VHDL Design Tools for IC Design

by

David C Blight

A thesis
presented to the University of Manitoba
in fulfilment of the
thesis requirement for the degree of
Master of Science
in
Electrical Engineering

Winnipeg,Canada 1990

©David C Blight 1990

National Library of Canada

Bibliothèque nationale du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-76598-4

Canada

VHDL DESIGN TOOLS FOR IC DESIGN

BY

DAVID C. BLIGHT

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

© 1990

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis. to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film, and UNIVERSITY MICROFILMS to publish an abstract of this thesis.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

As the density of Integrated Circuits continues to increase, larger and more complex systems are being implemented using VLSI technology. The increased complexity of such systems is mandating the uses of more sophisticated CAD tools in the design cycle. The traditional schematic-capture design-entry system used in IC design is unable to meet the requirements of system designers. Hardware descriptions languages like VHDL offer an alternative to schematic-capture which offer the designer more abstract modeling techniques, and higher levels of abstraction than are currently employed. This thesis discusses the design and implementation of IC design tools based on the VHDL language. Both structural compilation tools, and behavioral simulation tools are implemented.

Acknowledgements

I would like to thank my advisor, Dr. R. D. Mcleod for his advise, encouragement, and assistance throughout this thesis.

I would also like to acknowledge all the mebers of the VLSI labratory at the University of Manitoba, specifically Roland Schneider, Chris Schneider, Jay Diamond, Budi Rahardjo, for their comments and suggestions.

I would also like to acknowledge the equipment loans from the Canadian Microelectronics Corporation as well as financial support from the NSERC Strategic Grant Program.

Contents

1 Introduction 1

2 VHSIC Hardware Description Language 5

2.1 VHDL Design Environment 5

2.1.1 VHDL Analyzer 6

2.1.2 VHDL Reverse Analyzer 6

2.1.3 VHDL Intermediate Format 8

2.1.4 Design Library Manager 8

2.1.5 Circuit Elaboration 8

2.1.6 Netlist Generation 9

2.1.7 Simulation 9

2.2 VHDL Language Overview 9

2.2.1 Design Entity 11

2.2.2 Packages 13

2.2.3 Configuration 13

2.2.4 VHDL Types 13

2.2.5 VHDL Objects 15

2.2.6 Attributes 15

2.2.7 Assertions 16
2.2.8 Parallelism 16
2.3 Other Languages and Formats 17
2.3.1 EDIF 17
2.4 Cadence IC Design Platform 17

3 VHDL Analyzer 20

3.1 Syntactical Analysis 20
3.1.1 Lexical Analysis 22
3.1.2 Parsing 25
3.2 VHDL Semantic Analysis 28
3.2.1 Symbol Table 29
3.2.2 Pseudo Code Generation 30
3.2.3 Process / Memory Management 31
3.3 Intermediate Notations 32
3.3.1 Binary VIN 32
3.3.2 Textual VIN 33
3.4 VHDL Reverse Analyzer 34
3.5 Error Handling 34

4 Structural Compilation 35

4.1 Circuit Construction 37
4.1.1 Elaboration Instruction Interpretor 38
4.1.2 Circuit Representation 40
4.1.3 Hierarchical Elaboration 41

4.2 Circuit Simplification 43
4.3 Circuit Netlisters 44
4.3.1 EDIF Generation 44
4.3.2 VHDL Netlister 45

5 Design Simulation 48

5.1 Simulation Model 49
5.1.1 Simulation Processes 49
5.1.2 Signals 52
5.1.3 Simulation Event Queue 53
5.2 Simulation Algorithm 53
5.3 Hierarchical Simulation 56
5.3.1 Equivalent Nodes 57

6 Conclusions 59

6.1 Summary 59
6.2 Conclusions 60
6.3 Future Work 60

References 62

A VHDL Grammar 64

B Predefined Packages 81

B.1 STANDARD PACKAGE 81
B.2 COMPONENTS 83

C VIN Notation 89

C.1 Binary VIN 89

C.2 Textual VIN 103

D Design Example 106

D.1 VHDL Description 107

D.2 Textual VIN 110

D.3 EDIF 114

D.4 VHDL Netlist 116

D.5 Layout 119

D.6 Simulation 121

E User's Guide 122

E.1 Introduction 123

E.1.1 VHDL Language 123

E.1.2 VHDL Tools 124

E.2 Program Operation 126

E.2.1 Command Line Options 126

E.2.2 Initialization File 126

E.2.3 Files 127

E.3 Using Cadence With VHDL 128

E.3.1 Component Declarations 128

E.3.2 EDIF Netlist 129

E.3.3 Silos Extraction 129

E.4 VHDL Simulation 130

E.4.1 Component Declarations 130

E.4.2 Simulation Input 130

E.4.3 Simulation Output 130

E.5 Error Guide 131

E.6 Program Bugs 133

E.7 Unimplemented Features 133

E.8 Design Example 134

List of Tables

4.1 Elaboration Instructions 39

5.1 Simulation Instructions 50

List of Figures

1.1 IC Design Environment 3

2.1 VHDL Design Environment 7

2.2 VHDL Design Unit 10

2.3 VHDL Design Entity 11

2.4 VHDL General Format 12

2.5 VHDL example N Bit adder 19

3.1 VHDL Analyzer 21

4.1 Structural Compilation 36

4.2 Entity Representation 41

4.3 Circuit Representation 42

4.4 Component Attributes 46

4.5 EDIF vs VHDL netlists 47

5.1 Simulation Process 51

5.2 Signal Stability 55

5.3 Design Hierarchy 57

C.1 VIN Structure 90

C.2 Symbol Table 91

C.3 VIN Memory Management 93

C.4 VIN Expression 94

C.5 VIN Expression 95

C.6 VIN Types 97

C.7 VIN Types 98

C.8 VIN Types 99

C.9 VIN Component 100

C.10 VIN Attribute 101

C.11 VIN Attribute 102

D.1 ALU Schematic 107

Chapter 1

Introduction

The role of Computer Aided Design (CAD) tools is continually becoming more prominent in Integrated Circuit (IC) design. The size and complexity of VLSI systems is continually increasing due to improved fabrication techniques, as well as the development of sophisticated layout tools. Until recently the focus of CAD tools was on the automation of the mask layout generation, and physical design problems. Commercial layout design systems are now currently available with automated place and route tools which can synthesize mask layouts from a simple circuit description very efficiently. The focus of these tools is on the layout generation, and seldom are modeling techniques incorporated to facilitate the design of large systems. In system level design methodologies, circuit description must exploit more abstract modeling techniques, and higher levels of abstraction. CAD tools exploiting these techniques will enable system designers to implement designs into silicon in a relatively short amount of time. The major difficultly in the design process, is the consistent management of large system descriptions. Hardware Description Languages (HDL) such as VHDL (Very High Speed Integrated Circuits HDL) are intended to be high level languages capable of efficiently modeling large systems. These languages will convey design descriptions from the designer to the layout synthesis tools. This thesis discusses the implementation of a VHDL compiler for use in an integrated circuit design environment.

VHDL is a hardware description language initially developed by the United States Department of Defence (DoD) as part of its Very High Speed Integrated Circuits (VHSIC) program. The large amount of electronic system designs contracted out, to a multitude of vendors, left the DoD faced with problems of managing incompatable design specifications,

descriptions and documentation. The long life cycles of military systems compounded the problems, and in 1983 the DoD initiated the development a standard hardware description language (VHDL). By 1988 VHDL was modified and accepted as a standard by the IEEE [1]. The primary motivation of VHDL was to develop a hardware description language which meets the following criteria :

1. The input format should be a standard text file, instead of schematic capture or graphics based entry systems. This ensures portability of designs and tools, with independence from any one computer system or software platform. This allows VHDL descriptions to be entered using any text editor of the users choice.

2. Provide good documentation. The long life cycles, and wide deployment of military electronic systems dictate that stringent documentation exist for all designs. The documentation of a design should occur during the design stage rather than afterwards.

3. The language must be able to model/describe the wide range of digital IC components used in electronic systems, with emphasis on parameterized generic components (eg. a n-bit adder). By using parameterized generic descriptions, development time and cost can be reduced by reusing previously designed units.

4. Technology Independence. The description of a circuit must not be specifically bound to one technology. This implies that the lowest level of modeling can be the gate level.

5. Both behavioral and structural descriptions of circuits must be allowed. Description models must range from digital system level models down to gate level models.

6. The language should be capable of representing the wide range information that is required by Design Automation Tools. The rapid developement of new tools makes the prediction of future requirements impossible. It is important that the language have the flexibility to model unforseen information.

7. The language must support different design methodologies (eg Top - Down design and Bottom - Up design methodologies)

This thesis discusses the design and implementation of IC design tools based on the VHDL language. Figure 1.1 shows the IC design environment which incorporates VHDL

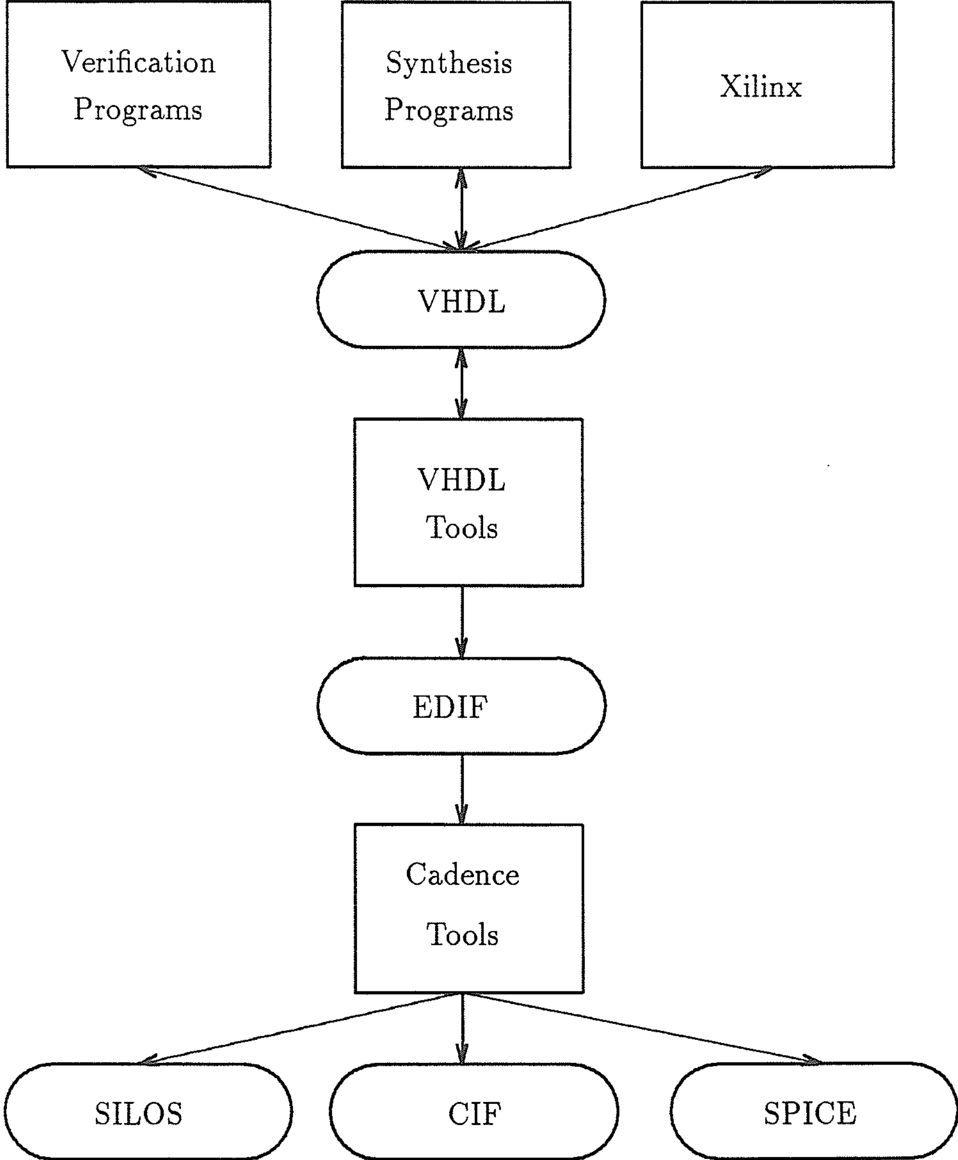


Figure 1.1: IC Design Environment

tools. VHDL may be used as the primary design entry format, or used as as design information interchange form when produced by high-level synthesis programs utilizing abstract VHDL behavioral descriptions or other alternative languages. The tools developed in this thesis are used for the synthesis of structural descriptions, and the simulation of behavioral descriptions. The actual mask layout generation is performed by the Cadence Tools [13] which reads the design information in the form of an EDIF netlist and creates a CIF (Caltech Intermediate Format) [7] layout description from which the actual mask may be fabricated.

The predominate design methodology for large digital IC's is the standard cell approach. A small set of basic circuit components referred to as gates have layouts generated. More complex circuits can be constructed by interconnecting these gates. Other prevalent techniques are gate array, PLA's, and PAL's. Primarily we are concerned with the implementation of IC design tools based on the standard cell approach. Specifically we are interested in a structural compiler to transform VHDL design descriptions into structural netlist, and the simulation of design hierarchies with behavioral models.

While this thesis is concerned with the use of VHDL in an IC design environment, it is important to realize that design entry mediums other than HDL's exists. Most common is the schematic capture systems, where designers use an interactive system to graphically represent design connectivity. The simplicity of such systems is hindered by the lack of high-level abstractions needed to model system-level design structures. The use of HDL's and schematic capture need not be mutually exclusive, as VHDL can be generated from schematic capture, although the converse is not as simple. The use of VHDL is not meant to be imposed on a designer, only to be available as an option.

Chapter two briefly introduces the VHDL language and environment, emphasising the aspects which are different from conventional programming languages. The tools with which VHDL must interface are also introduced. Chapter three discusses the implementation of the syntactical and semantical analysis progran for the VHDL language. Chapter four focuses on the implementation of a structure compiler which translates a VHDL description into a structural netlist, the information exchange format to layout generation programs. Chapter five covers the simulations of VHDL design hierarchies. The final chapter summarizes the results, and discusses future work.

Chapter 2

VHSIC Hardware Description Language

Before describing the implementation of VHDL design tools in an IC design environment, it is important to discuss both the language, and the environment in which they will be used. This chapter will give a brief overview of the VHDL language, highlighting those features which are important in the IC design process, and discuss the VHDL design environment which includes a description of the various tools and services that the VHDL design platform must implement. A brief overview of design tools and formats with which the VHDL system must interface are also discussed.

VHDL is both syntactically and semantically similar to procedural programming languages, especially the Ada programming language which was also standardized by the DoD. The final section of this chapter will compare and contrast VHDL to conventional procedural programming languages.

2.1 VHDL Design Environment

The design environment illustrated in figure 2.1 shows the relationship between the various modules of a VHDL design system. At the top of figure 2.1 is the VHDL design description. This may have been entered by the designer directly, or alternatively may have been produced by some other design tool. In either case this VHDL code will be a standard operating system text file. This description is parsed by the VHDL analyzer which produces

an equivalent internal notation (VIN). The VHDL library manager performs the storage and retrieval of design information from the proper design libraries. The current VHDL design entity is stored in the working library, and any external references to library entities are resolved by retrieving the appropriate VIN file. The complete design hierarchy is next elaborated into the desired circuit representation: architectural for layout synthesis, or behavioral for simulation. The structural representation may be extracted by the netlist generators to produce netlists in the desired format.

2.1.1 VHDL Analyzer

The VHDL analyzer closely resembles a programming language compiler (consisting of lexical analysis, syntactical parsing, and error detection/reporting routines). This program is responsible for converting the VHDL circuit description into an equivalent internal representation which is used by other design tools. During the analysis, the VHDL code is checked for both syntactical errors and simple static semantical errors. Other errors can not be detected until further processing. The output the the VHDL analyzer is an intermediate notation which is logically equivalent to the input. The semantic information from the VHDL description is retained, but syntactical information such as comments is lost.

2.1.2 VHDL Reverse Analyzer

The VHDL reverse analyzer performs the opposite function of the VHDL analyzer, specifically it converts the intermediate format of VHDL back into a VHDL text file. The text produced by reverse analysis is logically and functionally equivalent to the original VHDL code, but may be syntactically different. No comments will be included in the VHDL text produced by the reverse analyzer.

The primary purpose of this program is to guarantee portability of designs. VHDL systems using different intermediate formats (currently there is no accepted standard) may always exchange designs by using the reverse analyzer. Any design in the design library for which the original VHDL code is not available can have its library representation converted into equivalent VHDL text. Additional tools which modify VHDL intermediate representation can always have their effects translated into VHDL text, hence using VHDL as an design exchange language instead of a design entry language.

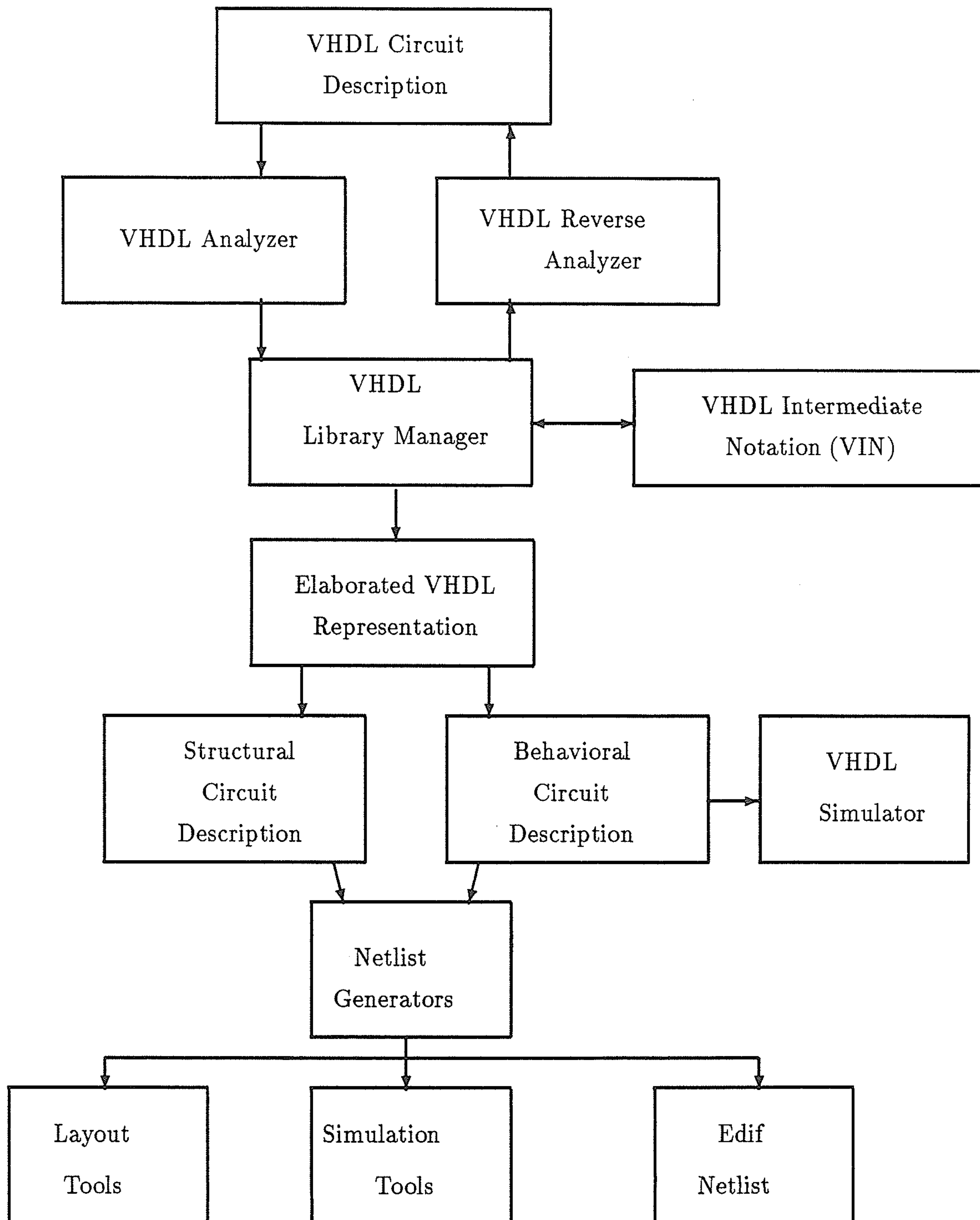


Figure 2.1: VHDL Design Environment

2.1.3 VHDL Intermediate Format

The VHDL language itself is not a convenient notation for programs to manipulate. For this reason, an intermediate notation is created which contains the same information as the original description, yet can be efficiently manipulated by the VHDL tools. Two internal representations are used. The first is a binary form, and the second is a textual form. Both express the same information, but the binary version is used internally in the programs, whereas the textual version is used as a storage format. The textual VIN is a language itself with the equivalent expressive power of the original VHDL code, but is designed to be both parsed more efficiently, and semantically interpreted more easily.

Currently there is no standard for this representation, although the IEEE is in the process of establishing one. An intermediate format developed by Intermetrics called IVAN (Intermediate VHDL Annotated Notation) was developed for VHDL version 7.2. The IEEE standard will probably adopt a format similar to this . This thesis uses its own intermediate format (both binary and textual representations instead of a text format used by IVAN). When a standard format is accepted, the tools can be modified to utilize that format, or a conversion program between the different formats could be written.

2.1.4 Design Library Manager

The design library manager (DLM) is responsible for the maintenance, storage, and retrieval of design information. The management of large system designs involves the managing of libraries of common components and packages which may be shared among members of a design team. The tools discussed in this thesis uses the UNIX operating system file system as the library manager. The large selection of utility programs, the high reliability, hierarchial structure, and proven performance of the file system eliminates the need to create a new database format. All designs are stored in a textual format in standard operating system files allowing designers a familar and modifiable database system.

2.1.5 Circuit Elaboration

The VHDL elaborator is the program which transforms the intermediate notation (a design description) into a logical representation of the design. Specifically, the VIN is a notational

representation which contains a sequence of instructions to be executed which will create circuit objects described in the original VHDL code. Elaboration is the process of executing these statements. After elaboration, an architectural description will be represented by a set of connected component instantiations and signals. A behavioral representation will consist of a set of signals and a set of processes, with the process being sensitive to transitions in the signal values. An elaborated design representation can be netlisted to be used as input to other programs, or simulated if it represents a behavioral circuit model.

2.1.6 Netlist Generation

One of the most important design considerations of any tools is its interface to other tools. The netlist generation programs are responsible for the translation of elaborated circuit representation into formats compatible with other tools. The large number of proprietary netlist formats exclude the possibility of interfacing to all such formats. Instead a most prevalent standardized format EDIF (Electronic Design Interchange Format) [15] was selected as the primary netlist format. VHDL is also used as a netlist language, but in the simplified form which employs no high-level abstractions.

2.1.7 Simulation

The structural representation of a design is used to create a physical design layout. The behavioral representation of design on the otherhand is used to simulate the dynamical behavior of circuit operation. It is important for the IC design to be able to incorporate simulation directly into the design cycle, without having to convert design information into external simulator formats. VHDL contains well defined behavioral models of typical digital circuit operation. The behavioral models are particularly well suited to discrete event simulation. The VHDL simulator uses the elaborated behavioral representation and simulates the operation by concurrently executing the processes and statements in a design.

2.2 VHDL Language Overview

The VHDL language is a fairly complex language, syntactically similar to the procedural programming language Ada. This section will discuss the general format of the language,

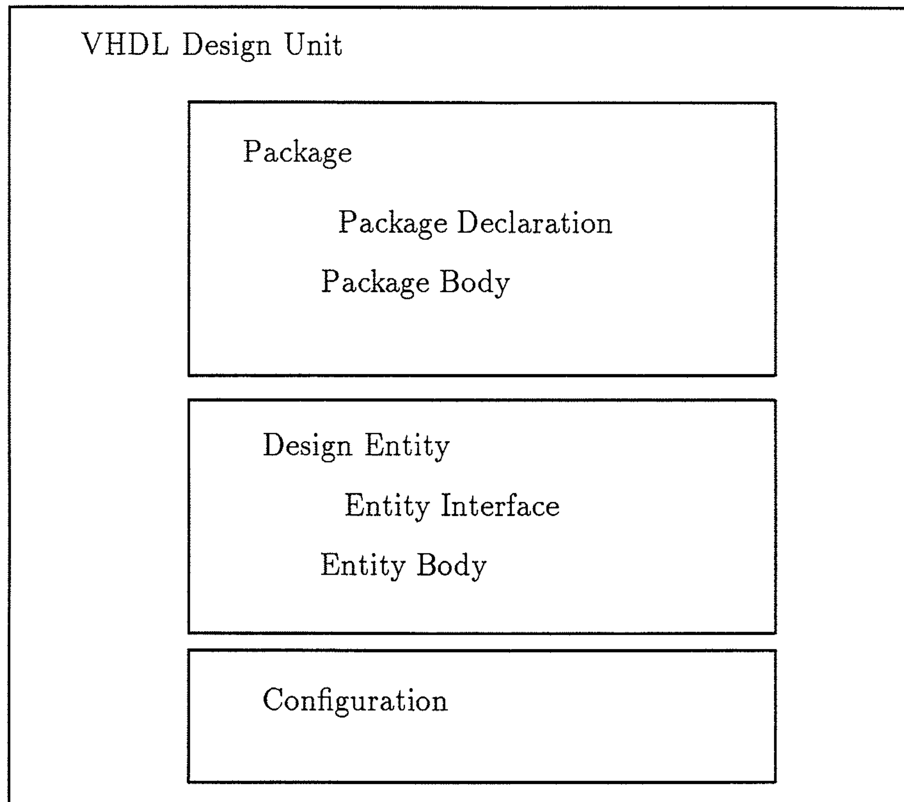


Figure 2.2: VHDL Design Unit

concentrating on the aspects which are unique to the language or directly related to implementation details discusses later in this thesis. The basic constructs which are common to most procedural programming languages are not discussed.

A typical VHDL design description consists of a series of design units. Each design unit describes a unique module representing either some portion of the design, or a group of common declarations. The basic format of the design unit is shown in figure 2.2 and consists of the following :

- Package : A collection of objects which are shared across different design units.
- Design Entity : The basic hardware abstraction of VHDL. The entity interface models the external connections and characteristics of a design, and the entity body models the internal structure / behavior of the entity.

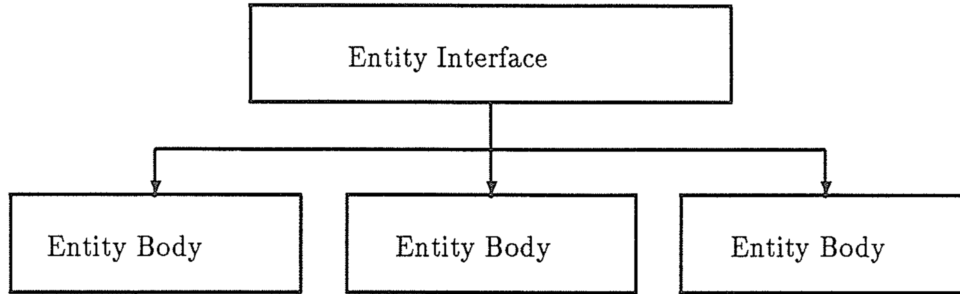


Figure 2.3: VHDL Design Entity

- Configuration : describes the selection of entity bodies and entity interfaces that are used in the construction of a specific design hierarchy.

2.2.1 Design Entity

The most basic hardware abstraction in VHDL is the design entity, which is shown in figure 2.3. The design entity is capable of describing any component in VHDL. Each design entity consists of two separate parts : an entity interface and one or more entity bodies.

The entity interface contains all the information describing the externally visible characteristics and signals of a design entity. All information contained in an entity interface is common to all entity bodies that are part of the design entity. Each entity interface contains port and generic declarations. A port is a communication channel through which electrical signals transfer between internal and external circuitry. A generic is a channel through which variable parameter information is transferred. Signals represent electrical states physically present in circuits, whereas variables represent values used to parameterize both the behavioral and structural nature of a design entity. Data contained in variables (including constants, generics, and variables) is used to control the elaboration of a design entity, the process of constructing an internal representation of the described entity. Data contained in signals (including both ports and signals) are manipulated during the simulation of behavioral descriptions.

Each entity body describes an alternative implementation of the design entity. The entity body consists of two parts. The first part defines all local variables, signals, types, and functions. The second part contains a set of concurrently executed statements. Multiple

```
ENTITY ARBITRARY_CIRCUIT IS
        GENERIC (Generic_variable : INTEGER);
        PORT(PORT1,PORT2 : IN BIT);
Begin
        – assertions
        – procedure calls
        – process statements
END ARBITRARY_CIRCUIT;

ARCHITECTURE Version1 OF ARBITRARY_CIRCUIT IS
        – declaration, specifications, concurrent statements
END Version1;

ARCHITECTURE Version2 OF ARBITRARY_CIRCUIT IS
        .
        .
        .
END Version2;
```

Figure 2.4: VHDL General Format

entity bodies are allowed for each entity interface to represent different implementations, abstractions, or versions of the same basic design. For example, for an ALU, a designer may wish to implement two bodies for the design. One version may describe a time efficient implementation containing look ahead adders, or a parallel multiplier, whereas the other version might describe one which is area efficient but slower. By using a common entity interface, the designer may easily chose the implementations of the design entities he requires, without modification of the descriptions. Similarily both behavioral and architectural descriptions can be specified for the same design, and older versions of the same entity may be stored in the different entity bodies. The selection of which entity body is to be used is specified in the configuration portion of the VHDL design.

