A VHDL Design Environment

Alan R. Martello
Department of Electrical Engineering
University of Pittsburgh
Pittsburgh, PA 15261

LIMITED DISTRIBUTION NOTICE

This report has been, or will be, submitted for publication outside of the University of Pittsburgh and will probably be copyrighted if accepted for publication. It has been issued as a Technical Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of the Univeristy of Pittsburgh prior to publication is limited to peer communications and specific requests.

Contents

1 Introduction **1**

2 VHDL Design Hierarchy and Database **2**

3 VHDL Code → Internal Database **5**

3.1 Concurrent and Sequential Operations 6

3.2 Concurrent Blocks 7

3.3 Process Blocks 9

4 Generating Compiled VHDL Output **12**

5 Simulating VHDL **14**

6 VHDL Design System Summary **15**

1 Introduction

The concept of describing circuits with some form of *structured language* description has been around for a considerable period of time. Many early hardware description languages were special purpose or otherwise did not provide for a totally general, hierarchical method of describing digital circuits. VHDL was designed to satisfy these broad requirements.[1]

At the University of Pittsburgh, VHDL research has gone on for a few years. Mears [5] implemented a VHDL version 1076/B compiler and simulator. The compiler was based on work by Frauenfelder [6] on a language analyzer for an earlier version of VHDL. Frauenfelder's basic design which Mears built on was in turn based on the sample compiler described in [7]. The event driven simulator Mears implemented was capable of simulating a subset of the compiled VHDL code. This simulator was based on the RSIM work performed by Chris Terman at MIT for switch level simulation of transistors [8].

The VHDL work which Mears performed demonstrated the usefulness and established the technological feasibility of VHDL as a research tool in an academic environment. This project built on Mears' work and consisted of refining those VHDL tools to produce an externally distributable software package for classroom and research use. A secondary goal was extending the implementation of the VHDL description and simulation constructs to meet the needs of both educators and researchers [9]. This section provides an overview of the VHDL compiler and simulator.

The VHDL compiler runs in a single pass as a monolithic compilation process. It performs the following functions:

1) read the input VHDL files and verify that they are valid;
2) build an internal representation of the VHDL code; and
3) translate the internal representation into a VHDL database file (.ivf) if there are no errors in step 1.

Since the compiler performs a single pass over all input files in the process of

[1]This section is not meant as an introduction to VHDL; that is beyond the scope of this paper. There are a number of good books available for a description and examples of VHDL (such as [1–3]). For the complete VHDL specification, refer to the current IEEE standard document [4]. The emphasis here is to provide a brief overview of the VHDL design system as implemented.

generating output, no forward declarations are permitted. Any item (such as an entity or entity/architecture pair) must be defined prior to being referenced by some other part of the database. The simulator can read the database which is output by the compiler; this makes it possible for the user to simulate their VHDL description.

The remainder of this discussion of the VHDL design project is organized as follows. First we discuss the concept of the VHDL design hierarchy and describe how the data structures used to implement this hierarchy are defined. Next, the method which the VHDL code is parsed and placed in the internal database is presented. This includes a general overview of the concurrent and sequential constructs which are supported. Finally, the process of creating output of the compiler from the internal compiler database is described, and an overview of the simulator is given.

2 VHDL Design Hierarchy and Database

The basic structure within a VHDL design is the *design entity*. A single design consists of (potentially) many design entities; each design entity describes a single portion of the design. Each design entity has a single *entity declaration* which describes the inputs and outputs of the design entity. The entity declaration does not describe *how* the design entity functions, it *defines* the inputs and outputs of the entity. Figure 1 shows a VHDL description for an 8-bit barrel shifter. The lines marked with `1` are the entity declaration.

Each design entity has at least one, but perhaps more, *architectures* associated with the entity declaration. Each architecture provides *one possible* way of describing the functionality of the design entity or one possible implementation of the design entity. Design entities may be hierarchically nested. This means that if one design entity is used in more than one part of the design, then it only needs to be *defined* once but may be referenced (or *instantiated*) multiple times. VHDL gives us the flexibility to describe designs while maintaining any type of hierarchical partitioning or nesting. The architecture in Figure 1 is designated by `2`.

