed in the previous section, made it necessary to synthesize each process in the Process Model Graph individually. Thus when the gate level design is synthesized for each process, the delays across each process can be calculated accurately and no information would be missing in the back-annotated VHDL behavioral description. BACKANN extracts each process from the Process Model Graph. This is done using the software routine **extract**. This program was modified to conform to the format required by BACKANN. **extract** returns the names of the process ports and their types for every process. When the Modeler's Assistant generates the VHDL behavioral description for any Process Model Graph, if any of the process ports is tied to a signal that connects processes, the process port's name is replaced by that signal's name in the VHDL description. In the description of the Process Model Graph shown in Figure 6, the name of a process port in process ADDER1 is SUM1 but it is connected to a signal IN0 and in the VHDL description generated by the Modeler's Assistant, shown in Figure 7, SUM1 does not appear and instead, has been replaced by IN0. The extraction program returns the original name of the process port - SUM1, but also specifies that it is connected to the signal IN0. Then, SUM1 has to be automatically replaced by IN0.

To synthesize each process, it has to be stored as an independent entity. The entity declaration for each process is constructed using the process port names (replaced by signal names wherever necessary) and their types, returned by the extraction program. The VHDL description for each process is saved in temporary files. The VHDL code for each process is extracted from the behavioral description using a

commercially available parser called the CLSI- VHDL Tool Integration Platform. The workings of this parser are explained below.

4.4 VHDL parser - CLSI VTIP, DLS and SPI

The CLSI-VTIP is a set of CAD tools that are used to parse the VHDL behavioral description. The analyzer processes the VHDL description, checks it for syntactic and semantic errors and then stores it in an internal format in the Design Library System (DLS) [28,29]. The DLS stores that data in an internal format that consists of a number of internally-defined types and structures, so that it can be accessed by different tools. It consists primarily of two parts, the DLS Data Definition and the Software Procedural Interface (SPI). The DLS Data Definition specifies the different data elements, objects and structures that are supported by the DLS. The SPI is implemented in the C programming language. It consists of data types and callable routines that allow access to the DLS [30].

The parsed VHDL description is stored by VTIP in the form of a tree of data structures whose top starts with a string that specifies the name of the file in which the description is stored. The tree then moves into the EntityDeclaration or the ArchitectureBody. In the EntityDeclaration the ports of the model and their types and the different generics are stored. In the ArchitectureBody are a number of regions like the DeclarationList, StatementList, Block etc., and each of these is further subdivided into a number of subtypes. For example, in the StatementList we have StatementTypes like Assignment (which indicates a signal assignment), WaitStatement, ProcessStatement etc. In the ArchitectureBody, the tree has a StatementList, which contains a series of

ProcessStatements (for the VHDL description generated by the Modeler's Assistant). The tree is then traversed by reading every Process, which then describes the VHDL statements inside the process. Each of the above StatementTypes is accompanied by attributes that specify the relevant parameters for the StatementType and the corresponding line number in the VHDL behavioral description that was parsed. For a SignalAssignment statement, the attributes defined would be the "Target" of the signal assignment, the "Source" of the signal assignment which may consist of a signal or a combination of signals, the generic associated with the signal assignment , if any, and the line no [30].

The SPI routines are made accessible by compiling the C source files of BACKANN with the SPI library of functions. The functions used by SPI have names which are the same as the data structures used by the DLS and are thus easy to use. These functions are called by BACKANN to extract the parsed information from the DLS database [30].

4.5 Construction of VHDL entities for processes in the PMG

As described in the section on Process Extraction from the PMG, each process in the Process Model Graph is constructed as an individual VHDL entity. The entity declaration for each process is constructed using **extract** which extracts the structure of the Process Model Graph from the Modeler's Assistant database. The VHDL description for each process is extracted from the VHDL file using the parsed data from the DLS. The SPI is called to open the DLS and then traverses to the top of the data tree. Then BACKANN traverses through the ArchitectureBody which has a StatementList that

specifies the line numbers for each statement. For the VHDL behavioral description generated by the Modeler's Assistant, the ArchitectureBody's StatementList consists of a series of ProcessStatements in the same order as the processes extracted from the PMG. The starting line number of each process is obtained from the DLS. Using these line numbers the VHDL file is read and the VHDL code for each process is extracted.

The algorithm to extract the each process, construct the entity declaration and extract the VHDL code is shown in Figure 11.

4.6 Generic Delays

VHDL allows the designer to specify the different delay values using generics. It is for these generics that BACKANN determines values by synthesizing the model and extracting the delays. However, the Synopsys Design Compiler cannot accept generics of type TIME [14]. This problem was solved by replacing the generics by unique numerical values. The Synopsys Design Compiler discards signal assignment delays when it synthesizes a VHDL description to a gate-level design [14]. Thus these symbolic delay values do not affect the manner in which the behavioral description is synthesized.

```
Start Construct_separate_VHDL_entities_for_each_process
        Run "extract" written for HBTG to extract information from the PMG
        Construct list of signals interconnecting processes in the PMG
        Open VHDL file
                For each process in PMG
                        "extract" returns name of each process port, type and signal
                                number it is connected to
                        Start Construct_entity_declaration
                                If port is connected to signal
                                        Obtain signal name from the list of interconnecting
                                                signals
                                        Replace port name by signal name
                                end if
                                Assemble entity declaration using port names (replaced
                                        where necessary) and their types.
                                Generics in process are not declared
                        End Construct_entity_declaration
                        Access DLS using SPI
                        Traverse to ArchitectureBody and obtain starting line number of
                                current process and next process
                        Read VHDL file and get lines starting at line number of current
                                process and ending at line number of next process
                        Store entity declaration and process' VHDL code in temporary file
                        with unique name
                end loop
End
```

Figure 11 Algorithm for constructing VHDL entities for each process

A signal assignment statement as shown below with a generic delay attached to it

```
A <= B and C after AND_DEL;
```

would become

```
A <= B and C after 0.012 ns;
```

The numeric values are generated starting from 0.01 ns, in increments of 0.001 ns. Each generic has a unique value associated with it and the generic's name and its symbolic numeric value are stored in a linked list. These numeric values are then used to determine the name of the generic when the delays are back-

annotated into the VHDL description. The algorithm for replacement of generics is shown in Figure 12.

```
Start Generic_Replacement
        Access DLS to obtain generics and generate unique symbolic numeric values for
                each
        Store generic name and symbolic numeric value in linked list
        For each process in PMG
                Create VHDL file and separate entity
                Parse created VHDL file for generics
                If generic specified
                        Match generic specified with generic_name in linked list
                        Replace generic specified by symbolic numeric value from linked
                                list
                end if
                Store new VHDL file without generics
        end loop
End
```

Figure 12 Algorithm for generic replacement

4.7 Parsing the Signal Assignments

After extracting each process and constructing separate VHDL entities for each, with generics replaced, the VHDL code to be synthesized is ready. Now, it has to be determined which are the delays that are required from the gate-level design generated by the Synopsys Design Compiler. This is done by parsing every signal assignment statement in the VHDL file and determining the paths for which the delays are required. The parsed signal assignments are stored in a linked list whose structure is explained in Appendix B. The data stored include the line number of each signal assignment statement, the target of the signal assignment, the type of the target and the different signals on the right hand side of the signal assignment.

The CLSI-SPI is used to access the DLS description of the VHDL file. The tree is traversed from the top to every branch and we check each StatementType to see if it is a SignalAssignment.. When a SignalAssignment is found the relevant attributes accompanying it are,

Name	*Type*	*Comment*
qLine	*Integer4*[1]	*Line number*
qTarget	*Node*	*Specifies the target signal*
qSource	*Node*	*Specifies the expression on the right hand side of a signal assignment*

The Target or the signal to which a value is being assigned is stored in the Node *qTarget.* The *qTarget* node specifies the name of the target signal and its type. The name and type are stored in the data structure [30].

The *qSource* node is traversed to determine the source signals. The source signals maybe either direct numeric values

```
A <= '1';
```

or a signal name

```
A <= B;
```

or a set of signals and the operations to be performed on them

```
A <= B and (not C) ;
```

In the third case the node is a tree with all the source signals and the relationships specified between them in the signal assignment statement.

[1] *Integer4, Node* etc., are types defined internally in DLS

Depending on the type of the target signal, the signal assignments are further classified as assigning values to

a) BIT

where the target signal is of type BIT.

b) BIT_VECTOR_FULL

where the target signal is of type BIT_VECTOR and all the bits of the BIT_VECTOR are assigned values.

c) BIT_VECTOR_SLICE

where the target signal is of type BIT_VECTOR but two or more bits but not all bits of the BIT_VECTOR are assigned values.

d) BIT_VECTOR_INDEX

where the target signal is of type BIT_VECTOR but only one bit in the BIT_VECTOR is assigned a value.

All the data extracted for a signal assignment is stored in the linked list and it is used to determine the delay values required from the Synopsys Design Compiler. The algorithm for parsing signal assignments is shown in Figure 13.

```
Start Parse_Signal_Assignments
        For each process in PMG
                Access DLS
                Search for Signal Assignment statements
                Determine target signal name and type
                Determine source - direct assign or signal(s)
                Store data in linked list
        end loop
End
```

Figure 13 Algorithm for parsing Signal Assignments

4.8 Invoking the Design Compiler

The Synopsys Design Compiler is invoked from within BACKANN through the UNIX Bourne shell. The Synopsys Design Compiler should be given a set of commands to execute so that it takes each process, synthesizes it and then returns the delay values required for that process. This is done by creating a command file into which all the commands that BACKANN requires the Synopsys Design Compiler to execute are written. This file is specified in the command line to the Synopsys Design Compiler and the required tasks are performed [27].

The specifications to the synthesizer include the design libraries to be used, the different components like specific flip-flop and latch types, maximum and minimum fanout and fanin. These specifications are written by the designer and stored in a log_file. This log_file is copied into the command file by BACKANN to synthesize the processes. A typical log_file is given in Figure 14.

For every process in the Process Model Graph the file with the VHDL description is read into the Synopsys Design Compiler. This is specified in the command file using the *read* command [31]. Then the log_file is copied into the command file. The log_file commands will synthesize the VHDL behavioral description to a gate level design. Then the commands to extract delays are written into the command file. The command to extract delays is the Synopsys Design Compiler command called *report_timing* [31]. This command is executed to determine the various delays and the output is written into a file called out_file which is later parsed.

```
search_path = ". /usr1/synopsys/libraries/syn /usr1/synopsys/libraries
/usr1/synopsys/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 "RCPORT"
 set_min_fault_coverage 95 -area_critical -timing_critical
 set_register_type -flip_flop "FJK2SP"
 set_register_type -exact -latch "LD2"
 set_boundary_optimization "RCPORT"
 link_library = "lsi_10k.db"
 link -all
compile  -map_effort high -verify -verify_effort high -boundary_optimization
```

Figure 14 A log file of user-specified commands to the synthesizer

The *report_timing* command's construction depends upon the type of the signal assignment statement. The various combinations are listed and explained below.

A target signal maybe triggered because of changes in a number of other signals and these changes can propagate through different paths leading to different delays for the change to take place in the target signal. The Synopsys Design Compiler calculates the different paths between signals and can return the delay values on the maximum delay and minimum delay paths for both a fall and rise in the value of the target signal.

If the signal assignment is the direct assignment of a '1' or a '0' to a signal of type BIT, then for a '1' the *report_timing* command specifies that the minimum and maximum rise times for the target signal should be written into the out_file and for a '0' the *report_timing* command specifies that the minimum and maximum fall times for the target signal should be written into the out_file.

e.g., A <= '1';

would generate

report_timing -delay min_rise -to A > out_file

report_timing -delay max_rise -to A > out_file

If the signal assignment is a direct assignment of a value to a signal of type BIT_VECTOR, then the *report_timing* command specifies that the minimum and maximum fall and rise times for the target signal should be written into the out_file.

e.g., B(7 downto 0) <= "00001010" ;

would generate

report_timing -delay min_rise -to B > out_file

report_timing -delay max_rise -to B > out_file

report_timing -delay min_fall -to B > out_file

report_timing -delay max_fall -to B > out_file

If the signal assignment has a target signal of type BIT and the source consists of other signal(s), then the *report_timing* command specifies that the maximum and minimum fall and rise times for the target signal from each of the source signals should be written into the out_file

e.g., C <= D and E ;

would generate

report_timing -delay min_fall -from D -to C > out_file

report_timing -delay max_fall -from D -to C > out_file

report_timing -delay min_rise -from D -to C > out_file

report_timing -delay max_rise -from D -to C > out_file

report_timing -delay min_fall -from E -to C > out_file

report_timing -delay max_fall -from E -to C > out_file

report_timing -delay min_rise -from E -to C > out_file

report_timing -delay max_rise -from E -to C > out_file

If the target signal is of type BIT_VECTOR and the source consists of other signal(s), the *report_timing* command specifies that the minimum and maximum rise and fall times for the target signals from each of the source signals should be written into the out_file.

e.g. F <= G & H(7 downto 1);

would generate

report_timing -delay min_fall -from G -to F > out_file

report_timing -delay max_fall -from G -to F > out_file

report_timing -delay min_rise -from G -to F > out_file

report_timing -delay max_rise -from G -to F > out_file

report_timing -delay min_fall -from H -to F > out_file

report_timing -delay max_fall -from H -toF > out_file

report_timing -delay min_rise -from H -to F > out_file

report_timing -delay max_rise -from H -to F > out_file

After all the *report_timing* commands, the *remove_design* command is invoked to clear the design out of the system, and then using *read* the next process' VHDL file is read in and the cycle continues till all the processes have been synthesized and the delays are written into the out_file [31]. A section of a typical command file is shown in Figure 15. The algorithm for invoking the Design Compiler is shown in Figure 16.

```
read -format vhdl /tmp/k84621.vhd
designer = " Sathya Narayanaswamy"
company = "Virginia Tech"
search_path = ". /usr1/synopsys/libraries/syn /usr1/synopsys/libraries
/usr1/synopsys/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 "RCPORT"
```

Figure 15 Command file for the Synopsys Design Compiler

```
set_min_fault_coverage 95 -area_critical -timing_critical
set_register_type -flip_flop "FJK2SP"
set_register_type -exact -latch "LD2"
set_boundary_optimization "RCPORT"
link_library = "lsi_10k.db"
link -all
compile  -map_effort high -verify -verify_effort high -boundary_optimization
report_timing -to UARTCLK -from CLKAA -delay min_fall >> report8462.out
report_timing -to UARTCLK -from CLKAA -delay max_fall >> report8462.out
report_timing -to UARTCLK -from CLKAA -delay min_rise >> report8462.out
report_timing -to UARTCLK -from CLKAA -delay max_rise >> report8462.out
report_timing -to UARTCLK -from OUTACLK -delay min_fall >> report8462.out
report_timing -to UARTCLK -from OUTACLK -delay max_fall >> report8462.out
report_timing -to UARTCLK -from OUTACLK -delay min_rise >> report8462.out
report_timing -to UARTCLK -from OUTACLK -delaay max_rise >> report8462.out
remove_design
read -format vhdl /tmp/k84622.vhd
```

Figure 15 Command file for the Synopsys Design Compiler (continued)

```
Start Invoke_Design_Compiler
        For each process in PMG
                Write read command into command file
                Copy log_file into command file
                Start Determine_delays_required
                        For each signal assignment statement
                                If source is a '1' and target is a signal of type BIT
                                        Delays required are min_rise, max_rise for target
                                If source is a '0' and target is a signal of type BIT
                                        Delays required are min_fall, max_fall for target
                                If source signals and target signal are type BIT
                                        Delays required are min_rise, max_rise, min_fall,
                                                max_fall to target signal from each source
signal
                                If source signals and target signal are type BIT_VECTOR
                                        Delays required are min_rise, max_rise, min_fall,
                                                max_fall to target signal from each source
signal
                        end loop
                End Determine_delays_required
        end loop
        Invoke Design Compiler with command file
End
```

Figure 16 Algorithm for invoking the Synopsys Design Compiler

4.9 Parsing the out_file

The out_file consists of the outputs of all the *report_timing* commands issued to the Synopsys Design Compiler. This file has to be parsed to extract the delay values required.