2.2.2 Packages

Communication between design entities neccessitates a mechanism to share common abstractions between design units. Declarations, specifications, and objects may be grouped together into a package and shared among design units. Without packages, design entities would be unable to incorporate abstract types into interface ports and generics, since these types must be declared outside of the entity. A predefined package called standard is assumed to be available to all design entities, even without the explicit declaration of it's use.

The package declaration is responsible for defining an interface to a package. The package body defines the implementation of package subprograms, and assigns values to deferred constant declarations in the the package interface. Not all packages are required to have bodies. For example the predefined package standard does not have an associated body. Package interfaces also be used to interface foreign language subprograms into VHDL code.

2.2.3 Configuration

The purpose of a configuration declaration is to bind component instances in entity bodies to other design entities. This is particularly important when more than one entity body exists for a particular design entity. A configuration declaration will select which specific body is to be used in each instantiation. Configuration declarations are not required during the definitions of design entities and packages, but only when a complete design is being specified.

2.2.4 VHDL Types

A type is an abstraction for a value, which includes both a constraint limiting the range of values it may represent, and a collection of operations which may be performed on the value. Operator overloading is permitted in VHDL which allows the predefined operators to be redefined so that they may be also used with user specified types. The following types are supported :

Scalar Types a type which represents a single value within a specified range (constraint). The four classes of scalar types are :

1. Integer type is used to represent an integer value.
2. Floating Point type is used to represent a real number.
3. Enumeration type is used represent an enumerated list of values.
4. Physical type is used to represent a physical quantity, a value with an associated unit.

Composite Types a type which represent an abstract collection of values. The two classes of composite types are :

1. Array type is used to represent a multidimensional collection of uniform type data.
2. Record type is used to represent a collection of varying type data.

File Type a type used to represent a operating system file. This is required to allow input/output operations.

Access Type a type to represent a memory address of a value.

The array type is allowed to have an undefined constraint associated with any or all of it index ranges. This allows generic array structures to be defined. An example of this is the predefined standard package type STRING [1] (see appendix). String is declared as an array of characters (another prefined type) with no range specifically specified for the index. Character arrays of range two, three, or more are all of type string, sharing the same properties and attributes. The standard package is a set prefined types which all VHDL programs may use.

The physical type is the only type which is not typically found in programming languages. This type is used to represent physical measures such as time, or voltages. Since timing specifications are an important part of hardware description models, this a necesary addition to the conventional types found in procedural programming languages. An example of a physical type is the standard package predefine type TIME. The value of a time variable must have both a value and a unit.

VHDL also supports subtypes which are composed of a type with an additional constraint. Subtypes are used to associate an additional constraint with already defined types. This allows the properties of a type to be inherited by other types. VHDL is a strongly typed language. The types of all expressions and objects can be determined during compilation.

2.2.5 VHDL Objects

Objects are the VHDL entities which contain values. The three classes of objects available in VHDL are signals, variables and constants. Variables and constants are VHDL objects which contain a single current value of specified type. The value contained in a constant is assigned during the declaration of the constant and may not be altered during program execution or simulation. The value contained in a variable may be manipulated and used to represent any information that changes. Both variables and constants are found in almost every programming language.

A signal is similar to a variable except in addition to containing the current value of the object it represents, it contains the object's past and future values. The function of a signal is to model electrical states in circuits. Signals are used to represent signals physically present in the circuit being described. Typically signals will be associated with buses, wires, datapaths. Only the present and future values may be set, and only the past and present values may be observed.

2.2.6 Attributes

With the rapid development of new DAT tools, covering a broad spectrum of purposes, it is impossible to design a HDL which meets the needs of all system designers. With this in mind, VHDL incorporates attributes in the language. An attribute allows values, functions, types, ranges, signals, or constants to be associated with any named entity. This feature gives designers the flexibility to incorporate design information directly into the VHDL description, even when there are no explicit models available to represent the information. Attributes are a means of syntactically expressing arbitrary information in the language, for which the language itself places no semantic meaning. It is up the individual DAT tools to interprete the information correctly and consistently. Likewise, the VHDL language can offer no syntactic error detection reporting of the information stored in attributes.

There are many situation where attributes will be used. In this thesis, attributes are used to store technology dependent information required by the Place and Route tools to create an IC layout. Information pertaining to cells names, pins names on cells(which are case sensitive in the Cadence Edge system, but VHDL supports only uppercase), and routing priority are incorporated in an attribute of components and signals called "CDS_INFO". In a similar manner other information including test information, even layout information could be expressed in attributes.

2.2.7 Assertions

Assertion are used to monitor specified values and signals during the execution and simulation of design entities. Each assertion has an associated condition which is continually monitored to ensure that it is satisfied. Upon detecting a false assertion condition, a report message can be generated, and an error level indication of predefined type SEVERITY_LEVEL is set.

Assertions are usefull in ensuring that circuit behavior is within circuit reasonable limits, as well as for verifing that constraints such as timing are met during circuit operation. For example a typical assertion declaration might specify that some clock signal must remain high for a least a specified period of time. This the component is used in a system with too high a clock frequency, the simulation of the design will report an error. This can be usefull in ensuring the proper use of generic components.

2.2.8 Parallelism

The operation of electronic hardware is an inherently parallel process. All of the components operate simultaneously. It is imperative that any HDL must model the concurrently operation of both structural components and behavioral models.

Parallel components are instantiated in a VHDL structural description through component instantiation statements. Each statement instantiates a component and connects its port to the specified signals. The ordering and execution sequence of these statements does not effect the operation of the circuit. For behavioral descriptions parallelism is

acheived through concurrent signal assignment statements, resolution functions, processes, and guarded assignemnt statements. These enable the modelling of parallel circuit behavior.

2.3 Other Languages and Formats

In this section a brief comparison is made between the features of VHDL and existing programming and hardware description languages. The purpose of this discussion to further illustrate some of the features of VHDL, and place emphasis on aspects which are different from other languages. These factors will be important in the discussion of the implementation of the VHDL analyzer.

2.3.1 EDIF

EDIF (Electronic Design Interchange Format) is another recently developed language for conveying circuit information. The main focus of EDIF is to describe circuit netlists, layout geometries, and more technology related data for the purpose of design communication using simple abstractions. EDIF's primary features include an easily generated and parsed syntax, flexibility, and an extensive range of allowed representations. The levels of abstractions that can be used in netlist descriptions are not as complete, or elegant as VHDL. VHDL on the otherhand is concerned with more abstract design models and is meant to be a design language, specifically it is easier to read and follow the design description, however it is more difficult to parse. While EDIF and VHDL do overlap slightly in their focuses, together they are complementary. For this reason one of the primary netlist formats used in the thesis is EDIF.

2.4 Cadence IC Design Platform

The Cadence Edge system [13] is a commercially available IC design platform which is used to generate the final IC layouts. The features of these tools that are used in conjunction with the VHDL discussed in this thesis are the EDIF input translators, place and route tools, the schematic capture, and the simulation interfaces. Cadence currently supports EDIF version 1 1 0 and EDIF version 2 0 0. The netlist view of EDIF is used as input, and

the place and route tools are used to produce the IC layouts from this netlist. The Cadence platform is primarily designed to use schematic capture as the design entry medium. The platform supports a flexible simulation interface which allows design information to be easily generated for different simulation tools. The primary simulator used are the SILOS II [12] switch level simulator, and the HSPICE [14] analog simulator. Although Cadence is capable of flattening netlists externally produced, the netlist constructs of EDIF 2 0 0 which Cadence utilizes are not as advanced as VHDL's requiring the production of flattened netlists.

```
PACKAGE BUS_PACKAGE IS
      TYPE BUSS IS ARRAY(0..15) OF BIT;
END BUS_PACKAGE;

USE BUS_PACKAGE;
ENTITY N_BIT_ADDER IS
      GENERIC(N : NATURAL);
      PORT(A,B : IN BUSS,
               CIN : IN BIT,
               SUM : OUT BUSS,
               COUT : OUT BIT);
END N_BIT_ADDER;

USE BUS_PACKAGE;
ARCHITECTURE IMPL1 OF N_BIT_ADDER IS
      SIGNAL  Carry : BIT_VECTOR( 0 TO N - 1);
      component fadd port(
               Ain,Bin,Cin : in Bit;
               Sum,Cout : out Bit );
BEGIN
      FOR I = 1 TO N GENERATE
               if i = 0 generate
                        fadd port map(A(0),B(0), CIN,SUM(0),Carry(0));
               end generate;
               if (i > 0) and (i < N) generate
                        fadd port map(A(i),B(i),Carry(i-1),SUM(i),Carry(i));
               end generate;
               if i = N generate
                        fadd port map(A(N),B(N),Carry(N-1),SUM(N),COUT);
               end generate;
      END GENERATE ;
END IMPL1;
```

Figure 2.5: VHDL example N Bit adder

Chapter 3

VHDL Analyzer

The VHDL analyzer is the first stage in the processing of a VHDL circuit description. As mentioned in the previous chapter, the purpose of the VHDL analyzer is to parse the input VHDL code, and translate the circuit description into both a convenient design notation and a sequence of elaboration instructions. The function and implementation of a VHDL analyzer closely parallels that of procedural programming language compiler. For this reason the terms VHDL analyzer and VHDL compiler will be used interchangably.

In this chapter, the implementation of the different aspects of the VHDL compiler shown in figure 3.1 will be discussed in detail. The compiler is implemented as a single pass compiler, implying that the three steps of the compilation process (lexical, syntactical and semantical analysis) are performed in parallel, with each process only executing on demand from the next level. An explanation of both the binary VIN and textual VIN will be given, as well as a brief discussion of the VHDL reverse analyzer.

3.1 Syntactical Analysis

The first step in the compilation of a VHDL description is the syntactical analysis of the input description. In general the syntactical analysis of languages is a complex task, and in order to simplify this task, the process is divided into two steps lexical analysis and parsing. The first step in the compilation process is lexical analysis (also referred to as scanning), which converts an input stream of characters into a sequence of tokens containing the

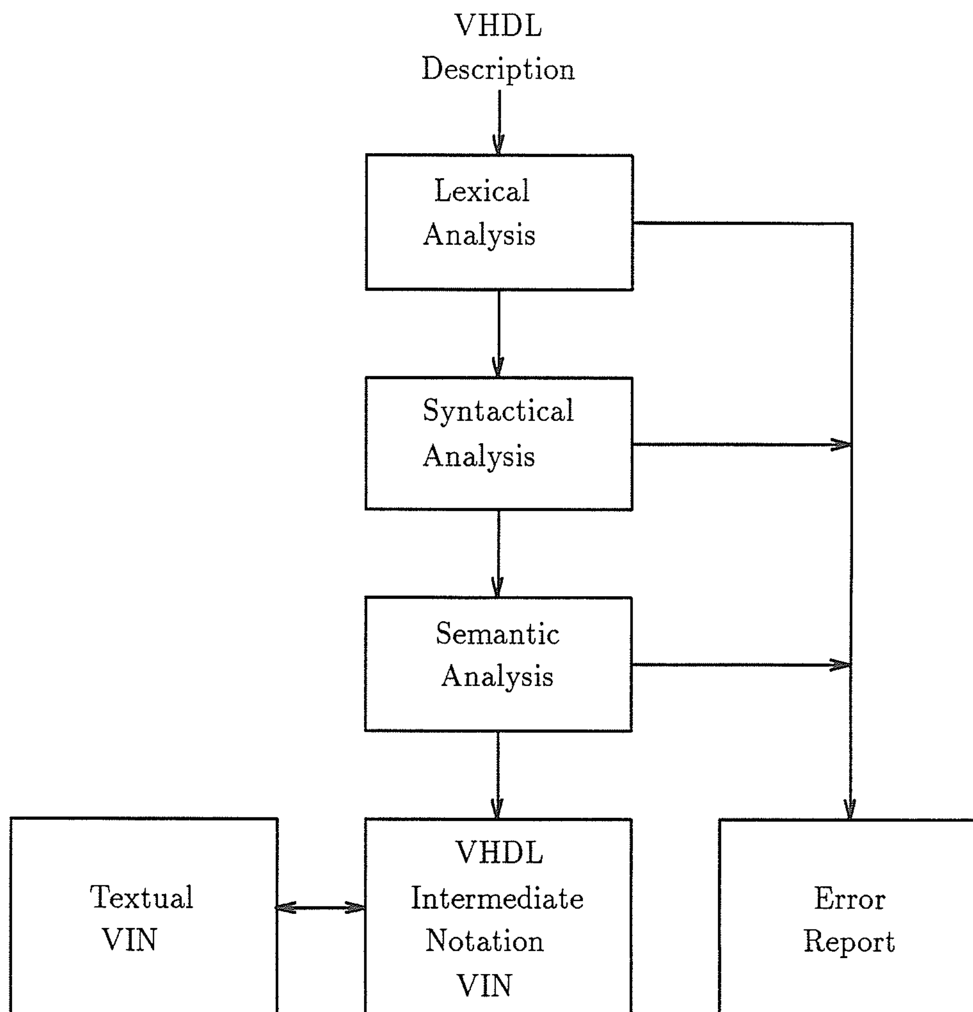


Figure 3.1: VHDL Analyzer

basic lexical elements of the language. These tokens represent the terminals (leaf nodes) in the syntax tree used by the parser. The syntax parser matches an input sequence of tokens, to the syntax tree representation of the language. The distinction between lexical analysis and parsing can also viewed in terms of the type of automaton which performs the analysis, and the languages they accept. Lexical analysis is implemented by a finite-state machine, which is capable of identifying regular expressions. A language like VHDL which can be represented as a context free grammar, requires a more complex push-down stack automaton. While the lexical analysis could in theory also be done by the parser, the simpler implementation and faster operation of finite automatons makes syntactical analysis more efficient with two distinct phases of operation.

After the syntax of the input description has been determined, the semantic analysis of the input description may begin. This section discusses the lexical analysis, and parsing of the VHDL language.

3.1.1 Lexical Analysis

The lexical analyzer implemented for the VHDL analyzer is a very simple finite state machine. The language specification for VHDL defines eight classes of lexical elements which must be identified. Using a double buffering technique [17], the lexical analyzer identifies the next lexical element in the input stream, and produces a corresponding token. This token which is placed in a token queue which acts as an interface between the parser and the lexical analyzer.

Since the VHDL analyzer is implemented as a single-pass compiler, tokens are only produced when the token queue becomes empty. Not all lexical elements produce a token (comments and separators are ignored), requiring that multiple lexical elements may have to be identified in order to produce a single token.

Lexical Elements

The following are the different types of lexical elements identified by the VHDL lexical analyzer. A complete description of lexical elements can be found in the IEEE standard.

1. **Identifier** is a sequences of alphanumeric characters which can be used to represent different named entities in VHDL. Identifiers include reserved words which are predefined lexical elements that may not be assigned any further meaning.

$$identifier ::= letter \; \{ \; [\; underline \;] \; letter_or_digit \; \}$$

2. **Character literal** is a lexical element corresponds to any single character contained in single quotes. Character literals are most commonly usually used as enumeration literals.

$$character ::= \; ' \, character \, '$$

3. **Separator** is an input character which separates adjacent lexical elements elements but which has no semantic significance. Separators can be used to increase readable of input code, or to explicitly separate lexical elements, that without separation, would be interpreted as a single lexical element. Separators are detected by the lexical analyzer but no token is produced to indicate there presence.

$$separator ::= LF \mid FF \mid CR \mid ' \;\; ' \mid VT$$

4. **Delimiters** are used to represent operators and syntactical separators. VHDL uses both single and double character delimiters which are shown below

$$\& \; ' \; (\;) \; \star \; + \; , \; - \; . \; / \; : \; ; \; < \; = \; > \; |$$
$$=> \quad \star\star \quad := \quad /= \quad >= \quad <= \quad <>$$

5. **Strings** are simple a variable length sequence of characters. Strings are typically used to represent textual information.

$$string ::= \; " \; \{ \; character \; \} \; "$$

6. **Bit Strings.** In describing digital circuits it is often advantageous to specify information in binary, octal, or hexadecimal notation. Bit strings provide one such method of representing this data.

$$bit_string ::= B \mid O \mid X \; "\{extended_digit \mid _\}"$$

7. **Comments.** The input VHDL program may contain comments which are used to express information which does not adhere to VHDL syntax nor have any semantic meaning to a VHDL compiler. Comments are identified by the lexical analyzer but no output tokens are produced.

$$comment ::= -- \{ character \} CR$$

8. **Abstract literals** are the lexical elements that correspond to numeric values in the input code. Numerical values can be either integer or floating point values, and may be expressed as either a decimal literal or a based literal. Decimal literals are numbers expressed in traditional decimal notation, whereas based literal are numbers expressed relative to a specific base. The allowed bases range from binary (base 2) to hexadecimal (base 16). The lexical analyzer produces one of two different token types for each abstract literal depending whether the abstract literal represents and integer value or decimal value.

$$\begin{aligned} abstract_literal &::= decimal_literal \mid based_literal \\ based_literal &::= base\#based_integer[.based_integer]\#[exponent] \end{aligned}$$

Lexical Tokens

The lexical analyzer recognizes the different lexical elements discussed in the previous section and produces the lexical tokens corresponding to these elements. In addition, an error token is produced whenever an error is detected during lexical analysis and compilation should not continue. This token is passed to the syntax analyzer, thereby passing error recovery responsibility to it. The following is a list of the token types produced by the lexical analyzer:

- **Reserved Word Token** produced when a lexical element matches a VHDL reserved word.
- **Identifier Token** corresponds to an identifier literal.
- **Character Token** corresponds to a character literal.

- **Delimiter Token** corresponds to a delimiter literal.
- **Floating Point Token** produced by a real valued based literal.
- **Integer Token** produced by an integer valued based literal.
- **String Token** corresponds to a string literal.
- **Bit String Token** corresponds to a bit_string literal.
- **Error Token** produced when an error is found in an lexical element.

3.1.2 Parsing

The IEEE Standard VHDL Reference Manual [1] gives a syntax specification of the VHDL language in a simple variant of the Backus-Naur form. The parser implemented in this thesis must be able to successfully parse this specification. While a compiler generation program such as Yacc [8] could have been used, it was decided to write a simple parser to ensure maximum portability of the compiler. This section discusses the implementation of such a parser.

There are two methodologies that can be used in the implementation of a parser: top-down parsing, and bottom-up parsing. In an attempt to keep the parsing routines simple, the top-down methodology was chosen.

Parsing Algorithm

Before discussing the parsing algorithm used, it is important to define a few terms which are used in the discussion. More complete and rigorous definitions may be found in compiler construction texts [16] [17].

- An **Alphabet** V is a set of symbols which can be used to construct a language. V_T is an alphabet with a finite number of symbols. The alphabet for the compiler is the set of tokens produced by the lexical analyzer.
- A **Language** L is a subset of V_T^*

- A **Grammar** G is a specification of a language containing the following : a set of terminals V_T ; a set of nonterminals V_N; a set of production rules Θ; and a starting nonterminal S. The nonterminals and production rules of VHDL are listed in appendix E.

- A **Production Rule** is a relation of the following form

$$(V_T \cup V_N)^\star V_N (V_T \cup V_N)^\star \to (V_T \cup V_N)^\star \tag{3.1}$$

which is essentially a mapping of sequences of terminals and nonterminals into another sequence. These mapping maybe recursive, by allowing the first sequence to appear in the second.

- A **Context Free Grammar** is a grammar where all of the production rules take the following form

$$V_N \to (V_T \cup V_N)^\star \tag{3.2}$$

By placing this restriction on the grammar, the process of parsing the language can be made simpler.

- An ϵ - **Rule** is a production rule of the form

$$\begin{aligned} V_N &\to \epsilon \\ or \quad V_N &\to \end{aligned} \tag{3.3}$$

The epsilon represents a null sequence used in the substitution of the nonterminal. This is often used to present optional constructs.

- A **First Terminals** of a nonterminal $FIRST(V_N)$ is equal to the nonempty set of terminals which satisfy

$$FIRST(V_N) = \{w | V_N \overset{*}{\Rightarrow} w \ldots and \quad w \in V_T\} \tag{3.4}$$

- A **LL(1) grammar** is a grammar in which all the production rules associated with a nonterminal have non intersecting FIRST sets.

Parsing is performed by repeatedly performing production rule substitution on the start symbol, until only terminals remain. The VHDL language has a context-free grammar representation which is used in the syntax parser. The first step in the parsing process is the generation of the syntax tree for the VHDL language. A set of production rules is read into the program and the syntax tree is generated. The syntax tree contains the following information for each nonterminal:

- a set of terminals representing the valid first terminals which may be produced by product rule substitutions.
- a set of production rules of the form specified in 3.2. Each production also has a set of terminals representing valid first terminals produced by production rule substitution.
- an ϵ - rule flag indicating the presence of an ϵ - rule.
- an LL(1) flag indicating whether the nonterminal obeys the LL(1) grammar restriction. The majority of the language obeys this restriction but it was violated in a few select places to simplify parsing.

The input language is parsed by selecting the top nonterminal in the syntax tree, and recursively applying the parse_nonterminal and apply_production_rules algorithms shown below.

parse_nonterminal algorithm

The parse_nonterminal algorithm selects which production rule for a nonterminal is to be applied. The next lexical token on the token stack is examined. If the LL(1) flag is set, the first production rule whose first set intersects the current token is selected. If the LL(1) flag is not set, each production whose first set matches the current token is recursively tested for matching the input stream. If no production rule is found which matches, and the ϵ rule flag is not set, an error is generated.

apply_production_rule algorithm

The apply_production_rule algorithm substitutes the specified production rule. For each V_N in the production rule, a START_VN marker is placed in the syntax queue. Likewise for each V_T another START_VT marker is placed in the syntax queue.

The syntax queue contains a list of production rule substitutions which have been performed during the syntactical analysis. The semantic analysis routines determine the selected substitutions by poping the next element from the stack. If the next element is a START_VN marker, the parse_nonterminal procedure is repeatedly called until a START_VT marker appears. The next lexical token is then popped from the lexical token stack and placed in the syntax queue.

The output of the parser is a sequence of syntax tokens, Each of these tokens has one of the following functions :

- Begin Syntax token: indicates the start of a production rule substitution.
- Lexical Element token: contains a pointer to a lexical element. The pointer will point to a symbol table entry for an identifier, or a memory location for data values (integer, reals, etc).
- Epsilon token: indicates the application of an epsilon production rule.
- End Syntax token: indicates the end of a production rule substitution.

The sequence of syntax tokens is stored in a queue which acts as an interface to the semantic analyzer routines. Since the parser generates syntactical tokens on demand from the semantic analysis routines, the parser has to be carefull about the use of backtracking during parsing.

3.2 VHDL Semantic Analysis

Whereas the previous section dealt with the syntactic analyzer of an input description, semantic analysis is concerned with determining the meaning of the description. The primary duties of the semantic analyzer routines can be summarized as follows :

- maintain a symbol table of all currently defined identifiers and assigned meanings.
- create internal models of language constructs. Models for declared objects such as types, components, and other parameterized objects must be initialized. These will be elaborated during circuit construction.
- maintain an ordered list of all statements parsed. A proper ordering of VHDL statements is essential to the generation VHDL in the reverse analyzer, as well the creation of textual VIN for design library storage.
- maintain a list of currently visible entities and objects. The object based structure of VHDL gives the designer the ability to specify packages of constructs, and to control visibility of objects in the VHDL description.
- create a sequence of executable statements which can be executed during the elaboration of a design.
- initialize process and memory management information required to elaborate a design.

3.2.1 Symbol Table

The symbol table used by the VHDL compiler maintains a hash table of parsed lexical elements. Each entry in the symbol table points to a list of defined meanings which has been assigned to the identifier. It is the responsibility of the visibility checker to ensure the proper semantic meaning is determined for each reference. If an entry has multiple visible entries at one time, the proper information must be determined from the context of the identifier. For example, predefined operators such as addition have many different functional meanings corresponding to the types of operands used.

Also associated with each entry in the symbol table is a list of specified attributes for an object. Each named object in VHDL may have attributes specified for it, and this information is maintained in the symbol library.

The symbol table also contains all the predefined identifiers and reserved words. Upon invocation of the VHDL analyzer, the initial symbol table is loaded from a stored file.

3.2.2 Pseudo Code Generation

A VHDL design description is not of much use unless it can be converted into a circuit representation. The VHDL analyzer does not create this circuit representation but does generate a series of executable instructions which will be executed during the elaboration stage of processing (Circuit Construction). These executable statements are called pseudo-code. The defined pseudo-code operations will be discussed in the next chapter.

As each statement is interpreted by the VHDL analyzer, a sequence of statements is produced which will be passed to the Circuit Construction program with the declared object templates to produce the circuit representation which will be used for either simulation or implementation. The statements in VHDL can be classified into the following catagories :

- **declarations** these statements define design types and values.
- **specifications** these statements assign values to declared objects.
- **generation statements** these statements define the execution of statements used in the elaboration of a design.
- **behavioral statements** these statements define the concurrent behavior of the elaborated circuit during simulation.

The use of parameterized generic values in an design entity description, implies that expression evaluation can not occur until elaboration. The pseudo-code ($PCODE$) statements will take the form

$$PCODE = (ACTION, EXPRESSIONS, ENTITIES)$$

where $ACTION$ is a pseudo op-code, $EXPRESSIONS$ is a set of parsed but not evaluated expressions, and $ENTITIES$ is a set of VHDL entities.

Each declaration statement defines a new entity, and each specification statement associates additional information with the named object. For an entity such as a type, the declaration statement must create an internal model of the abstract type, and a series of pseudo-code statements to assign object attributes specific values.

Generation statements are statements which are explicitly stated in the VHDL design description which can be sequentially executed during elaboration to instantiate parallel circuit structures. For example, a series of component instantiation statements will produces a set of components in the circuit representation independent of the order of the original pseudo-code statements.

Behavioral statements are used to describe the behavior of signals during the simulation of a design. While they have no explicit meaning during elaboration, VHDL processes have to be initialized to reflect the behavior represented by these statements.

3.2.3 Process / Memory Management

Memory management is one of the most complex aspects of the VHDL analyzer. The memory management functions are responsible for the allocation, freeing, garbage collection, and referencing of memory locations for distinct purposes in different portions of the compilation process.

Constant Memory This type of memory is used to hold values whose size and value are both known at compile time. The address is stored relative to the "cmemory" pointer in current process structure.

Static memory This type of memory is used to store values whose size is known at compile time, but whose value is determined during elaboration. Typically this is used to store predefined attributes of declared objects.

Dynamic memory This type of memory is used to store information whose size and value is not determined until elaboration. This type of memory is usually used to hold values of signals, and variables. The use of generic values to parameterize type definitions implies that signal and variable memory requirements are not defined until circuit construction. Dynamic memory is implemented by storing a pointer to the allocated memory in a static memory location.

Each logical design unit, design block, VHDL process, or subprogram has associated with it an process control structure, and every declared entity is associated with one and

only one process control structure. This process control structure is not to be confused with VHDL processes.

3.3 Intermediate Notations

The VHDL analyzer creates an intermediate notation describing the VHDL design in an internally consistent format. This format called VIN (VHDL Internal Notation) can exist in two formats. The first of which is the binary format which is produced by the VHDL analyzer. An alternative format is a textual notation which is used to store design descriptions in library databases. This section discusses both the binary and text versions of the VIN.

3.3.1 Binary VIN

The binary VIN is intended to be the primary notation used to describe parsed designs. This notation contains the following information :

- Symbol table contains all named indentifiers in the design, and pointers to information and models of each identifier.
- Elaboration instructions are a sequence of instructions to be executed during design elaboration.
- VHDL Statements are a sequence of VHDL statements corresponding to the original VHDL design description. This information is used by the reverse analyzer to print out VHDL is proper sequence.
- Process Control information is used to control visibililty and memory management of different VHDL structures. Every named VHDL entity has an associated VHDL process.

A more complete description on the binary VIN notation is given in appendix C.

3.3.2 Textual VIN

Binary formats are efficient for storage in computer memory, but create difficulties for storage in design libraries. In order to ensure portability of design libraries, and offer additional debugging facilities, it was decided to create a text based intermediate format. The textual VIN notation is an intermediate notation used to represent design information in the design library.

The lack of an accepted standard for the intermediate format led to the creation of the following format. The primary guideline was to devise a simple language which would meet the following constraints : It must be both simple to generate and parse; it must have the same expression capabilities as VHDL. The context-free grammar representation of this notation is defined as follows :

V_N : the set of nonterminals is may be found in the appendix C. Most nonterminals correspond directly to binary VIN structures or VHDL statements.

V_T : { INTEGERS, REALS, STRINGS, CHARACTERS, BIT_STRINGS, IDENTIFIERS, NULL, '{', '}' }. The NULL terminal is used to represent null pointers or unspecified information.

S : VIN_TOP.

Θ : The productions are similar in structure to list oriented languages such as EDIF. Each production rule contains a left brace as the first elements and a right brace as the final element in the production rule. The production rules are of the format

$$V_N \rightarrow \text{`\{`} \ (V_N \ \cup V_T \ )^* \ \text{`\}`} \tag{3.5}$$

The production rules are shown in the appendix.

Since the VIN notation is intended to only be produced by the VHDL analyzer, no error detection beyond simple syntactical checking is performed.

3.4 VHDL Reverse Analyzer

The translation of the VIN notation back into VHDL code is performed by the reverse analyzer. The reverse analyzer produces a semantically equivalent but syntactically different VHDL description from the original description. The reverse analyzer serves two primary purposes. If the original VHDL code is unavailable, the VIN library entry may be translated back to VHDL. Secondly, any synthesis or model manipulation performed on an unelaborated design may be translated into a VHDL description. Elaborated VHDL may be output as a VHDL netlist (see next chapter).