Architectures describe how a particular implementation of a design entity should function. Architectures consist of two parts: the *architecture declarations* (`3` in Figure 1) and the *architecture body* (`4` in Figure 1). The

```
1     entity BARREL is
1            port (data_in:   in bit_vector(7 downto 0);
1                  shift:     in bit_vector(7 downto 0);
1                  data_out:  out bit_vector(7 downto 0)
1            );
1     end BARREL;

2     architecture logic of BARREL is
3     2            signal buffer_a:  bit_vector(7 downto 0);
3     2            signal buffer_b:  bit_vector(7 downto 0);
2     begin
4     2            -- stage one, shift one bit if needed
4     2            buffer_a(7 downto 0) <=
4     2                  data_in(6 downto 0) & data_in(7) when shift(0)
4     2                  else data_in(7 downto 0);
4     2
4     2            -- stage two, shift two bits if needed
4     2            buffer_b(7 downto 0) <=
4     2                  buffer_a(5 downto 0) & buffer_a(7 downto 6)
4     2                  when shift(1) else buffer_a(7 downto 0);
4     2
4     2            -- stage three, shift four bits if needed
4     2            data_out(7 downto 0) <=
4     2                  buffer_b(3 downto 0) & buffer_b(7 downto 4)
4     2                  when shift(2) else buffer_b(7 downto 0);
2     end logic;
```

Figure 1: VHDL Description of 8-Bit Tree Structured Barrel Shifter

architecture declaration describes the various items used within the architecture body. These items include *signals* (which can be considered the same as wires) and forward references to other design entities which are nested inside this architecture.

Within an architecture body may appear two types of statements: concurrent statements and sequential statements. These statements are used to describe the functionality of the architecture. VHDL's concurrent and sequential statements are an attempt to describe circuits which exhibit parallel behavior and serial behavior respectively. The basic data construct within both sequential and concurrent code is the assignment construct.

To internally manipulate the VHDL design, an internal database format was defined which modeled the VHDL design hierarchy described above. Although the VHDL compiler parses the entire VHDL grammar as defined in [10], it does not build the entire language into its internal database. Only those items described below are built into the internal VHDL representation.

Figure 2 shows the uppermost level of the internal VHDL database. The entire design description is composed of a list of entities. Each entity contains the information which defines its inputs and outputs. Each entity has one or more architectures which describe possible implementations of the entity. If an entity has multiple architectures (or representations) then these are chained in a list which is linked to the entity.

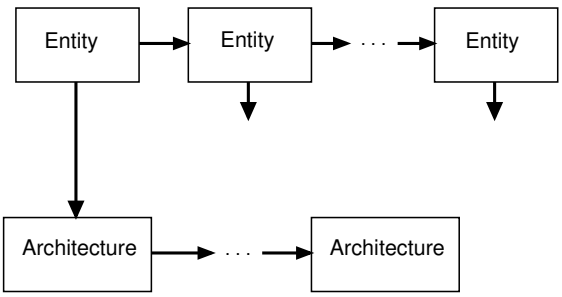

Figure 2: VHDL Compiler Entity and Architecture Structure

Within each architecture are blocks which contain the actual implementation of the architecture. Each architecture contains at least one top-level block (shown in Figure 3 as Block a. This block is a concurrent block. This block may contain other blocks nested within it (for example, Block aa and ab) and any block may contain other nested blocks (such as Block aaa and aab. This block structured nesting capability is similar to programming languages such as Pascal.