The *report_timing* command in the Synopsys Design Compiler returns not only the delay values required from it but also other information. This includes the technology library used, the wire loading model which determines the delay for the propagation of a signal through a wire between two cells, The start and endpoints of the signal, the type of delay required - max or min and the cells through which the signal propagated and the delays through each of these cells. A typical output is shown in Figure 17.

```
Operating Conditions: BCCOM   Library: lsi_10k
Wire Loading Model Mode: top
Design                Wire Loading Model          Library
i8212          20x20                              lsi_10k
        Startpoint: DS2 (input port)
        Endpoint: DO[0] (output port)
        Constraint Group: (none)
        Path Type: min
Point                                  Incr          Path
  -----------------------------------------------------------
input external delay                   0.00          0.00 r
DS2 (in)                               0.00          0.00 r
U86/Z (ND2)                            0.35          0.35 f
U89/Z (OR2P)                                  0.49          0.84 f
DO[0] (inout)                                 0.00          0.84 f
data arrival time                             0.84
```

Figure 17 Output from the *report_timing* command

4.10 Back annotation of delay values

The linked list of data mentioned in the preceding sections, now contains all the information required to calculate the delay values associated with each signal assignment.

There are two modes in which the designer may specify if the minimum delays or the maximum delays are used. These modes are ***minimum delay mode*** and ***maximum delay mode*** and are defined below. The choice of modes is offered as a command line option. For a signal assignment

A <= B;

let the delay values returned be

min_fall	0.7 ns
max_fall	1.1 ns
min_rise	0.85 ns
max_rise	1.23 ns

In the ***minimum delay mode***, the delay values used for risetime and falltime would be 0.85 ns and 0.7 ns respectively as the minimum rise and fall values. In the second or ***maximum delay mode***, the values used would be 1.1 ns and 1.23 ns.

There are also two models for assigning rise and fall delays. These models differ only for assignments to signals of type BIT. In the first model called ***separate rise-and-fall model*** the designer can specify that different values be used for the rise and fall of the signal while in the second model called the ***combined rise-and-fall model*** the designer can specify that the delay value used be an average of the rise and fall times.

The ***separate rise and fall model*** in the ***maximum delay mode*** would generate

```
if ( B = '1' ) then
        A <= '1' after 1.23 ns;
elsif ( B = '0' ) then
        A <= '0' after 1.1 ns;
end if;
```

The risetime delay of 1.23 ns is the max_rise delay value.

The falltime delay of 1.1 ns is the max_fall delay value.

The ***separate rise-and-fall model*** in the ***minimum delay mode*** would generate

```
if ( B = '1' ) then
        A <= '1' after 0.85 ns;
elsif ( B = '0' ) then
        A <= '0' after 0.7 ns;
end if;
```

The risetime delay of 0.85 ns is the min_rise delay value.

The falltime delay of 0.7 ns is the min_fall delay value.

The ***combined rise-and-fall model*** in the ***maximum delay mode*** would generate

```
A <= B after 1.165 ns;
```

The delay value of 1.165 ns is obtained by calculating the average of the two delay values - max_fall (1.1 ns) and max_rise (1.23 ns).

The ***combined rise-and-fall model*** in the ***minimum delay mode*** would generate

```
A <= B after 0.775 ns;
```

The delay value of 0.775 ns is obtained by calculating the average of the two delay values - min_fall (0.7 ns) and min_rise (0.85 ns).

The user specifies if the ***maximum delay mode*** or ***minimum delay mode*** is to be used and if the ***separate rise-and-fall model*** or the ***combined rise-and-fall model*** should be used, when BACKANN is run.

If in a signal assignment, the source or the right hand side is a combination of signals, the delay for the target with respect to each source signal is calculated and the maximum or minimum value out of all the delay values returned is assigned to that signal assignment.

For a signal assignment

A <= B and C ;

the delays returned by the Synopsys Design Compiler would be as shown in Table 1.

Table 1 Delay values returned by the Synopsys Design Compiler

Type of delay	B to A (ns)	C to A (ns)
min_fall	0.9	1.2
max_fall	1.3	1.5
min_rise	1.1	1.25
max_rise	1.4	1.6

For the above signal assignment the ***separate rise-and-fall model*** in the ***maximum delay mode*** would generate

```
if ( B and C = '0' ) then
        A <= '1' after 1.5 ns;
elsif ( B and C = '1' ) then
        A <= '1' after 1.6 ns;
end if;
```

The delay value of 1.5 ns is calculated by taking the max_fall delay values from B to A (1.5 ns) and from C to A (1.3 ns) and taking the maximum of the two values.

The delay value of 1.6 ns is calculated by taking the max_rise delay values from B to A (1.4 ns) and from C to A (1.6 ns) and taking the maximum of the two values.

The ***separate rise-and-fall model*** in the ***minimum delay mode*** would generate

```
if ( B and C = '0' ) then
        A <= '0' after 0.9 ns;
elsif ( B and C = '1' ) then
        A <= '1' after 1.1 ns;
end if;
```

The delay value of 0.9 ns is calculated by taking the min_fall delay values from B to A (0.9 ns) and from C to A (1.2 ns) and taking the minimum of the two values.

The delay value of 1.1 ns is calculated by taking the min_rise delay values from B to A (1.1 ns) and from C to A (1.25 ns) and taking the minimum of the two values.

The ***combined rise-and-fall model*** in the ***maximum delay mode*** generates

```
A <= B and C after 1.55 ns;
```

The delay value of 1.55 ns is calculated by first taking the maximum of the two max_fall delays (1.3 ns and 1.5 ns) and the maximum of the two max_rise delays (1.4 ns and 1.6 ns), which gives 1.5 ns and 1.6 ns and then averaging these two delays.

The ***combined rise-and-fall model*** in the ***minimum delay mode*** generates

A <= B and C after 1.0 ns;

The delay value of 1.0 ns is calculated by first taking the minimum of the two min_fall delays (0.9 ns and 1.2 ns) and the minimum of the two min_rise delays (1.1 ns and 1.25 ns), which gives 0.9 ns and 1.1 ns and then averaging the two delays.

For every signal assignment statement, it is checked to see if there is a generic delay associated with it. If there is a generic delay, the signal assignment is not rewritten with the numeric delay values as above, instead the generic is initialized to the calculated delay value in the generic clause in the entity declaration.

In the signal assignment,

A <= B after A_DEL;

if the delay is calculated as 0.8 ns, then the signal assignment statement would be unchanged in the VHDL description but the generic clause in the entity declaration is altered as

A_DEL : TIME := 0.8 ns;

BACKANN does not impose any naming conventions for generics on the designer [33], but it is good modeling practice to use a consistent and readable notation.

The back-annotated VHDL description is then written back into a new file. The whole process is depicted as a flowchart in Figure 18.

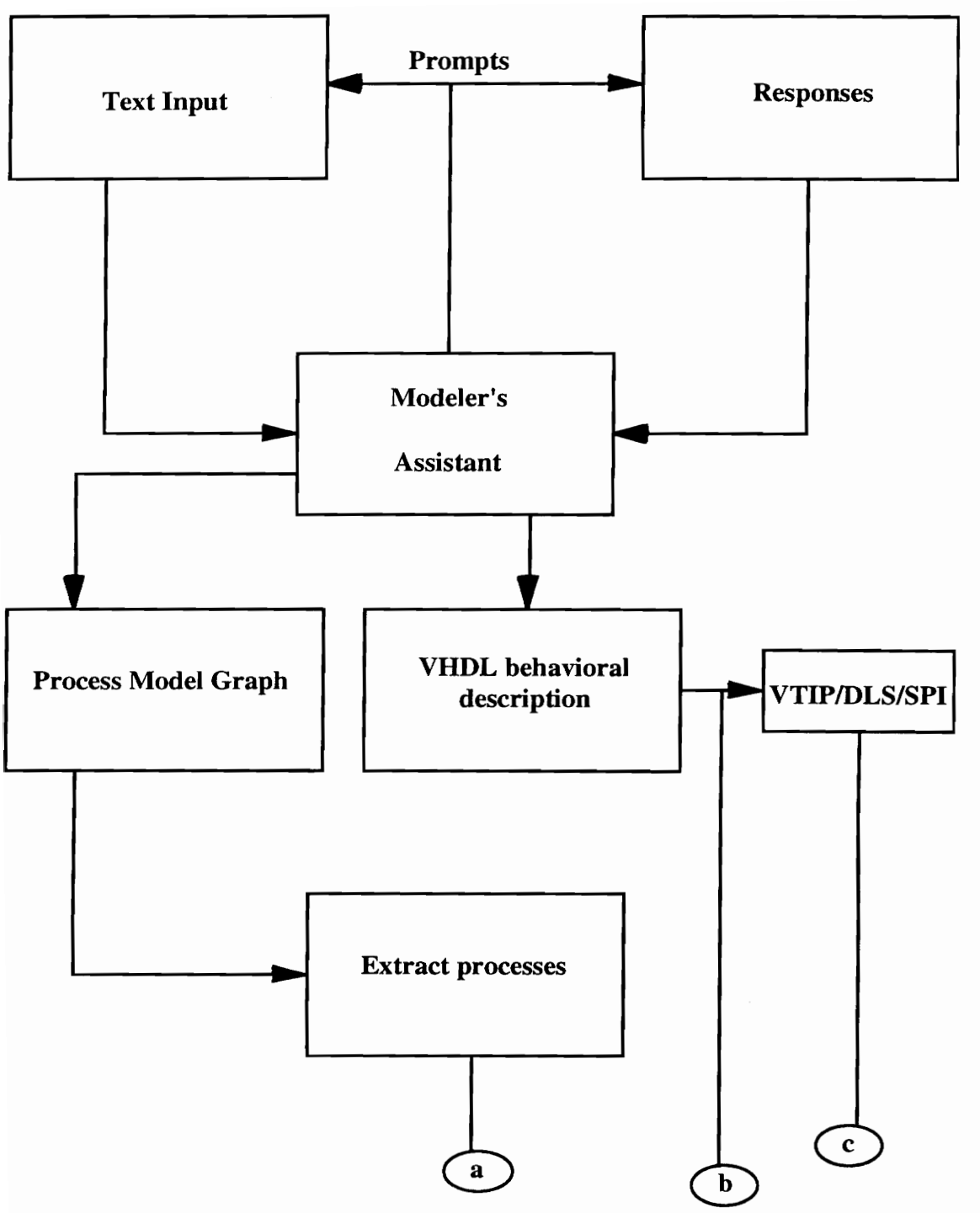


Figure 18 Flowchart for BACKANN

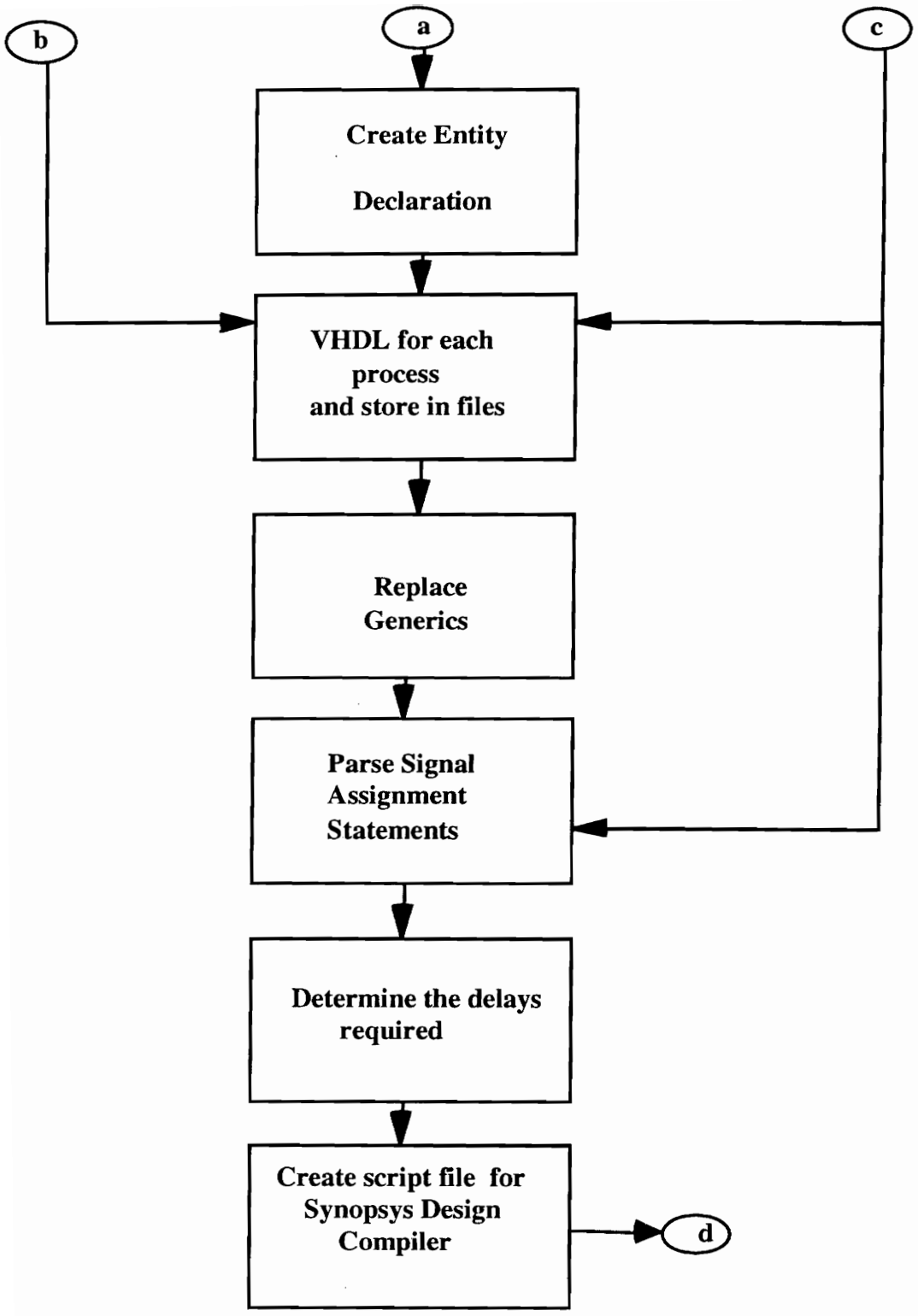


Fig 18 Flowchart for BACKANN

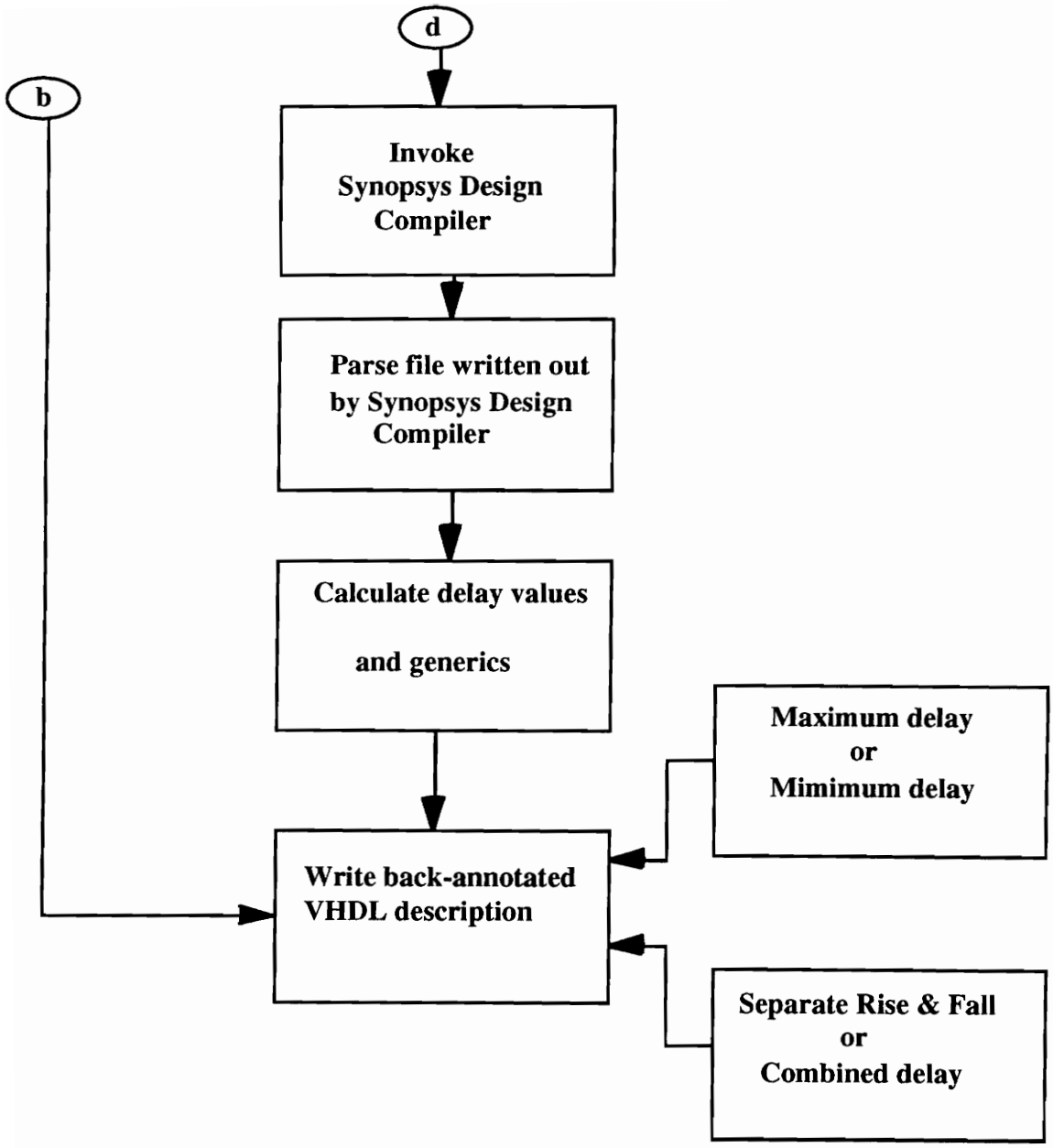


Figure 18 Flowchart for BACKANN

Chapter 5

Results

In this chapter the back-annotated models produced by BACKANN for two designs will be analyzed. The first chip is RCPORT- a digital/analog interface for RS-232 and the second chip is SIMCPU - a simple CPU implementation.