In order to guarantee a correct VHDL description, the VHDL statements must be produced in the same order as they are interpreted. For example, a type declaration must come before a signal declaration which uses that type. To ensure that this order is maintained, the VHDL analyzer generates a list of statements. A single statement in the VHDL description with multiple entity declarations is treated as multiple statements.

3.5 Error Handling

It is important for a compiler to be able to handle errors in the input syntax. Ideally, the compiler should be able to both report and recover from errors.

Both the lexical and syntactical analyzers produce error tokens which are passed on to the semantical analysis phase. If a sufficient number of errors are detected in either the lexical or semantic analyzer stages, compilation is aborted.

The semantic analysis stage is responsible for any attempts at error recovery. For common errors and simple problems, special recover routines have been implemented. Whenever an error is detected, a message is added to the analyzer's error log. Upon completion of the compilation, or on abortion due to errors, the error log is printed. Each message includes a source line and position number to help identify the location of the error.

Error recovery is in general quite a difficult task for compilers. Some of the common error have special error recovery procedure associated with them to increase the possibility of continued compilation. More drastic and uncommon errors will tend to cause the immediate termination of the compilation with the current error log printed out.

Chapter 4

Structural Compilation

The VHDL structural description is the design abstraction most closely linked to the layout implementation process. Whether the VHDL structural description was entered directly by the IC designer or generated by a synthesis program, the structural compiler is responsible for producing a circuit description that is compatible with the physical layout generation program.

The primary design methodology employed is based upon the standard and macro cell design approachs. In the standard cell approach, an IC layout is generated by connecting the ports of simple logic gates. These logic gates implement the basic and most commonly used digital logic functions, and are typically designed with strict structural constraints such as constant power rail separations. The macro cell design approach allows the basic building blocks to be less constrained than the standard cell approach, and generally implements higher complexity functions. Macro cells may be composed of standard cells and/or macro blocks. The Cadence Edge system is capable of placing and routing both macro blocks and standard cells. It is important that the structure compiler be able to both describe circuits built with standard and macro cells, as well describe the structure of macro cells.

The structural compilation process shown in figure 4.1 involves first the elaboration of design descriptions. Each component instance in the design which does not correspond to a standard cell is hierarchically elaborated. A circuit simplification procedure is then invoked which removes any redundant nodes and connections introduced in the hierarchical expansions of the design. Upon completion of this process, a flat circuit representation of the

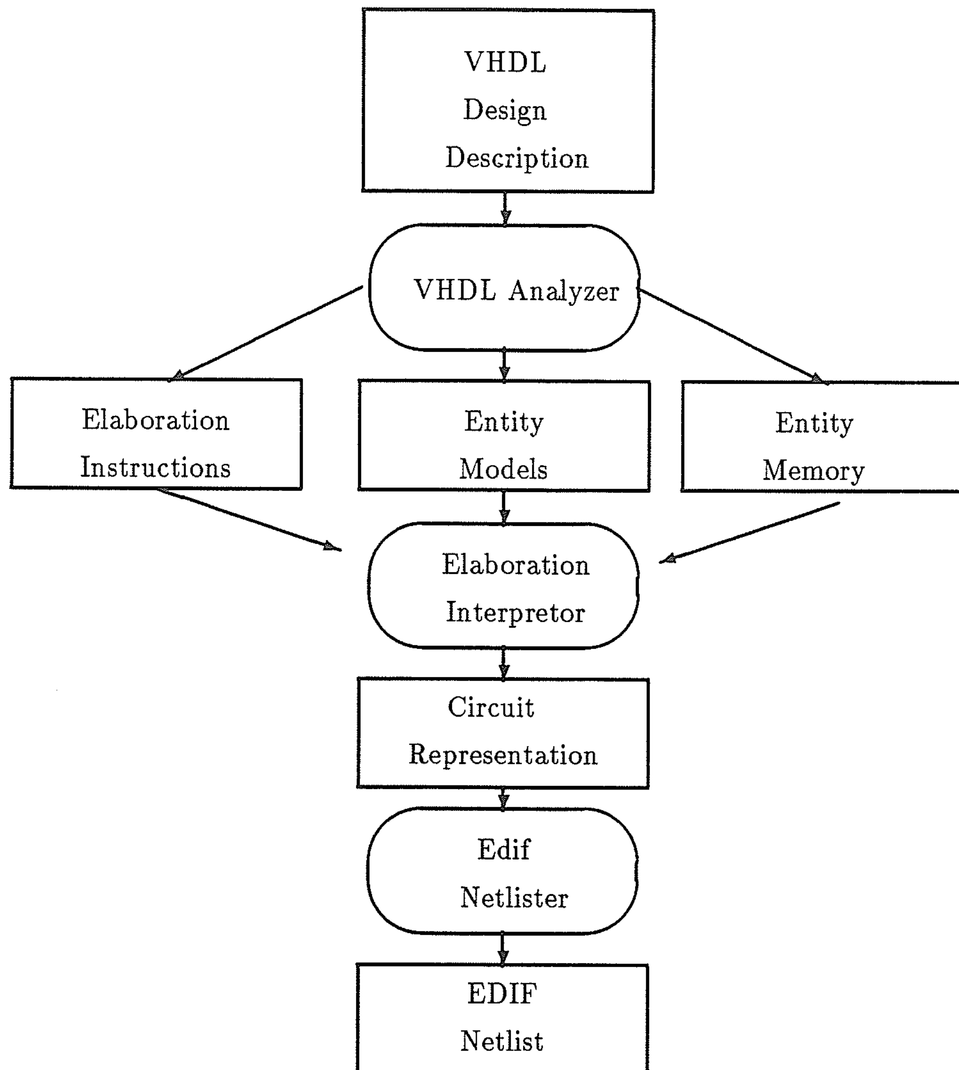


Figure 4.1: Structural Compilation

design is available. Two netlist extractors are available to interface the circuit information to other design tools. The EDIF netlister produces a netlist primarily intended for the layout generation programs, whereas the VHDL netlister produces a simplified VHDL structural description of the design.

This chapter specifically examines the implementation of the design tools used in each step of the structural compilation process.

4.1 Circuit Construction

The basic modules of the VHDL structure compiler implemented are shown in figure 4.1. The VHDL analyzer which is responsible for parsing the input VHDL description, produces a VHDL Internal Notation (VIN) which is an intermediate description of the design information. Included in this information is the following :

- **Entity Models.** Each named entity in VHDL, as well as derived objects and types has an associated internal model which reflects the semantics of the entity.
- **Entity Memory.** Entities which correspond to distinct modules such as entity interfaces, entity bodies, and subprograms, each required local memory which must be maintained during elaboration (circuit construction).
- **Elaboration Instructions.** Each design unit has a sequence of elaboration instructions which must be executed during elaboration. These instructions perform functions such as object initialization, attribute initialization, component instantiation, and process creation, memory allocation, assignments, and conditional branching.

Elaboration is the process of initializing VHDL objects, types, components (through instantiation), declarations, and specifications. For example, during elaboration of a generic description of the n-bit adder in figure 2.4, the value of n is initialized and a corresponding circuit representation of an adder is produced with n full adders instantiated. Upon completion of the elaboration process, the elaborated circuit representation contains a structural model of the design

Elaboration of a VHDL design description is performed by the execution of sequences of elaboration instructions produced by the VHDL analyzer which act on abstract generic datatypes. The execution of these instructions is similar to the execution of machine instructions produced by a programming language compiler.

4.1.1 Elaboration Instruction Interpretor

Each design module in a VHDL description contains a sequence of elaboration instructions which must be executed during elaboration. All actions of the elaboration interpretor are specified as instructions for the Elaboration Instruction Interpretor which sequentially executes these instructions.

The interpreter operates by mimicking a complex processor. This processor can be viewed as operating on two separate memories, one for instructions, and one for data. The instructions are produced by the VHDL analyzer, and contain indirect pointers to values and objects in the data memory. Data may be stored with the instructions, but only if it is of predetermined size (determined during analysis), and is allocated before elaboration commences. Instructions may not be modified during elaboration. The data memory is used to hold the generated circuit representation, and to hold any memory allocated during elaboration. Static memory referred to in the previous chapter is in instruction memory, whereas dynamic memory is held in data memory.

The instructions which the interpreter recognizes range in complexity from relatively simple operations, similar to those found in conventional processors, to complex and specialized operations unique to the interpreter. Many of the complex instructions could be converted into sequences of simpler operations, but the use of these instructions was found to make the intrepretor both simplier to implement, and more efficient in operation. The instructions are shown in table 4.1.

The control flow instructions (group CF) are used to control the execution of the elaborator. The start and return are the first and last statements respectively in each module. The noop and break operations are used primarily for debugging purposes. Branch, conditional branch, and call are used to divert instruction flow.

The module control instructions (group MC) are used to manipulate the module control

Instruction	Function	Group
1	Start	CF
2	Return	CF
3	NOOP	CF
4	Operation	EV
5	Operations	EV
6	Component Instantiation	CG
7	Conditional Branch	CF
8	Branch	CF
9	Assignment	EV
10	Start Process	MC
11	Stop Process	MC
12	Elaborate DU	MC
13	Clear Process	MC
14	Allocate Cvspg	EV
15	Allocate Memory	EV
16	Allocate Attribute	EV
17	Break	CF
18	Entity Ports	CG
19	Entity Generics	CG
20	Use statement	CG
21	Allocate Type	EV
22	Subprogram Call	CF
23	Conc. Signal Assignment	SM
24	Assertion	SM
25	Process	SM

Table 4.1: Elaboration Instructions

information. This is used to allocate memory, and set process information. Each module has an associated process information structure with it. Since a module may exist more than once in any given design, the memory information associated with each module is kept distinct.

Each module has three distinct sets of instructions which are executed one after another, and serve distinct purposes. The first set represents the initialization instructions which correspond to the declaration and specification statements in the original VHDL source, as well as any objects not explicitly declared (for example types associated with index ranges). These instructions are primarily concerned with initializing the module memory blocks, and

initializing the type declarations for the module. The allocation of memory in expression evaluation must also be performed in this section.

The second set of instructions corresponds to the statements in the main body of the VHDL descriptions. Included in these statements are the sequential and concurrent statements, including the generation statements with instantiate components.

The third set of instructions is the module termination set which is responsible for the releasing of allocated resources of the module, and the return of the designated values from subprogram calls. Any garbage collection tasks are included in this set of instructions.

In the event that an error is detected during the execution of the the elaboration instructions, an error message is generated and execution is terminated. Further processing is disallowed. Currently no debugging facilities are including with the interpretor, but a log file of structural compiler operation does contain object initialization values which may be used to aid debugging.

4.1.2 Circuit Representation

Through the execution of the the circuit generation instructions by the elaborator, a circuit representation of the design is created. The circuit is a collection of circuit elements, each of which belongs to one of the following sets.

Nodes a node is circuit element which represents a signal, port, or a component instance port in a design.

Variables a variable is a data abstraction whose values are used during the elaboration of a design. These value serve no purpose after the elaboration process.

Instances an instance is an instantiation of a component in a design. Each instance either represents another VHDL module (hierarchical descriptions) or a standard cell.

Connections a connection represents the logical connectivity between nodes in a design.

Processes behavioral statements are represented by VHDL processes which are concurrently executable instruction streams.

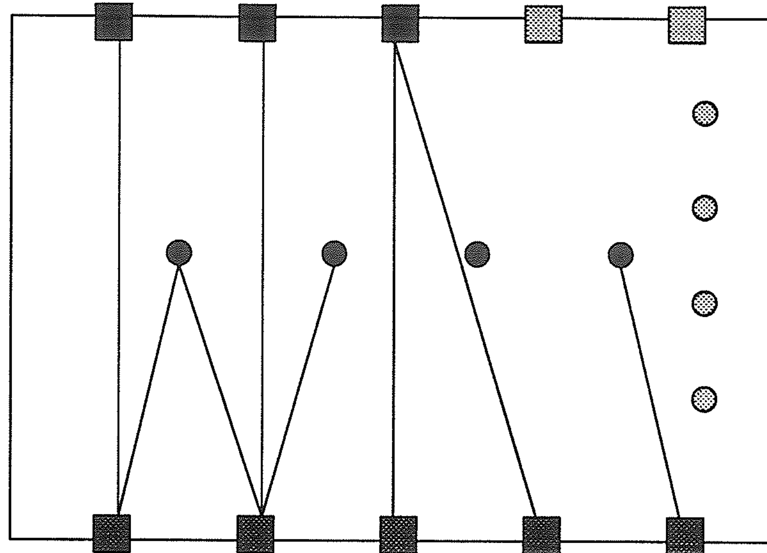


Figure 4.2: Entity Representation

Module Control Information these elements are used to represent the memory allocation information of the current design module.

Attributes attributes are used to store and retrieve various forms of circuit information.

The connectivity information of a single entity is illustrated in figure 4.2. The ports are shown as black rectangles on the top of each entity, and the generics are shown as gray rectangles on the top. Component ports and generics are shown on the bottom of each entity. Circular dots represent signals and variables within design entity. It is important to note that ports and signals are used in identical manners, and that no connections are allowed between ports and signals (or between signals or between ports).

4.1.3 Hierarchical Elaboration

The final step in the elaboration of a design entity, the instantiated components with corresponding structural VHDL descriptions must also be recursively elaborated to completely elaborate the design hierarchy. The execution of each component instantiation statement causes the component instance to be placed in a queue. The execution of the Elaborate DU instruction causes the structural compiler to elaborate the corresponding design units of

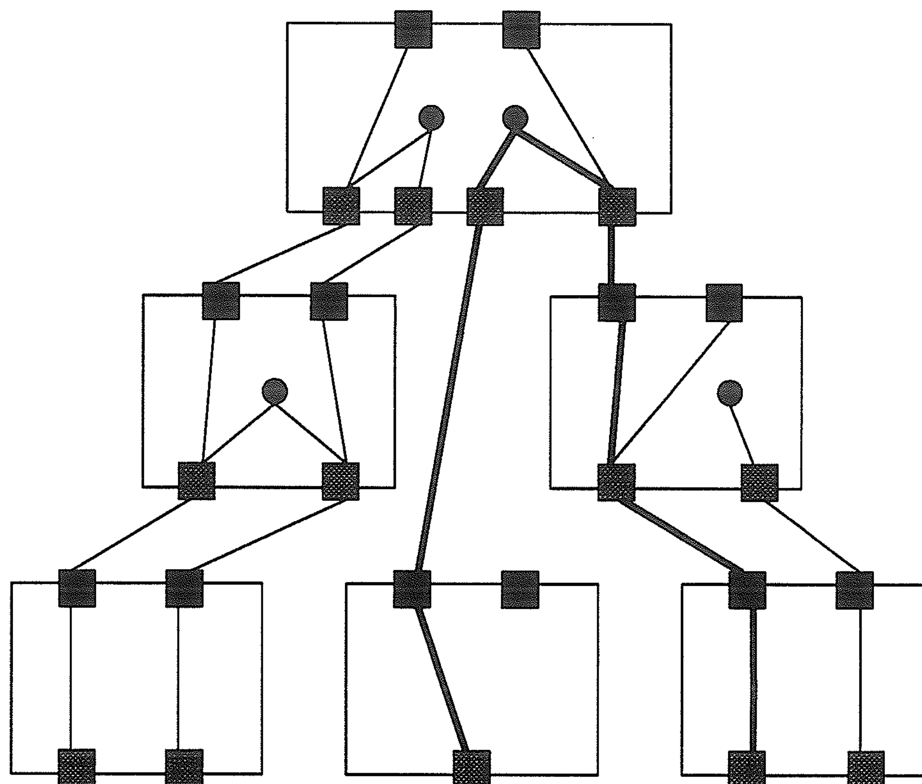


Figure 4.3: Circuit Representation

each hierarchical component recursively until only the standard cell / macro cell component instances remain in the circuit.

There are two important differences in the elaboration of the a design entity (child instance) which corresponds to a component in another design entity (parent instance). The parent instance must pass to the child instance both the generic values, and port connections assigned to the instance during the instantiation instruction execution. A top level design instance determines its generic parameters from the entity interface declaration, and has no external port connections. A child instance, on the otherhand, is dependent on the generic variable values, and must be connected to the component instance's ports.

The basic connectivity of a design hierarchy is shown in figure 4.3. Each module represents a design entity. The ports are shown as black rectangles on the top of each entity, and the component ports are shown on the bottom of each entity. Circular dots represent

signals within design entities. Some observations should be made about the connectivity :

- A logical net in the hierarchical can extend across several entities and levels of the hierarchy. The thick line in diagram shows the signals and ports that are all part of the same net.
- The design hierarchy is in the topology of a tree. Each entity may have several components beneath it, but these components are unique to that entity, and may not have connections to other entities.
- Not all ports and signals need to have connections attached to them. Signals may be used for purposes other that connectivity.

4.2 Circuit Simplification

The circuit representation produced by the design entity elaboration, is a more complex circuit representation than is allowed or required by most netlist formats. The circuit simplification program is designed to reduce the circuit complexity to a simpler model which meets the constraints of the structural netlisters. At the same time, the design is checked for potential design errors, and whether or not it meets the constraints of the netlist formats. The simplified circuit representation will also be flat, containing no instances other than the top level and references to external standard cells.

The circuit representation generated by the design entity elaboration contains a one to one mapping between VHDL objects and current nodes. In a hierarchical design, two nodes will be created in the circuit representation which will correspond to the same logical node. A port of a component instance, and the elaborated design entity's port will represent the same circuit node, which in turn may be part of a larger circuit net. This circuit representation keeps those nodes as separate entities for two reasons. The separation of the nodes in the circuit representation closely ties the original VHDL structure to the circuit format, allowing distinct levels in the design hierarchy to be identified in the circuit. Secondly, since the two ports are associated with different processes in the symbol VIN, and have unique assertions, attributes and other objects associated with them, it is more logical to maintain the distinction.

The desired simplified circuit contains only four circuit elements : nets, ports, instances, and connections. Ports are used to represent the external connections of the both the entity ports, and the component instance ports. Nets are used to represent common connectivity of ports, and connections explicitly represent each connection. Connections may exist between ports and and nets. Connections between ports, or between nets are forbidden.

4.3 Circuit Netlisters

There are currently only two ways of accessing the elaborated circuit representation of a VHDL design. The first method is to write a program to access the circuit representation directly, and the second is to convert the circuit representation into one of the two supported netlist formats. Two netlist formats are currently used for design information interchange, although other interfaces could easily be written.

The primary netlist format is the EDIF [15] format. This format was chosen for its wide acceptance among design tools, as well its support of different design "views". The netlist view is used to descibe design connectivity and properties which are required by the Cadence EDGE tools.

A second netlist format used is VHDL itself, although restricted to a smaller subset than that implemented for this thesis. This subset is simple enough to allow interfacing to other design tools using VHDL subsets and general enough that simple translation can produce netlist for other tools.

4.3.1 EDIF Generation

The circuit representation generated by the elaboration process is a hierarchically expanded structural equivalent of the original VHDL description. The conversion of this design model to an EDIF netlist view proceedes as follows :

- generate a list of all standard cell components used in the design.
- for each standard cell instantiation, generate an EDIF instance statement. If port mapping are specified in the component attributes, the specified names are substituted

for the VHDL names. Each standard cell component must then have its implementation attributes mapped into the appropriate EDIF properties. Labels on instantiation statements are used to generate unique instance identifiers in the EDIF syntax.

- Each signal used in the circuit representation is mapped into an EDIF net construct, and all appropriate attributes are specified as EDIF properties. Composite type signals are transformed into multiple single bit signals, each uniquely identified. Each single bit signal is than connected to the corresponding component ports.

- Technology dependent signals, connections, and properties are next inserted into the EDIF description. This includes the addition of power and ground supply connections, and any global properties.

Figure 4.4 shows an attribute declaration and specification used to convey technology dependent information in VHDL which the EDIF netlister will place into the appropriate EDIF property constructs in the netlist. The constant CDS_MAX_PORTS and types cds_port_type, cds_port, cds_ports, and cds_struct declare the type used in the CDS_INFO attribute. This attribute is used with each standard cell. The attribute specification is shown for the inverter standard cell. Since the type cds_struct is a compound type, an aggregate is used to specify its values. The enumeration type cds_port_type defines the type of the port. The component declaration for the invertor contains only two ports (input and output), these are mapped to names In and Out (case sensitive). Two additional ports are added in the attribute specification, one for power and one for ground. It is possible to add other properties in a similar manner.

4.3.2 VHDL Netlister

While the EDIF netlist is the primary output format of the structural compiler, a VHDL netlist was also determined to be benificial. The VHDL netlister produces an extremely simple, non hierarchical structural description of the design. The primary purpose of this netlist is to allow the interfacing of the VHDL tools to other design tools, which utilize only a subset of the VHDL language. Two examples of this are the QUISC silicon compiler [18], and the Xilinx VHDL compiler [19]. The netlister is distinct from the VHDL reverse analyzer which reproduces the original VHDL code.

```
package components is
    constant CDS_MAX_PORTS : INTEGER := 7;
    type cds_port_type is ( POWER,GROUND,CDS_SIGNAL,DUPLICATE ) ;
    type cds_port is record
        vhdl_name : string(1 to 32);
        cds_name : string(1 to 32);
        port_type : cds_port_type;
    end record;
    type cds_ports is array ( integer range <> ) of cds_port;
    type cds_struct is record
        num_ports : integer;
        cell_name : string(1 to 32);
        ports : cds_ports ( 1 to CDS_MAX_PORTS ) ;
    end record;
    attribute CDS_INFO : cds_struct;
    component inv port
        ( input : inout Bit;
        output : out Bit );
    attribute CDS_INFO of inv:component is
        (4,"inv ",
        (("INPUT ",
        "In ",
        CDS_SIGNAL),
        ("OUTPUT ",
        "Out ",
        CDS_SIGNAL),
        ("VDD ",
        "vdd! ",
        POWER),
        ("GND ",
        "gnd! ",
        GROUND)));
end components;
```

Figure 4.4: Component Attributes

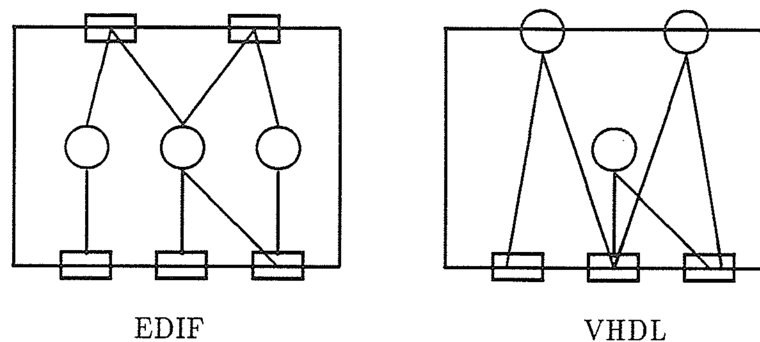


Figure 4.5: EDIF vs VHDL netlists

The VHDL netlister only uses a small subset of the VHDL language, containing only the following statements.

- Entity Interface with only port declarations.
- Architectural Body
- Signal Declarations.
- Component Declarations.
- Component Instantiations.

The only types that may be used are those predefined in the standard VHDL package. Net type declarations are not allowed.

The production of the VHDL netlist from the circuit procedes in much the manner as the EDIF netlist generation previously discussed. There are two fundamental differences in the handling of ports and node types. EDIF netlists view ports as similar to component ports. whereas VHDL views entity ports as similar to signals. The difference can be seen in figure 4.5. Objects on the top of a cell are ports, and objects on the bottom are component ports. Rectangles represent ports, and circle represent signals. The edif netlist treats cell ports as distinct from signals. Cell ports must be connected to nets.

VHDL on the otherhand, treats entity ports as signals.

Another major difference is that in the EDIF netlist, all nets are single bit. The VHDL netlist allows multiple bit (using predefined TYPE bit_vector) signals and connections.

Chapter 5

Design Simulation

Simulation is one of the most critical steps in the design process of integrated circuits. The high cost and the slow turn around time of IC fabrication, make it imperative that a design be thoroughly simulated before implementation. Typically simulators can be divided into two classes, functional simulators, and circuit simulators. Circuit simulation is concerned with the simulation of the timing characteristics based on the physical geometries of a design layout. Functional simulation, on the other hand, is primarily concerned with characterizing functional behavior, and verifing design structures. Since simulation can flag design errors, it is important that simulation be carried out concurrently with system design. In this chapter, the use of VHDL for functional simulation of a design is discussed.

Since a VHDL design description typically is describing technology independent circuit information, it is logical to implement a functional simulator for use with behavioral designs. Circuit simulation may be performed after a design layout is produced, and parasitic value extracted. The VHDL language was specifically designed for use with discrete event simulation algorithms

In the previous chapter we were concerned with transforming the structural description of a design into a circuit representation. In this chapter we will be using the behavioral statements of the VHDL language, and transforming them into a model which can be used to simulate a design. In the first section, the basic processor model which is used to represent behavioral statements is discussed. The next section, discusses the basic simulation algorithm, and the final section discusses the simulation of hierarchical designs.

5.1 Simulation Model

The behavior statements of VHDL may be simulated using a discrete event simulator by producing one or more simulation processes for each behavioral statement. It is through the execution of these processes that the dynamical behavior is observed.

In this section, the modeling of behavioral statements in VHDL as processes is discussed. The basic model of the process is introduced. In addition a more sophisticated model is used to represent circuit signals than was required for structural compilation. This is also discussed as well as the discrete event simulation queue.

5.1.1 Simulation Processes

VHDL behavioral statements are the subset of the VHDL language which can be catagorized as statements responsible for signal assignments. Whether these statements appear by themselves or a part of a more complex module like a VHDL process, the basic function is the same, to assign values to signals at specified time under specified conditions. Typical examples of behavioral statements are as follows :

- Concurrent Signal Assignments are a set of behavioral signal assignment statements which operate in parallel.
- VHDL processes are modules which describe more complex behavior than is possible is simple concurrent signal assignment statements.
- Assertions are used to impose conditions on the allowed values of VHDL objects.

Each behavioral statement is converted into a simulation process. A simulation process consists of a sequence of instructions which are sequentially executed when a process is active. These instructions are shown in table 5.1. While this set of instructions is very small, they provide the basic operations sufficient to simulate a design. A process which is not active will have its current instruction pointer (pointer to instruction being executed) pointing to a wait instruction. When the condition on the instruction becomes true, the process execution will continue. Most processes will execute an infinite loop, after the

0	NOOP
1	Waveform Assignment
2	Wait on Change
3	Wait on Condition
4	Wait for Condition
5	Branch on Condition
6	Assertion

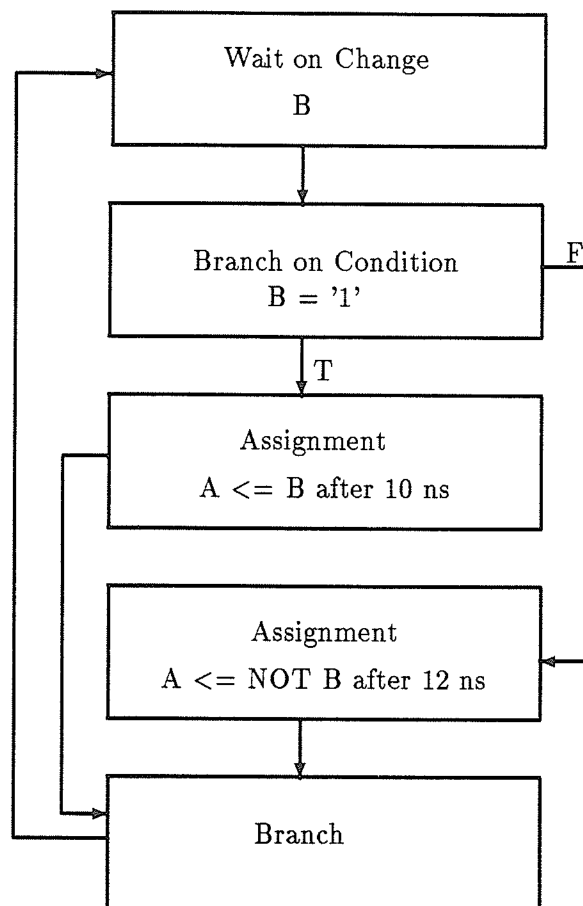
Table 5.1: Simulation Instructions

intended assignment statement is executed, the process will branch to the wait statement, and wait for further change. A process waiting for a condition is said to be idle.

The "wait on change" instruction causes the simulation process to become idle until on the specified signals under go transition at a future simulation time interval. The wait on condition instruction performs a similar function, except that it waits for an arbitrary condition to become true. The condition may be any valid VHDL boolean expression. Since most processes are executing infinite loops, on the execution of the wait instructions, the condition is not evaluated. The evaluation only occurs after the simulator clock is advanced. This prevents processes from never terminating their execution. The "Wait for Condition" instruction does not implement this waiting period, otherwise it is identical to the "Wait on Condition" instruction. The branch statement is used to control the process instruction flow. A branch instruction is achieved by using a conditional branch with a constant logically true expression.

The assertion instruction is used to monitor the values of signals and verify that they meet both the constraints of their declared types, as well as any explicitly stated assertion conditions. The assertion instructions evaluates the expressions specified in the assertion statement, reports the appropriate severity level and message in the simulation log. If severity level of error or fatal is detected, the simulation is terminated.

A simple simulation process instruction sequence is shown in figure 5.1. This process corresponds to a simple conditional signal assignment statement. The wait on change instructions causes the process to become active only when the value of signal B changes. When this occurs, the process becomes active and continues execution with the next instruction. The conditional branch statement selects which waveform assignment instruction



A <= B after 10ns when b = '1' else

A <= NOT B after 12 ns;

Figure 5.1: Simulation Process

is to be executed. The selected signal assignment statements in VHDL are converted to a sequence of "branch on condition" instructions which will select the proper waveform assignment instruction to execute. The branch statement at the end of the process returns the process to the wait on change statement where the process become inactive until signal B undergos another transition.