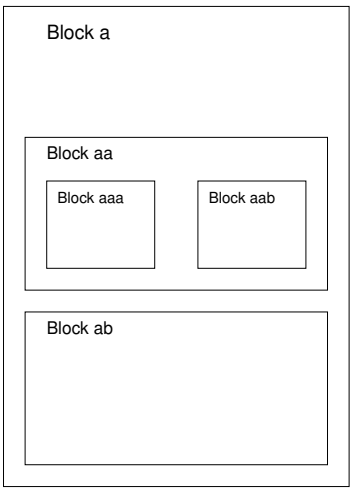

Figure 3: VHDL Compiler Block hierarchy

Each block (other than the top-level block in the architecture) is either a *process block* or a *concurrent block*. A process block contains process statements which by definition are "executed" sequentially. A concurrent block contains concurrent statements which "execute" in parallel. Concurrent blocks may contain other nested blocks (either process or concurrent) but process blocks may not contain additional hierarchy. For this reason, the top-level block in the architecture is a concurrent block.

Blocks contain two basic parts: a declaration part and a statement part. The

declaration part contains definitions of local data items. A *data item* could be a signal, a variable, a constant, or a variety of other data types, depending on the type of block as defined in the VHDL Language Reference Manual [10]. Scoping rules for data items referenced in the statement part of a block is the same as in other block structured languages. If a reference to a data item within a block appears ambiguous due to multiple declarations of the same item within the block hierarchy, the reference is resolved statically by using the most recently declared version of the data item.

Within the statement part of a block may appear a variety of items depending of the type of block. In concurrent blocks, the following items are supported within the statement part:

- nested process blocks,
- nested concurrent blocks,
- component instantiations, and
- assignment statements.

Process blocks only support sequential control and assignment statements; no hierarchy of components or blocks are permitted within process blocks. Inside every block in the database is a list of statements which appear within the block. These statements define the functionality of the block and ultimately the entire design.

3 VHDL Code → Internal Database

The part of the compiler which parses the VHDL input file is automatically created from an input grammar utilizing the parser generator *bison*[11]. Parse trees for each VHDL assignment statement are created and stored in the internal database. Each node of the parse tree contains an operation to perform and pointers to the operands. These operand pointers may reference other operation nodes. This allows for arbitrarily complex expressions to be represented by the parse trees.

The primitive operations supported for both concurrent and sequential blocks are shown in Table 1. The operations common to both concurrent and sequential statements are discussed in the next section and the operations unique to concurrent and sequential statements are covered in Sections 3.2 and 3.3 respectively.

Operation	Concurrent	Sequential	Group
null	√	√	I
and	√	√	
or	√	√	
nand	√	√	
nor	√	√	
xor	√	√	
not	√	√	
concat	√	√	
eq	√	√	
ne	√	√	
cond	√		II
select	√		
plus		√	III
minus		√	
lt		√	
le		√	
gt		√	
ge		√	
forup		√	IV
fordown		√	
bag		√	
rising		√	
falling		√	

Table 1: Primitive Internal Operations Supported

3.1 Concurrent and Sequential Operations

The Group I operations of Table 1 are common to both sequential and concurrent statements. The operations *and, or, nand, nor, xor,* and *not* are the standard logic operations. The operations *and, or,* and *xor* are multiple input single output operations; *not* is a single input operation; and *nand* and *nor* are two input operations. The *width* (or bit size) of all inputs to these operations should be identical and must match the width of the output.

The *null* operation is a single input operation and is used to represent instructions such as `A <= B`. The *null* operation is also used during parsing to represent and to insert expression evaluation precedence into an expression. The input and output widths of the operands for the *null* operation should be identical.

The *eq* and *ne* operations are used to express equality and inequality operations. These are two input operations whose operands must have the same size. The result of the operation is a single bit which indicates if the operands are identical or different.

The *concat* operation performs concatenation of all its input operands into a single bit string. Each operand may be any size and the result of the operation is a bit string whose size is the summation of the sizes of all the operands.

3.2 Concurrent Blocks

Concurrent blocks are utilized as the method for expressing parallel execution and provide VHDL hierarchical capabilities. Within concurrent blocks may appear component instantiations, concurrent statements, nested concurrent blocks, and process blocks.