5.1 RCPORT

This chip implements a UART in combination with a Pulse Width Modulator and a clock generator circuit. The Process Model Graph for the chip is shown in Figure 19. The delays for the different signal assignments are specified in the form of generics. The VHDL description of the chip as generated by the Modeler's Assistant is shown in Figure 20.

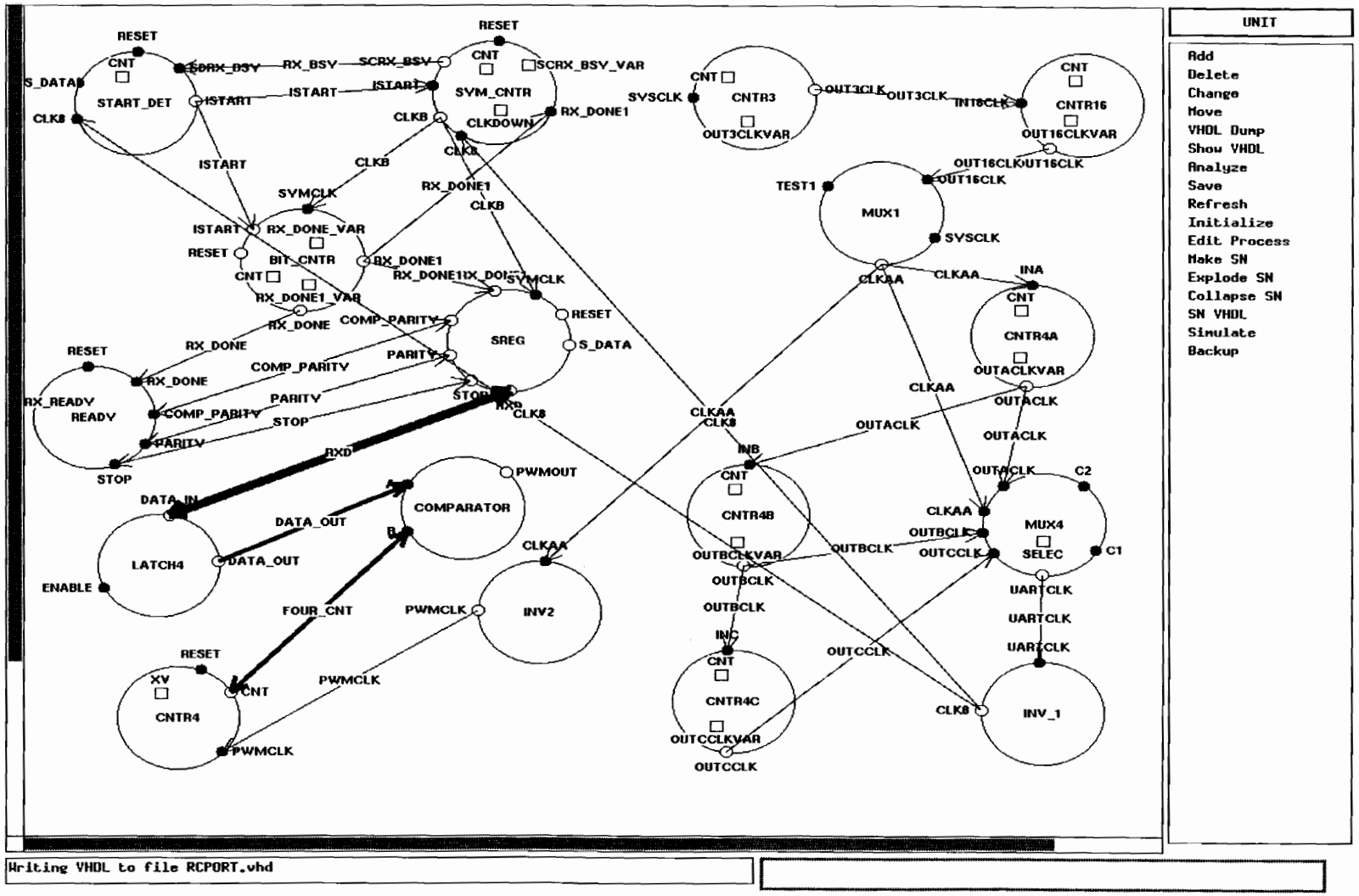


Figure 19 Process Model Graph of RCPORT

```
**********************************************************
entity RCPORT is
generic (MUX_DEL: TIME;
CNTR4_DEL: TIME;
INV_DEL: TIME;
OUT16CLKDEL: TIME;
OUT3CLKDEL: TIME;
OUTCCLKDEL: TIME;
OUTBCLKDEL: TIME;
OUTACLK_DEL: TIME;
COMP_DEL: TIME;
DATA_OUTDEL: TIME;
RX_READYDEL: TIME;
COMP_PARITYDEL: TIME;
PARITY_DEL: TIME;
STOP_DEL: TIME;
RXD_DEL: TIME;
RX_DONE1DEL: TIME;
RX_DONEDEL: TIME;
CLKB_DEL: TIME;
RX_BSYDEL: TIME;
ISTART_DEL: TIME
);
port (C1: in BIT;
C2: in BIT;
RESET: in BIT;
TEST1: in BIT;
SYSCLK: in BIT;
PWMOUT: out BIT;
ENABLE: in BIT;
RX_READY: out BIT;
S_DATA: in BIT);
```

Figure 20 VHDL behavioral description of RCPORT

```
end RCPORT;

-- ************************************************************
architecture BEHAVIORAL of RCPORT is
signal OUTCCLK: BIT;
signal OUTBCLK: BIT;
signal OUTACLK: BIT;
signal CLKAA: BIT;
signal UARTCLK: BIT;
signal PWMCLK: BIT;
signal FOUR_CNT: BIT_VECTOR(3 downto 0);
signal CLK8: BIT;
signal OUT16CLK: BIT;
signal OUT3CLK: BIT;
signal DATA_OUT: BIT_VECTOR(3 downto 0);
signal RXD: BIT_VECTOR(7 downto 0);
signal RX_DONE: BIT;
signal PARITY: BIT;
signal COMP_PARITY: BIT;
signal STOP: BIT;
signal RX_DONE1: BIT;
signal CLKB: BIT;
signal ISTART: BIT;
signal RX_BSY: BIT;
begin

  -- ---------------------------------------------------
 -- Process Name: MUX4
 -- ---------------------------------------------------
 MUX4_4: process (OUTCCLK,OUTBCLK,OUTACLK,CLKAA,C1,C2)
       variable SELEC: BIT_VECTOR(1 downto 0);
 begin
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
        SELEC := C2&C1;
        case SELEC is
                when "00" => UARTCLK <= CLKAA after MUX_DEL;
                when "01" => UARTCLK <= OUTACLK after MUX_DEL;
                when "10" => UARTCLK <= OUTBCLK after MUX_DEL;
                when "11" => UARTCLK <= OUTCCLK after MUX_DEL;
        end case;
end process MUX4_4;

 -- ------------------------------------------------------
 -- Process Name: CNTR4
 -- ------------------------------------------------------
CNTR4_15: process (RESET,PWMCLK)
        variable XV: BIT_VECTOR(3 downto 0);
begin
        if (RESET = '1') then
                FOUR_CNT <= "0000" after CNTR4_DEL;
        elsif ( PWMCLK = '1' and PWMCLK'EVENT ) then
                XV := FOUR_CNT;
                for I in 0 to 3 loop
                        if XV(I) = '0' then
                                XV(I) := '1';
                                exit;
                        else XV(I) := '0';
                        end if;
                end loop;
                FOUR_CNT <= XV after CNTR4_DEL;
        end if;
end process CNTR4_15;

 -- ------------------------------------------------------
 -- Process Name: INV2
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
-- ------------------------------------------------
INV2_23: process (CLKAA)
begin
        PWMCLK <= not CLKAA after INV_DEL;
end process INV2_23;

 -- ------------------------------------------------
 -- Process Name: INV_1
 -- ------------------------------------------------
INV_1_28: process (UARTCLK)
begin
        CLK8 <= not UARTCLK after INV_DEL;
end process INV_1_28;

 -- ------------------------------------------------
 -- Process Name: MUX1
 -- ------------------------------------------------
MUX1_33: process (TEST1,SYSCLK,OUT16CLK)
begin
         case TEST1 is
                when '0' => CLKAA <= OUT16CLK after MUX_DEL;
                when '1' => CLKAA <= SYSCLK after MUX_DEL;
        end case;
end process MUX1_33;

 -- ------------------------------------------------
 -- Process Name: CNTR16
 -- ------------------------------------------------
CNTR16_40: process (OUT3CLK)
        variable CNT: INTEGER range 0 to 15;
        variable OUT16CLKVAR : BIT;
begin
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
        if(OUT3CLK = '1') then
                if(CNT < 15) then
                        CNT := CNT + 1;
                else
                        CNT := 0;
                        OUT16CLKVAR := not OUT16CLKVAR ;
                end if;
        end if;
        OUT16CLK <= OUT16CLKVAR after OUT16CLKDEL;
end process CNTR16_40;

 -- ---------------------------------------------------
 -- Process Name: CNTR3
 -- ---------------------------------------------------
CNTR3_46: process (SYSCLK)
variable CNT: INTEGER range 0 to 2;
variable OUT3CLKVAR : BIT;
begin
        if(SYSCLK = '1') then
                if(CNT < 2) then
                        CNT := CNT + 1;
                else
                        CNT := 0;
                        OUT3CLKVAR := not OUT3CLKVAR;
                end if;
        end if;
        OUT3CLK <= OUT3CLKVAR after OUT3CLKDEL;
end process CNTR3_46;

 -- ---------------------------------------------------
 -- Process Name: CNTR4C
 -- ---------------------------------------------------
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
CNTR4C_52: process (OUTBCLK)
        variable CNT: INTEGER range 0 to 3;
        variable OUTCCLKVAR : BIT;
begin
        if(OUTBCLK = '1') then
                if(CNT < 3 )then
                        CNT := CNT + 1;
                else
                        CNT := 0;
                        OUTCCLKVAR := not OUTCCLKVAR;
                end if;
        end if;
        OUTCCLK <= OUTCCLKVAR after OUTCCLKDEL;
end process CNTR4C_52;

 -- ------------------------------------------------------
 -- Process Name: CNTR4B
 -- ------------------------------------------------------
CNTR4B_58: process (OUTACLK)
        variable CNT: INTEGER range 0 to 3;
        variable OUTBCLKVAR : BIT;
begin
        if(OUTACLK = '1') then
                if(CNT < 3 )then
                        CNT := CNT + 1;
                else
                        CNT := 0;
                        OUTBCLKVAR := not OUTBCLKVAR;
                end if;
        end if;
        OUTBCLK <= OUTBCLKVAR after OUTBCLKDEL;
end process CNTR4B_58;
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
  -- ------------------------------------------------------
  -- Process Name: CNTR4A
  -- ------------------------------------------------------
CNTR4A_64: process (CLKAA)
        variable CNT: INTEGER range 0 to 3;
        variable OUTACLKVAR : BIT;
begin
        if(CLKAA = '1') then
                if(CNT < 3 )then
                        CNT:= CNT + 1;
                else
                        CNT := 0;
                        OUTACLKVAR := not OUTACLKVAR;
                end if;
        end if;
        OUTACLK <= OUTACLKVAR after OUTACLK_DEL;
end process CNTR4A_64;

  -- ------------------------------------------------------
  -- Process Name: COMPARATOR
  -- ------------------------------------------------------
COMPARATOR_70: process (FOUR_CNT,DATA_OUT)
begin
        if DATA_OUT < FOUR_CNT then
                PWMOUT <= '1' after COMP_DEL;
        elsif DATA_OUT = FOUR_CNT then
                PWMOUT <= '0' after COMP_DEL;
        elsif DATA_OUT > FOUR_CNT then
                PWMOUT <= '0' after COMP_DEL;
        end if;
end process COMPARATOR_70;
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
 -- ------------------------------------------------------
 -- Process Name: LATCH4
 -- ------------------------------------------------------
LATCH4_76: process (ENABLE)
begin
        if(ENABLE = '1') then
                DATA_OUT <= RXD(7 downto 4) after DATA_OUTDEL;
        end if;
end process LATCH4_76;

 -- ------------------------------------------------------
 -- Process Name: READY
 -- ------------------------------------------------------
READY_82: process (RESET,RX_DONE,PARITY,COMP_PARITY,STOP)
begin
        if(RESET = '1') then
                RX_READY <= '1' after RX_READYDEL;
        elsif(COMP_PARITY = PARITY) then
                if(STOP = '1' and RX_DONE = '1') then
                        RX_READY <= '1' after RX_READYDEL;
                end if;
        end if;
end process READY_82;

 -- ------------------------------------------------------
 -- Process Name: SREG
 -- ------------------------------------------------------
SREG_91: process (CLKB)
begin
        if(RESET = '1') then
                RXD <= "00000000" after RXD_DEL;
                STOP <= '0' after STOP_DEL;
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
                PARITY <= '0'  after PARITY_DEL;
                COMP_PARITY <= '0' after COMP_PARITYDEL;
        end if;
        if(CLKB = '1') then
                RXD <=  STOP & RXD(7 downto 1) after RXD_DEL;
                STOP <= PARITY after STOP_DEL;
                PARITY <= S_DATA after PARITY_DEL;
                if(RXD(7) = '1') then
                        COMP_PARITY <= not COMP_PARITY after
COMP_PARITYDEL;
                end if;
        elsif (RX_DONE1 = '1') then
                COMP_PARITY <= '0' after COMP_PARITYDEL;
        end if;
end process SREG_91;

 -- ---------------------------------------------------
 -- Process Name: BIT_CNTR
 -- ---------------------------------------------------
BIT_CNTR_105: process (CLKB)
        variable RX_DONE1_VAR: BIT;
        variable CNT: INTEGER range 0 to 3;
        variable RX_DONE_VAR: BIT;
begin
        if(RESET = '1')then
                CNT := 0;
                RX_DONE1 <= '0' after RX_DONE1DEL;
                RX_DONE <= '0'  after RX_DONEDEL;
        elsif(CLKB = '1')then
                if (CNT < 9) then
                        CNT := CNT + 1;
                else
                        CNT := 0;
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
                    RX_DONE <= '1' after RX_DONEDEL;
                    RX_DONE1 <= '1' after RX_DONE1DEL;
                    RX_DONE_VAR := '1';
                end if;
        end if;
        if(RX_DONE_VAR = '1' and CLKB = '0')then
                RX_DONE1 <= '0' after RX_DONE1DEL;
        end if;
end process BIT_CNTR_105;

  -- ------------------------------------------------------
  -- Process Name: SYM_CNTR
  -- ------------------------------------------------------
SYM_CNTR_117: process (RESET,CLK8,RX_DONE1,ISTART)
        variable SCRX_BSY_VAR: BIT;
        variable CLKDOWN: BIT;
        variable CNT: INTEGER range 0 to 5;
begin
        if(RESET = '1')then
                RX_BSY <= '0' after RX_BSYDEL;
                CLKB <= '0' after CLKB_DEL;
                CNT := 0;
        elsif( ISTART = '1' and SCRX_BSY_VAR = '0') then
                RX_BSY <= '1';
                SCRX_BSY_VAR := '1';
                CLKDOWN := '1';
                CLKB <= '0' after CLKB_DEL;
        elsif(CLK8'EVENT and CLK8 = '1') then
                if( ISTART = '1' and SCRX_BSY_VAR = '0') then
                        RX_BSY <= '1' after RX_BSYDEL ;
                        SCRX_BSY_VAR := '1';
                        CLKDOWN := '1' ;
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
            end if;
            if(CLKDOWN = '1' and SCRX_BSY_VAR = '1') then
                if(CNT < 5) then
                    CNT := CNT + 1;
                else
                    CNT := 0;
                    CLKB <= '1'after CLKB_DEL ;
                    CLKDOWN := '0';
                end if;
            elsif(CLKDOWN = '0' and SCRX_BSY_VAR = '1') then
                if(CNT < 1) then
                    CNT := CNT + 1;
                else
                    CNT := 0;
                    CLKB <= '0' after CLKB_DEL;
                    CLKDOWN := '1';
                end if;
             else
                    CLKB <= '0' after CLKB_DEL;
            end if;
        end if;
        if(RX_DONE1 = '1')then
            RX_BSY <= '0' after RX_BSYDEL;
            SCRX_BSY_VAR := '0';
        end if;
end process SYM_CNTR_117;

 -- ------------------------------------------------------
 -- Process Name: START_DET
 -- ------------------------------------------------------
START_DET_130: process (RX_BSY,RESET,CLK8,S_DATA)
        variable CNT: INTEGER range 0 to 3;
```

Figure 20 VHDL behavioral description of RCPORT (continued)

```
begin
        if(CLK8'EVENT and CLK8 = '1') then
                if(S_DATA = '1' or RESET = '1' or RX_BSY = '1') then
                        CNT := 0;
                        ISTART <= '0' after ISTART_DEL;
                elsif(S_DATA = '0') then
                        if(CNT < 3)then
                                CNT := CNT + 1;
                        else
                                CNT := 0;
                                ISTART <= '1' after ISTART_DEL;
                        end if;
                end if;
        end if;
end process START_DET_130;
end BEHAVIORAL;
```

Figure 20 VHDL behavioral description of RCPORT (continued)

In the VHDL description the declarations of variables as INTEGER type was modified to include the range for the variable, if not the Synopsys Design Compiler allocates 32 flip-flops to cover the whole range for an INTEGER type [14].