Processes are created during the elaboration of a design entity. For example the conditional signal assignment statement shown in figure 5.1 will create an exec_csa elaboration instruction which when executed will produce the corresponding simulation process. During the creation of a process, all expressions are transformed into a simple expression format where only absolute addresses are used in signal references. This not only eliminates the need to maintain process memory information during simulation, but reduces simulation time by streamlining memory addressing. In addition, any variable references are replaced by a constant reflecting the current value during elaboration. This freezes the variable information to its correct value at the process create time and any further changes in its value will not affect process operation..

At the commencement of simulation, all signals are considered to to be stable for 0 time units (ie just under went transition), and all processes are active. Each process is executed, and simulation continues until the simulation time is reached, errors force termination, or the event queue is empty.

5.1.2 Signals

Signals are the VHDL objects which represent the current state of a design. In a structural description there will be a one to one correspondence between signals in the VHDL descriptions, and nets in the layout. In a behavioral description, signals are only required to represent ports of a module, but may also be used to represent the internal state of a circuit, for which there is no correspondence to actual nets or physical meaning in the circuit.

Each signal in the circuit under simulation not only contains the current value of the signal, but also maintains a history of past values. This list of previous values and time is used not only to store simulation results, but is required for the implementation of the predefined stable and quiet attributes of signals. These attributes are essential in the modelling of digital systems [2].

In addition to the state of a signal, the list of connections produced by the structural compiler is maintained. This allows a mixture of structural and behavioral descriptions to be used in a design hierarchy. In a hierarchical simulation, the equivalent nodes (and subnodes) of each node are required. This is discussed later in this chapter.

5.1.3 Simulation Event Queue

The simulation event queue, is the part of simulator which maintains a list of events to happen at future times. The execution of processes do not cause events to occur, but causes events to scheduled to occur at specific times. If the scheduled event is not cancelled before the simulator reaches the specified time, the event transaction will take place.

Each signal assignment instruction has an associated delay value (defaults to zero if not explicitly specified). Whenever a simulation process executes an assignment instruction, the delay value on the assignment is added to the current simulation time, and the assignment value is placed in the simulation event queue. This queue holds all the future values of a signal. Each signal contains a list of all pending events in the event queue which reflect that signal.

VHDL supports two delay models for a signal assignment. Inertial delays represent the typical delay model of a logical gate. After an input signal changes values, the output value will take a new value after a specified delay. However if the input signal undergoes another transition before the output value is assigned, this output value is cancelled (removed from the queue) and a new output waveform is assigned. The second delay model, transport delays, are used to represent delays in transmission lines where input transition do not effect previously scheduled output assignments. The simulation event queue must support the deletion of waveforms.

5.2 Simulation Algorithm

In the previous section, the three components of the simulator were discussed. In this section, the operation of the simulator is detailed.

The VHDL simulator does not currently allow any input signal values to be described that are not part of the VHDL description undergoing simulation. Test vector to the design

are specified by creating a new level of hierarchy in the description and specifing test vectors as concurrent signal assignment statements. This was done for three reasons :

1. This eliminates the need for an additional test vector file and formats.
2. The VHDL language contains more than enough constructs to describe input waveforms.
3. This simplifies the design of the simulator.

The simulation of a behavioral description is performed by the repeated execution of the two phases of the simulation cycle. In the first phase, all active simulation processes are executed. This may produce simulation actions (assignments) in the event queue. In the second phase, the actions for the current time interval are performed. This may cause simulation processes to become active, and/or cause the deletion of other events in the queue.

The simulation time steps are represented by a two dimensional vector (time_unit, offset). The time_unit value is an integer presentation of the simulator time. This represents the discrete time intervals in which the simulator events are specified. Typically this time_unit will represent a nanosecond. Only the time_step intervals are observable to the user. The offset represents a further division of each time step into smaller intervals. While these time steps all logically occur at the same time, the division is made to ensure consistency in parallel operations. If a waveform assignment instruction is executed at time (m,n) with a zero delay specified, the actual assignment will occur at time (m,n+1). If the assignment had occured at time (m,n), this may have introduced hazards and races which would be dependent upon the execution order of parallel processes. The offset time intervals allow a consistent methodology for the execution of parallel processes.

The simulation cycle is repeatedly executed until either no more events are scheduled to occur, or the user specified maximum time is exceeded.

The first step in the simulation cycle is the execution of actions in the event queue scheduled for the current time interval. Each action specifies a target for the value to be assigned. This target may be a node in the circuit, or a portion of a node in the case of signals which are compound types. These actions update the current value of the specified

```
type control_struct is record
    line1 : bit;
    line2 : bit;
end record;
type bus is record
    address_bus : bit_vector(0 to 16);
    data_bus : bit_vector(0 to 16);
    control_bus : control_struct;
end record;
signal buss : bus;
buss.data_bus(5) <= '0' after 10ns;
buss.control_bus.line1 <= '1" after 20 ns;
```

Signal	'Last_event
buss	20 ns
buss.address_bus	0 ns
buss.data_bus	10 ns
buss.data_bus(0)	0 ns
buss.data_bus(5)	10 ns
buss.control_bus	20 ns
buss.control_bus.line1	20 ns
buss.control_bus.line1	0 ns

Figure 5.2: Signal Stability

signals. The current value of the signal is stored in a list of past values for the signal, and the newly assigned value becomes the current value. Once a new value has been assigned, a signal transition is said to have occured for the specified node. A list of changed nodes and their targeted portions is maintained for each time interval, reflecting the changes in the circuit. After all scheduled actions have been executed, the changed list will be examined to determine active processes. If any future events in the event queue have targeted the same node, and have an inertial delay specified, they are removed.

It is important to distinguish which portions of a node were targeted in the signal transitions. In the case of a composite record type if one field changes value, the entire signal is considered to have undergone transition. However other fields of the record are not considered to have under gone a transition. This is demonstrated in figure 5.2. The stability of signals in a composite bus structure is illustrated.

The second step of the simulation cycle is the execution of processes which have become active. All targets which underwent a transition are then checked to see if they are on the sensitivity list of any simulation processes. If there exist processes waiting on these signals, they are executed.

If more than one event is scheduled for a target at a specific time, the bus resolution feature should select the correct driving value. Currently this feature is not supported, and buses are handled on a first come first serve basis.

The final step in the simulation cycle is the advancement of the current simulator time. The simulation time is sent to the time value of the next scheduled event in the event queue. If no further events are scheduled, simulation terminates.

5.3 Hierarchical Simulation

The primary purpose of the design tools developed in this thesis was to aid in the design of VLSI ICs. In order to obtain the most benefit from the simulator, it is imperative the the design hierarchy to be simulated include both structural and behavioral modules. This can be seen in figure 5.3. For each standard cell in the design hierarchy, a behavioral representation of the entity is elaborated.

The structural modules in the the design hierarchy are modelled as signals and connections. In order to simulate the structural components, each structural module must be transformed into a behavioral representation. This is achieved by creating a simulation process for each connection. Some special provisions are placed on these processes.

1. The delay on the assignment instructions must be (0,0).

2. The process must be able to model a bidirectional connection. Both ends of such a connection must not be driving the other end at the same time.

These requirements are implemented by allowing each assignment action to have multiple targets. At the start of the simulation, logically equivalent signals are identified, and stored for each node. Assignment to any of these nodes, causes the same value to be assigned to all the other logically equivalent signals at the same time.

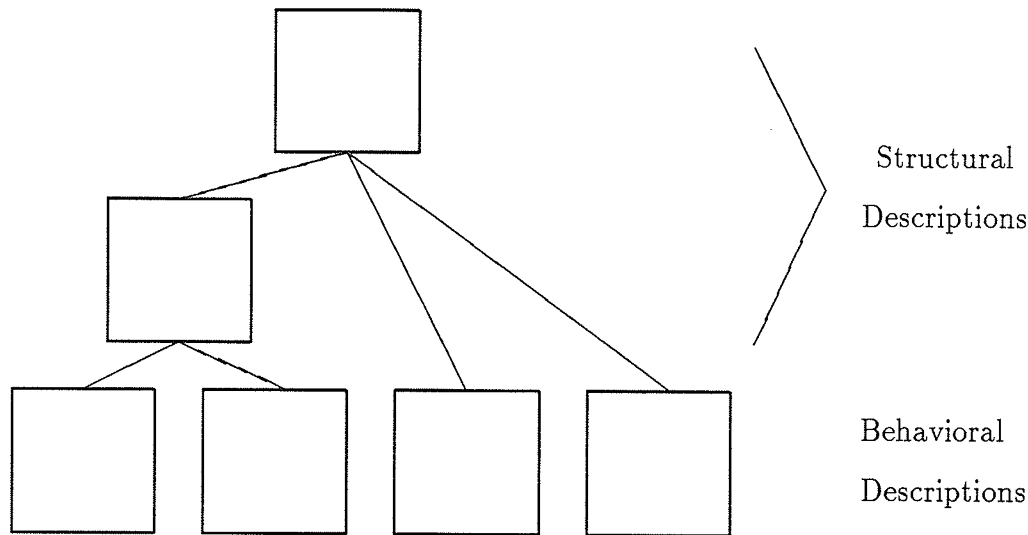


Figure 5.3: Design Hierarchy

5.3.1 Equivalent Nodes

In order to efficientally and correctly simulate a hierarchical VHDL design description it is important that each node in the design know of all the other nodes in the circuit to which it is logically equivalent. Two nodes are defined to be equivalent if they are part of the same net in the structural hierarchy. For example, the component port created in one level is equivalent to the entity interface port of the elaborated component in the next level. Both these nodes represent the same net, and any value assigned to one, must also be assigned to the other. Before simulation can begin, the set of equivalent nodes of each node in the design must be determined.

In the case of composite type signals, certain portions of signals may be equivalent.

Equivalent node information is stored using the standard connection mechanism discussed in the previous chapter, only associating a flag with each connection indicating an equivalent connection. The equivalent node information of a design is determined as follows. Each connection in the design is duplicated, and marked as equivalent. Each time a new equivalent connection is added to the circuit, the nodes at both ends of the connection propagate this new connection to all previously specified equivalent connections of the node.

This is repeated until all connection are found.

Chapter 6

Conclusions

A set of tools based on the VHDL language for use in an IC design environment was designed and implemented. The tools are responsible for analyzing a VHDL design description, performing structural compilation and behavioral simulation of design hierarchies. Interfaces exists to other IC CAD tools through both EDIF and VHDL netlists. This chapter summarizes the results of the thesis, presents conclusions that may drawn from the work, and presents possible future work that may be carried out as extensions of the work presented in this thesis.

6.1 Summary

The tools developed can be thought of as three separate components of a simple VHDL design platform. Included are :

1. A VHDL analyzer which can parse and perform analysis on a VHDL design description. This program produces an intermediate notation describing the design information, which may be stored in a design library, and merged with other designs. Also included is a reverse analyzer for translating the intermediate notation back to VHDL code.

2. A VHDL structural compiler for transforming structural design descriptions into a circuit representation. This representation may be used directly, or exported to other

design tools through either the EDIF or VHDL netlists. Specifically declared attributes may be used to specify Cadence related parameters, so the place and route tools may generate layout geometries from the design netlists.

3. A VHDL simulator was implemented in order that design descriptions may be functionally verified using the same syntax as the design description, thereby keeping verification closely tied in with the design environment.

6.2 Conclusions

1. VHDL is a very suitable language for use in an Integrated Circuit design environment. The emphasis in this thesis was its use based on the standard cell design methodology. The language was found to be flexible enough to support the requirements of the IC design tools with which it was intended to interface.

2. VHDL offers a convenient behavioral description for use with a discrete event simulator.

3. EDIF was found to be a powerful design interchange format, capable of incorporating sufficient constructs to be used in a design environment.

6.3 Future Work

In a field such as VLSI design, where new tools and fabrication techniques are being introduced at a fast place, there is a myriad of desired tools which can aid in the design process. The goal of this thesis was to implement some design tools based on the higher level constructs of a language such as VHDL. The basic analysis, structural compilation, and simulation tools have been implemented but further development of tools is still mandatory. Specifically the following works needs to be done :

- The full VHDL language has not been incorporated into the tools as of yet. The mechanisms are in place to support the full VHDL language, but the complexity and size of the language did not leave time to support all the features of the language.

The currently unsupported features should be added without much modification to the existing code.

- No software is devoid of bugs and these tools are no exceptions. More testing of the tools needs to be done to ensure correctness of operation. To realistically use these tools, it is important to ensure designers of the validity of their operation.

- One of weakest area of the tools is the area of error detection and correction. More work should be placed in the user friendly reporting or errors. Emphasis should also be placed in detection of errors which escape detection currently.

- Synthesis of behavioral design descriptions is an exciting area or research. Currently behavioral descriptions are used only for simulation, but it would be beneficial to allow behavioral descriptions to be transformed into structural descriptions for implementation. This is currently an active area of research.

- The automatic insertion of testing structures into designs is a further extension of these tools which should be examined. Structured testing methodologies have been devised which should be incorporated automatically into structural design descriptions. This is particularly important for structural design descriptions produced through behavioral synthesis.

Bibliography

[1] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987.

[2] Armstrong J. A., "Chip-Level Modeling with VHDL," Prentice Hall, 1989.

[3] Lewi J., De Vlaminck K., Huens J., and Steegmans E., "A Programing Methodolgy in Compiler Construction," Part2 : Implementation, North-Hooland, 1982. Addison-Wesley, 1980.

[4] Bhasker J., "Implementation of an Optimizing Compiler for VHDL," *SIGPLAN Notices,* Vol. 23, pp92-108.

[5] Gilman A. S., "VHDL - The Designers Environment," *IEEE Design and Test,* pp. 42 - 47, April 1986.

[6] Hines J., "Where VHDL Fits Within the CAD Environment," *24th IEEE/ACM Design Automation Conference,* pp. 491 - 494, 1987.

[7] Mead C., Conway L., "Introduction to VLSI System,"

[8] Johnson, Stephen, "Yet Another Compiler - Compiler". Bell Labratories, Murray Hill, New Jersey.

[9] Shadad M., "An Interface Between VHDL and EDIF," *24th IEEE/ACM Design Automation Conference,* pp. 472 - 478, 1987.

[10] Dewey A., Gadient A., "VHDL Motivation," *IEEE Design and Test,* pp. 12 - 16, 1986.

[11] Shahdad M., Lipsett R., Marschner E., Sheehan K., Cohen H., "VHSIC Hardware Description Language," *IEEE Computer,* pp. 94 - 103, 1985.

[12] Silos II Logic and Fault Simulator, User's Manual, Simucad Inc, 1988.

[13] Cadence EDGE system, CADENCE, 1989.

[14] HSPICE Users' Manual H8801, Meta-Software Inc, 1988.

[15] EDIF Specification, Electronic Design Interchange Format, Version 1 1 0, EDIF Steering Committee, 1985.

[16] Fischer, C. N., LeBlanc, R. J., *Crafting a Compiler*, The Benjamin/Cummings Publishing Co., 1988.

[17] Tremblay, J-P, Sorenson, P. G., *The Theory and Practice of Compiler Writing*, McGraw-Hill, 1985.

[18] Kostiuk, Andrew R., *QUISC : An Interactive Silicon Compiler*, MSc. Thesis, Queens University, 1987.

[19] Liu, Bing, *A VHSIC Hardware Description Language Compiler for Logic Cell Arrays*, MSc. Thesis, University of Manitoba, 1990.

[20] *Integrated Circuit Design using VHDL*, Canadian Conference of Electrical and Computer Engineering,1990.

Appendix A

VHDL Grammar

```
TOP                    ->  DESIGN_UNIT TOP
TOP                    ->  e
ABSTRACT_LITERAL       ->  DECIMAL_LITERAL
ABSTRACT_LITERAL       ->  INTEGER_LITERAL
ACCESS_TYPE_DF         ->  ACCESS SUBTYPE_INDCTN
ACTUAL_DESGNTR         ->  EXPRESSION
ACTUAL_DESGNTR         ->  OPEN
ACTUAL_PARM_PART       ->  ASSC_LIST
_AGGREGATE1            ->  , ELEMENT_ASSCTN _AGGREGATE1
_AGGREGATE1            ->  e
AGGREGATE              ->  ( ELEMENT_ASSCTN _AGGREGATE1 )
ALIAS_DECL             ->  ALIAS IDENTIFIER :
                           SUBTYPE_INDCTN IS NAME ;
ALLOCATOR              ->  NEW SUBTYPE_INDCTN
ARCH_BODY              ->  ARCHITECTURE IDENTIFIER OF NAME
                           IS ARCH_DECL_PART BEGIN
                           ARCH_STMT_PART END [ IDENTIFIER ]
                           ;
ARCH_DECL_PART         ->  BLOCK_DECL_ITEM
                           ARCH_DECL_PART
ARCH_STMT_PART         ->  CONC_STMT ARCH_STMT_PART
ARCH_STMT_PART         ->  e
ARRAY_TYPE_DF          ->  ARRAY INDEX_CONSRT2 OF
                           SUBTYPE_INDCTN
_X_ASSERTION_STMT1     ->  REPORT EXPRESSION
_X_ASSERTION_STMT1     ->  e
```

_X_ASSERTION_STMT2 -> SEVERITY EXPRESSION
_X_ASSERTION_STMT2 -> e
ASSC_ELEMENT_X_2 -> OPEN
ASSC_ELEMENT_X_2 -> EXPRESSION
ASSERTION_STMT -> ASSERT EXPRESSION _X_ASSERTION_STMT1 _X_ASSERTION_STMT2 ;
ASSC_ELEMENT_X_1 -> IDENTIFIER => ASSC_ELEMENT_X_2
ASSC_ELEMENT -> ASSC_ELEMENT_X_1
ASSC_ELEMENT -> ASSC_ELEMENT_X_2
ASSC_ELEMENT -> e
_X_ASSC_LIST1 -> , ASSC_ELEMENT _X_ASSC_LIST1
_X_ASSC_LIST1 -> e
ASSC_LIST -> ASSC_ELEMENT _X_ASSC_LIST1
ATTRIBUTE_DESGNTR -> IDENTIFIER
_X_ATTR_NAME2 -> ' NAME_A2 _X_ATTR_NAME2
_X_ATTR_NAME2 -> e
_X_ATTR_NAME1 -> (EXPRESSION)
_X_ATTR_NAME1 -> e
ATTRIBUTE_NAME -> NAME_A1 _X_ATTR_NAME2 _X_ATTR_NAME1
ATTRIBUTE_SPEC -> ATTRIBUTE ATTRIBUTE_DESGNTR OF ENTITY_SPEC IS EXPRESSION ;
_X_ATTR__1 -> QXATTRIBUTE1
_X_ATTR__1 -> QXATTRIBUTE2
ATTRIBUTE__1 -> ATTRIBUTE IDENTIFIER _X_ATTR__1
QXATTRIBUTE1 -> OF ENTITY_SPEC IS EXPRESSION ;
QXATTRIBUTE2 -> : SUBTYPE_INDCTN ;
BASE_UNIT_DECL -> IDENTIFIER ;
BINDING_INDCTN -> ENTITY_ASPECT GENERIC_MAP_ASPECT PORT_MAP_ASPECT
_X_BLOCK_CONF1 -> USE_CLAUSE _X_BLOCK_CONF1
_X_BLOCK_CONF1 -> e
_X_BLOCK_CONF2 -> CONF_ITEM _X_BLOCK_CONF2
_X_BLOCK_CONF2 -> e
BLOCK_CONF -> FOR BLOCK_SPEC _X_BLOCK_CONF1 _X_BLOCK_CONF2 END FOR ;
BLOCK_CONF -> FOR BLOCK_SPEC _X_BLOCK_CONF1 _X_BLOCK_CONF2 END FOR ;
BLOCK_CONF -> e

```
BLOCK_DECL_PART        -> BLOCK_DECL_ITEM
                          BLOCK_DECL_PART
BLOCK_DECL_PART        -> e
_X_BLOCK_HEADER1       -> GENERIC_CLAUSE
                          GENERIC_MAP_ASPECT
_X_BLOCK_HEADER1       -> e
_X_BLOCK_HEADER2       -> PORT_CLAUSE PORT_MAP_ASPECT
_X_BLOCK_HEADER2       -> e
BLOCK_HEADER           -> _X_BLOCK_HEADER1
                          _X_BLOCK_HEADER2
_X_BLOCK_SPEC2         -> ( INDEX_SPEC )
_X_BLOCK_SPEC2         -> e
BLOCK_SPEC             -> NAME
_X_BLOCK_STMT1         -> ( EXPRESSION )
_X_BLOCK_STMT1         -> e
BLOCK_STMT             -> BLOCK _X_BLOCK_STMT1
                          BLOCK_HEADER BLOCK_DECL_PART
                          BEGIN BLOCK_STMT_PART END BLOCK
                          [ IDENTIFIER ] ;
BLOCK_STMT_PART        -> CONC_STMT BLOCK_STMT_PART
BLOCK_STMT_PART        -> e
_X_CASE_STMT1          -> CASE_STMT_ALTRN _X_CASE_STMT1
_X_CASE_STMT1          -> e
CASE_STMT              -> CASE EXPRESSION IS
                          CASE_STMT_ALTRN _X_CASE_STMT1
                          END CASE ;
CASE_STMT_ALTRN        -> WHEN CHOICES => SQN_OF_STMTS
CHOICE                 -> IDENTIFIER
CHOICE                 -> DISCRETE_RANGE
CHOICE                 -> EXPRESSION
CHOICE                 -> OTHERS
_X_CHOICES1            -> — CHOICE _X_CHOICES1
_X_CHOICES1            -> e
CHOICES                -> CHOICE _X_CHOICES1
_X_COMP_CONF1          -> USE BINDING_INDCTN ;
_X_COMP_CONF1          -> e
COMP_DECL              -> COMPONENT IDENTIFIER
                          GENERIC_CLAUSE_Y_1
                          PORT_CLAUSE_Y_1 [ END ] [
                          COMPONENT ] ;
COMP_INSTT_STMT        -> IDENTIFIER GENERIC_MAP_ASPECT
                          PORT_MAP_ASPECT ;
```

COMP_SPEC	– >	INSTT_LIST : NAME
_X_C_S_A_S1	– >	NAME _X_C_S_A_S2
_X_C_S_A_S2	– >	_X_C_S_A_S3
_X_C_S_A_S2	– >	_X_C_S_A_S4
_X_C_S_A_S3	– >	: _X_C_S_A_S5
_X_C_S_A_S5	– >	SLCTD_SIGNAL_ASSGN
_X_C_S_A_S5	– >	CNDTNL_SIGNAL_ASSGN
_X_C_S_A_S4	– >	<= OPTIONS CNDTNL_WVFMS ;
ENTITY_STMT	– >	XLABEL ENTITY_STMT_1
CONC_STMT	– >	XLABEL CONC_STMT_1
XLABEL	– >	IDENTIFIER :
XLABEL	– >	e
ENTITY_STMT_1	– >	IDENTIFIER
ENTITY_STMT_1	– >	PROCESS_STMT
ENTITY_STMT_1	– >	PROC_CALL_STMT
ENTITY_STMT_1	– >	ASSERTION_STMT
ENTITY_STMT_1	– >	IDENTIFIER
CONC_STMT_1	– >	BLOCK_STMT
CONC_STMT_1	– >	PROCESS_STMT
CONC_STMT_1	– >	PROC_CALL_STMT
CONC_STMT_1	– >	ASSERTION_STMT
CONC_STMT_1	– >	GENERATE_STMT
CONC_STMT_1	– >	COMP_INSTT_STMT
CONC_STMT_1	– >	SLCTD_SIGNAL_ASSGN
CONC_STMT_1	– >	CNDTNL_SIGNAL_ASSGN
CONDITION_CLAUSE	– >	UNTIL EXPRESSION
CONDITION_CLAUSE	– >	e
CNDTNL_SIGNAL_ASSGN	– >	TARGET <= OPTIONS CNDTNL_WVFMS ;
_X_CNDTNL_WVFMS1	– >	WHEN EXPRESSION ELSE WVFM _X_CNDTNL_WVFMS1
_X_CNDTNL_WVFMS1	– >	e
CNDTNL_WVFMS	– >	WVFM _X_CNDTNL_WVFMS1
CONF_DECL	– >	CONF IDENTIFIER OF NAME IS CONF_DECL_PART BLOCK_CONF END [IDENTIFIER] ;
CONF_DECL_ITEM	– >	USE_CLAUSE
CONF_DECL_ITEM	– >	ATTRIBUTE_SPEC
CONF_DECL_PART	– >	CONF_DECL_ITEM CONF_DECL_PART
CONF_DECL_PART	– >	e

CONF_ITEM	->	FOR _X_CONF_ITEM0 CONF_ITEM__1 END FOR ;
CONF_ITEM__1	->	CONF_ITEM
CONF_ITEM__1	->	USE__2
CONF_ITEM__1	->	e
USE__2	->	USE USE__3 ;
USE__3	->	BINDING_INDCTN
USE__3	->	USE__4
USE__4	->	NAME USE__5
USE__5	->	, NAME USE__5
USE__5	->	e
_X_CONF_ITEM0	->	_X_CONF_ITEM1 _X_CONF_ITEM4
_X_CONF_ITEM1	->	_X_CONF_ITEM2
_X_CONF_ITEM1	->	OTHERS
_X_CONF_ITEM1	->	ALL
_X_CONF_ITEM2	->	NAME _X_CONF_ITEM3
_X_CONF_ITEM3	->	, _X_CONF_ITEM2 _X_CONF_ITEM3
_X_CONF_ITEM3	->	e
_X_CONF_ITEM4	->	: NAME
_X_CONF_ITEM4	->	e
CONF_SPEC	->	FOR COMP_SPEC USE BINDING_INDCTN ;
_X_CNST_DECL1	->	:= EXPRESSION
_X_CNST_DECL1	->	e
CNST_DECL	->	CONSTANT IDENTIFIER : SUBTYPE_INDCTN _X_CNST_DECL1 ;
CONSRT	->	RANGE_CONSRT
CONSRT	->	INDEX_CONSRT
CONSRT	->	e
CONTEXT_CLAUSE	->	CONTEXT_ITEM CONTEXT_CLAUSE
CONTEXT_CLAUSE	->	e
CONTEXT_ITEM	->	LIBRARY_CLAUSE
CONTEXT_ITEM	->	USE_CLAUSE
_X_DESIGN_FILE1	->	DESIGN_UNIT _X_DESIGN_FILE1
_X_DESIGN_FILE1	->	e
DESIGN_UNIT	->	CONTEXT_CLAUSE LIBRARY_UNIT
DESGNTR	->	IDENTIFIER
DESGNTR	->	STRING_LITERAL
DIRECTION	->	TO
DIRECTION	->	DOWNTO

DISCONNECTION_SPEC	– >	DISCONNECT GUARDED_SIGNAL_SPEC AFTER EXPRESSION ;
DISCRETE_RANGE_1	– >	IDENTIFIER RANGE <>
DISCRETE_RANGE2	– >	RANGE
DISCRETE_RANGE2	– >	DISCRETE_RANGE_1
DISCRETE_RANGE2	– >	SUBTYPE_INDCTN
DISCRETE_RANGE	– >	RANGE
DISCRETE_RANGE	– >	SUBTYPE_INDCTN
DISCRETE_RANGE	– >	RANGE
DISCRETE_RANGE	– >	SUBTYPE_INDCTN
DISCRETE_RANGE	– >	e
_X_ELEMENT_ASSCTN1	– >	CHOICES => EXPRESSION
ELEMENT_ASSCTN	– >	_X_ELEMENT_ASSCTN1
ELEMENT_ASSCTN	– >	EXPRESSION
ELEMENT_DECL	– >	IDENTIFIER_LIST : SUBTYPE_INDCTN ;
_X_ENTITY_ASPECT3	– >	(IDENTIFIER)
_X_ENTITY_ASPECT3	– >	e
_X_ENTITY_ASPECT1	– >	ENTITY NAME _X_ENTITY_ASPECT3
_X_ENTITY_ASPECT2	– >	CONF NAME
ENTITY_ASPECT	– >	_X_ENTITY_ASPECT1
ENTITY_ASPECT	– >	_X_ENTITY_ASPECT2
ENTITY_ASPECT	– >	OPEN
ENTITY_CLASS	– >	ENTITY
ENTITY_CLASS	– >	PROCEDURE
ENTITY_CLASS	– >	TYPE
ENTITY_CLASS	– >	SIGNAL
ENTITY_CLASS	– >	LABEL
ENTITY_CLASS	– >	ARCHITECTURE
ENTITY_CLASS	– >	FUNCTION
ENTITY_CLASS	– >	SUBTYPE
ENTITY_CLASS	– >	VARIABLE
ENTITY_CLASS	– >	CONF
ENTITY_CLASS	– >	PACKAGE
ENTITY_CLASS	– >	CONSTANT
ENTITY_CLASS	– >	COMPONENT
_X_ENTITY_DECL1	– >	BEGIN ENTITY_STMT_PART
_X_ENTITY_DECL1	– >	e