Component instantiations are used to define that a *copy* of an entity implemented by a specified architecture is present within a concurrent block. The component instantiation behaves similar to a macro definition in function (versus a procedure call) since it *creates a copy* of the entity and architecture within the hierarchy when the database output is generated.

When a component is instantiated, the following checks are performed.

- Any constants which are in the port specification of the instantiation must have a signal direction '*in*' within the instantiated entity.
- Any signals which are in the port specification of the instantiation and have a signal direction '*in*' must map to ports which also have a signal direction '*in*' within the instantiated entity.

In addition to these two instantiation checks, a check is made on each assignment statement to verify that the left-hand side of the assignment statement is not a signal which has a signal direction '*in*'. These three checks combined allows constants to be passed into component instantiations freely and not be corrupted within any arbitrary hierarchy.

In the database, the entity / architecture pair which refers to a component instantiation are *pointed* to by the component instantiation statement; they are not explicitly copied. The names to use for the port mapping are stored

within the instantiation for use during the database output generation phase of compilation.

The Group II operations of Table 1 are used only in concurrent statements. The operation *cond* is used to represent a conditional signal assignment as in

```
a <= b when c else d.
```

Three operands are required for this operation. The first operand (`b` in our example) is the value to be assigned when the second operand (`c`) is true. The final operand (`d`) is the value to be assigned when `c` is false. The sizes of `a, b,` and `d` must be the same and the size of `c` must be one (i.e. a single bit). All of `b, c,` and `d` may be an arbitrary expression.

The operation *select* signifies a selected signal assignment and is used to choose a single value for an assignment. The VHDL syntax for the instruction is:

$$\begin{array}{lllll} \texttt{with } b \texttt{ select } c \texttt{ <=} & x_1 & \texttt{when} & y_1, \\ & x_2 & \texttt{when} & y_2, \\ & & \vdots & \\ & x_n & \texttt{when} & y_n; \end{array}$$

This statement operates by comparing b first with y_1, then y_2, etc. and stopping when the first match is found. When this match is found, the corresponding value of x is assigned to c.

To implement this construct, a four input, single output operation is represented in the parse tree. The first operand is x, the second operand is b, the third operand points to the output of a "nested" *select* expression which represents the succeeding `when` clause, and the fourth operand is y. Because the third operand can point to another *select* operation, an arbitrary VHDL *select* instruction with n `when` clauses can be represented in the parse tree with n select operations. The sizes of the first and third operands of a *select* operation are the same size as the output; the sizes of the second and fourth operands must be identical since they are compared for equality.

The parsing of a concurrent assignment statement has four phases. The first phase is the building of the parse tree for the expression in the internal database. The second phase is a check that all operand sizes are correct depending on the operations being performed. The third phase consists of checking that the left-hand side expression of the assignment statement is not an entity input port since we never allow input ports to be modified. The final phase is the optimization of the parse tree by removing *null gates*.

During concurrent statement parsing, many *null* gates are added to disambiguate the expression and to correctly capture delay expressions within the concurrent signal assignment. After the entire statement has been parsed, some of these gates may be superfluous and can be removed. This optimization phase is performed to remove the superfluous gates.

The nesting of concurrent blocks provides for modularity within a block. Everything which may appear in the uppermost block of an architecture may also appear in any nested concurrent blocks. Static scoping rules apply to signals which are declared within nested blocks.

VHDL also allows nested concurrent blocks to have *guards* associated with them. A *guard* is an enabling signal which may be generated by an arbitrary VHDL concurrent expression. The *guard* is then used to selectively enable and disable concurrent assignment statements within the block which are *guarded*. This construct is useful for modeling buffers or entire circuit partitions which are selectively enabled and disabled by simple signal expressions.

3.3 Process Blocks

The process blocks are used within VHDL to express sequential execution. Process blocks which are encountered are stored within the database and are also translated to 'C' language code for simulation. If used during simulation, the complex sequential code executes at native machine speeds since it is executed directly and is not interpreted. Two types of translations take place in mapping the sequential code to 'C' code: translation of control constructs and translation of data constructs.