When the VHDL description of the Process Model Graph for RCPORT is input to BACKANN and then passed from BACKANN to the Synopsys Design Compiler, the gate-level design produced for the whole chip, is as shown in Figure 21. The gate level designs were optimized by the Synopsys Design Compiler to occupy minimum area.

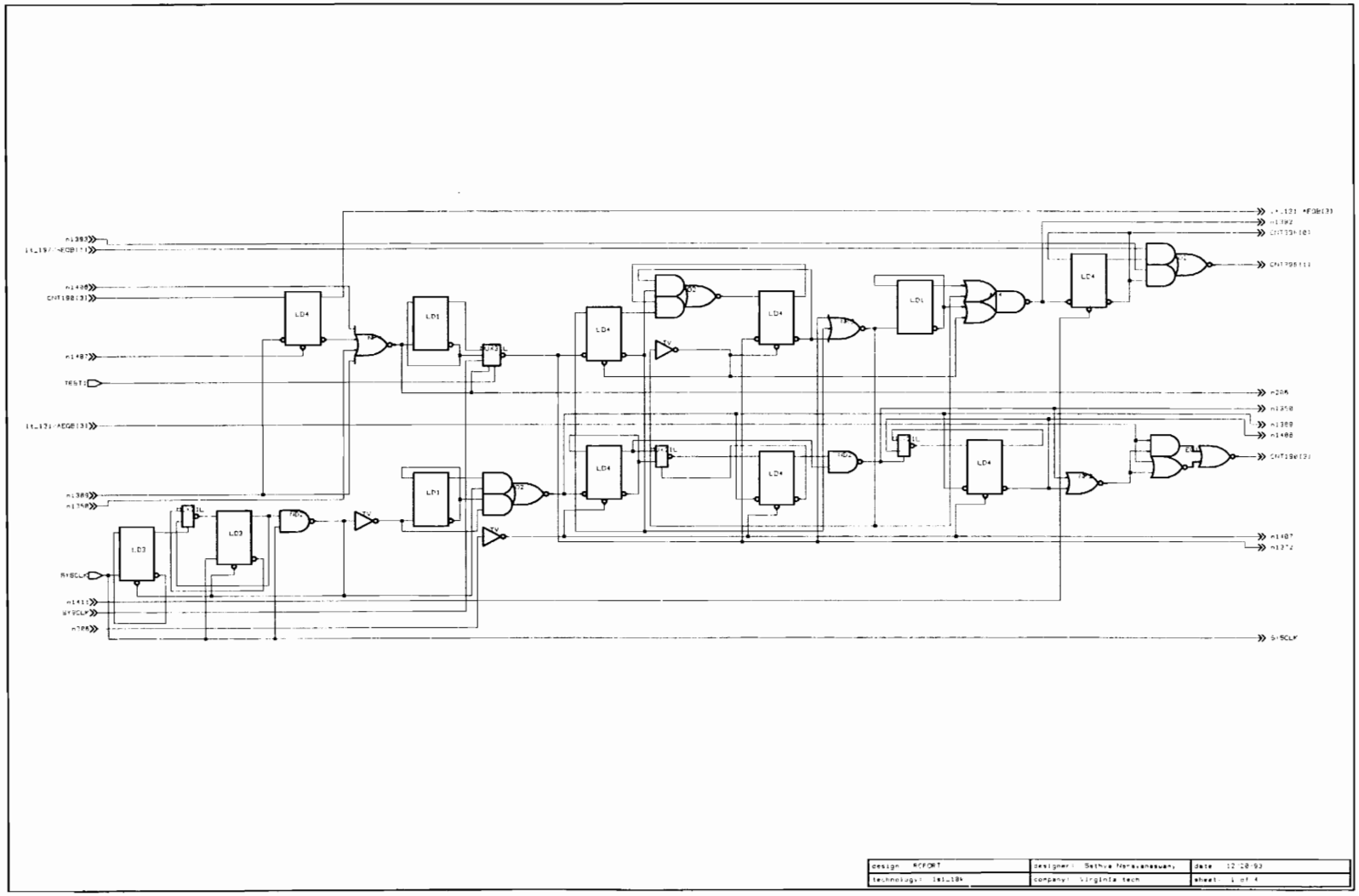


Figure 21 Gate level design of RCPORT

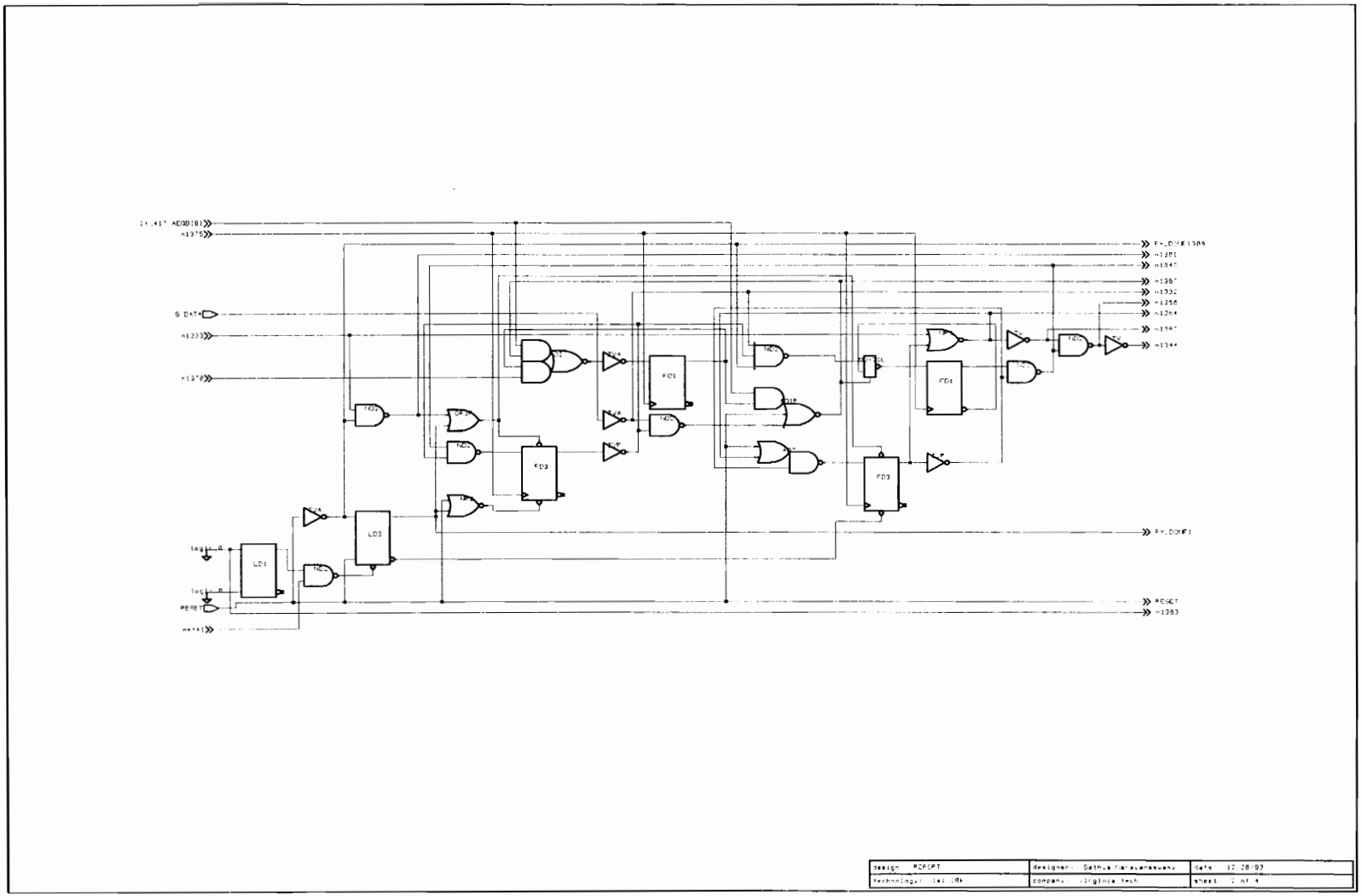


Figure 21 Gate level design of RCPORT (continued)

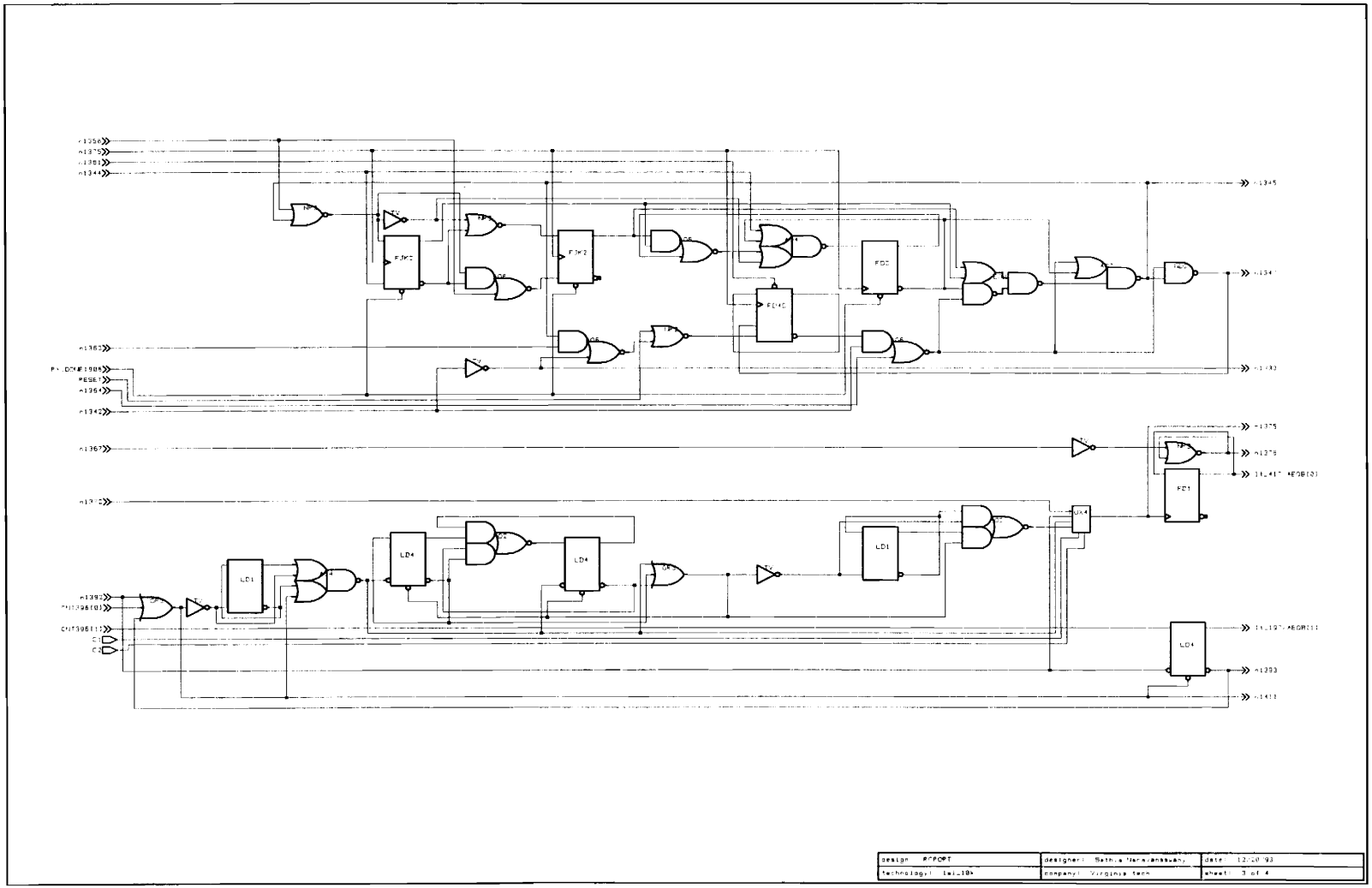


Figure 21 Gate level design of RCPORT (continued)

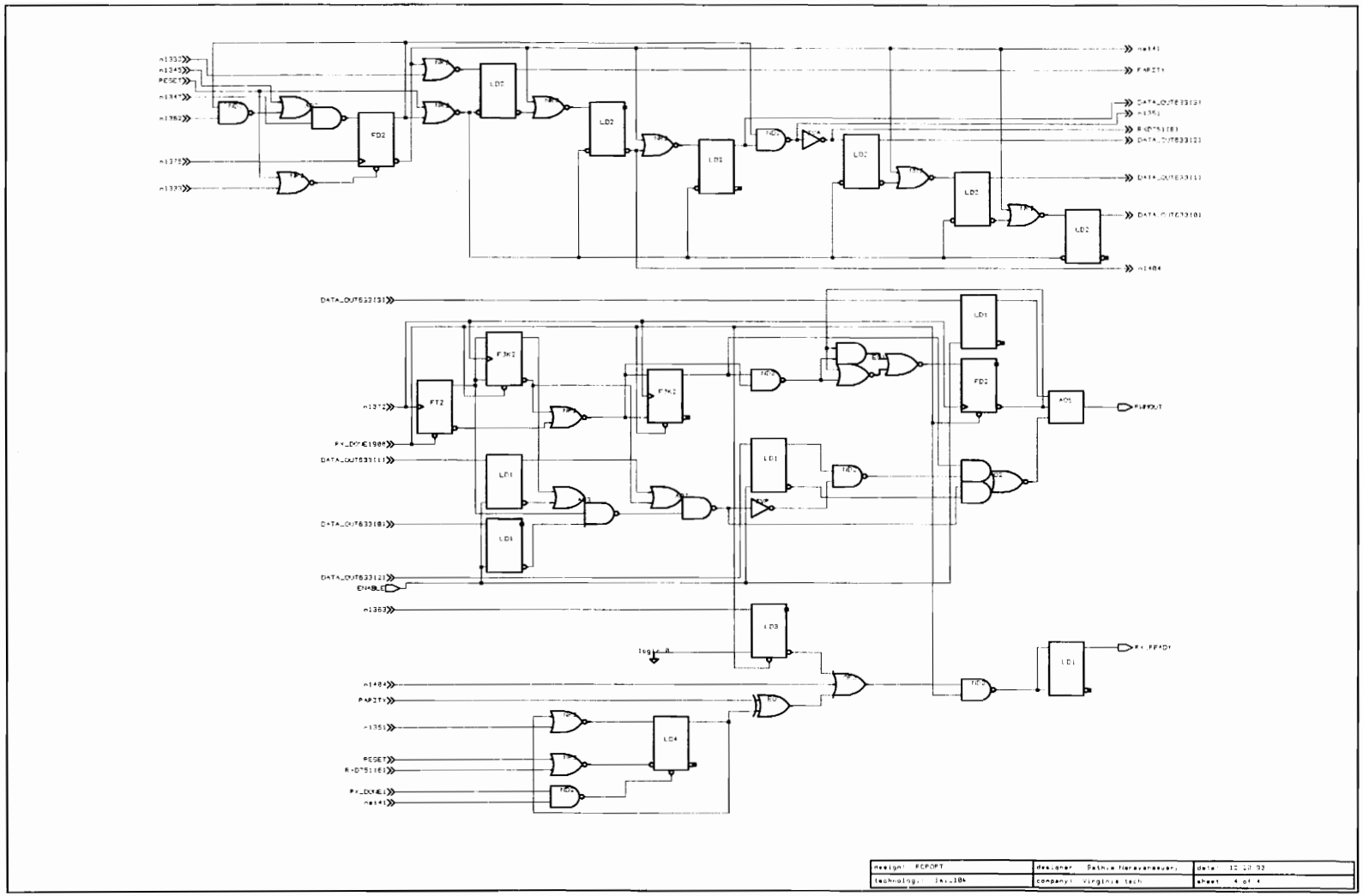


Figure 21 Gate level design of RCPORT (continued)