ENTITY_DECL	->	ENTITY IDENTIFIER IS ENTITY_HEADER ENTITY_DECL_PART _X_ENTITY_DECL1 END [IDENTIFIER] ;
ENTITY_DECL_PART	->	ENTITY_DECL_ITEM ENTITY_DECL_PART
ENTITY_DECL_PART	->	e
ENTITY_DESGNTR	->	IDENTIFIER
ENTITY_DESGNTR	->	STRING_LITERAL
ENTITY_HEADER	->	GENERIC_CLAUSE PORT_CLAUSE
ENTITY_HEADER	->	e
_X_ENTITY_NAME_LIST1	->	ENTITY_DESGNTR _X_ENTITY_NAME_LIST2
_X_ENTITY_NAME_LIST2	->	, ENTITY_DESGNTR _X_ENTITY_NAME_LIST2
_X_ENTITY_NAME_LIST2	->	e
ENTITY_NAME_LIST	->	_X_ENTITY_NAME_LIST1
ENTITY_NAME_LIST	->	OTHERS
ENTITY_NAME_LIST	->	ALL
ENTITY_SPEC	->	ENTITY_NAME_LIST : ENTITY_CLASS
ENTITY_STMT_PART	->	ENTITY_STMT ENTITY_STMT_PART
ENTITY_STMT_PART	->	e
ENMRT_LITERAL	->	IDENTIFIER
ENMRT_LITERAL	->	CHARACTER_LITERAL
_X_ENMRT_TYPE_DF1	->	, ENMRT_LITERAL _X_ENMRT_TYPE_DF1
_X_ENMRT_TYPE_DF1	->	e
ENMRT_TYPE_DF	->	(ENMRT_LITERAL _X_ENMRT_TYPE_DF1)
_X_EXIT_STMT1	->	WHEN EXPRESSION
_X_EXIT_STMT1	->	e
EXIT_STMT	->	EXIT [IDENTIFIER] _X_EXIT_STMT1 ;
OPERATOR	->	AND
OPERATOR	->	OR
OPERATOR	->	XOR
OPERATOR	->	NAND
OPERATOR	->	NOR
OPERATOR	->	+
OPERATOR	->	–
OPERATOR	->	ABS
OPERATOR	->	NOT

OPERATOR	->	&
OPERATOR	->	*
OPERATOR	->	/
OPERATOR	->	MOD
OPERATOR	->	REM
OPERATOR	->	=
OPERATOR	->	/=
OPERATOR	->	>=
OPERATOR	->	<=
OPERATOR	->	**
OPERATOR	->	<
OPERATOR	->	>
EXPRESSION_1	->	OPERATOR FACTOR EXPRESSION_1
EXPRESSION_1	->	e
EXPRESSION	->	FACTOR EXPRESSION_1
EXPRESSION	->	FACTOR EXPRESSION_1
EXPRESSION	->	e
FACTOR_1	->	+
FACTOR_1	->	-
FACTOR_1	->	ABS
FACTOR_1	->	NOT
FACTOR_1	->	e
FACTOR_2	->	** PRIMARY
FACTOR_2	->	e
FACTOR	->	FACTOR_1 PRIMARY FACTOR_2
FILE_DECL	->	FILE IDENTIFIER : SUBTYPE_INDCTN IS MODE FILE_LGCL_NAME ;
FILE_LGCL_NAME	->	EXPRESSION
FILE_TYPE_DF	->	FILE OF SUBTYPE_INDCTN
FORMAL_PARM_LIST	->	INTFC_LIST
_X_FUNCTIONAL_CALL1	->	(ACTUAL_PARM_PART)
_X_FUNCTIONAL_CALL1	->	e
GENERATE_STMT	->	GENRTN_SCHEME GENERATE ARCH_STMT_PART END GENERATE [IDENTIFIER] ;
_X_GENRTN_SCHEME1	->	FOR PARM_SPEC
_X_GENRTN_SCHEME2	->	IF EXPRESSION
GENRTN_SCHEME	->	_X_GENRTN_SCHEME1
GENRTN_SCHEME	->	_X_GENRTN_SCHEME2
GENERIC_CLAUSE	->	GENERIC (GENERIC_LIST) ;
GENERIC_CLAUSE	->	e

GENERIC_CLAUSE_Y_1 – > GENERIC (GENERIC_LIST)

GENERIC_CLAUSE_Y_1 – > e

GENERIC_LIST – > INTFC_LIST

GENERIC_MAP_ASPECT – > GENERIC MAP (ASSC_LIST)

GENERIC_MAP_ASPECT – > e

GUARDED_SIGNAL_SPEC – > SIGNAL_LIST : SUBTYPE_INDCTN

_X_IDENTIFIER_LIST1 – > , IDENTIFIER _X_IDENTIFIER_LIST1

_X_IDENTIFIER_LIST1 – > e

IDENTIFIER_LIST – > IDENTIFIER _X_IDENTIFIER_LIST1

_X_IF_STMT1 – > ELSIF EXPRESSION THEN SQN_OF_STMTS _X_IF_STMT1

_X_IF_STMT1 – > e

_X_IF_STMT2 – > ELSE SQN_OF_STMTS

_X_IF_STMT2 – > e

IF_STMT – > IF EXPRESSION THEN SQN_OF_STMTS _X_IF_STMT1 _X_IF_STMT2 END IF ;

_X_INDEX_CONSRT1Z – > , DISCRETE_RANGE2 _X_INDEX_CONSRT1Z

_X_INDEX_CONSRT1Z – > e

_X_INDEX_CONSRT1 – > , DISCRETE_RANGE _X_INDEX_CONSRT1

_X_INDEX_CONSRT1 – > e

INDEX_CONSRT2 – > (DISCRETE_RANGE2 _X_INDEX_CONSRT1Z)

INDEX_CONSRT – > (DISCRETE_RANGE _X_INDEX_CONSRT1)

INDEX_SPEC – > EXPRESSION DISCRETE_RANGE

INDEX_SUBTYPE_DF – > SUBTYPE_INDCTN RANGE <>

_X_INDEX_NAME1 – > , EXPRESSION _X_INDEX_NAME1

_X_INDEX_NAME1 – > e

INDEXED_NAME – > IDENTIFIER (EXPRESSION _X_INDEX_NAME1)

_X_INSTT_LIST2 – > , IDENTIFIER _X_INSTT_LIST2

_X_INSTT_LIST2 – > e

_X_INSTT_LIST1 – > IDENTIFIER _X_INSTT_LIST2

INSTT_LIST – > _X_INSTT_LIST1

INSTT_LIST – > OTHERS

INSTT_LIST – > ALL

_X_INTFC_CNST_DECL1 – > := EXPRESSION

_X_INTFC_CNST_DECL1 – > e

```
INTFC_DECL                  -> INTFC_TYPE IDENTIFIER_LIST :
                               MODE SUBTYPE_INDCTN [ BUS ]
                               _X_INTFC_CNST_DECL1
INTFC_TYPE                  -> CONSTANT
INTFC_TYPE                  -> SIGNAL
INTFC_TYPE                  -> VARIABLE
INTFC_TYPE                  -> e
INTFC_ELEMENT               -> INTFC_DECL
_X_INTFC_LIST1              -> ; INTFC_ELEMENT _X_INTFC_LIST1
_X_INTFC_LIST1              -> e
INTFC_LIST                  -> INTFC_ELEMENT _X_INTFC_LIST1
_X_ITERATION_SCHEME1        -> WHILE EXPRESSION
_X_ITERATION_SCHEME2        -> FOR PARM_SPEC
ITERATION_SCHEME            -> _X_ITERATION_SCHEME1
ITERATION_SCHEME            -> _X_ITERATION_SCHEME2
ITERATION_SCHEME            -> e
LIBRARY_CLAUSE              -> LIBRARY LGCL_NAME_LIST ;
LIBRARY_UNIT                -> ARCH_BODY
LIBRARY_UNIT                -> CONF_DECL
LIBRARY_UNIT                -> ENTITY_DECL
LIBRARY_UNIT                -> PKG__1
_X_LGCL_NAME_LIST1          -> , IDENTIFIER _X_LGCL_NAME_LIST1
_X_LGCL_NAME_LIST1          -> e
LGCL_NAME_LIST              -> IDENTIFIER _X_LGCL_NAME_LIST1
LOOP__1                     -> ITERATION_SCHEME LOOP
                               SQN_OF_STMTS END LOOP [
                               IDENTIFIER ] ;
MODE                        -> IN
MODE                        -> OUT
MODE                        -> INOUT
MODE                        -> BUFFER
MODE                        -> LINKAGE
MODE                        -> e
NAME_A2                     -> SLCTD_NAME
NAME_A2                     -> IDENTIFIER
NAME_A2                     -> SLICE_NAME
NAME_A2                     -> INDEXED_NAME
NAME_A2                     -> IDENTIFIER
NAME_A2                     -> RANGE
NAME_A1                     -> SLCTD_NAME
NAME_A1                     -> IDENTIFIER
```

NAME_A1	->	SLICE_NAME
NAME_A1	->	INDEXED_NAME
NAME_S2	->	INDEXED_NAME
NAME_S2	->	IDENTIFIER
NAME_S2	->	CHARACTER_LITERAL
NAME_S2	->	ALL
NAME_S1	->	IDENTIFIER
NAME_S1	->	INDEXED_NAME
NAME	->	SLCTD_NAME
NAME	->	ATTRIBUTE_NAME
NAME	->	SLICE_NAME
NAME	->	INDEXED_NAME
NAME	->	IDENTIFIER
_X_NEXT_STMT1	->	WHEN EXPRESSION
_X_NEXT_STMT1	->	e
NEXT_STMT	->	NEXT [IDENTIFIER] _X_NEXT_STMT1
NULL_STMT	->	NULL
NUMERIC_LITERAL	->	PHYS_LITERAL
OPTIONS	->	[GUARDED] [TRANSPORT]
OPTIONS	->	e
PKG__1	->	PACKAGE PKG__2
PKG__2	->	PKG__3
PKG__2	->	PKG__4
PKG__3	->	IDENTIFIER IS PKG_DECL_PART END [IDENTIFIER] ;
PKG__4	->	BODY IDENTIFIER IS PKG_BODY_DECL_PART END [IDENTIFIER] ;
PKG_BODY	->	PACKAGE BODY IDENTIFIER IS PKG_BODY_DECL_PART END [IDENTIFIER] ;
PKG_BODY_DECL_ITEM	->	SPRGM__1
PKG_BODY_DECL_ITEM	->	TYPE_DECL
PKG_BODY_DECL_ITEM	->	SUBTYPE_DECL
PKG_BODY_DECL_ITEM	->	CNST_DECL
PKG_BODY_DECL_ITEM	->	IDENTIFIER
PKG_BODY_DECL_ITEM	->	FILE_DECL
PKG_BODY_DECL_ITEM	->	ALIAS_DECL
PKG_BODY_DECL_ITEM	->	IDENTIFIER
PKG_BODY_DECL_PART	->	PKG_BODY_DECL_ITEM PKG_BODY_DECL_PART

PKG_BODY_DECL_PART	->	PKG_BODY_DECL_ITEM PKG_BODY_DECL_PART
PKG_BODY_DECL_PART	->	e
PKG_DECL	->	PACKAGE IDENTIFIER IS PKG_DECL_PART END [IDENTIFIER] ;
PKG_DECL_ITEM	->	SPRGM_1
PKG_DECL_ITEM	->	TYPE_DECL
PKG_DECL_ITEM	->	SUBTYPE_DECL
PKG_DECL_ITEM	->	CNST_DECL
PKG_DECL_ITEM	->	SIGNAL_DECL
PKG_DECL_ITEM	->	FILE_DECL
PKG_DECL_ITEM	->	ALIAS_DECL
PKG_DECL_ITEM	->	COMP_DECL
PKG_DECL_ITEM	->	ATTRIBUTE_1
PKG_DECL_ITEM	->	IDENTIFIER
PKG_DECL_ITEM	->	DISCONNECTION_SPEC
PKG_DECL_ITEM	->	USE_CLAUSE
ENTITY_DECL_ITEM	->	SPRGM_1
ENTITY_DECL_ITEM	->	TYPE_DECL
ENTITY_DECL_ITEM	->	SUBTYPE_DECL
ENTITY_DECL_ITEM	->	CNST_DECL
ENTITY_DECL_ITEM	->	SIGNAL_DECL
ENTITY_DECL_ITEM	->	FILE_DECL
ENTITY_DECL_ITEM	->	ALIAS_DECL
ENTITY_DECL_ITEM	->	IDENTIFIER
ENTITY_DECL_ITEM	->	ATTRIBUTE_1
ENTITY_DECL_ITEM	->	IDENTIFIER
ENTITY_DECL_ITEM	->	DISCONNECTION_SPEC
ENTITY_DECL_ITEM	->	USE_CLAUSE
BLOCK_DECL_ITEM	->	SPRGM_1
BLOCK_DECL_ITEM	->	TYPE_DECL
BLOCK_DECL_ITEM	->	SUBTYPE_DECL
BLOCK_DECL_ITEM	->	CNST_DECL
BLOCK_DECL_ITEM	->	SIGNAL_DECL
BLOCK_DECL_ITEM	->	FILE_DECL
BLOCK_DECL_ITEM	->	ALIAS_DECL
BLOCK_DECL_ITEM	->	COMP_DECL
BLOCK_DECL_ITEM	->	ATTRIBUTE_1
BLOCK_DECL_ITEM	->	CONF_SPEC
BLOCK_DECL_ITEM	->	DISCONNECTION_SPEC
BLOCK_DECL_ITEM	->	USE_CLAUSE

PKG_DECL_PART	->	PKG_DECL_ITEM PKG_DECL_PART
PKG_DECL_PART	->	e
PARM_SPEC	->	IDENTIFIER IN DISCRETE_RANGE
PHYS_LITERAL	->	ABSTRACT_LITERAL [IDENTIFIER]
PHYS_LITERAL	->	e
_X_PHYS_TYPE_DF1	->	SECONDARY_UNIT_DECL _X_PHYS_TYPE_DF1
_X_PHYS_TYPE_DF1	->	e
PORT_CLAUSE	->	PORT (PORT_LIST) ;
PORT_CLAUSE	->	e
PORT_CLAUSE_Y_1	->	PORT (PORT_LIST)
PORT_CLAUSE_Y_1	->	e
PORT_LIST	->	INTFC_LIST
PORT_MAP_ASPECT	->	PORT MAP (ASSC_LIST)
PORT_MAP_ASPECT	->	e
PRIMARY	->	NAME
PRIMARY	->	PHYS_LITERAL
PRIMARY	->	BIT_STRING_LITERAL
PRIMARY	->	NULL
PRIMARY	->	ALLOCATOR
PRIMARY	->	PARENTHESIS__1
PRIMARY	->	CHARACTER_LITERAL
PRIMARY	->	STRING_LITERAL
PARENTHESIS__1	->	PARENTHESIS__2
PARENTHESIS__1	->	AGGREGATE
PARENTHESIS__2	->	(EXPRESSION)
_X_PROC_CALL_STMT1	->	(ACTUAL_PARM_PART)
_X_PROC_CALL_STMT1	->	e
PROC_CALL_STMT	->	IDENTIFIER _X_PROC_CALL_STMT1 ;
PROCESS_DECL_ITEM	->	SPRGM__1
PROCESS_DECL_ITEM	->	TYPE_DECL
PROCESS_DECL_ITEM	->	SUBTYPE_DECL
PROCESS_DECL_ITEM	->	CNST_DECL
PROCESS_DECL_ITEM	->	IDENTIFIER
PROCESS_DECL_ITEM	->	FILE_DECL
PROCESS_DECL_ITEM	->	ALIAS_DECL
PROCESS_DECL_ITEM	->	IDENTIFIER
PROCESS_DECL_ITEM	->	ATTRIBUTE__1
PROCESS_DECL_ITEM	->	USE_CLAUSE
PROCESS_DECL_ITEM	->	VARIABLE_DECL

PROCESS_DECL_PART	- >	PROCESS_DECL_ITEM PROCESS_DECL_PART
PROCESS_DECL_PART	- >	e
PROCESS_STMT	- >	PROCESS _X_PROCESS_STMT1 PROCESS_DECL_PART BEGIN PROCESS_STMT_PART END PROCESS [IDENTIFIER] ;
_X_PROCESS_STMT1	- >	(SENSITIVITY_LIST)
_X_PROCESS_STMT1	- >	e
PROCESS_STMT_PART	- >	SEQNTL_STMT PROCESS_STMT_PART
PROCESS_STMT_PART	- >	e
RANGE	- >	_RANGE1
RANGE	- >	ATTRIBUTE_NAME
_RANGE1	- >	EXPRESSION DIRECTION EXPRESSION
RANGE_CONSRT	- >	RANGE RANGE
_X_RECORD_TYPE_DF1	- >	ELEMENT_DECL _X_RECORD_TYPE_DF1
_X_RECORD_TYPE_DF1	- >	e
RECORD_TYPE_DF	- >	RECORD ELEMENT_DECL _X_RECORD_TYPE_DF1 END RECORD
RETURN_STMT	- >	RETURN EXPRESSION ;
_X_SCALAR_TYPE_DF3	- >	SECONDARY_UNIT_DECL _X_SCALAR_TYPE_DF3
_X_SCALAR_TYPE_DF3	- >	e
_X_SCALAR_TYPE_DF2	- >	UNITS BASE_UNIT_DECL _X_SCALAR_TYPE_DF3 END UNITS
_X_SCALAR_TYPE_DF1	- >	RANGE_CONSRT _X_SCALAR_TYPE_DF2
SECONDARY_UNIT_DECL	- >	IDENTIFIER = PHYS_LITERAL ;
SLCTD_NAME	- >	NAME_S1 SLCTD_NAME_1
SLCTD_SIGNAL_ASSGN	- >	WITH EXPRESSION SELECT TARGET <= OPTIONS SLCTD_WVFMS ;
_X_SLCTD_WVFMS1	- >	, WVFM WHEN CHOICES _X_SLCTD_WVFMS1
_X_SLCTD_WVFMS1	- >	e
SLCTD_WVFMS	- >	WVFM WHEN CHOICES _X_SLCTD_WVFMS1
SENSITIVITY_CLAUSE	- >	ON SENSITIVITY_LIST
SENSITIVITY_CLAUSE	- >	e
_X_SENSITIVITY_LIST1	- >	, NAME _X_SENSITIVITY_LIST1
_X_SENSITIVITY_LIST1	- >	e
SENSITIVITY_LIST	- >	NAME _X_SENSITIVITY_LIST1
_X_SQN_OF_STMTS1	- >	SEQNTL_STMT _X_SQN_OF_STMTS1

```
_X_SQN_OF_STMTS1        -> e
SQN_OF_STMTS            -> SEQNTL_STMT _X_SQN_OF_STMTS1
_X_SEQNTL_STMT5         -> <= [ TRANSPORT ] WVFM ;
_X_SEQNTL_STMT4         -> := EXPRESSION ;
_X_SEQNTL_STMT3         -> : LOOP__1
_X_SEQNTL_STMT2         -> _X_SEQNTL_STMT3
_X_SEQNTL_STMT2         -> LOOP__1
_X_SEQNTL_STMT2         -> ;
_X_SEQNTL_STMT2X        -> _X_SEQNTL_STMT2
_X_SEQNTL_STMT2X        -> _X_SEQNTL_STMT4
_X_SEQNTL_STMT2X        -> _X_SEQNTL_STMT5
_X_SEQNTL_STMT2X        -> e
_X_SEQNTL_STMT1         -> NAME _X_SEQNTL_STMT2X
SEQNTL_STMT             -> WAIT_STMT
SEQNTL_STMT             -> ASSERTION_STMT
SEQNTL_STMT             -> _X_SEQNTL_STMT1
SEQNTL_STMT             -> IF_STMT
SEQNTL_STMT             -> CASE_STMT
SEQNTL_STMT             -> NEXT_STMT
SEQNTL_STMT             -> LOOP__1
SEQNTL_STMT             -> EXIT_STMT
SEQNTL_STMT             -> RETURN_STMT
SEQNTL_STMT             -> NULL_STMT
_X_SIGNAL_DECL1         -> := EXPRESSION
_X_SIGNAL_DECL1         -> e
SIGNAL_DECL             -> SIGNAL IDENTIFIER_LIST :
                           SUBTYPE_INDCTN SIGNAL_KIND
                           _X_SIGNAL_DECL1 ;
SIGNAL_KIND             -> REGISTER
SIGNAL_KIND             -> BUS
SIGNAL_KIND             -> e
_X_SIGNAL_LIST2         -> , NAME _X_SIGNAL_LIST2
_X_SIGNAL_LIST2         -> e
_X_SIGNAL_LIST1         -> NAME _X_SIGNAL_LIST2
SIGNAL_LIST             -> _X_SIGNAL_LIST1
SIGNAL_LIST             -> OTHERS
SIGNAL_LIST             -> ALL
SLICE_NAME              -> IDENTIFIER ( DISCRETE_RANGE )
SPRGM_DECL              -> SPRGM_SPEC ;
SPRGM_DECL_ITEM         -> SPRGM__1
SPRGM_DECL_ITEM         -> TYPE_DECL
```

SPRGM_DECL_ITEM	->	SUBTYPE_DECL
SPRGM_DECL_ITEM	->	CNST_DECL
SPRGM_DECL_ITEM	->	IDENTIFIER
SPRGM_DECL_ITEM	->	FILE_DECL
SPRGM_DECL_ITEM	->	ALIAS_DECL
SPRGM_DECL_ITEM	->	IDENTIFIER
SPRGM_DECL_ITEM	->	ATTRIBUTE__1
SPRGM_DECL_ITEM	->	USE_CLAUSE
SPRGM_DECL_ITEM	->	VARIABLE_DECL
SPRGM_DECL_PART	->	SPRGM_DECL_ITEM SPRGM_DECL_PART
SPRGM_DECL_PART	->	e
_X_SPRGM_SPEC3	->	(FORMAL_PARM_LIST)
_X_SPRGM_SPEC3	->	e
_X_SPRGM_SPEC1	->	PROCEDURE DESGNTR _X_SPRGM_SPEC3
_X_SPRGM_SPEC2	->	FUNCTION DESGNTR _X_SPRGM_SPEC3 RETURN NAME
SPRGM_SPEC	->	_X_SPRGM_SPEC1
SPRGM_SPEC	->	_X_SPRGM_SPEC2
SPRGM_STMT_PART	->	SEQNTL_STMT SPRGM_STMT_PART
SPRGM_STMT_PART	->	SEQNTL_STMT SPRGM_STMT_PART
SPRGM_STMT_PART	->	e
SUBTYPE_DECL	->	SUBTYPE IDENTIFIER IS SUBTYPE_INDCTN ;
_X_SPRGM__1	->	IS SPRGM_DECL_PART BEGIN SPRGM_STMT_PART END [IDENTIFIER]
_X_SPRGM__1	->	e
SPRGM__1	->	SPRGM_SPEC _X_SPRGM__1 ;
SUBTYPE_INDCTN_1	->	IDENTIFIER IDENTIFIER CONSRT
SUBTYPE_INDCTN_2	->	IDENTIFIER CONSRT
SUBTYPE_INDCTN	->	SUBTYPE_INDCTN_1
SUBTYPE_INDCTN	->	SUBTYPE_INDCTN_2
TARGET	->	NAME
TARGET	->	AGGREGATE
TIMEOUT_CLAUSE	->	FOR EXPRESSION
TIMEOUT_CLAUSE	->	e
_X_TYPE_DECL1	->	IS TYPE_DF
_X_TYPE_DECL1	->	e
TYPE_DECL	->	TYPE IDENTIFIER _X_TYPE_DECL1 ;
TYPE_DF	->	ENMRT_TYPE_DF

TYPE_DF	->	_Y_SCALAR_TYPE_DF1
TYPE_DF	->	ARRAY_TYPE_DF
TYPE_DF	->	RECORD_TYPE_DF
TYPE_DF	->	ACCESS_TYPE_DF
TYPE_DF	->	FILE_TYPE_DF
_X_UCN_ARRAY_DF1	->	, INDEX_SUBTYPE_DF _X_UCN_ARRAY_DF1
_X_UCN_ARRAY_DF1	->	e
_X_USE_CLAUSE2	->	, NAME _X_USE_CLAUSE2
_X_USE_CLAUSE2	->	e
USE_CLAUSE	->	USE NAME _X_USE_CLAUSE2 ;
_X_VARIABLE_DECL1	->	:= EXPRESSION
_X_VARIABLE_DECL1	->	e
VARIABLE_DECL	->	VARIABLE IDENTIFIER_LIST : SUBTYPE_INDCTN _X_VARIABLE_DECL1;
WAIT_STMT	->	WAIT SENSITIVITY_CLAUSE CONDITION_CLAUSE TIMEOUT_CLAUSE;
_X_WVFM1	->	, WVFM_ELEMENT _X_WVFM1
_X_WVFM1	->	e
WVFM	->	WVFM_ELEMENT _X_WVFM1
_X_WVFM_ELEMENT1	->	AFTER EXPRESSION
_X_WVFM_ELEMENT1	->	e
WVFM_ELEMENT	->	EXPRESSION _X_WVFM_ELEMENT1
SLCTD_NAME_1	->	. NAME_S2 SLCTD_NAME_1
SLCTD_NAME_1	->	e
_Y_SCALAR_TYPE_DF1	->	_X_SCALAR_TYPE_DF1
_Y_SCALAR_TYPE_DF1	->	RANGE_CONSRT

Appendix B

Predefined Packages

This appendix lists the predefined packages and available packages for use with this program.

B.1 STANDARD PACKAGE

```
--
-- This is the predefined standard package
-- This is defind in the  IEEE Standard VHDL
-- Language Reference Manual
--

package STANDARD is

   -- predefined enumeration types

   type BOOLEAN is (FALSE,TRUE);

   type BIT is ('0','1');

   type CHARACTER is (
      NULL,   SOH,   STX,   ETX,   EOT,   ENQ,   ACK,   BEL,
      BS,     HT,    LF,    VT,    FF,    CR,    SO,    SI,
      DLE,    DC1,   DC2,   DC3,   DC4,   NAK,   SYN,   ETB,
      CAN,    EM,    SUB,   ESC,   FSP,   GSP,   RSP,   USP,

      ' ',    '!',   '"',   '#',   '$',   '%',   '&',   ''',
      '(',    ')',   '*',   '+',   ',',   '-',   '.',   '/',
```

```
    '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',
    '8',    '9',    ':',    ';',    '<',    '=',    '>',    '?',

    '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
    'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
    'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
    'X',    'Y',    'Z',    '[',    '\',    ']',    '^',    '_',

    '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',
    'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',
    'p',    'q',    'r',    's',    't',    'u',    'v',    'w',
    'x',    'y',    'z',    '{',    '|',    '}',    '~',    DEL);

type SEVERITY_LEVEL is (NOTE,WARNING,ERROR,FAILURE);

-- predefined numeric types

type INTEGER is range  -2147483647 to 2147483647;

type real is range -1E38 to 1E38;

--predefined type TIME

type TIME is range -2147483647 to 2147483647
   units
      fs;                     -- femtosecond
      ps     = 1000 fs;       -- picosecond
      ns     = 1000 ps;       -- nanosecond
      us     = 1000 ns;       -- microsecond
      ms     = 1000 us;       -- millisecond
      sec    = 1000 ms;       -- second
      min    = 60 sec;        -- minute
      hr     = 60 min;        -- hour
   end units;

-- function to return the current simulation time

function NOW return TIME;

-- predefined numeric subtypes;
```

```
   subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;

   subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;

   -- predefined array type

   type STRING is array (POSITIVE range <>) of CHARACTER;

   type BIT_VECTOR is array (NATURAL range <>) of BIT;

end STANDARD;
```

B.2 COMPONENTS

```
--
-- PACKAGE Components is used to declare structures needed
-- for producing EDIF output compatible with Cadence 2.1
-- This is assuming EDIF 2 0 0
--

package components is

   -- define the maximun number of ports allowed on gate

   constant CDS_MAX_PORTS : INTEGER := 8;

   -- cds_port_type define port purpose

   type cds_port_type is (POWER,GROUND,CDS_SIGNAL,DUPLICATE);

   -- cds_port_direction defines port directions

   type cds_port_direction is (CDS_INPUT,CDS_OUTPUT,CDS_INOUT);

   -- cds_port is a structure containing port info

   type cds_port is record
      vhdl_name : string(1 to 32);     -- port name in comp decl
      cds_name  : string(1 to 32);     -- Cadence sc name
```

```
      port_type : cds_port_type;
      port_dir  : cds_port_direction;
   end record;

   -- cds_ports define info for standard cell

   type cds_ports is array(integer range <>) of cds_port;
   type cds_struct is record
      num_ports : integer;
      cell_name : string(1 to 32);
      ports     : cds_ports(1 to CDS_MAX_PORTS);
   end record;

   -- The following define structures for signal information

   type cds_signal_type is  (CDS_SIGNAL,CDS_GROUND,CDS_POWER);

   type CDS_SIGNAL_STRUCT is record
      global      : boolean;          -- global signal ?
      signal_type : cds_signal_type;
      cds_name    : string(1 to 32);  -- name to use
   end record;

   -- define attributes for signals and components

   attribute CDS_INFO : cds_struct;
   attribute CDS_SIGNAL_INFO : CDS_SIGNAL_STRUCT;

   --  define standard cell information for University of Manitoba
   --  CMOS3DLM Standard cell library.

   component inv port
      (
      input : inout Bit;
      output : out Bit
      );
   attribute CDS_INFO of inv:component is
      (4,"inv                              ",
       (("INPUT                            ",
         "In                               ",
         CDS_SIGNAL,
```

```
        CDS_INPUT),
       ("OUTPUT                        ",
        "Out                           ",
        CDS_SIGNAL,
        CDS_OUTPUT),
       ("VDD                           ",
        "vdd!                          ",
        POWER,
        CDS_OUTPUT),
       ("GND                           ",
        "gnd!                          ",
        GROUND,
        CDS_OUTPUT)));

component nand2 port
   (
   Ain     : in Bit;
   Bin     : in Bit;
   Output  : out Bit
   );
attribute CDS_INFO of nand2:component is
   (5,"inv                             ",
    (("Ain                             ",
      "A                               ",
      CDS_SIGNAL,
      CDS_INPUT),
    ("Bin                             ",
      "B                               ",
      CDS_SIGNAL,
      CDS_INPUT),
     ("OUTPUT                          ",
      "Out                             ",
      CDS_SIGNAL,
      CDS_OUTPUT),
     ("VDD                             ",
      "vdd!                            ",
      POWER,
      CDS_OUTPUT),
     ("GND                             ",
      "gnd!                            ",
      GROUND,
```

```
        CDS_OUTPUT)));

  component fadd port
     (
     Ain,Bin,Cin : in Bit;
     Sum,Cout    : out Bit
     );
  attribute CDS_INFO of fadd:component is
     (7,"fadd                          ",
      (("Ain                           ",
        "Ain                           ",
        CDS_SIGNAL,
        CDS_INPUT),
      ("Bin                           ",
        "Bin                           ",
        CDS_SIGNAL,
        CDS_INPUT),
      ("Cin                           ",
        "Cin                           ",
        CDS_SIGNAL,
        CDS_INPUT),
       ("SUM                           ",
        "Sum                           ",
        CDS_SIGNAL,
        CDS_OUTPUT),
       ("Cout                          ",
        "Cout                          ",
        CDS_SIGNAL,
        CDS_OUTPUT),
       ("VDD                           ",
        "vdd!                          ",
        POWER,
        CDS_OUTPUT),
       ("GND                           ",
        "gnd!                          ",
        GROUND,
        CDS_OUTPUT)));

   component inpad port
     (
     Input   : inout Bit
```

```
    );
  attribute CDS_INFO of inpad:component is
    (3,"inpad                          ",
     (("VDD                            ",
       "vdd!                           ",
       POWER,
       CDS_INPUT),
      ("GND                            ",
       "gnd!                           ",
       GROUND,
       CDS_INPUT),
      ("Input                          ",
       "Input                          ",
       CDS_SIGNAL,
       CDS_INOUT)));

  component outpad port
    (
    Output  : inout Bit
    );
  attribute CDS_INFO of outpad:component is
    (3,"inpad                          ",
     (("VDD                            ",
       "vdd!                           ",
       POWER,
       CDS_INPUT),
      ("GND                            ",
       "gnd!                           ",
       GROUND,
       CDS_INPUT),
      ("Output                         ",
       "Output                         ",
       CDS_SIGNAL,
       CDS_INOUT)));

  component vddpad;
  attribute CDS_INFO of vddpad:component is
    (3,"vddpad                         ",
     (("VDD                            ",
       "vdd!                           ",
       POWER,
```

```
          CDS_INPUT),
        ("GND                         ",
          "gnd!                        ",
          GROUND,
          CDS_INPUT),
        ("VDDCore                     ",
          "vdd!                        ",
          CDS_SIGNAL,
          CDS_OUTPUT)));

   component gndpad;
   attribute CDS_INFO of gndpad:component is
       (3,"gndpad                       ",
        (("VDD                         ",
          "vdd!                        ",
          POWER,
          CDS_INPUT),
        ("GND                         ",
          "gnd!                        ",
          GROUND,
          CDS_INPUT),
        ("GNDCore                     ",
          "gnd!                        ",
          CDS_SIGNAL,
          CDS_OUTPUT)));

    -- define vdd and gnd global signals

    signal vdd : Bit;
    attribute CDS_SIGNAL_INFO of vdd:signal is
       (TRUE,CDS_POWER,"vdd!                         ");
    signal gnd : Bit;
    attribute CDS_SIGNAL_INFO of gnd:signal is
       (TRUE,CDS_GROUND,"gnd!                         ");

end components;
```

Appendix C

VIN Notation

This appendix describes both the binary and textual VIN.