The sequential control constructs supported are similar to ones found in most high level programming languages. These control constructs may be arbitrarily nested to allow for complex sequential control description. When encountered, these sequential constructs are translated to 'C' code. The translations performed are shown in Table 2 on page 11.

The Group III and IV operations of Table 1 are used only in sequential statements. Group III operations are mathematical in nature while Group IV operations are used for control constructs.

Since the sequential constructs are translated to 'C' language statements,

then these operations are dependent on the lower-level 'C' representation for their functionality. The Group III operations of *plus* and *minus* signify standard two's complement addition and subtraction. Both operations require two operands and no checking of sizes of the operands is performed. Similarly the *lt, le, gt,* and *ge* operands perform the respective two's complement comparison operation.

The Group IV items are not true "operations" but are constructs which allow the representation of looping and signal changes within the parse tree structure. The *forup* and *fordown* "operations" signify ascending and descending loops respectively. The *bag* "operation" is used to hold values to compare against for VHDL 'case' statements. The *rising* and *falling* "operations" are used to represent the locally defined signal attributes `'rising` and `'falling`.

VHDL Sequential Construct	'C' Construct

if *expression* **then**
sequential-statements
[**elsif** *expression* **then**
sequential-statements]∗
[**else**
sequential-statements]
end if ;
	if (*expression*) {
	translated-sequential-statements
	[} **else if** (*expression*) {
	translated-sequential-statements]∗
	[} **else** {
	translated-sequential-statements]
	}

case *expression* **is**
when *choices* =>
sequential-statements
[**when** *choices* =>
sequential-statements]∗
end case ;
	switch (*expression*) {
	case $choice_1$**: case** $choice_2$**:** ...
	translated-sequential-statements
	break;
	[**case** $choice_i$**: case** $choice_{i+1}$**:** ...
	translated-sequential-statements
	break;]∗
	}

loop
sequential-statements
end loop ;
	for (;;) {
	translated-sequential-statements
	}

while *expression* **loop**
sequential-statements
end loop ;
	while (*expression*) {
	translated-sequential-statements
	}

for *iden* **in** *range* **loop**
sequential-statements
end loop ;
	for (*iden* = *start-range* ;
	iden comparison end-range ;
	iden inc/dec) {
	translated-sequential-statements
	}

exit [**when** *expression*];	[**if** (*expression*)] **break ;**
next [**when** *expression*];	[**if** (*expression*)] **continue ;**
null ;	;

Table 2: Translations Performed from VHDL Sequential Statements to 'C' Statements

The sequential data constructs are treated in a generic manner. All data is assumed to consist of a collection of bits which is not greater than the maximum size of a long integer for the host machine's 'C' compiler. This size is 32 bits for VAX and SUN systems. Each data construct (*integer*, *bit vector*, *variable*, or *bit*) is mapped into a single long integer.

The advantage of utilizing this mapping is that mathematical operations have a predefined functionality on the host machine for long integers and operations such as subtraction or arithmetic comparisons are not ambiguous. A disadvantage of this approach is the limitation imposed by the length of a long integer on the host machine. This limitation effects the concurrent portion as well as the sequential portion of the compiler; thus all data objects within the VHDL design system are currently limited to a maximum length of 32 bits.

4 Generating Compiled VHDL Output

Once the input VHDL is successfully parsed and the entire database built in memory, output is generated. The output generated needs to be referenced from some uppermost point called the "top-level". This top-level of the hierarchy consists of both an entity and an architecture specification. Because the compiler performs its work in one pass, the internal database is guaranteed to have a valid hierarchy rooted at every entity / architecture pair within the database. The output generated consists of two versions of the internal database: a hierarchical version and a flattened version.

The hierarchical database maintains much of the look of the original VHDL but has the following additions:

- all expressions are fully parenthesized,
- all concurrent expressions have undergone type and size checking, and
- all signals and variables used appear in the declarations.