The VHDL description in Figure 20 had all delays specified using generics and in the back-annotated model of RCPORT produced by BACKANN, the architecture body of RCPORT remains the same while the generic declaration changes to specify the values for the generics. The results of using different technology libraries is shown in the following figures.

```
generic (MUX_DEL: TIME := 0.690000 ns ;
 CNTR4_DEL: TIME := 1.120000 ns ;
 INV_DEL: TIME := 0.150000 ns ;
 OUT16CLKDEL: TIME := 1.455000 ns ;
 OUT3CLKDEL: TIME := 1.455000 ns ;
 OUTCCLKDEL: TIME := 1.550000 ns ;
 OUTBCLKDEL: TIME := 1.550000 ns ;
 OUTACLK_DEL: TIME := 1.550000 ns ;
 COMP_DEL: TIME := 1.855000 ns ;
 DATA_OUTDEL: TIME := 0.000000 ns ;
 RX_READYDEL: TIME := 0.360000 ns ;
 COMP_PARITYDEL: TIME := 0.410000 ns ;
 PARITY_DEL: TIME := 0.410000 ns ;
 STOP_DEL: TIME := 0.410000 ns ;
 RXD_DEL: TIME := 0.815000 ns ;
 RX_DONE1DEL: TIME := 0.380000 ns ;
 RX_DONEDEL: TIME := 0.380000 ns ;
 CLKB_DEL: TIME := 0.460000 ns ;
 RX_BSYDEL: TIME := 0.485000 ns ;
 ISTART_DEL: TIME := 0.455000 ns
 );
```

Figure 22 Modified generic clause for RCPORT with lsi_10k technology

```
generic (MUX_DEL: TIME := 4.100000 ns ;
        CNTR4_DEL: TIME := 8.070000 ns ;
        INV_DEL: TIME :=  0.650000 ns ;
        OUT16CLKDEL: TIME := 0.000000 ns ;
        OUT3CLKDEL: TIME := 0.000000 ns ;
        OUTCCLKDEL: TIME := 8.550000 ns ;
        OUTBCLKDEL: TIME := 8.550000 ns ;
        OUTACLK_DEL: TIME := 8.550000  ns ;
        COMP_DEL: TIME := 7.930000 ns ;
        DATA_OUTDEL: TIME := 0.000000 ns ;
        RX_READYDEL: TIME := 1.850000 ns ;
        COMP_PARITYDEL: TIME := 4.460000 ns ;
        PARITY_DEL: TIME := 3.67000 ns ;
        STOP_DEL: TIME := 3.670000 ns ;
        RXD_DEL: TIME := 6.925000 ns ;
        RX_DONE1DEL: TIME := 3.500000 ns ;
        RX_DONEDEL: TIME := 3.500000 ns ;
        CLKB_DEL: TIME :=  2.460000 ns ;
        RX_BSYDEL: TIME :=  2.765000 ns ;
        ISTART_DEL: TIME := 2.605000 ns
);
```

Figure 23 Modified generic clause for RCPORT with vsc100 technology

When the processes are synthesized to the gate-level individually there is a small increase in the number of components generated because when the Synopsys Design Compiler works on the whole chip, it optimizes the gate level design and processes in the Process Model Graph may share some hardware. In Tables 2 and 3, the gate level design generated for the whole PMG and each individual process is quantized in terms of the total number of cells generated, the total area, the number of

combinational cells generated, the area occupied by the combinational cells, the number of non-combinational cells generated and their area.

Table 2 Synthesis report for RCPORT with lsi_10k technology

Process name	# of cells in process	Area	# of combi. cells	Combi. area	# of non combi cells	area of non comb cells
MUX4	1	6	1	6	0	0
CNTR4	11	50	7	14	4	36
INV_1	1	1	1	1	0	0
INV2	1	1	1	1	0	0
MUX1	1	4	1	4	0	0
CNTR16	14	41	9	16	5	25
CNTR3	7	22	4	7	3	15
CNTR4C	2	23	4	8	3	15
CNTR4B	2	23	4	8	3	15
CNTR4A	2	23	4	8	3	15
COMPARATOR	8	15	8	15	0	0
LATCH4	4	20	0	0	4	20
READY	4	12	3	7	1	5
SREG	27	80	16	25	11	55
BIT_CNTR	5	18	2	3	3	15
SYM_CNTR	37	109	30	41	7	68
START_DET	11	33	8	12	3	21
TOTAL	153	481	103	176	50	305
RCPORT	133	428	87	140	46	288
RATIO	1.15	1.12	1.18	1.26	1.09	1.06

Table 3 Synthesis report for RCPORT with vsc100 technology

Process name	# of cells in process	Area	# of combi. cells	Combi. area	# of non combi cells	area of non comb cells
MUX4	1	4.42	1	4.42	0	0
CNTR4	11	31.7	7	9.58	4	22.12
INV_1	1	0.74	1	0.74	0	0
INV2	1	0.74	1	0.74	0	0
MUX1	1	2.21	1	2.21	0	0
CNTR16	13	30.6	8	12.53	5	18.07
CNTR3	7	16.22	4	5.53	3	10.69
CNTR4C	7	16.59	4	5.90	3	10.69
CNTR4B	7	16.59	4	5.90	3	10.69
CNTR4A	7	16.59	4	5.90	3	10.69
COMPARATOR	12	14.01	12	14.01	0	0
LATCH4	4	13.27	0	0	4	13.27
READY	4	8.85	3	4.79	1	4.06
SREG	27	57.14	16	20.28	11	36.86
BIT_CNTR	5	12.9	2	2.21	3	10.69
SYM_CNTR	39	81.1	32	39.44	7	41.66
START_DET	9	22.12	6	8.85	3	13.27
TOTAL	156	345.79	106	143.03	50	202.76
RCPORT	139	318.14	93	128.66	46	189.49
RATIO	1.12	1.09	1.14	1.11	1.09	1.07

From Table 2 we see that the area occupied by an optimized gate-level design for RCPORT synthesized as a whole, using lsi_10k technology, is 428. However, when BACKANN synthesizes RCPORT, it synthesizes each process individually. If the areas occupied by the gate level designs for each process are summed, the value is 481. This is a 12% increase from the optimized area of 428 for RCPORT, for lsi_10k technology.

From Table 3, for vsc100 technology the optimized area is 318.14. The area for all individual processes summed is 345.79. This is an increase of 9% over the optimized area, for vsc100 technology.

Thus the extra hardware generated because each process is synthesized individually is not more than 12% for RCPORT.

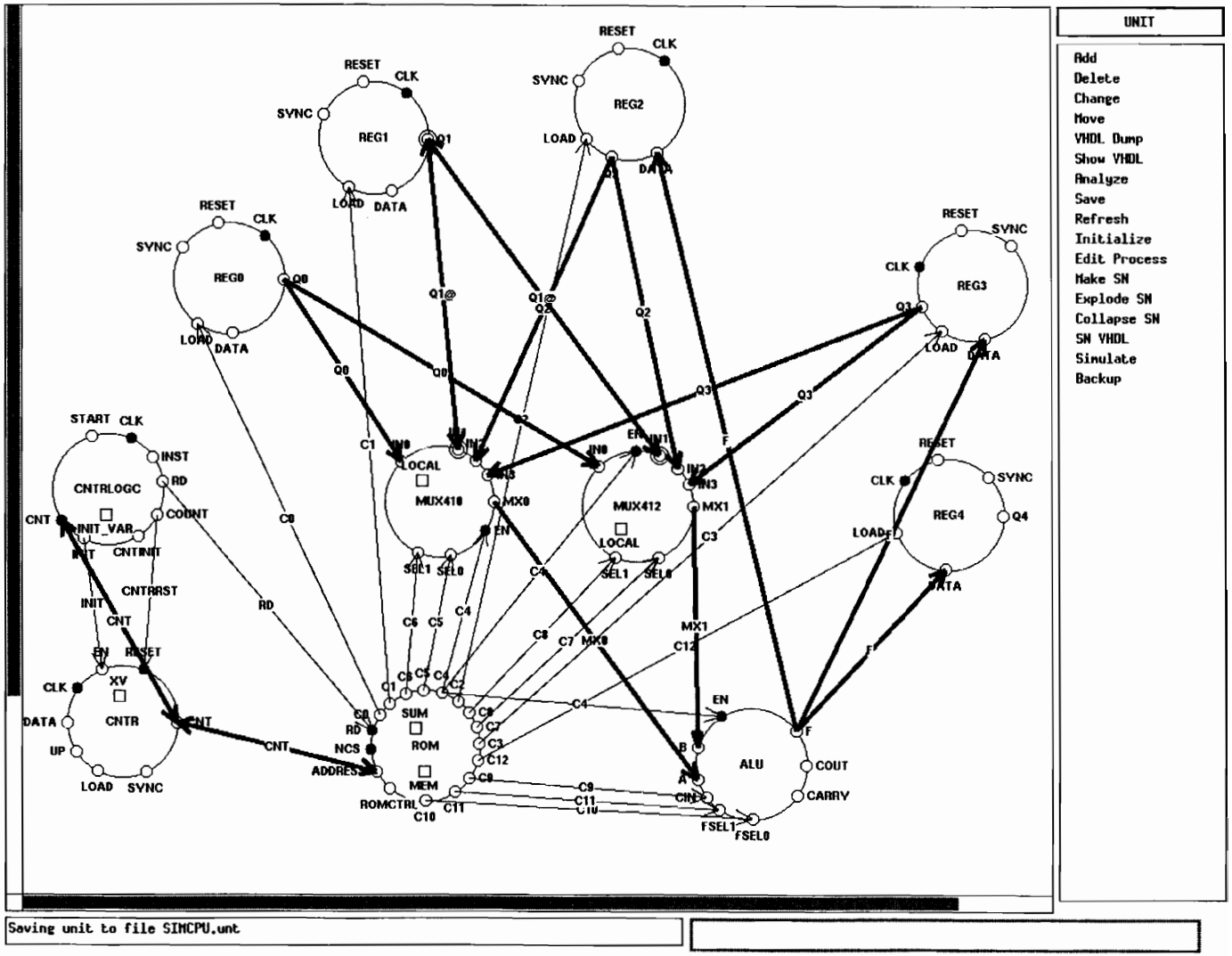


Figure 24 Process Model Graph of SIMCPU

5.2 SIMCPU

SIMCPU is a simple CPU implementation with a ROM and a set of registers, an Arithmetic and Logic Unit and two multiplexers. The Process Model Graph of SIMCPU is given in Figure 24. The VHDL description of SIMCPU is given in Figure 25.

```
-- ************************************************************
entity SIMCPU is
generic (RD_DEL: TIME;
CNTRRST_DEL: TIME;
INIT_DEL: TIME;
CNT_INI_DEL: TIME;
COUT_DEL: TIME;
CARRY_DEL: TIME;
F_DEL: TIME;
REG4_DEL: TIME;
REG3_DEL: TIME;
REG2_DEL: TIME;
MUX1_DEL: TIME;
REG1_DEL: TIME;
MUX2_DEL: TIME;
REG0_DEL: TIME;
ROMCTRL_DEL: TIME;
CNT_DEL: TIME
);
port (CLK: in BIT;
START: in BIT;
CNTINIT: out BIT_VECTOR(3 downto 0);
```

Figure 25 VHDL behavioral description of SIMCPU

```
INST: in BIT;
COUT: out BIT;
CARRY: inout BIT_VECTOR(4 downto 0);
Q4: out BIT_VECTOR(3 downto 0);
SYNC: in BIT;
RESET: in BIT;
DATA: in BIT_VECTOR(3 downto 0);
ROMCTRL: inout BIT_VECTOR(12 downto 0);
NCS: in BIT;
LOAD: in BIT;
UP: in BIT;
Q1: inout BIT_VECTOR(3 downto 0)
);
end SIMCPU;
-- ************************************************************
architecture BEHAVIORAL of SIMCPU is
signal INIT: BIT;
signal CNT: BIT_VECTOR(3 downto 0);
signal RD: BIT;
signal CNTRRST: BIT;
signal C10: BIT;
signal C11: BIT;
signal F: BIT_VECTOR(3 downto 0);
signal C4: BIT;
signal C9: BIT;
signal MX1: BIT_VECTOR(3 downto 0);
signal MX0: BIT_VECTOR(3 downto 0);
signal C12: BIT;
signal Q3: BIT_VECTOR(3 downto 0);
signal C3: BIT;
signal Q2: BIT_VECTOR(3 downto 0);
signal C2: BIT;
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

```
signal C7: BIT;
signal C8: BIT;
signal Q0: BIT_VECTOR(3 downto 0);
signal C1: BIT;
signal C5: BIT;
signal C6: BIT;
signal C0: BIT;
begin
  -- ---------------------------------------------------
  -- Process Name: CNTRLOGC
  -- ---------------------------------------------------
CNTRLOGC_4: process (CLK,CNT)
variable INIT_VAR : BIT;
begin
        if (START = '1' ) then
                if(CLK = '1' ) then
                        if INST = '0' then
                                CNTINIT <= "0000" after CNT_INI_DEL;
                                INIT_VAR := not INIT_VAR ;
                                CNTRRST <= '1' after CNTRRST_DEL;
                                RD <= '1' after RD_DEL
                        end if;
                        if INST = '1' then
                                CNTINIT <= "1010" after CNT_INI_DEL;
                                 INIT_VAR := not INIT_VAR;
                                CNTRRST <= '1' after CNTRRST_DEL;
                                RD <= '1' after RD_DEL;
                        end if;
                end if;
                if (CNT = "1001" or CNT = "1110") then
                        CNTRRST <= '0' after CNTRRST_DEL;
                end if;
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

```
                RD <= '1' after RD_DEL;
        end if;
        INIT <= INIT_VAR after INIT_DEL;
end process CNTRLOGC_4;

 -- ---------------------------------------------------
 -- Process Name: ALU
 -- ---------------------------------------------------
ALU_19: process (C4)
begin
        if(C4 = '1') then
                if (C11 = '0' and C10 = '0') then
                        F <= MX0 after F_DEL;
                end if;
                if (C11 = '0' and C10 = '1') then
                        F <= not MX0 after F_DEL;
                end if;
                if (C11 = '1' and C10 = '0') then
                        CARRY <= "00000" after CARRY_DEL;
                        CARRY(0) <= C9 after CARRY_DEL;
                        for I in 0 to 3 loop
                                CARRY(I + 1) <= (MX0(I) and MX1(I)) or (MX0(I) and
CARRY(I)) or
                                (MX1(I) and CARRY(I)) after CARRY_DEL;
                        end loop;
                        COUT <= CARRY(4) after COUT_DEL;
                        for I in 0 to 3 loop
                                F(I) <= (MX0(I) xor MX1(I) xor CARRY(I)) after F_DEL;
                        end loop;
                end if;
                if (C11 = '1' and C10 = '1') then
                        F <= MX0 and MX1 after F_DEL;
                 end if;
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

```
        end if;
end process ALU_19;