C.1 Binary VIN

The binary VIN is the internal representation of parsed VHDL description. This section illustrates the structure of the major parts of the VIN. The pictures are not intended to show all the fields or information associated with an particular structure, but rather to show the general connectivity of information. The actual format of the internal VIN may be found in the VHDL analyzer source code file analyzer.h.

The illustrations in this appendix show structures are boxes with the structure name in the box. An optional number is sometimes shown with the name. This number represents the identification code stored with the structure. This is used for error detection during program operation, and to aid in debugging. Arrows represent pointers to structures. If the structure is in a list, typically more than one of the structure is shown.

Figure C.1 shows the top structure in the binary VIN which is internally referred to as ENTITY_UNIT. This structure points to the major components of the VIN. The Symbol_table is the symbol table containing all declared identifiers. This VIS_STRUCT is a structure containing visibility information for the current design. NSYNTAX_LIST contains the syntax tree used during parsing. PCODE_STRUCT is the elaboration instruction sequence produced by the analyzer. The PROCESS_STRUCT contains memory management

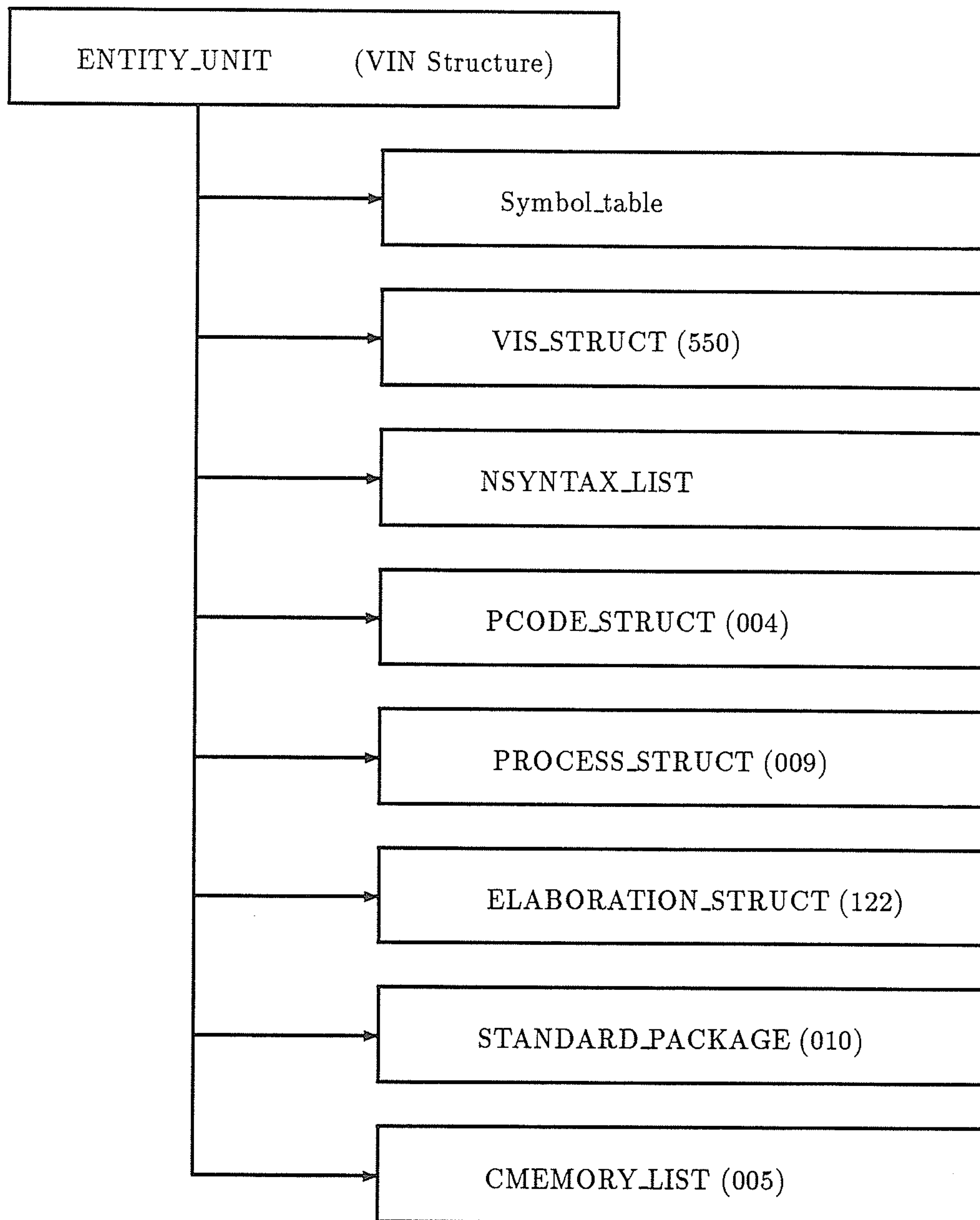


Figure C.1: VIN Structure

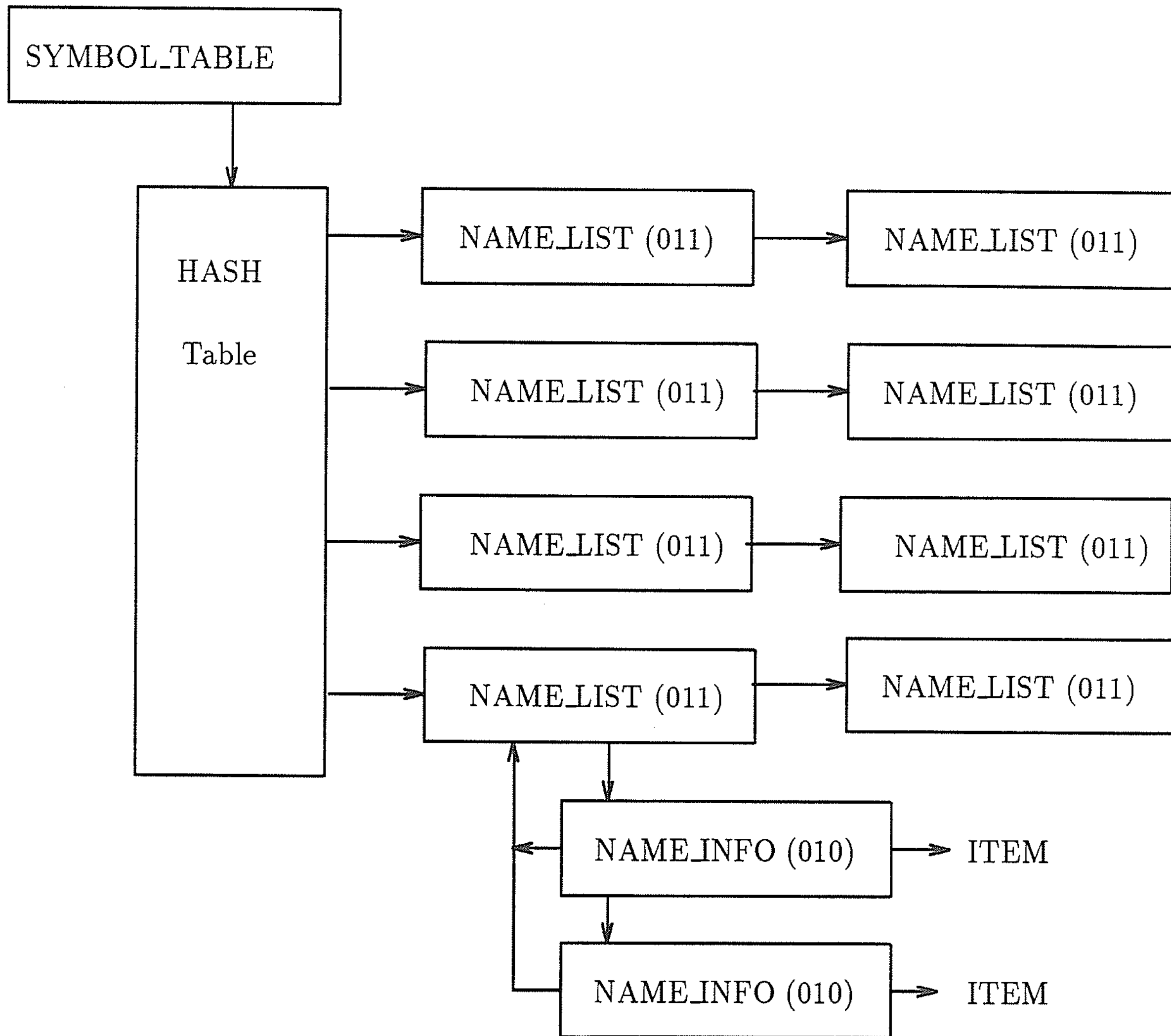


Figure C.2: Symbol Table

information for the various processes in the design. Elaboration struct is a structure containing information produced during elaboration. The standard_package contains pointers to symbol_table entries which make up the predefined package standard. CMEMORY_LIST contains memory management information used to store contant values parsed in the input VHDL description.

The symbol table shown in figure C.2 contains an entry for each named entity in the VHDL description. The hash table is used to increase search time. Each entry in the table is referred to as a NAME_LIST structure. This structure points to both the actual name, as well as a list of declared meaning for the entry. Each meaning is represented by an NAME_INFO structure which identifies the meaning, points the the appropiate information, and contains process information used by the visibility controller.

The memory management information is shown in figure C.3. Each VHDL design entity has an associated PROCESS_STRUCT. Each VHDL module which requires its own local memory has an asociated procesess information stored in the PROCESS_LIST structure. Each PROCESS_LIST has PROCESS_MEMORY structures associated with it which store the actual memory allocation information for the process. The DMEMORY_LIST is used to maintain dynamic memory allocation during elaboration and simulation.

The structure of stored VHDL expressions is shown in the two figures C.4 and C.5. The EXPR_STRUCT is used to maintain expression information during parsing. This is later converted to the EXPRESSION structure which is the format stored in the VIN. The expression contains a sequence of pseudo code instructions which when executed calculate the desired expression value. Each instruction also contains pointers to the instruction which generated its operands. This is used by the reverse analyzer to reproduce expressions.

Type informations is stored in structure TYPE as illustrated in figures C.6, C.7, and C.8. Type also contains pointers to the commonly used type attributes (not shown) such as RIGHT, LEFT, HIGH, and LOW, as well as others defined in the IEEE standard. The logical and physical size of the type are also stored as attributess, but are not directly user accessable. Scalar information is represented by structure SCALAR. This stores both the left and right extremes of the scalar range. Enumerated types are stored as a sequence of enumeration literals. Each literal has stored with it the symbol table entry for the literal, and its positional value. Physical types are stored as a linked list of unit specifications, each

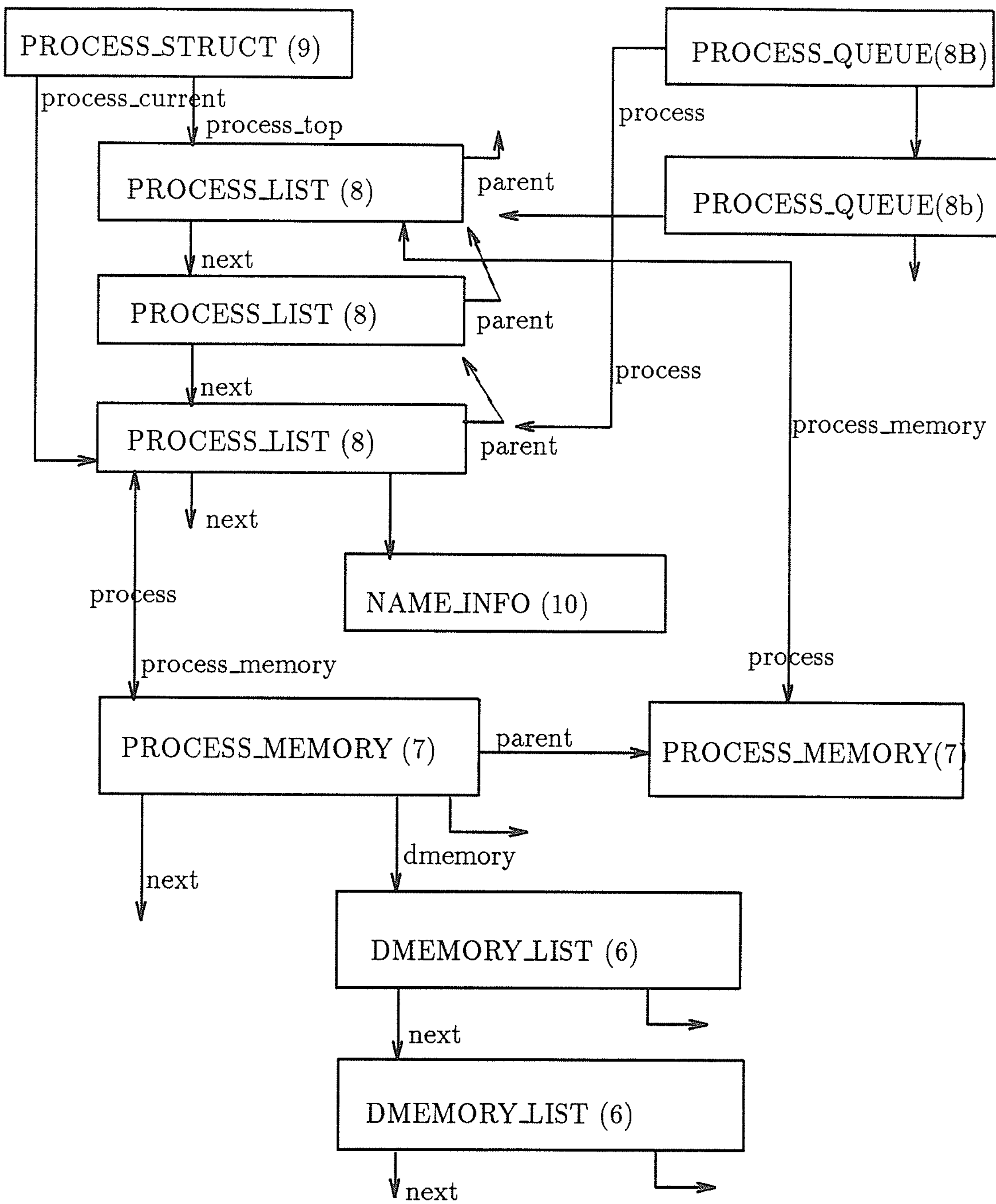


Figure C.3: VIN Memory Management

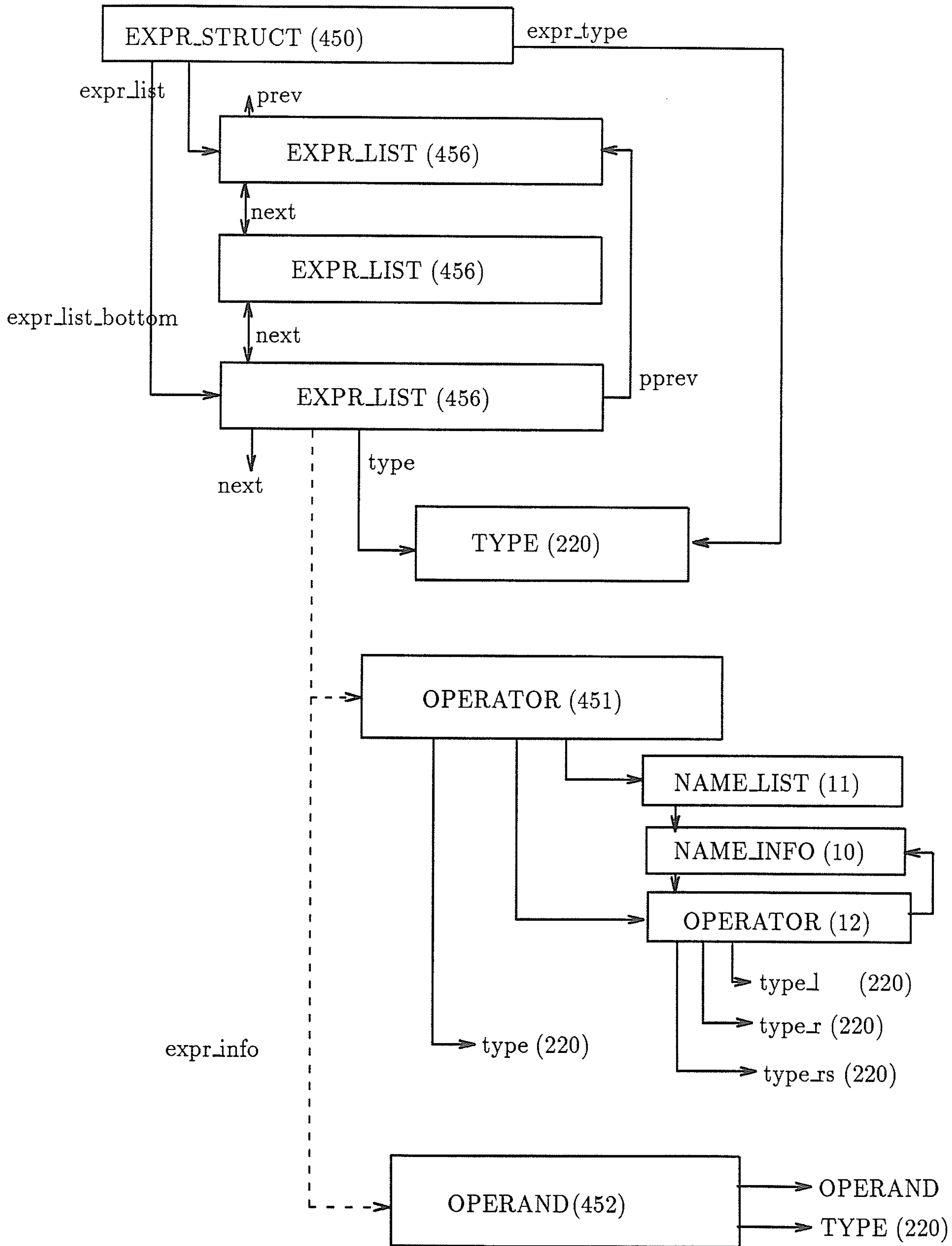


Figure C.4: VIN Expression

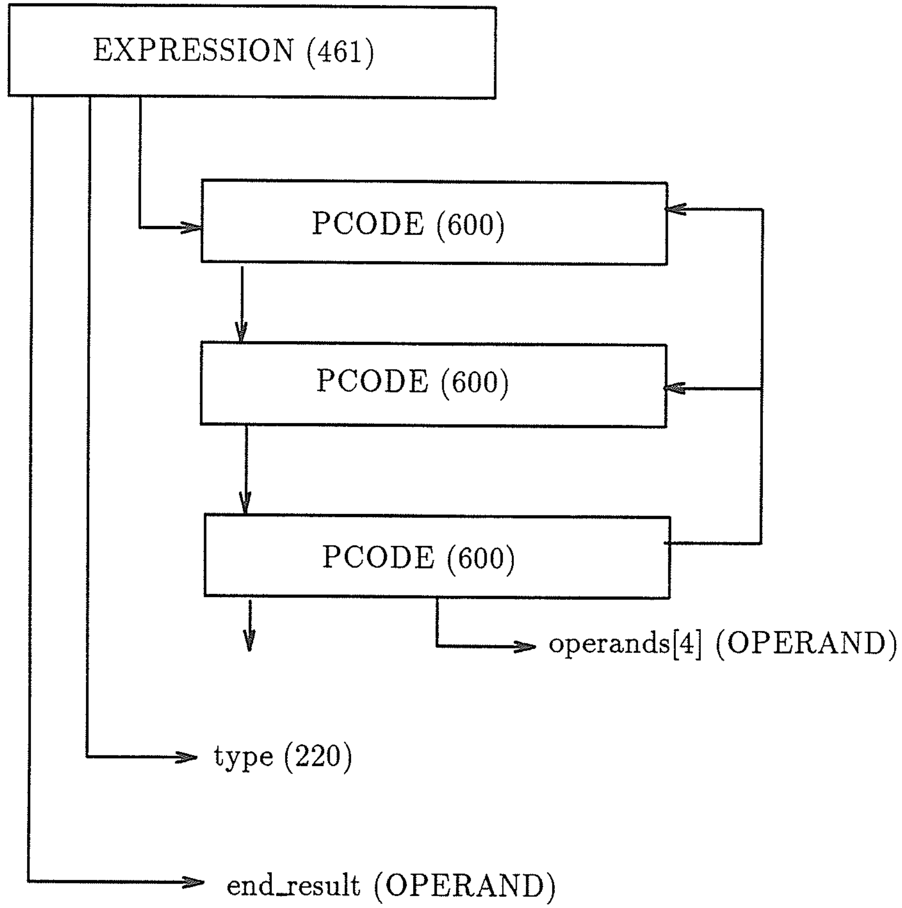


Figure C.5: VIN Expression

of which stores a factor and another unit as reference (except the base unit).

The Array type information is stored as a table of scalar type references each of which represent the range of the index. The logical and physical offset and dimension sizes are also stored for each array dimension. Record fields are stored as a linked list of field structures, each of which contains not only a pointer to field symbol table entry, but also the logical and physical offsets of each field.

Figure C.9 shows the structure used to represent a component.

Figure C.10 illustrates the format of attributes associated with each symbol table entry.

Figure C.11 shows the VIN representation of the structure used to represent name references. A name reference may be a simple signal reference, a indexed name reference, field reference or some combination of the above. The LDAE information is used primarily for the reverse analysis of VIN.

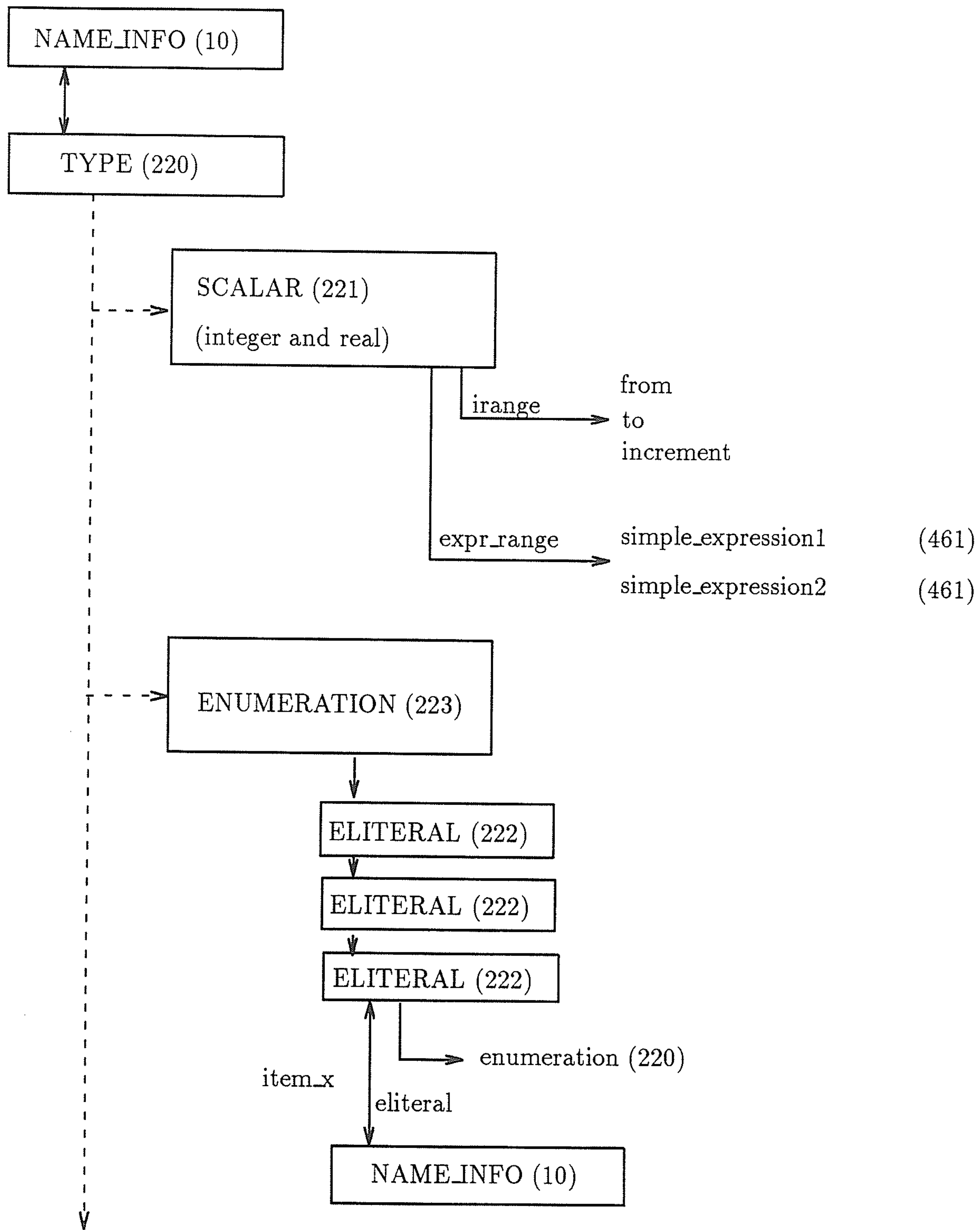


Figure C.6: VIN Types

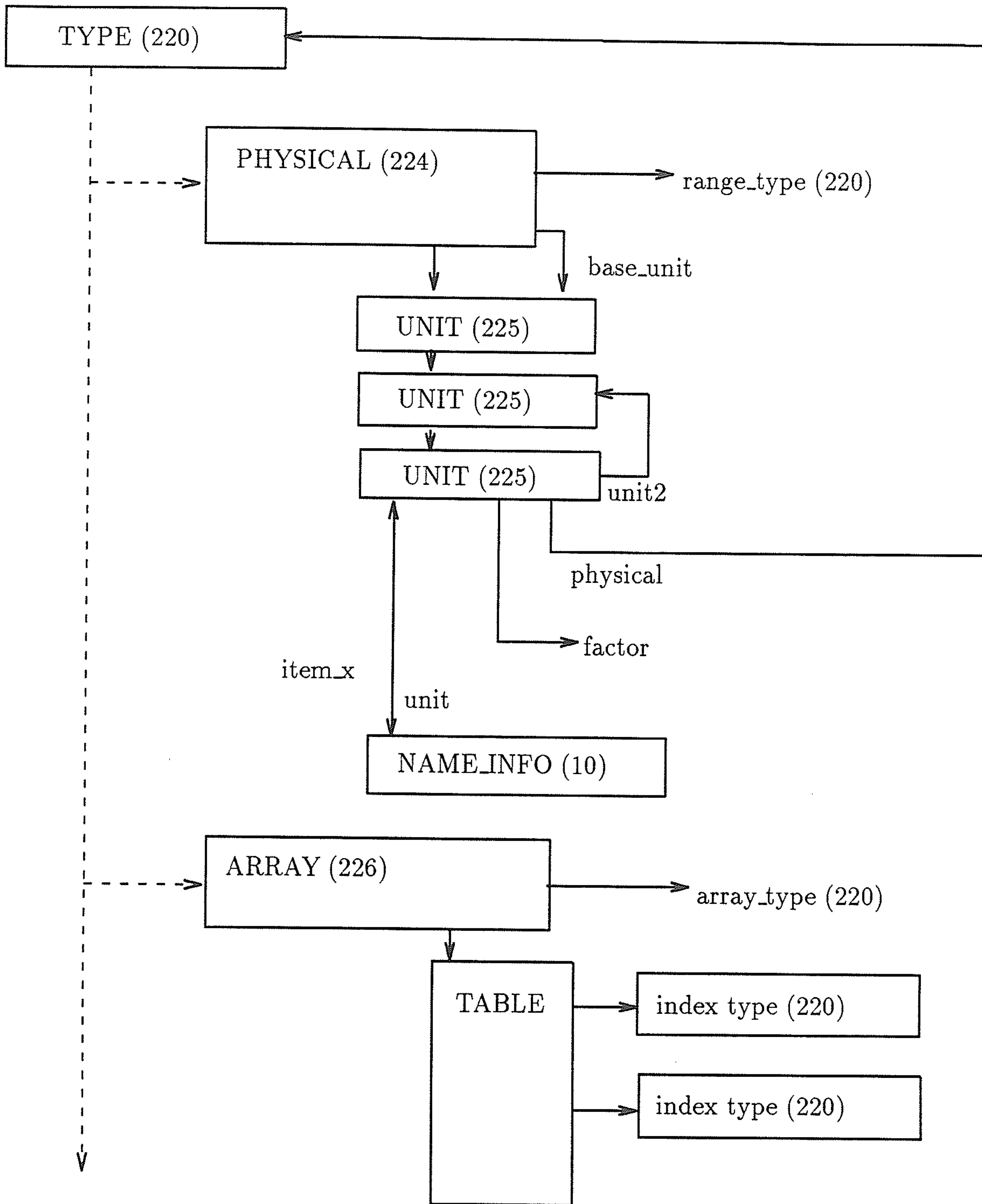


Figure C.7: VIN Types

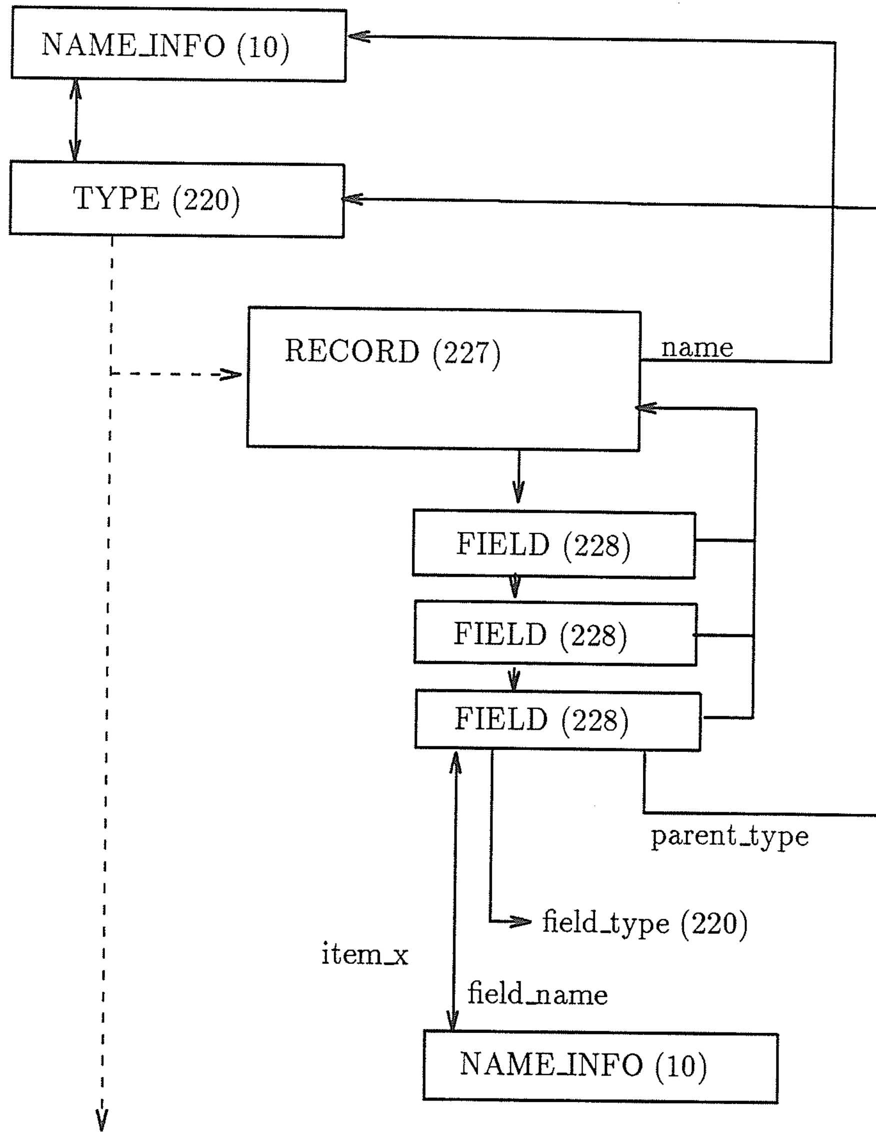


Figure C.8: VIN Types

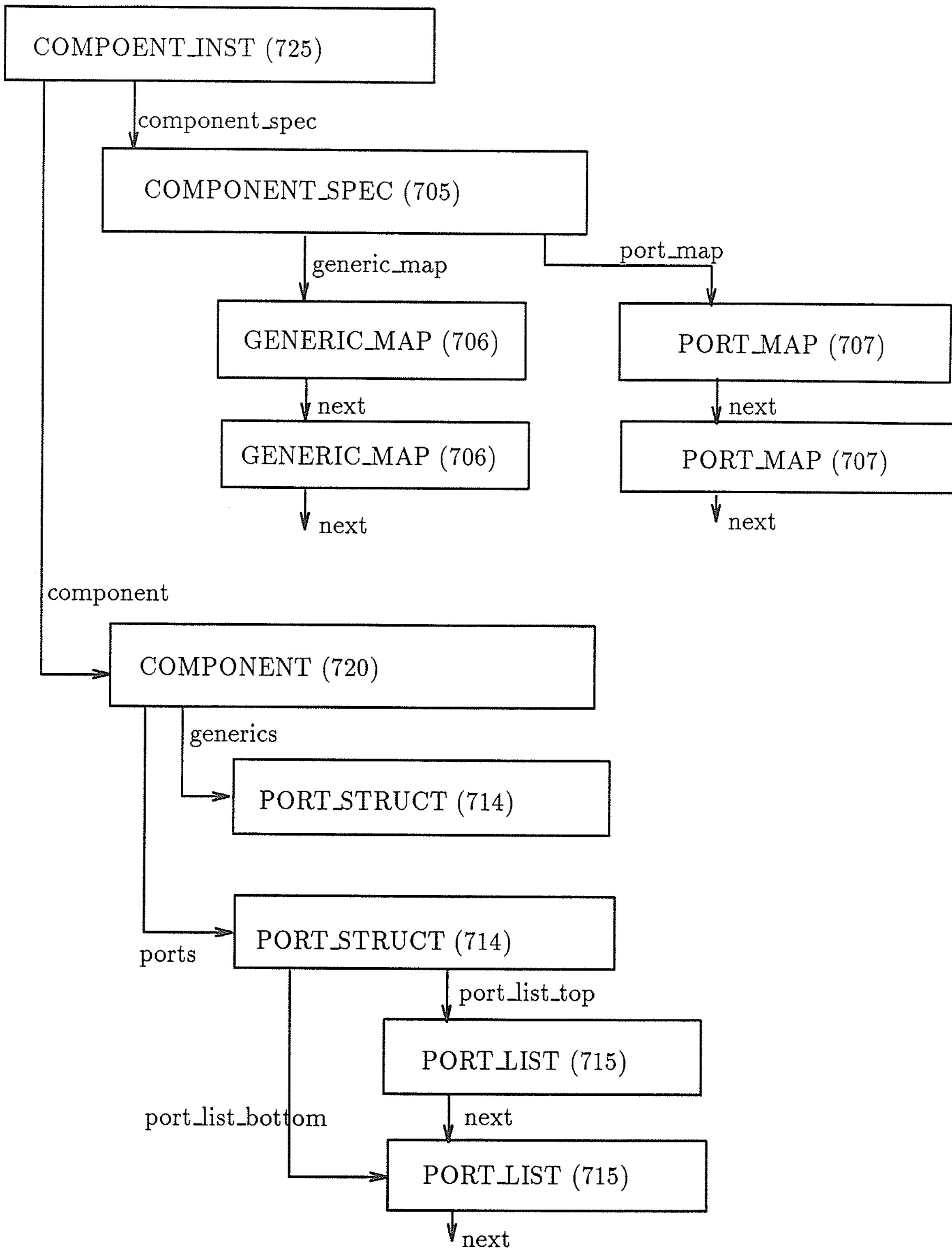


Figure C.9: VIN Component

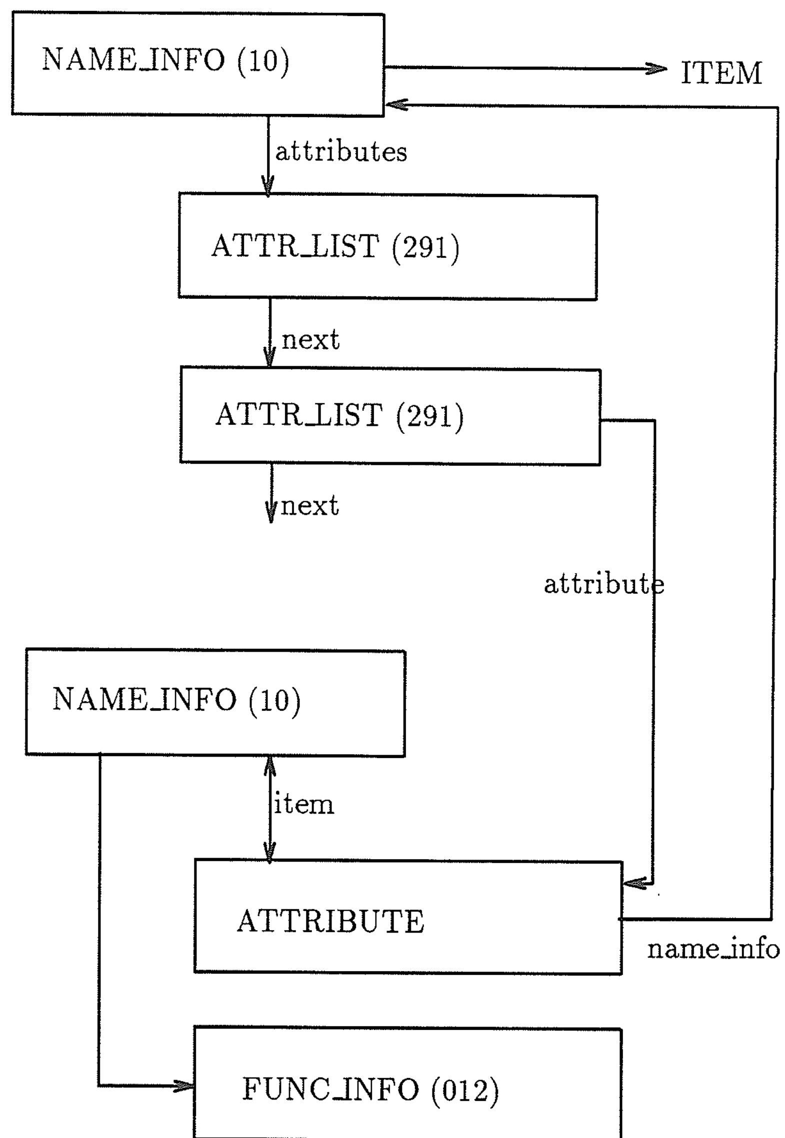


Figure C.10: VIN Attribute

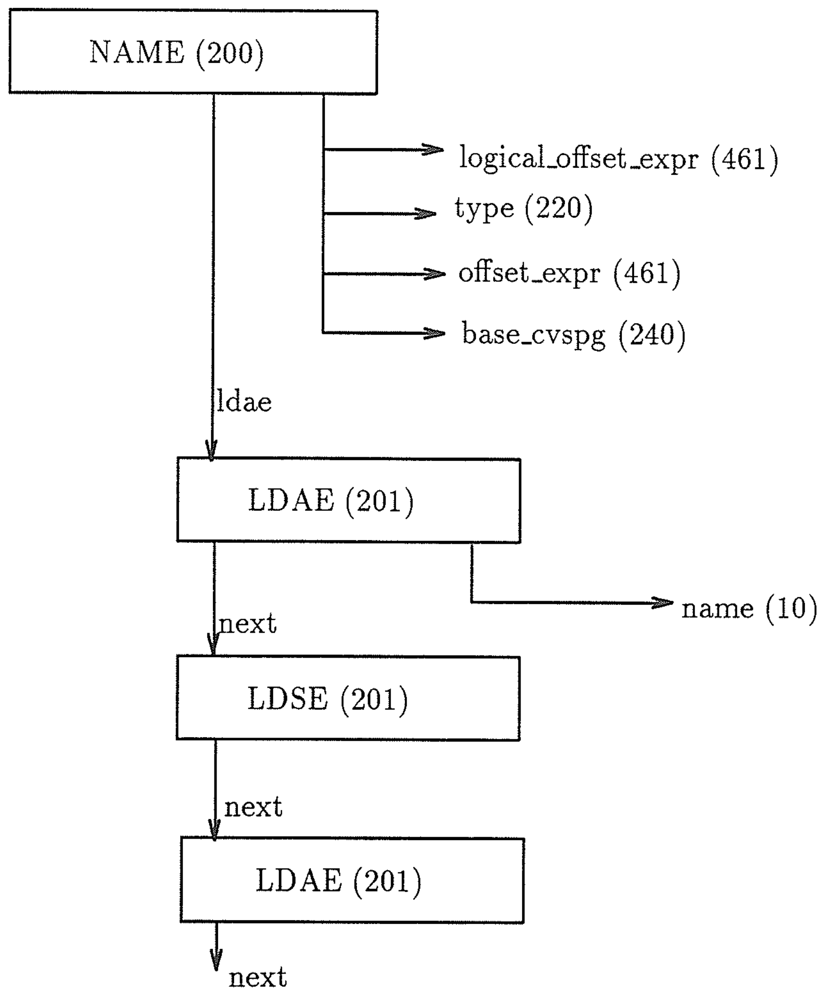


Figure C.11: VIN Attribute

C.2 Textual VIN

This section describes the textual VIN notation used to store parsed designs in a library. The format of the description is a modified Backus-Naur form, using "(" and ")" in place of "{" and "}" respectively to indicate repeated rules. Parenthesis are use in place of braces since braces are used to identify the begin and end of lists. Emphasized text represents NONTERMINALS, and bold text indicates reserved words. Reserved words follow immediately after the opening brace.

TOP → { **VIN** (*DESIGN_UNIT* })
DESIGN_UNIT → { **du** *COMMANDS* }
COMMANDS → { **commands** (*COMMAND* }
COMMAND → { **use** *NAME* }
COMMAND → { **PACKAGE** *SIMPLE_NAME COMMANDS* }
COMMAND → { **ENTITY** *SIMPLE_NAME PORT_STRUCT PORT_STRUCT COMMAND*
COMMAND → { **ARCHITECTURE** *SIMPLE_NAME COMMANDS* }
COMMAND → { **cvspg** *SIMPLE_NAME* integer *SUBTYPE_IND EXPR* }
COMMAND → *TYPE_DECL*
COMMAND → *ATTR_DECL*
COMMAND → *COMP_DECL*
COMMAND → *ATTR_SPEC*
COMMAND → *FOR*
COMMAND → *COM_INST*
COMMAND → *IF*
IF → { **ifs** *NAME_LIST EXPR COMMANDS* }
COM_INST → { **[illegible]_inst** *NAME_LIST NAME_INFO COMPS* }
COMPS → { **comps** *MAP MAP* }
MAP → { **map** (*PM*) }
PM → { **pm** integer *EXPR* }
FOR → { **fors** *NAME_LIST NAME_LIST EXPR* to — downto *EXPR COMMANDS* }
ATTR_SPEC → { **attr_spec** *NAME_INFO NAME_INFO EXPR* }
COMP_DECL → { **compd** *SIMPLE_NAME PORT_STRUCT PORT_STRUCT* }

PORT_STRUCT	→ { **portstruct** (*PORTLIST*) }
PORTLIST	→ { **portlist** *SIMPLE_NAME* integer integer *SUBTYPE_IND EXPR* }
ATTR_DECL	→ { **attr_decl** *SIMPLE_NAME SUBTYPE_IND* }
NAME	→ { **name** integer integer *TYPE CSA LDAES* }
SIMPLE_NAME	→ identifier (.identifier) }
TYPE	→ NULL
TYPE	→ identifier
CSA	→ { **CSA** (*TARGETX*) }
TARGETX	→ { **targetx** *NAME_LIST CSA_COND* }
CSA_COND	→ { **csa_cond** integer *NAME* integer *WAVEFORM* }
WAVEFORM	→ { **waveform** *SIMPLE_NAME* integer *EXPR PVALUE* }
PVALUE	→ { **pvalue** integer *NAME_INFO* }
LDAES	→ { **ldae** integer *SIMPLE_NAME CSA* [*indexes*] }
indexes	→ { **indexes** integer (*SUBTYPE_IND*) }
SUBTYPE_IND	→ NULL
SUBTYPE_IND	→ { **sub_ind** *SIMPLE_NAME* [*SUBTYPE_IND2*] }
SUBTYPE_IND2	→ *INDCON*
SUBTYPE_IND2	→ *RANGE*
EXPR	→ { **subexpr** (*OP_OR_VALUE*) }
EXPR	→ NULL
OP_OR_VALUE	→ *OP*
OP_OR_VALUE	→ *VALUE*
OP_OR_VALUE	→ *AGGREGATE*
OP_OR_VALUE	→ *NAME*
AGGREGATE	→ { **aggr** *SUBTYPE_IND* integer *aggrl* }
aggrl	→ { **aggrl** integer *SIMPLE_NAME EXPR* }
OP	→ { **op** *SIMPLE_NAME* integer }
VALUE	→ { **ivalue** integer }
VALUE	→ { **fvalue** real }
VALUE	→ { **svalue** real }
VALUE	→ { **cvalue** real }
VALUE	→ { **bsvalue** real }

TYPE_DECL → { **type** *SIMPLE_NAME DIFF_TYPES* }
DIFF_TYPES → { **enumeration** integer (*ELIT*) }
DIFF_TYPES → { **record** integer (*FIELD*) }
DIFF_TYPES → { **array** integer *SUBTYPE_IND* (*SUBTYPE_IND*) }
ELIT → { **elit** integer (*NAME_LIST*) }
NAME_LIST → { **nl** integer *SIMPLE_NAME* }
NAME_LIST → { **field** *SIMPLE_NAME SUBTYPE_IND* }
INDCON → { **indcon** integer *SUBTYPE_IND* (*SUBTYPE_IND*) }
RANGE → { **range** *EXPR* to | downto *EXPR* }

Appendix D

Design Example

This appendix gives a complete design example which demonstrates both the structural compiler and simulator. The example is of an n-bit ALU shown in figure D.1, which is slightly more complex than the design used in the lab guide in the next appendix.

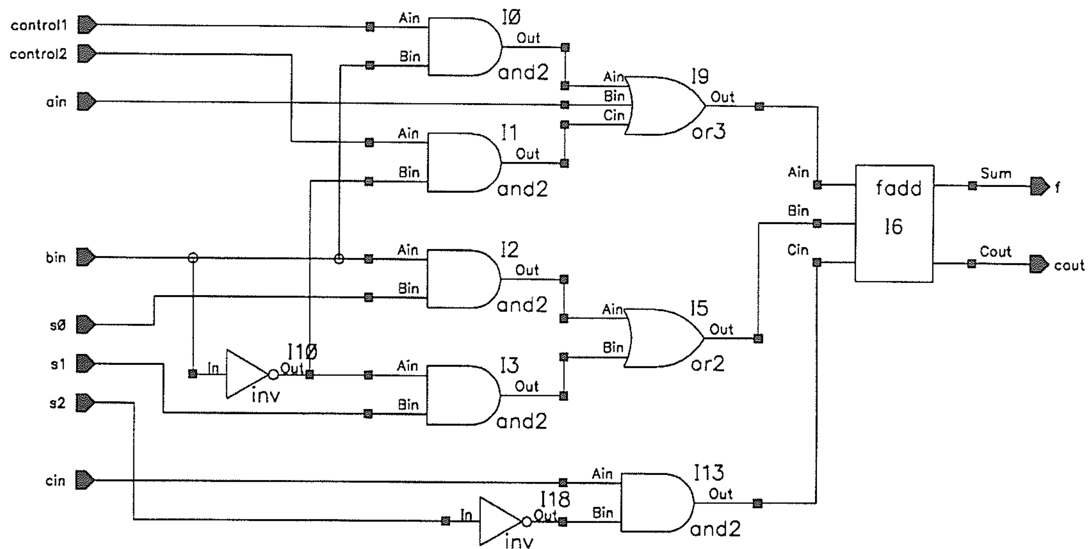


Figure D.1: ALU Schematic

D.1 VHDL Description

```
--
--        ALU example for c24.450 laboratory
--

entity alu_slice is
   port
      (
      Selection_lines : bit_vector(0 to 2);
      Control_lines   : bit_vector(0 to 1);
      A : bit;
      B : bit;
      Cin : bit;
      Output : bit;
      Cout : bit
      );
end alu_slice;

use components;
architecture alu_slice1 of alu_slice is
```

```
   signal b_not,s2_not          : bit;
   signal x1,x2,x3,x4,x6,x7,x8 : bit;
begin
   inv  port map (B,b_not);

   and2 port map (Control_lines(0),B,x1);
   and2 port map (Control_lines(1),b_not,x2);

   and2 port map (B,Selection_lines(0),x3);
   and2 port map (b_not,Selection_lines(1),x4);

   inv  port map (Selection_lines(2),s2_not);

   or3  port map (x1,A,x2,x6);
   or2  port map (x3,x4,x7);
   and2 port map (Cin,s2_not,x8);

   fadd port map (x6,x7,x8,Output,Cout);

end alu_slice1;

entity nbit_alu is
   generic
      (
      N : natural := 3
      );
   port
      (
      A : bit_vector(0 To N);
      B : bit_vector(0 To N);
      Output : bit_vector(0 to N);
      Select_lines : bit_vector( 0 to 2);
      Cin : bit;
      Cout : bit
      );
end nbit_alu;

use components;
architecture nbit_alu1 of nbit_alu is
   signal select_lines_not : bit_vector(0 to 2);
   signal control_lines    : bit_vector(0 to 1);
```

```
   signal cx,cy            : bit_vector(0 to N);
   component alu_slice1
      port
         (
 Selection_lines : bit_vector(0 to 2);
 Control_lines   : bit_vector(0 to 1);
         A : bit;
         B : bit;
         Cin : bit;
 Output : bit;
         Cout : bit
         );
begin
   --
   -- generate control lines
   --
   inpad port map(Select_lines(0));
   inpad port map(Select_lines(1));
   inpad port map(Select_lines(2));
   inpad port map(Cin);
   outpad port map(Cout);
   vddpad;
   gndpad;
   inv port map(Select_lines(0),select_lines_not(0));
   inv port map(Select_lines(1),select_lines_not(1));
   inv port map(Select_lines(2),select_lines_not(2));

   and3 port map(select_lines_not(0),select_lines_not(1),Select_lines(2),control_lines(
   and3 port map(select_lines_not(0),Select_lines(1),Select_lines(2),control_lines(1));

   and2 port map(Cin,select_lines_not(2),cx(0));

   for i in 0 To N generate
      inpad port map(A(i));
      inpad port map(B(i));
      outpad port map(Output(i));

      alu_slice1 port map(Select_lines,control_lines,A(i),B(i),cx(i),Output(i),cy(i));

      if (i = N) generate
         and2 port map(cy(i),select_lines_not(2),Cout);
```

```
      end generate;
      if (i /= N) generate
         and2 port map(cy(i),select_lines_not(2),cx(i+1));
      end generate;
   end generate;

end nbit_alu1;
```

D.2 Textual VIN

The length of the VIN design description precludes it's complete inclusion in this appendix, but a portion is included here to demonstrate the format of the notation. The complete VIN file is appropimately 17,000 lines long.

```
{COMMENT Example VIN for for ALU}
{COMMENT Shorten to show only important parts}
{COMMENT ONLY ALU_SLICE ENtity Decl is showm}
{
VIN
  {du #I32 { commands
      { use { name #I0 #I0 <<NULL>> { nptr }
 { ldae #I4096 STANDARD.STANDARD.du { nptr } } }
      }

      {
      ENTITY ALU_SLICE.ALU_SLICE.du  { portstruct }
        { portstruct
          { portlist SELECTION_LINES.ALU_SLICE.du #I8 #I1
            { sub_ind BIT_VECTOR.STANDARD.du #I0
              {
              sub_ind BIT.STANDARD.du
              }
              {
              indcon #I1
                {
                sub_ind BIT.STANDARD.du
                }
                {
                sub_ind universal_integer.STANDARD.du #I1
                  {
```

```
          range
            {
            expr
              {
              subexpr
                {
                op null_operator.STANDARD.du #I50
                }
                {
                ivalue #I0
                }
              }

            }
          to
            {
            expr
              {
              subexpr
                {
                op null_operator.STANDARD.du #I50
                }
                {
                ivalue #I2
    } } } } } } }

    { expr }
  }
  { portlist CONTROL_LINES.ALU_SLICE.du #I8 #I1
    { sub_ind BIT_VECTOR.STANDARD.du #I0
      {
      sub_ind BIT.STANDARD.du
      }
      {
      indcon #I1
        {
        sub_ind BIT.STANDARD.du
        }
        {
        sub_ind universal_integer.STANDARD.du #I1
          {
```

```
        range
          {
          expr
            {
            subexpr
              {
              op null_operator.STANDARD.du #I50
              }
              {
              ivalue #I0
              }
            }
          }
        to
          {
          expr
            {
            subexpr
              {
              op null_operator.STANDARD.du #I50
              }
              {
              ivalue #I1
  } } } } } } }
  {
  expr
  }
}
{
portlist A.ALU_SLICE.du #I8 #I1
  {
  sub_ind BIT.STANDARD.du
  }
  {
  expr
  }
}

{
portlist B.ALU_SLICE.du #I8 #I1
  {
```

```
            sub_ind BIT.STANDARD.du
            }
            {
            expr
            }
          }
          {
          portlist CIN.ALU_SLICE.du #I8 #I1
            {
            sub_ind BIT.STANDARD.du
            }
            {
            expr
            }
          }
          {
          portlist OUTPUT.ALU_SLICE.du #I8 #I1
            {
            sub_ind BIT.STANDARD.du
            }
            {
            expr
            }
          }
          {
          portlist COUT.ALU_SLICE.du #I8 #I1
            {
            sub_ind BIT.STANDARD.du
            }
            {
            expr
            }
          }
        }
        {
        commands
        }
      }
    }
  }
```

D.3 EDIF

Like the VIN description, the EDIF netlist produced is also to long for inclusion in this appendix. Again a sample of the file is shown to give an example of the format.

```
(
EDIF VHDL_design (EDIFversion 2 0 0) (EDIFlevel 0)
      (keywordmap (keywordLevel 0))
  (
  status
    (
    written
      (
      timeStamp 1990 24 7 19 30 27
      )
    )
  )

  (
  EXTERNAL SC_LIB (EDIFlevel 0) (technology (numberDefinition
( scale 1 (e 1 -9) (unit Distance))))
    (
    cell and3 ( cellType GENERIC )
      ( view abstract (viewType NETLIST)
        ( interface
          ( port ( Rename GND "gnd!" ) (direction input) )
          ( port ( Rename VDD "vdd!" ) (direction input) )
          ( port ( Rename OUTPUT "Out" ) (direction output) )
  ( port ( Rename CIN "Cin" ) (direction input) )
  ( port ( Rename BIN "Bin" ) (direction input) )
  ( port ( Rename AIN "Ain" ) (direction input) )
        )
      )
    (COMMENT ..... other cells delete  from this description .... )
    )
  )

  (
  library (rename DEFAULT_LIB ".") (EDIFlevel 0) (technology
      (numberDefinition
     ( scale 1 (e 1 -9) (unit Distance))))
```

```
(
cell NBIT_ALU1  ( cellType GENERIC )
  (
  view autoLayout (viewType NETLIST)
    (
    interface
    )
    (
    contents
      (
      instance AND377
        (
        viewref abstract
          (
          cellref and3
            (
            libraryRef SC_LIB
            )
          )
        )
      )

      (
      instance AND3
        (
        viewref abstract
          (
          cellref and3
            (
            libraryRef SC_LIB
            )
          )
        )
      )
      (COMMENT    ........ other cell instances deleted .....)
      (
      net ( Rename VDD "vdd!")
        ( Joined (
          ( portref VDD ( instanceref INPAD85 ) )
  ( portref VDD ( instanceref INPAD84 ) )
  ( COMMENT ... rest of VDD connects deleted ..)
```

```
          ( net B_2_
     ( Joined (
     ( portref INPUT ( instanceref INPAD66 ) )
             ( portref AIN  ( instanceref AND248 ) )
             (Property sigType (string "signal") (owner "Cadence" ))
          )

          (COMMENT rest of nets deleted ...)
) ) ) ) )
```