 -- ---------------------------------------------------
 -- Process Name: REG4
 -- ---------------------------------------------------
REG4_33: process (CLK)
begin
        if(RESET = '1') then
                Q4 <= "0000" after REG4_DEL;
        end if;
        if(CLK = '1') then
                if(C12 = '1' and RESET = '0') then
                        Q4 <= F after REG4_DEL;
                end if;
         end if;
        if( (CLK = '1') and ((SYNC = '1' and
                RESET = '1') or (SYNC = '0' and RESET = '1') ) ) then
                Q4 <= "0000" after REG4_DEL;
        end if;
end process REG4_33;

 -- ---------------------------------------------------
 -- Process Name: REG3
 -- ---------------------------------------------------
REG3_42: process (CLK)
begin
        if(RESET = '1') then
                Q3 <= "0000" after REG3_DEL;
        end if;
        if(CLK = '1') then
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

```
                if(C3 = '1' and RESET = '0') then
                        Q3 <= F after REG3_DEL;
                 end if;
        end if;
        if( (CLK = '1') and ((SYNC = '1' and RESET = '1') or (SYNC = '0' and RESET =
'1') ) ) then
                Q3 <= "0000" after REG3_DEL;
        end if;
end process REG3_42;

  -- ------------------------------------------------------
  -- Process Name: REG2
  -- ------------------------------------------------------
REG2_51: process (CLK)
begin
        if(RESET = '1') then
                Q2 <= "0000"  after REG2_DEL;
        end if;
        if(CLK = '1') then
                if(C2 = '1' and RESET = '0') then
                        Q2 <= F after REG2_DEL;
                end if;
        end if;
        if( (CLK = '1') and ((SYNC = '1' andRESET = '1') or (SYNC = '0' and RESET =
'1') ) ) then
                Q2 <= "0000" after REG2_DEL;
        end if;
end process REG2_51;
  -- ------------------------------------------------------
  -- Process Name: MUX412
  -- ------------------------------------------------------
MUX412_60: process (C4)
variable LOCAL: BIT_VECTOR(1 downto 0);
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

```
begin
        if C4 = '1' then
                LOCAL := C8&C7;
                 case LOCAL is
                        when "00" => MX1 <= Q0 after MUX1_DEL;
                        when "01" => MX1 <= Q1 after MUX1_DEL;
                        when "10" => MX1 <= Q2 after MUX1_DEL;
                        when "11" => MX1 <= Q3 after MUX1_DEL;
                end case;
        end if;
end process MUX412_60;

 -- ---------------------------------------------------
 -- Process Name: REG1
 -- ---------------------------------------------------
REG1_72: process (CLK)
begin
        if(RESET = '1') then
                Q1 <= "0000" after REG1_DEL;
        end if;
        if(CLK = '1') then
                if(C1 = '1' and RESET = '0') then
                        Q1 <= DATA after REG1_DEL;
                end if;
        end if;
        if( (CLK = '1') and ((SYNC = '1' and RESET = '1') or (SYNC = '0' and RESET =
'1') ) ) then
                        Q1 <= "0000" after REG1_DEL;
        end if;
end process REG1_72;

 -- ---------------------------------------------------
 -- Process Name: MUX410
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

```
 -- ------------------------------------------------------
MUX410_81: process (C4)
variable LOCAL: BIT_VECTOR(1 downto 0);
begin
        if C4 = '1' then
                LOCAL := C6&C5;
                case LOCAL is
                        when "00" => MX0 <= Q0 after MUX2_DEL;
                        when "01" => MX0 <= Q1 after MUX2_DEL;
                        when "10" => MX0 <= Q2 after MUX2_DEL;
                        when "11" => MX0 <= Q3 after MUX2_DEL;
                end case;
        end if;
end process MUX410_81;

 -- ------------------------------------------------------
 -- Process Name: REG0
 -- ------------------------------------------------------
REG0_93: process (CLK)
begin
        if(RESET = '1') then
                Q0 <= "0000" after REG0_DEL ;
        end if;
        if(CLK = '1') then
                if(C0 = '1' and RESET = '0') then
                        Q0 <= DATA after REG0_DEL;
                end if;
        end if;
        if( (CLK = '1') and ((SYNC = '1' and RESET = '1') or (SYNC = '0' and RESET =
'1') ) ) then
                Q0 <= "0000" after REG0_DEL;
        end if;
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

```
end process REG0_93;

  -- ---------------------------------------------------
  -- Process Name: ROM
  -- ---------------------------------------------------
ROM_102: process (RD,NCS)
variable SUM: INTEGER range 0 to 15;
type MEMORY is array (0 to 14) of BIT_VECTOR(12 downto 0) ;
variable MEM: MEMORY := ("0000000000001",
 "0000000000010",
 "0010000111000",
 "0110110001000",
 "0010001111000",
 "0010000000100",
 "0110011010100",
 "0010001000100",
 "0110111010100",
 "1010001000000",
 "0000000010001",
 "0000000000010",
 "0010000110100",
 "0101100000100",
 "1100101010000")
;
begin
        if (NCS = '0' and RD = '1' ) then
                for N in CNT'LOW to CNT'HIGH loop
                        if CNT(N) = '1' then
                                SUM := SUM + (2 ** N);
                        end if;
                end loop;
                ROMCTRL <= MEM(SUM) after ROMCTRL_DEL;
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

```
                C12 <= ROMCTRL(12) after ROMCTRL_DEL;
                C11 <= ROMCTRL(11) after ROMCTRL_DEL;
                C10 <= ROMCTRL(10) after ROMCTRL_DEL;
                C9 <= ROMCTRL(9) after ROMCTRL_DEL;
                C8 <= ROMCTRL(8) after ROMCTRL_DEL;
                C7 <= ROMCTRL(7) after ROMCTRL_DEL;
                C6 <= ROMCTRL(6) after ROMCTRL_DEL;
                C5 <= ROMCTRL(5) after ROMCTRL_DEL;
                C4 <= ROMCTRL(4) after ROMCTRL_DEL;
                C3 <= ROMCTRL(3) after ROMCTRL_DEL;
                C2 <= ROMCTRL(2) after ROMCTRL_DEL;
                C1 <= ROMCTRL(1) after ROMCTRL_DEL;
                C0 <= ROMCTRL(0) after ROMCTRL_DEL;
        end if;
end process ROM_102;

 -- ------------------------------------------------------
 -- Process Name: CNTR
 -- ------------------------------------------------------
CNTR_126: process (CNTRRST,CLK)
variable XV: BIT_VECTOR(3 downto 0);
begin
        if (CNTRRST = '1'and SYNC = '0') or (CNTRRST = '1' and SYNC = '1' and CLK
= '1' ) then
                CNT <= "0000" after CNT_DEL;
        end if;
        if (LOAD = '1' and CLK = '1' ) then
                CNT <= DATA after CNT_DEL;
        end if;
        if (INIT = '1' and CLK = '1' ) then
                if UP = '1' then
                        XV := CNT;
                        for I in 0 to 3 loop
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

```
                    if XV(I) = '0' then
                        XV(I) := '1';
                        exit;
                    else
                        XV(I) := '0';
                    end if;
                end loop;
            else
                XV := CNT;
                for I in 0 to 3 loop
                    if XV(I) = '1' then
                        XV(I) := '0';
                        exit;
                    else
                        XV(I) := '1';
                    end if;
                end loop;
            end if;
            CNT <= XV after CNT_DEL;
        end if;
end process CNTR_126;
end BEHAVIORAL;
```

Figure 25 VHDL behavioral description of SIMCPU (continued)

In the VHDL behavioral description of SIMCPU all the signal assignments had generic delays associated with them and after the back annotation of the delays by BACKANN the architecture body of SIMCPU remains unchanged, but the generic clause is changed to initialize the generic delays and these are shown in Figures 27 and 28.

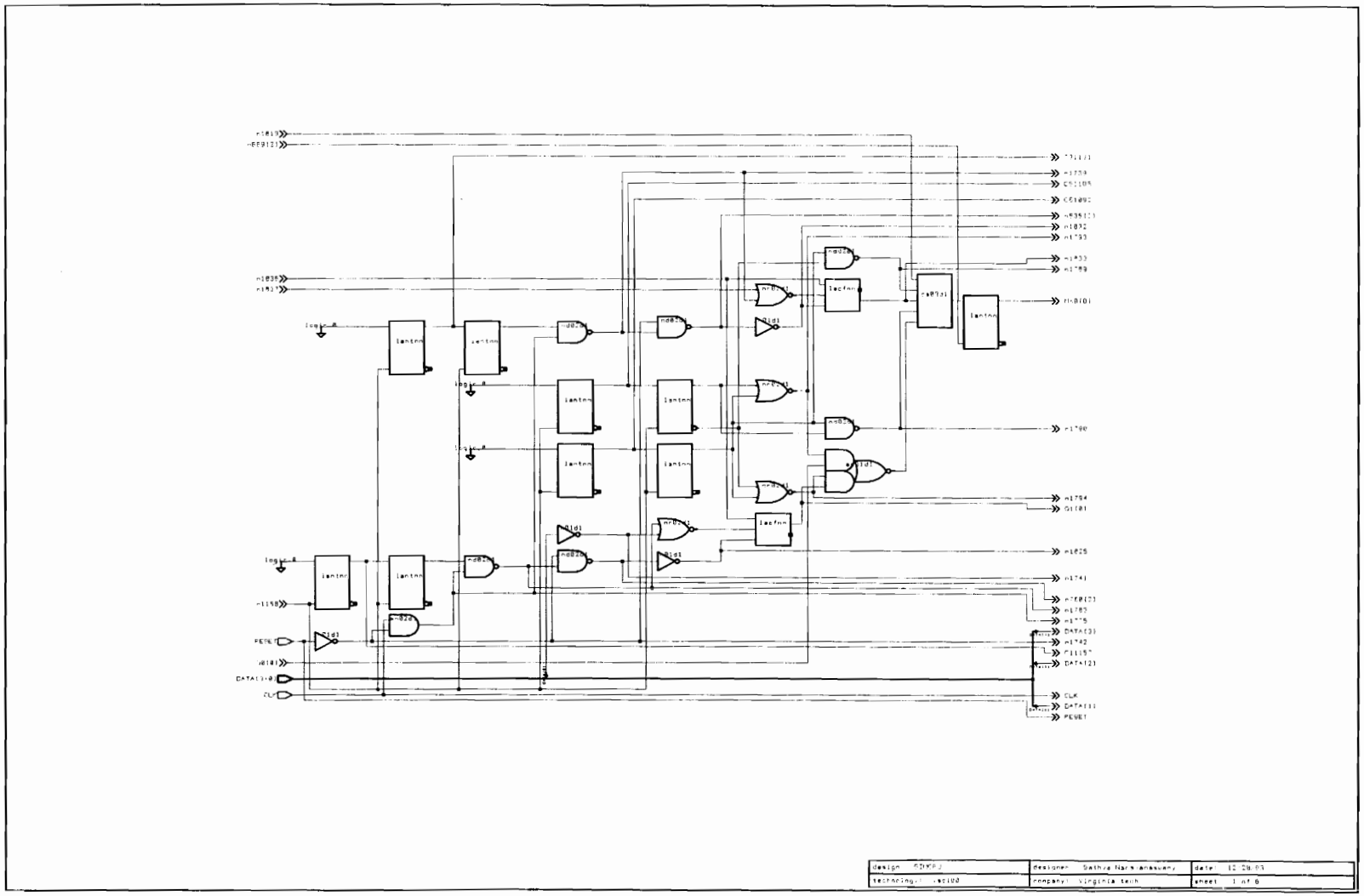


Figure 26 Gate level design of SIMCPU

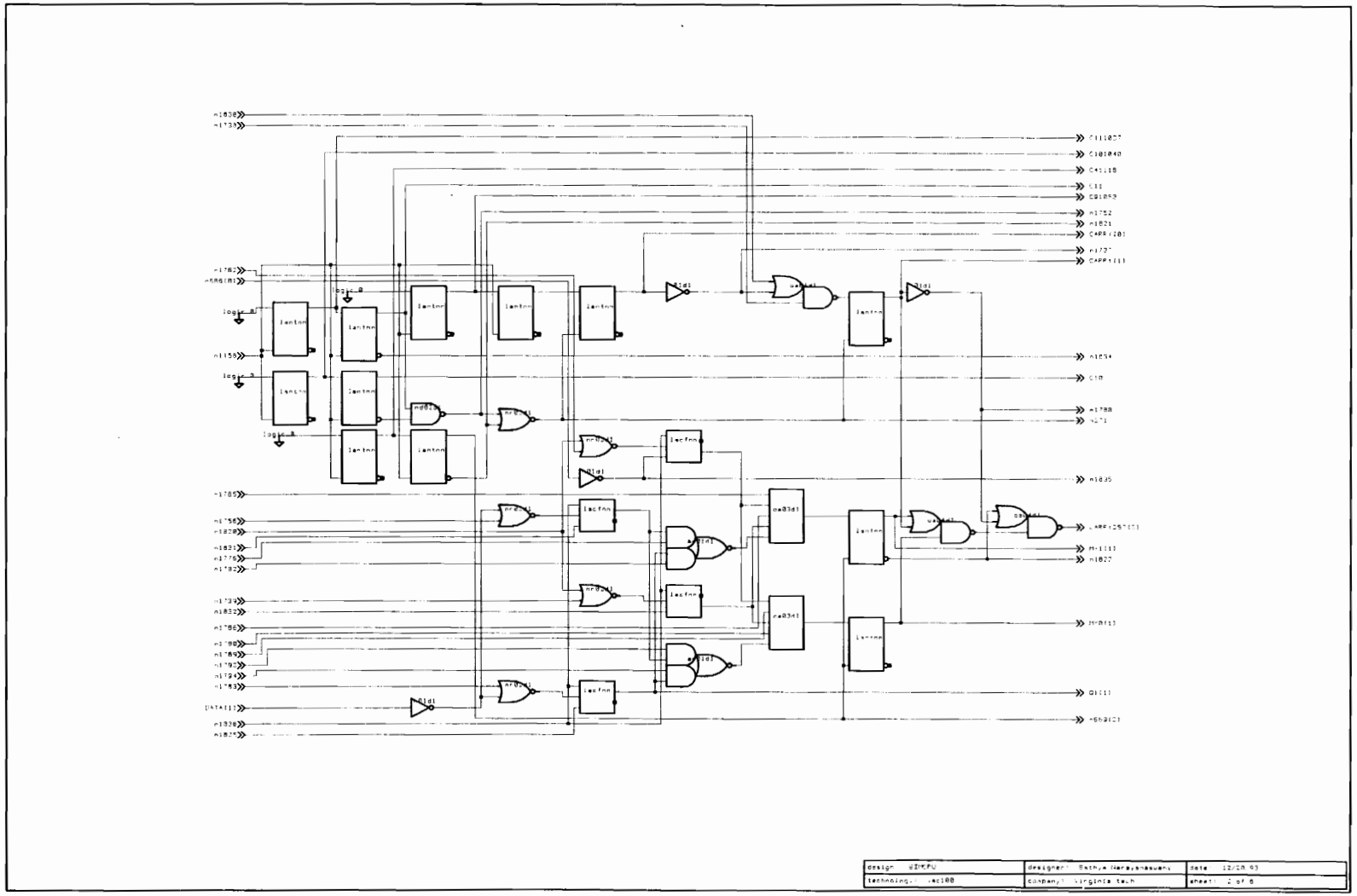


Figure 26 Gate level design of SIMCPU (continued)

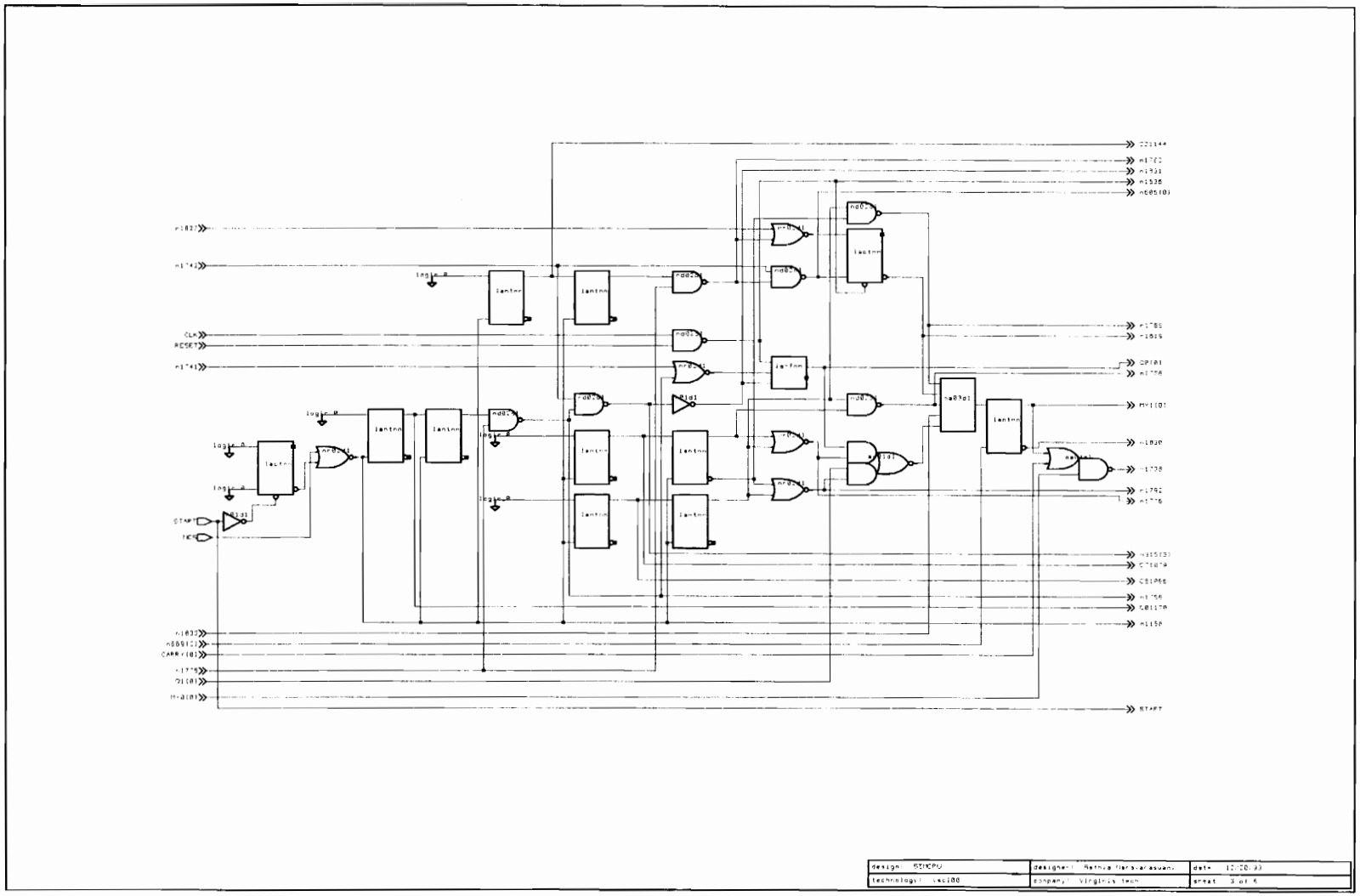


Figure 26 Gate level design of SIMCPU (continued)

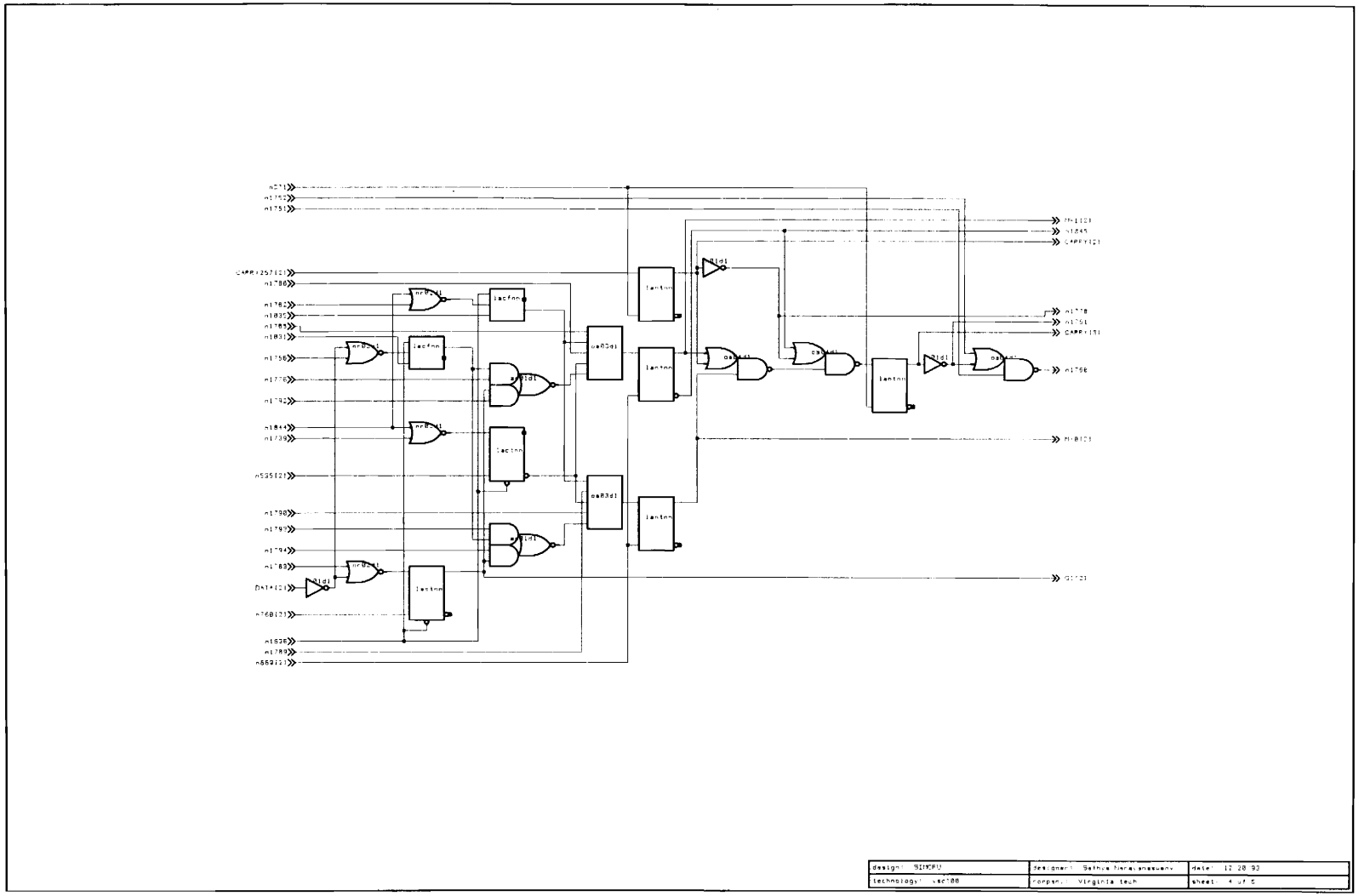


Figure 26 Gate level design of SIMCPU (continued)

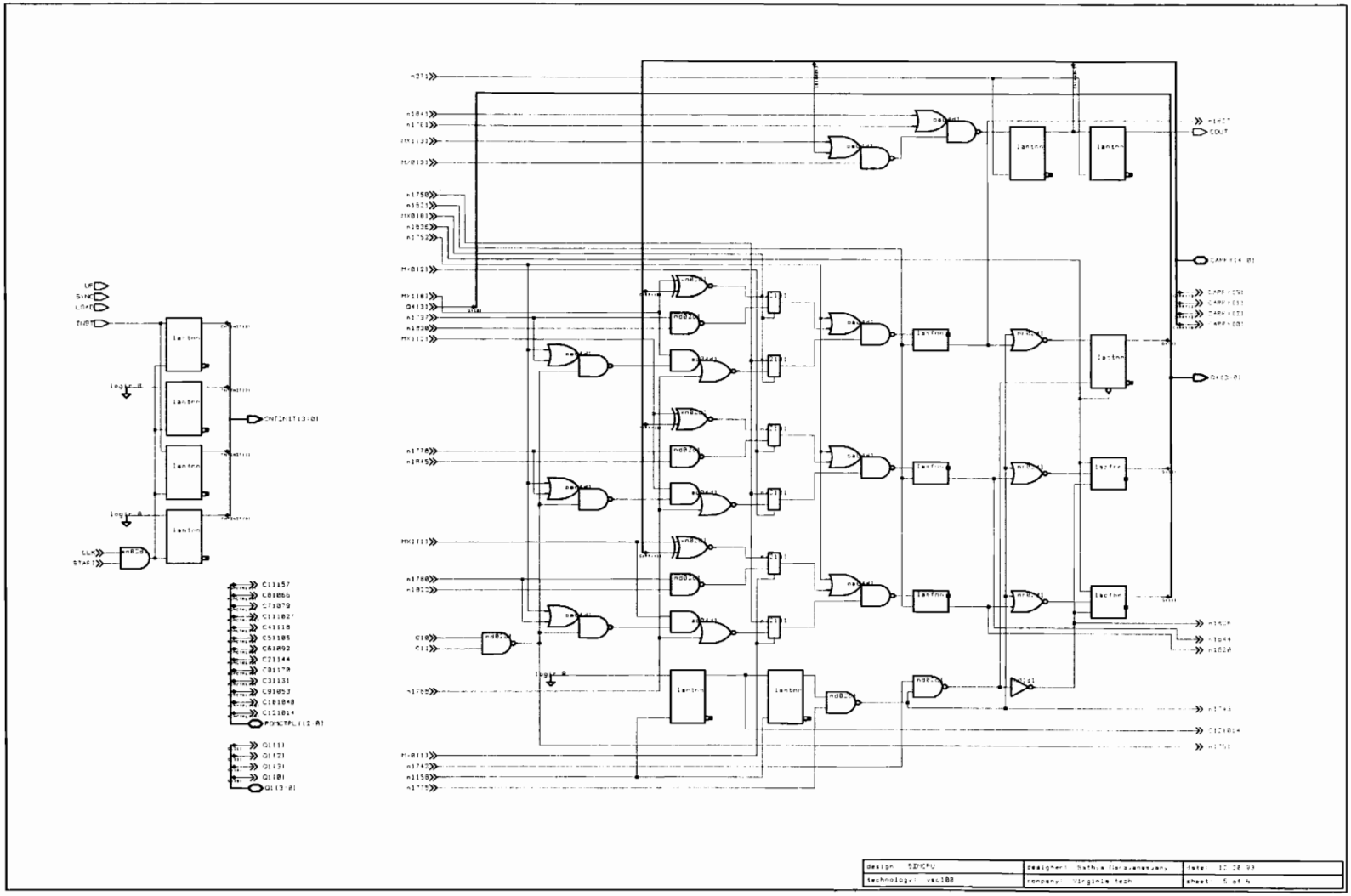


Figure 26 Gate level design of SIMCPU (continued)

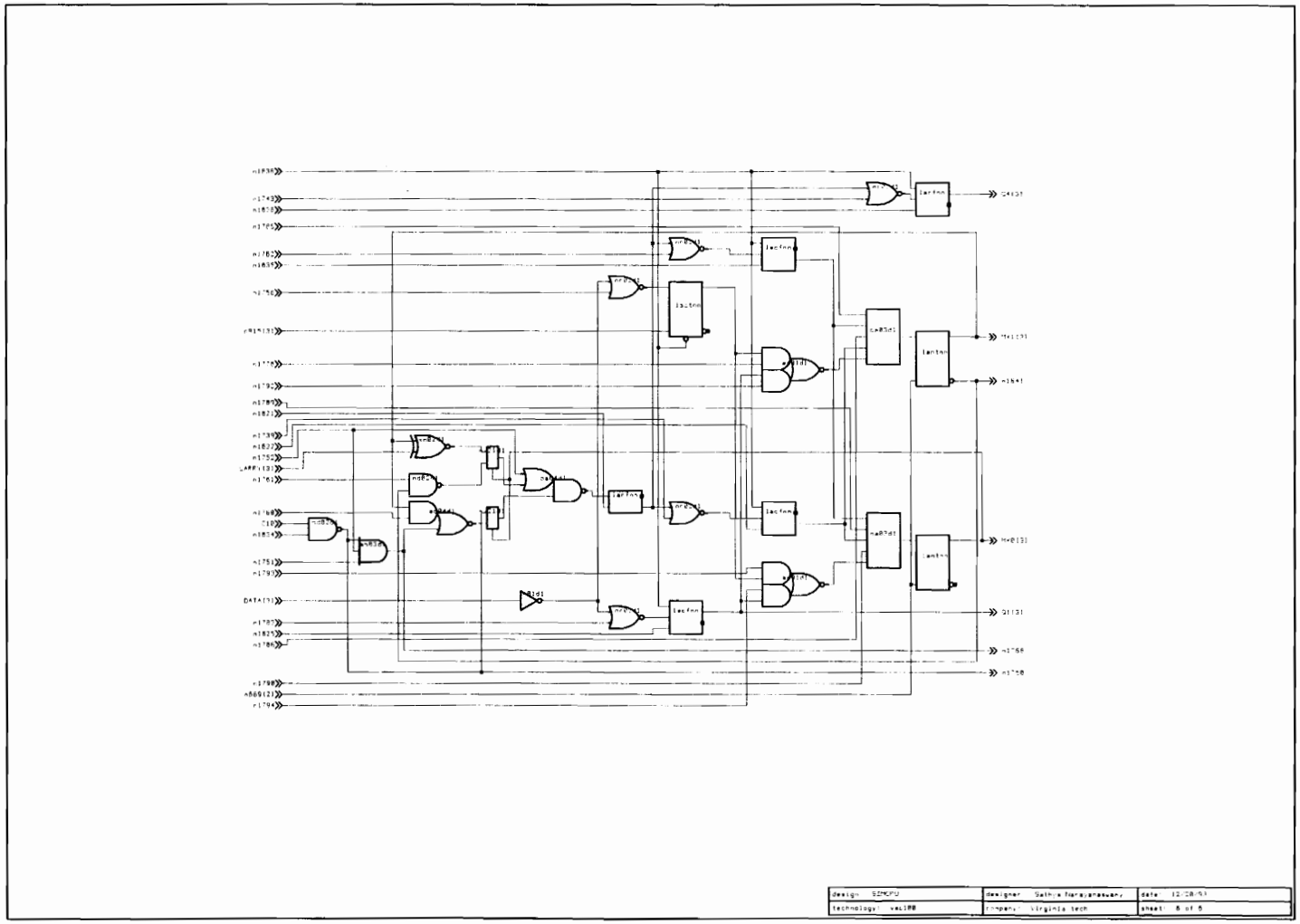


Figure 26 Gate level design of SIMCPU (continued)

```
generic (RD_DEL: :TIME := 0.355000 ns ;
        CNTRRST_DEL: :TIME := 0.310000 ns ;
        INIT_DEL: :TIME := 0.950000 ns ;
        CNT_INI_DEL: :TIME := 0.495000 ns ;
        COUT_DEL: :TIME := 0.495000 ns ;
        CARRY_DEL: :TIME := 0.850000 ns ;
        F_DEL: :TIME := 0.000000 ns ;
        REG4_DEL: :TIME := 0.685000 ns ;
        REG3_DEL: :TIME := 0.685000 ns ;
        REG2_DEL: :TIME := 0.685000 ns ;
        MUX1_DEL: :TIME := 0.000000 ns ;
        REG1_DEL: :TIME := 0.785000 ns ;
        MUX2_DEL: :TIME := 0.000000 ns ;
        REG0_DEL: :TIME := 0.685000 ns ;
        ROMCTRL_DEL: :TIME := 0.575000 ns ;
        CNT_DEL: TIME := 1.320000 ns
);
```

Figure 27 Modified generic clause for SIMCPU with lsi_10k technology

```
generic (RD_DEL: TIME := 1.170000 ns ;
        CNTRRST_DEL: TIME := 1.005000 ns ;
        INIT_DEL: TIME := 2.570000 ns ;
        CNT_INI_DEL: TIME := 1.860000 ns ;
        COUT_DEL: TIME := 1.860000 ns ;
        CARRY_DEL: TIME := 2.130000 ns ;
        F_DEL: TIME := 0.000000 ns ;
        REG4_DEL: TIME := 2.220000 ns ;
        REG3_DEL: TIME := 2.220000 ns ;
        REG2_DEL: TIME := 2.220000 ns ;
        MUX1_DEL: TIME := 0.000000 ns ;
```

Figure 28 Modified generic clause for SIMCPU with vsc100 technology