D.4 VHDL Netlist

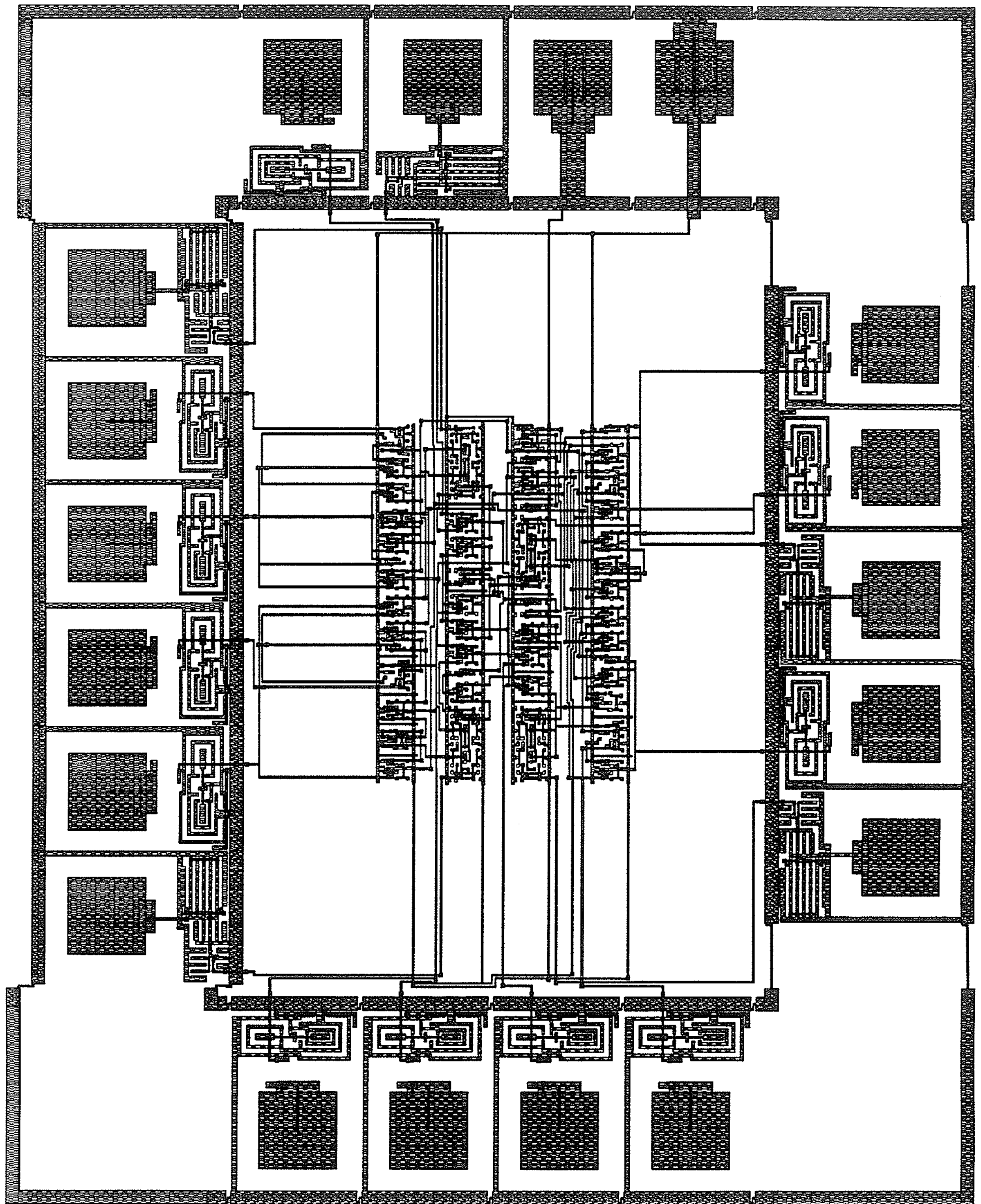
```
--
-- Simplified VHDL circuit netlist
--
entity NBIT_ALU is
   port(
   A : BIT_VECTOR(0 to 3) ;
   B : BIT_VECTOR(0 to 3) ;
   OUTPUT : BIT_VECTOR(0 to 3) ;
   SELECT_LINES : BIT_VECTOR(0 to 2) ;
   CIN : BIT ;
   COUT : BIT);
end NBIT_ALU

architecture NBIT_ALU1 of NBIT_ALU is
   SIGNAL X8 : BIT ;
   SIGNAL X7 : BIT ;
   SIGNAL X6 : BIT ;
   SIGNAL X4 : BIT ;
   SIGNAL X3 : BIT ;
   SIGNAL X2 : BIT ;
   SIGNAL X1 : BIT ;
   SIGNAL S2_NOT : BIT ;
   SIGNAL B_NOT : BIT ;
   SIGNAL X8__15_ : BIT ;
   SIGNAL X7__16_ : BIT ;
   SIGNAL X6__17_ : BIT ;
   SIGNAL X4__18_ : BIT ;
   SIGNAL X3__19_ : BIT ;
```

```
   SIGNAL X2__20_ : BIT ;
   SIGNAL X1__21_ : BIT ;
   SIGNAL S2_NOT__22_ : BIT ;
   SIGNAL B_NOT__23_ : BIT ;
   SIGNAL X8__24_ : BIT ;
   SIGNAL X7__25_ : BIT ;
   SIGNAL X6__26_ : BIT ;
   SIGNAL X4__27_ : BIT ;
   SIGNAL X3__28_ : BIT ;
   SIGNAL X2__29_ : BIT ;
   SIGNAL X1__30_ : BIT ;
   SIGNAL S2_NOT__31_ : BIT ;
   SIGNAL B_NOT__32_ : BIT ;
   SIGNAL X8__33_ : BIT ;
   SIGNAL X7__34_ : BIT ;
   SIGNAL X6__35_ : BIT ;
   SIGNAL X4__36_ : BIT ;
   SIGNAL X3__37_ : BIT ;
   SIGNAL X2__38_ : BIT ;
   SIGNAL X1__39_ : BIT ;
   SIGNAL S2_NOT__40_ : BIT ;
   SIGNAL B_NOT__41_ : BIT ;
   SIGNAL CY : BIT_VECTOR(0 to 3) ;
   SIGNAL CX : BIT_VECTOR(0 to 3) ;
   SIGNAL CONTROL_LINES : BIT_VECTOR(0 to 1) ;
   SIGNAL SELECT_LINES_NOT : BIT_VECTOR(0 to 2) ;
   SIGNAL GND : BIT ;
   SIGNAL VDD : BIT ;
BEGIN
   FADD port map (X6,X7,X8,OUTPUT(0),CY(0));
   AND2 port map (CX(0),S2_NOT,X8);
   OR2 port map (X3,X4,X7);
   OR3 port map (X1,A(0),X2,X6);
   INV port map (SELECT_LINES(2),S2_NOT);
   AND2 port map (B_NOT,SELECT_LINES(1),X4);
   AND2 port map (B(0),SELECT_LINES(0),X3);
   AND2 port map (CONTROL_LINES(1),B_NOT,X2);
   AND2 port map (CONTROL_LINES(0),B(0),X1);
   INV port map (B(0),B_NOT);
   FADD port map (X6__17_,X7__16_,X8__15_,OUTPUT(1),CY(1));
   AND2 port map (CX(1),S2_NOT__22_,X8__15_);
```

```
OR2 port map (X3__19_,X4__18_,X7__16_);
OR3 port map (X1__21_,A(1),X2__20_,X6__17_);
INV port map (SELECT_LINES(2),S2_NOT__22_);
AND2 port map (B_NOT__23_,SELECT_LINES(1),X4__18_);
AND2 port map (B(1),SELECT_LINES(0),X3__19_);
AND2 port map (CONTROL_LINES(1),B_NOT__23_,X2__20_);
AND2 port map (CONTROL_LINES(0),B(1),X1__21_);
INV port map (B(1),B_NOT__23_);
FADD port map (X6__26_,X7__25_,X8__24_,OUTPUT(2),CY(2));
AND2 port map (CX(2),S2_NOT__31_,X8__24_);
OR2 port map (X3__28_,X4__27_,X7__25_);
OR3 port map (X1__30_,A(2),X2__29_,X6__26_);
INV port map (SELECT_LINES(2),S2_NOT__31_);
AND2 port map (B_NOT__32_,SELECT_LINES(1),X4__27_);
AND2 port map (B(2),SELECT_LINES(0),X3__28_);
AND2 port map (CONTROL_LINES(1),B_NOT__32_,X2__29_);
AND2 port map (CONTROL_LINES(0),B(2),X1__30_);
INV port map (B(2),B_NOT__32_);
FADD port map (X6__35_,X7__34_,X8__33_,OUTPUT(3),CY(3));
AND2 port map (CX(3),S2_NOT__40_,X8__33_);
OR2 port map (X3__37_,X4__36_,X7__34_);
OR3 port map (X1__39_,A(3),X2__38_,X6__35_);
INV port map (SELECT_LINES(2),S2_NOT__40_);
AND2 port map (B_NOT__41_,SELECT_LINES(1),X4__36_);
AND2 port map (B(3),SELECT_LINES(0),X3__37_);
AND2 port map (CONTROL_LINES(1),B_NOT__41_,X2__38_);
AND2 port map (CONTROL_LINES(0),B(3),X1__39_);
INV port map (B(3),B_NOT__41_);
AND2 port map (CY(3),SELECT_LINES_NOT(2),COUT);
OUTPAD port map (OUTPUT(3));
INPAD port map (B(3));
INPAD port map (A(3));
AND2 port map (CY(2),SELECT_LINES_NOT(2),CX(3));
OUTPAD port map (OUTPUT(2));
INPAD port map (B(2));
INPAD port map (A(2));
AND2 port map (CY(1),SELECT_LINES_NOT(2),CX(2));
OUTPAD port map (OUTPUT(1));
INPAD port map (B(1));
INPAD port map (A(1));
AND2 port map (CY(0),SELECT_LINES_NOT(2),CX(1));
```

```
   OUTPAD port map (OUTPUT(0));
   INPAD port map (B(0));
   INPAD port map (A(0));
   AND2 port map (CIN,SELECT_LINES_NOT(2),CX(0));
   AND3 port map (SELECT_LINES_NOT(0),SELECT_LINES(1),SELECT_LINES(2),CONTROL_LINES(1))
   AND3 port map (SELECT_LINES_NOT(0),SELECT_LINES_NOT(1),SELECT_LINES(2),CONTROL_LINES
   INV port map (SELECT_LINES(2),SELECT_LINES_NOT(2));
   INV port map (SELECT_LINES(1),SELECT_LINES_NOT(1));
   INV port map (SELECT_LINES(0),SELECT_LINES_NOT(0));
   GNDPAD;
   VDDPAD;
   OUTPAD port map (COUT);
   INPAD port map (CIN);
   INPAD port map (SELECT_LINES(2));
   INPAD port map (SELECT_LINES(1));
   INPAD port map (SELECT_LINES(0));
end NBIT_ALU1
```

D.5 Layout



D.6 Simulation

```
assignment at time = 100 node = A    value : (0,0,0,0)
assignment at time = 100 node = B    value : (0,0,0,0)
assignment at time = 100 node = SELECT_LINES    value : (0,0,0)
assignment at time = 100 node = CIN    value : 0
assignment at time = 125 node = OUTPUT    value : (0,0,0,0)
assignment at time = 135 node = OUTPUT    value : (0,0,0,0)
assignment at time = 145 node = OUTPUT    value : (0,0,0,0)
assignment at time = 155 node = OUTPUT    value : (0,0,0,0)
assignment at time = 160 node = COUT   value : 0

assignment at time = 200 node = A    value : (0,1,0,1)
assignment at time = 200 node = B    value : (1,0,1,0)
assignment at time = 200 node = SELECT_LINES    value : (0,0,0)
assignment at time = 200 node = CIN    value : 0
assignment at time = 225 node = OUTPUT    value : (0,0,0,0)
assignment at time = 235 node = OUTPUT    value : (0,1,0,0)
assignment at time = 245 node = OUTPUT    value : (0,0,0,0)
assignment at time = 255 node = OUTPUT    value : (0,1,0,1)
assignment at time = 260 node = COUT   value : 0

assignment at time = 300 node = A    value : (0,1,0,1)
assignment at time = 300 node = B    value : (1,0,0,0)
assignment at time = 300 node = SELECT_LINES    value : (1,0,1)
assignment at time = 300 node = CIN    value : 0
assignment at time = 325 node = OUTPUT    value : (1,0,0,0)
assignment at time = 335 node = OUTPUT    value : (1,1,0,0)
assignment at time = 345 node = OUTPUT    value : (1,1,0,0)
assignment at time = 355 node = OUTPUT    value : (1,1,0,1)
assignment at time = 360 node = COUT   value : 0

assignment at time = 300 node = A    value : (0,1,0,1)
assignment at time = 300 node = B    value : (1,0,1,0)
assignment at time = 300 node = SELECT_LINES    value : (1,0,0)
assignment at time = 300 node = CIN    value : 1
assignment at time = 325 node = OUTPUT    value : (0,1,1,1)
assignment at time = 335 node = OUTPUT    value : (0,0,1,1)
assignment at time = 345 node = OUTPUT    value : (0,0,0,1)
assignment at time = 355 node = OUTPUT    value : (0,0,0,0)
assignment at time = 360 node = COUT   value : 1
```

Appendix E

User's Guide

E.1 Introduction

VHDL (Very High Speed Integrated Circuits Hardware Description Language) is textual means of describing systems design information. Structural descriptions may be converted into EDIF netlists which may be used with the Cadence Edge tools to produce an IC layout. Behavioral VHDL descriptions may be simulated using the VHDL simulator.

This guide is an introduction to the use of the VHDL tools implemented at the University of Manitoba. This guide does not discuss the internal operation of the tools, but only the operation from a user's perspective.

The first section gives a brief overview of the most commonly used features of the VHDL language. The most commonly used aspects of the language are briefly reviewed. The following sections discusses the different VHDL tools incorporated into the VHDL **analyzer** program.

E.1.1 VHDL Language

The VHDL language is an IEEE standard language [1]. This section gives a brief introduction to the language. An example of a VHDL description is available in section 8. This section will consider only those aspects of the language which are currently supported, although not all supported features are discussed. This section does not describe the total VHDL language, but rather introduces only enough to get the first time user started. More explicit VHDL information may be found in [2].

The VHDL design description consists of a sequence of design units. Each design unit is of one of the following types :

- A Package contains a set of declarations and specifications which may be shared by other design units.
- An Entity (also called entity interface) describes the externally visible connections of a design. All information declared in an entity is visible to all architecural bodies associated with the entity. Ports are used to represent external connections, and generics are used to represent design values.

- An Architecture (also called entity body) contains the design implementation information. For a structural description this will contain components and connectivity information. For a behavioral description, this will contain signal assignment statements.

VHDL offers many of the features and constructs, like programming languages, to aid in system design. Many predefined types are available to the user in the package "standard" which contains the basic logical and numeric types. VHDL allows the user to define abstract types in addition to those found in the predefined package "standard". Scalar types represent integer and floating values. Arrays types may be used to represent collections of similar typed data values. Record types are used to represents collections of non homogeneous data types.

Structural descriptions are used to describe connectivity information of a design. Connectivity may be expressed by set of node declarations representing nets, and a set of component instantiation statements representing components, either hierarchal objects or standard cells. Component instantiations can be nested in for and if statement loops.

Behavioral descriptions are represented as a set of signal assignment statements. Each statement describes a relation between a node (called target) and an expression containing other nodes. Changes in expression values cause the target value to change after an appropriate delay.

E.1.2 VHDL Tools

The tools implemented for use with VHDL are all incorporated in the the single program **analyzer** and for the most part the operation is transparent to the user. It is important to be aware of the distinctions since different errors are reported from different programs. The different tools are :

1. The VHDL Analyzer is responsible for parsing the input VHDL description and converting the VHDL into an internal format (called VIN). VIN may be stored in libraries and shared with other designs.

2. The VHDL Reverse Analyzer may convert VIN into VHDL code functionally equivalent to the VHDL code which produced it.

3. The elaborator (also referred to as the structural compiler) is responsible for transforming the VIN format into a circuit representation. After elaboration, a model of the described circuit is available for netlisting or simulation.

4. The EDIF netlister produces an EDIF "netlist" view of the elaborated circuit representation. This produces the edif netlist which may be read into Cadence, or any other tools supporting EDIF.

5. The VHDL netlister produces a flat simple structural VHDL description of the circuit representation. This is different from the reverse analyzer which simply translate VIN to VHDL. This is usefull for interfacing to VHDL programs utilizing only a small subset of the VHDL language such as the Xilinx VHDL compiler.

6. The VHDL Simulator simulates the VHDL behavioral description. A structural hierarchy and behavioral models for standard cells may be mixed to allow simulations of structural descriptions.

E.2 Program Operation

This section discusses the operation of the **analyzer** program which not only performs the functions of the VHDL analyzer, but coordinates the activities of the other VHDL tools.

The first subsection discusses the command line arguments the analyzer program. The most common sequence of actions are executed by a single option.

Not only may options be specified on the command line, but information may be stored in a .vhdl initialization file. The format of this is discussed in the next subsection. The final subsection lists the various files used and produced by the by the analyzer program.

E.2.1 Command Line Options

The **analyzer** program may be called with one and only one of the following options.

-I initialize symbol table, create prefined packages, and create syntax tree. This is used only during installation of the program, and is noramlly disabled.

-G file_name analyze filename, elaborate, EDIF and VHDL netlist. The file_name must have a .hdl suffix.

-S file_name analyze filename, elaborate, and simulate the specified file_name.The file_name must have a .hdl suffix.

-V file_name read the library file, elaborate, EDIF and VHDL netlist. The file_name must have a .vin suffix.

-L file_name analyze filename, and write to working library.

E.2.2 Initialization File

The analyzer program after reading the command line options, read a .vhdl file in the current working directory. This is file exists, certain default values may be overwritten. The file contains a sequence of commands, one per line. The following cammands are valid.

VHDL_START This specifies the start of the .vhdl file. If this is not the first command, the file contents are ignored.

SET *library file_name* This command is used to set the specified directory to the proper directory. Reference, Work, and Simulation are three predefined libraries.

CLEAR *library* remove library entry.

SIMULATION NOISEY print all simulator assignments. Useful in debugging.

SIMULATION QUIET Default simulator assignment option.

CORE DUMP enable core dumps.

NO LOG disable printing of log information.

BANANARAMA set special flag. Normally disabled.

VHDL_END Specifies the end of the initialization file. All commands after this are ignored.

E.2.3 Files

The following files are used or produced by the analyzer program. Prefix xxx refers to abritrary file names.

.vhdl user defined initialization file.

xxx.hdl VHDL soruce file.

xxx.vin VIN library file.

xxx.vhdl VHDL netlist.

xxx.edif EDIF netlist.

xxx.sim Simulation output.

vhdl.log Log file for analyzer program.

core core file sometimes produced.

E.3 Using Cadence With VHDL

This section describes using the structural compiler and EDIF netlister for use with the Cadence Place and Route tools. This section does not discuss the operation of the Cadence tools. User's should be familar with these tools before reading any further.

E.3.1 Component Declarations

The package components in the default reference library contains most of the component declarations for cells in the University of Manitoba standard cell library (CMOS3DLM). Each component declaration has an associated attribute "CDS_INFO" which contains the following information :

num_ports defines the number ports on the cell. This includes those not visible in VHDL such as power and ground.

cell_name define the cadence blockname of the standard cell. This is case sensitive.

In addition, for each port, the following information is defined

vhdl_name identifies the VHDL port name.

cds_name specifies the cadence port name.

port_type specifies the cadence purpose for the port. Values are one of POWER, GROUND, CDS_SIGNAL,or DUPLICATE. Duplicate implies the port is logically the same as another port on the cell.

port_dir specifies the port direction. value may be one of CDS_INPUT, CDS_OUTPUT, or CDS_INOUT.

Likewise the attribute "CDS_SIGNAL_INFO" is defined for signals. Two signals, VDD and GND are defined in the package components. The fields of attribute "CDS_SIGNAL_INFO" are :

global specifies whether the signal is global or not.

signal_type specifies the type of signal. Allowed values include : CDS_SIGNAL, CDS_GROUND, or CDS_POWER.

cds_name specifies the name given to the signal. This is used to use signal names not allowed in VHDL such as "vdd!".

E.3.2 EDIF Netlist

The EDIF netlist generated is a flat netlist corresponding to the top level of hierarchy in the design. The netlist uses the netlist constructs of the EDIF 2 0 0 standard and supports the properties of Cadence's edifin program. A Cadence autoLayout represenation may be created from the edif file by running the edif200_to_cds program. If no errors are found, an autolayout representation is created. This may be used in the Place and Route environment in a similar manner to any autoLayout generated from a schematic.

The operation of the Place and Route Tools is not discussed here.

E.3.3 Silos Extraction

After generation of the final layout, it is wise to perform a simulation on the design. The prlabel skill program should be run on the layout, to create pins with the net names on the appropiate wires. This will allow net names to be visible in the simulation file. Extract the silos simulation deck using the layout_silos_rules_2.1 and netlist the extracted rep. The simulation should not be performed from within Cadence since no schematic exists to select waveforms from.

E.4 VHDL Simulation

This section discusses the operation of the VHDL simulator. Emphasis is placed on the simulation of structural descriptions using behavioral standard cell entities.

E.4.1 Component Declarations

In order to simulate a structural design hierarchy, behavioral models must be used with the standard cell components. This are already define and available in the predefined simulation library which is automatically loaded when the analyzer is run with the -s command line option. The library may be replaced with another library by setting the SIMULATION library in the .vhdl file.

E.4.2 Simulation Input

No external means of specifing test vectors for the circuit currently exists. All values must be applied using signal assignment statements.

E.4.3 Simulation Output

Simulator output is written to the .sim file corresponding to the current design. Only values that correspond to nodes in the top level of hierarchy are written by default. Optionally if the attribute SIM_MONITOR exists for a node, it's value is also displayed.

E.5 Error Guide

This section discusses the error codes generated by the VHDL compiler. Errors number greater than 500 reflect errors that occurred after compilation. All error codes over 1000 reflect internal errors of the compiler and should be reported.

0. A reserved word was expected but not found.
1. Syntax error occured. Check syntax at error point.
2. The subtype declaration contained a type reference that was not able to be resolved.
3. The subtype declaration contained a type reference that was not able to be resolved.
4. The expression in a constant declaration did not match the specified type.
5. The range of scalar type declaration was not a scalar of proper type. Check expression in range.
6. The expression in a signal declaration did not match the specified type.
7. A delimiter was expected but not found. If this error is found in a component instantiation statement, check whether the component name corresponds to a component declaration.
8. An identifer was expected but not found.
9. Aggregates in targets of conditional signal assignment statements are not supported.
10. A subtype indication contained an array range for a non array type (eg BIT(0 to 2)). Make sure type is an array.
11. An operaor was expected by not found.
12. The name (signal,constant,variable) was not correctly specified. Possible undeclared identifier.
13. Attributes not supported in range. This is an allowed syntax in VHDL 1076, but not yet supported.

14. Positional aggrgate entity not allowd after named aggregated entity.

15. Invalid index in array constraint.

16. The specified attribute was not declared. This error is detected during an attribute specification.

17 A operator was used with incompatible operands.

19 An invalid or missing declaration or specification was found.

20 The enumeration Literal is not part of the specified type.

21 Lexical Analyzer error. The token produced does not match VHDL define types.

400 Invalid function in evaluate operation. A function was used that is not implemented.

500. Incompatible sizes were found in attribute initialization. Probably caused by incorrect type in attribute specification. This error should have been detected earlier.

501. A attribute was not initialized with a value.

800. Net was not a global net. When the port of a component which is a power or ground is added to the edif output, the global net with the same name was not located.

801 No CDS_INFO attribute was found for the specified component.

802 The specified component name was not found. This is typically the result of a 801 error.

900. No name for the edif netlist cell was specified, default name of NEW_CELL was used.

1000. Internal parsing error (psyntax end).

1001. Internal parsing error (epsilon rule).

9999. Internal error in program. Could be anything. Do not waste your time trying to debug this one, this is as bad as it gets.

E.6 Program Bugs

This section contains a list of currently known bugs for the VHDL tools. Bug numbers are kept contant across different versions of the programs.

5 Bug in lexical analysis. Certain lexical elements ending at end of buffer cause incorrect results. Syntax error usually appears as a result, which is incorrect. Lexical analysis errors are being included in analyzer error log which should help eliviate this. Usual action is to add spaces to beginning of input file.

7 Memory allocation error. Not all simulation memory is properly discarded when finished. Long simulations may run out of memory.

8 Not all user errors cause message to be included in error log. Please report core dumps and error 9999.

13 No type checking is performed on component declarations, and function calls. This may lead to unexplained errors later on.

22 Some incorrect syntaxes in input deck give rather cryptic messages. More help should be provided. Example component instantiation statement.

E.7 Unimplemented Features

1. Bus resolution function not supported.
2. Operator overloading not currently supported.
3. Configurations not supported.
4. Blocks not supported (soon to be implemented).
5. Access types supported but no where to use them yet.
6. File Type support but no i/o functions yet.
7. Package Bodies implemented but not needed yet.

E.8 Design Example

This sections gives an example of VHDL code describing a simple n -bit ALU. The package components is not shown.

```
--
--       test vhdl program for analyzer 1.00
--


--
-- Define a package misc which contains
-- a type declaration used by otherdesign units
--
package misc is

   type control_struct is record
      con0   : bit;
      con1   : bit;
   end record;

end misc;

--
-- The use statement tells the analyzer to
-- use the packages components, and misc.
-- components is assumed to be in an accessable
-- library.
--
use components,misc;

-- Begin Entity declaration.
--      N reprsents the number of bits in alu
```

```
--      Ports represent external connections of ALU
--
entity alu_example is
   generic
      (
      N : natural := 3
      );
   port
      (
      A : bit_vector(0 To N);
      B : bit_vector(0 To N);
      Output : bit_vector(0 to N);
      Con : control_struct;
      Cin : bit;
      Cout : bit
      );
end alu_example;

--
-- The architecture part contains the connectivity
-- information for the alu
--
use components,misc;
architecture aalu_example of alu_example is
   signal B_not    : bit_vector(0 To N);  -- signal declarations
   signal X1,X2,X3 : bit_vector(0 To N);
   signal Carry    : bit_vector(0 To N-1);

begin
   for i in 0 To N generate
      inv port map(B(i),B_not(i));          -- component instantiations
      nand2 port map(B(i),Con.con0,X1(i));
```

```
      nand2 port map(B_not(i),Con.con1,X2(i));
      nand2 port map(X1(i),X2(i),X3(i));

      if i = 0 generate
         fadd port map(a(0),X3(0),Cin,Output(0),Carry(0));
      end generate;

      if (i > 0) and (i < N) generate
         fadd port map(a(i),X3(i),Carry(i-1),Output(i),Carry(i));
      end generate;

      if i = N generate
         fadd port map(a(N),X3(N),Carry(N-1),Output(N),Cout);
      end generate;

   end generate;
end aalu_example;

-- chip1 is the description of the chip
-- it incorporates aalu_example through
-- a component declaration statement and
-- component instantiation statement
--
--  Inpads and out pads are also specified
--
use components,misc;
entity chip1 is
   generic
      (
      N : natural := 3
      );
   port
```

```
      (
      A : bit_vector(0 To N);
      B : bit_vector(0 To N);
      Output : bit_vector(0 to N);
      Con : control_struct;
      Cin : bit;
      Cout : bit
      );
end chip1;

use components,misc;
architecture version1 of chip1 is
   subtype shit_for_brains is bit_vector(0 to n);
   signal cout_tmp : Bit;
   component aalu_example
      generic
         (
         N : natural
         )
      port
         (
         A : bit_vector(0 to n);
         B : shit_for_brains;
         Output : shit_for_brains;
         Con : control_struct;
         Cin : bit;
         Cout : bit
         )
   end component;
   signal xyz,xyz1 : bit;
begin
```

```
   xyz <= xyz1 after 10 ns;            -- signal assignment statement
   ALU  : aalu_example generic map(N)
      port map(A,B,Output,Con,Cin,cout_tmp);
   CON0 : inpad port map(Con.con0);
   CON1 : inpad port map(Con.con1);
   CIN  : inpad port map(Cin);
   COUT : outpad port map(Cout);
   for i in 0 To N generate
      inpad port map(A(i));
      inpad port map(B(i));
      outpad port map(Output(i));
   end generate;
   DUMMY : inv port map(cout_tmp,Cout);
   vddpad;
   gndpad;
end version1;
```