```
        REG1_DEL: TIME := 2.285000 ns ;
        MUX2_DEL: TIME := 0.000000 ns ;
        REG0_DEL: TIME := 2.220000 ns ;
        ROMCTRL_DEL: TIME := 1.940000 ns ;
        CNT_DEL: TIME := 3.160000 ns
    );
```

Figure 28 Modified generic clause for SIMCPU with vsc100 technology (continued)

The tables below list the hardware synthesized for the different processes and the model as a whole, by the Synopsys Design Compiler.

Table 4 Synthesis report for SIMCPU with lsi_10k technology

Process name	# of cells in process	Area	# of combi. cells	Combi. area	# of noncombi cells	Area of noncombi cells
CNTRLOGC	23	59	16	24	7	35
ALU	63	135	53	85	10	50
REG0	14	34	10	14	4	20
REG1	14	34	10	14	4	20
REG2	14	34	10	14	4	20
REG3	14	34	10	14	4	20
REG4	14	34	10	14	4	20
MUX40	22	46	18	26	4	20
MUX41	22	46	18	26	4	20
ROM	38	142	12	12	26	130
CNTR	40	81	36	61	4	20
TOTAL	278	679	203	304	75	375
SIMCPU	226	548	157	203	69	345
RATIO	1.15	1.23	1.29	1.49	1.08	1.09

Table 5 Synthesis report for SIMCPU with vsc100 technology

Process name	# of cells in process	Area	# of combi. cells	Combi. area	# of noncombi cells	Area of noncombi cells
CNTRLOGC	20	40.92	13	17.33	7	23.59
ALU	67	110.59	57	77.41	10	33.18
REG0	14	26.54	10	11.80	4	14.74
REG1	14	26.54	10	11.80	4	14.74
REG2	14	26.54	10	11.80	4	14.74
REG3	14	26.54	10	11.80	4	14.74
REG4	14	26.54	10	11.80	4	14.74
MUX40	8	30.97	4	17.70	4	13.27
MUX41	8	30.97	4	17.70	4	13.27
ROM	38	97.32	12	11.06	26	86.26
CNTR	53	74.10	49	60.83	4	13.27
TOTAL	246	517.57	189	261.03	75	243.27
SIMCPU	223	425.78	154	189.11	69	236.67
RATIO	1.10	1.21	1.22	1.31	1.08	1.03

From Table 4 we see that the area occupied by an optimized gate-level design for SIMCPU synthesized as a whole, using lsi_10k technology, is 548. However, when BACKANN synthesizes SIMCPU, it synthesizes each process individually. If the areas occupied by the gate level designs for each process are summed, the value is 679. This is a 23% increase from the optimized area of 548 for SIMCPU, for lsi_10k technology.

From Table 5, for vsc100 technology the optimized area is 425.78. The area for all individual processes summed is 517.57. This is an increase of 21% over the optimized area, for vsc100 technology.

Thus the extra hardware generated because each process is synthesized individually is not more than 23% for SIMCPU.

Chapter 6

Suggestions

BACKANN is a powerful tool, but certain inadequacies do exist in it. These inadequacies arise because it has to work with other software which were designed with the same philosophy or for back-annotation.

6.1 Synopsys Design Compiler - Non-synthesizable VHDL constructs

When the individual processes in a VHDL model are synthesized, there might be some VHDL constructs which are not synthesizable. This could result in warning messages being issued by the Synopsys Design Compiler on the computer's console. BACKANN however will not be able to deduce that an error has occurred and thus proceeds with its task of back-annotation and will exit in the middle of the process because the required data would not be available. Thus, The user has to keep watch on the console to monitor the error messages issued by the Synopsys Design Compiler. It coule be possible with future versions of the Synopsys Design Compiler and BACKANN to get the error messages about non-synthesizable code to be written to an error file which

can be analyzed by BACKANN and then used by the designer to correct the VHDL model and conform it to a synthesizable format.

6.2 Redundant Synthesis

BACKANN synthesizes each process in the VHDL model created using the Modeler's Assistant. The Modeler's Assistant aids the development of VHDL models using a library of process primitives. BACKANN could be modified such that it also has a library of delays for the process primitives in the Modeler's Assistant library. Whenever a process primitive is found in a VHDL model, BACKANN can access its library to get the delays for that particular technology. This speeds up the design cycle as it is the synthesizer that uses up most of the time required to produce a working design. BACKANN's library could contain all the delays for all the process primitives, for various technologies.

Chapter 7

Conclusion

This thesis has presented a design tool - BACKANN. The tool is used to back annotate realistic timing delays into VHDL behavioral models. The use of the Process Model Graph and the Modeler's Assistant, in the task was explained. The working of BACKANN, using the CLSI tools and the Synopsys Design Compiler to evaluate the timing delays was explained in detail in Chapter 4.

The software was tested on two complex VHDL behavioral models and the results obtained were presented. These results were analyzed and the penalty arising due to the synthesis of each process in the Process Model Graph separately was found. Suggestions to enhance BACKANN have also been presented.

VHDL behavioral models with realistic timing can be developed by back-annotating delays using BACKANN.

Bibliography

[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] M. Donlin, "ASIC Complexity fuels drive to HDL design," Computer Design, May 1991.

[3] D. D. Gajski, "Essential Issues and Possible Solutions in High-Level Synthesis," High Level VLSI Synthesis, Kluwer Academic Publishers, 1991.

[4] J. R. Armstrong and F. G. Gray, "Structured Logic Design with VHDL," Prentice Hall, 1993.

[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] IEEE Standard VHDL Language Reference Manual, 1988.

[9] D. Giles, C. Berking, K. Wacks, "Integrated Functional/Structural Timing for Digital Simulation," IEEE Test Conference, pp.153-160, April 1982.

[10] A. Gadagkar and J.R.Armstrong, "Timing Distribution in VHDL behavioral Models," Proceedings of ICCAD 1992, pp. 82-89.

[11] A. Gadagkar, "Timing Distribution in VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.

[12] S. R. Rao, B. Pan, J. R. Armstrong, "Hierarchical Test Generation for VHDL Behavioral Models,"Proceedings of the European Design Automation Conference," Feb 1993.

[13] S. R. Rao, "A Hierarchical Approach to Effective Test Generation for VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1993.

[14] Synopsys Inc., "VHDL Compiler Reference Manual", Nov 1992

[15] B. Singh, J. Wicks, P. Wright, J. R. Armstrong, "The Modeler's Assistant : A CAD Tool for Behavioral Model Development", Proceedings of CHDL 1993, pp. 347-354.

[16] F. Vahid, S. Narayan, D. D. Gajski,, "SpecCharts: A Language for System Level Synthesis," Proceedings of 15 1991, pp. 165-174.

[17] J. Lahti, J. Kivela, "Logic Compilation from Graphical Dependency Notation," ICCAD 90, pp.474-477, Nov 1990.

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

[19] B. Singh, "A Parametrized CAD tool for VHDL Model Development with X-Windows," Master's Thesis, Virginia Polytechnic Institute and State University, 1990.

[20] P. A. Wright, "Rapid Development of VHDL Behavioral Models," Master's Thesis, Virginia Polytechnic Institute and State University, 1992.

[21] C. S. Kosaraju and J. R. Armstrong, "A Set of Behavioral Modeling Primitives," SOUTHEASTCON 1990, pp.610-613.

[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] B. W. Kernighan and D. M. Ritchie, "The C Programming Language," Prentice Hall, 1977.

[25] E. Horowitz, S. Sahni, S. Anderson-Freed, "Fundamentals of Data Structures in C," Computer Science Press, 1993.

[26] R. Camposano, "From Behavior to Structure: High-Level Synthesis," IEEE Design and Test of Computers, pp.8-19.

[27] Synopsys Inc., "Design Compiler Reference Manual," Dec 1992.

[28] CAD Language Systems, "VHDL Analyzer Designer's Manual," April 1993.

[29] CAD Language Systems, "Design Library System," April 1991.

[30] CAD Language Systems, "DLS Application Development : The Software Procedural Interface," March 1993.

[31] Synopsys Inc., "Command Reference ," Dec 1992.

[32] E. A. Rundensteiner and D Gajski, "Functional Synthesis using Area and Delay Optimization," 29th ACM/IEEE Design Automation Conference, pp.291-296.

[33] D. Coelho, "VHDL : A Call for Standards," 25th ACM/IEEE Design Automation Conference, pp.40-47.

Appendix A

User Guide for BACKANN

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

Behavioral Model Creation

The designer creates a behavioral VHDL 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 BACKANN. The command to analyze the model is

vhdl -preserve_case *VHDL_filename*

Creating the log_file

The user has to create a log_file which contains the specifications and constraints to be adhered to by the Synopsys Design Compiler. This file is named

entity_name_of_model.log

The log_file also has as its last line the *compile* command with the different options to be used for it. This is the command to generate a gate-level design. A typical log_file is shown in Figure 15.

Invoking BACKANN

BACKANN is invoked with the name of a VHDL file as input. It also has three different command-line options which may or may not be specified. The command line options are specified using the letters *m*, *s* and *d*.

The format for invoking BACKANN is

backann *VHDL_filename [m] [s] [d]*

The letters *m*,*s* and *d* are listed in brackets as they are optional. These options are discussed below.

The option *m* specifies that the ***minimum delay mode*** be used when back-annotating delays. If this option is not specified on the command line, BACKANN assumes that the ***maximum delay mode*** has to be used.

The option *s* specifies that the ***separate rise-and-fall model*** be used when back-annotating delays. If this option is not specified on the command line, BACKANN assumes that the ***combined rise-and-fall model*** is to be used.

The option *d* specifies that a the gate level design for the model be written out in *db* format. The *db* format is an internal format of Synopsys. The gate-level design maybe viewed by loading this *db* file into the Synopsys Design Compiler. The *db* file is named

entity_name_of_model.db

and is stored in the directory from which BACKANN is invoked.

BACKANN stores the back-annotated VHDl code in a file named

VHDL_filename.new

BACKANN takes the name of the VHDL file given to it as input and appends the extension ".new" to it. This file contains the back-annotated VHDL and is stored in the directory from which BACKANN was invoked.

Appendix B

Programmer's Guide for BACKANN

The Programmer's Guide explains the different data structures used by BACKANN and the procedures for compilation. The software is structured into a number of C language source files. They are also explained briefly.

annot.c

This file contains the *main* routine of a C program. It invokes all the other routines that are in the other files. It starts by checking the comman line options specified and sets the appropriate flags. The functions it calls and their task is listed below.

get_entity_name	This function extracts the entity name of the VHDL model.
get_arch_name	This function extracts the architecture name of the VHDL model.

build_port_list	This function builds a linked list of the names of the odel's ports and generics and their types.
replace_generic_decln	This function replaces the generic clause in the entity declaration and stores the new code in a temporary file.
replace_generic	This function replaces the generics in the architecture body by symbolic values, following the lgorithm in Figure 12.
before_build_sigass_list	This function builds a linked list of all the signal assignments, following the algorithm in Figure 13.
extract	This is the routine which extracts information about the Process Model Graph and also creates VHDL entities for each process foloowing the algorithm in Figure 11.
build_synopsys_script t	This function invokes the Design Compiler followin e algorithm in Figure 16.
get_output_sig	This function parses the out_file generated by the Synopsys Design Compiler for the different delays required.
proc_vhdl	This function back-annotates the delays and writes the new code into a file.

extract.c

This file contains the extract routine and was developed for the Hierarchical Behavioral Test Generator. It invokes the functions in ***his.c***. It was modified to conform to the requirements of BACKANN.

his.c

This file contains the functions that access the Modeler's Assistant database.

misc.c

This file contains the functions *replace_generic* and *replace_generic_decln.*

parse.c

This file contains the functions *get_entity_name, get_arch_name, build_port_list, build_synopsys_script* and *before_build_sigass_list.* It also contains the functions *build_sigass_list, copy_log_file, return_generic_number* and *return_generic_name*, which are called by the other functions in the file.

sigext.c

This file contains the function *get_output_sig*.

vhdl.c

This file contains the function *proc_vhdl.* It also has the functions *delay_return* and *port_delay*, which are invoked by *proc_vhdl.*

vhdlm.h, externs.h and ***macrosm.h***

These are the header files that are used by the Modeler's Assistant and are included to extract the information from the Modeler's Assistant database.

decl.h

This header file defines the data structures used by BACKANN. It also has a number of *#define* statements that declare constants The data structures used are defined below.

The linked list *out_sig* is used to store the delays extracted for the various output ports of the VHDL model.

```
struct out_sig
        {
            struct out_sig *prev_out      /* points to the previous member */
            char *signal_name             /* stores the name of the output port
                                             */
            float *max_rise               /* stores the max_rise delay */
            float *max_fall               /* stores the max_fall delay */
            float *min_rise               /* stores the min_rise delay */
            float *min_fall                     /* stores the min_fall delay */
            struct out_sig *next_out      /* points to the next member */
        }
```

The linked list *sig_list* is used to store the names and types of the ports in the behavioral model.

```
struct sig_list
        {
            char *signal_name             /* stores the name of the port */
            int *type                     /* stores the type of the port */
            struct sig_list *next_sig
```

```
}
```

The linked list *dep_delay* is used to store the delay values for every source signal in a signal assignment. It is used as an element of the linked list *sigs,* which is defined later.

```
struct dep_delay
        {
                struct dep_delay *prev_dep    /* points to the previous member */
                char *sig_name                /* stores the name of the source
                                                 signal */
                float *minfall                /* stores the min_fall delay */
                float *maxfall                /* stores the maxfall delay */
                float *minrise                /* stores the min_rise delay */
                float *minfall                /* stores the min_fall delay */
                struct dep_delay *next_dep    /* points to the next member */
        }
```

The linked_list *sigs* is used to store the information about a single signal assignmnet statement and is an element of the linked list *ass_proc_signals,* which is defined later.

```
struct sigs
    {
        struct sigs *prev_sig          /* points to the previous member */
        int *line_no                   /* the line number os the signal
                                          assignment */
        char *lhs_signsl               /* stores the name of the target
                                          signal */
        int *lhs_sig_type              /* stores the type of the target signal
                                          */
        int *flags                     /* defines the flags used to identify
                                          the type of the signal assignment */
        float *generic_id              /* stores the symbolic value of the
                                          generic, if any */
        struct dep_delay *dep_delays
        struct sigs *next_sig          /* points to the next member */
    }
```

The linked list *ass_proc_signals* is used to store information about the signal assignment statements in a process.

```
struct ass_proc_signals
    {
        struct ass_proc_signals *prev_proc  /* points to the previous member */
        int *currt_proc_lineno              /* stores the starting line number of
                                               the process */
        struct sigs *assign_sigs
        struct ass_proc_signals *next_proc  /* points to the next member */
    }
```

The linked list *extract_signals* is used to store the names and numbers of the signals, as defined in the Modeler's Assistant database.

```
struct extract_signals
    {
        char *sig_name                      /* stores the name of the signal */
        int *sig_no                         /* stores the symbolic value for the
                                               signal */
        struct extract_signals *next_sig    /* points to the next member */
    }
```

The linked list *generic_list_type* is used to store the names of the generics, the numeric values associated with each and the delay calculated for each.

```
struct generic_list_type
    {
        char *generic_name                  /* stores the generic name */
        float *generic_num                  /* stores the unique numeric value
                                               for the generic */
        float *generic_value                /* delay value calculated for the
                                               generic */
        struct generic_list_type *next_gen  /* points to the next member */
    }
```

func.h

This header file defines the various function declarations.

SPI.h

This is the header file which is included to access the Software Procedural Interface.

Building BACKANN

BACKANN is built by compiling the different source code files using a Makefile as given below.

```
CC =        cc - I$(V_DLS)/dev/include -I$(V_DLS)/spi -L$(V_DLS)/spi -Bdynamic
            lSPI
backann :   annot.c sigext.o misc.o vhdl.o parse.o decl.h extract.o his.o vhdlm.h
            macrosm.h func.h externs.h
            $(CC) sigext.o misc.o vhdl.o parse.o extract.o his.o annot.c -o backann
sigext.o :  sigext.c decl.h func.h
            $(CC) -c sigext.c
misc.o :    misc.c decl.h func.h
            $(CC) -c misc.c
vhdl.o :    vhdl.c decl.h func.h
            $(CC) -c vhdl.c
parse.o :   parse.c decl.h func.h
            $(CC) -c parse.c
extract.o : extract.c his.o macrosm.h vhdlm.h externs.h
            cc -c extract.c
his.o :     his.c macrosm.h vhdlm.h extersns.h
            cc -c his.c
```

Vita

Sathyanarayanan Narayanaswamy was born in Coimbatore, India. He graduated with a Bachelor of Engineering degree in Electronics and Communication Engineering from the Coimbatore Institute of Technology, in April 1992. He went to graduate school at Virginia Polytechnic Institute and State University and received a Master of Science degree in Electrical Engineering in February 1994. He has been employed with COMSAT Labs, Maryland since January 1994.