This portion of the database maintains the VHDL hierarchy and is useful for processing by tools which manipulate and optimize high-level or hierarchical constructs; examples of this usage of the hierarchical database are given in [9].

The hierarchical output is generated by scanning the entire database, one entity at a time, and writing all information associated with each node in the

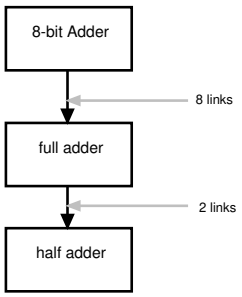

Figure 4: 8-bit Adder Hierarchy

hierarchy to the output file. The top-level node is noted in the hierarchical output and is used to limit the amount of the internal database written to the output; only that part of the database which is referenced within the hierarchy rooted at the specified "top-level" node is written to the hierarchical database.

For example, if a design of an 8-bit ripple carry adder consisted of 8 full adders and each full adder was further described by two half adders, the design database would appear as shown in Figure 4. The hierarchical output of the design contains three major entries: one for the entire adder (top-level node), one for the full adder, and one for the half adder. The full adder is composed of two half adders and a single *or* gate. The half adder is composed of three *and* gates, two *not* gates, and one *or* gate. Each entity has multiple links within the hierarchical database as shown in Figure 4; thus ten links are present within the hierarchical representation of the adder.

The flattened database contains the information in the internal database in a totally expanded format. The entire hierarchy is flattened down to the operation level (also referred to at this point as the gate level). This means that the flattened database assumes operations are the primitive elements. For the 8-bit adder example, the flattened representation is composed of 104 gate equations which are broken down as follows:

$$\begin{array}{ccccccccccc} \left(\right. & \frac{\#\ \text{gates}}{\text{half adder}} & \times & \frac{\#\ \text{half adders}}{\text{full adder}} & + & \frac{1\ or\ \text{gate}}{\text{full adder}} & \left. \right) & \times & \#\ full\ adders & = & \#\ gates \\ (& 6 & \times & 2 & + & 1 &) & \times & 8 & = & 104\ gates \end{array}$$

The operation of flattening the database consists of performing a tree traversal of the entire database starting at the node specified as the top-level. The database is traversed in a depth-first manner and a gate is generated each time a leaf node in the tree is encountered.

Each operation (or gate) equation within the flattened database contains the following information:

- a unique gate name;
- the type of gate/operation;
- the delay (if any) associated with the operation;
- the type of delay (if any) associated with the operation;
- the number of inputs;
- a list of inputs;
- the number of outputs;
- the list of outputs.

In addition to removing hierarchy, the flattened database represents all multi-bit signals such as bit vectors as single data bits. Because the interconnection within the netlist is specified by name, each node within the net is labeled. This information in the flattened database was designed to be used as the input to the simulator discussed in the next section. Although specific decisions were made regarding the flattened format to accommodate our simulator, the flattened format is in a general netlist form suitable for other applications and has been used as input for automatic schematic generation [12].

5 Simulating VHDL

The VHDL simulator which has been implemented is an event-driven simulator. It has a user friendly command line interface which supports wildcarding, command name aliases, macro expansion, and command files.

The simulator's internal structure supports two main data types which are read from the flattened database description: *nodes* and *objects*. *Nodes* are the data items represented within the simulator. *Objects* are the gates or operations represented within the simulator.

For simulation, the process blocks which have been translated to sequential code are placed in a file with a simulator specific header and footer. The header and footer allow the translated sequential code to have read and write access to the nodes. The translated process blocks are compiled and linked with a simulator library to form a VHDL model specific simulator. Once this custom simulator is created, the process blocks are not (in general) treated specially within the simulator but are simply viewed as large, multi-input,

multi-output "gates" or "operations."

Since the simulator is an event driven simulator, it maintains a list of events sorted by the time at which the events should occur. The events maintained are changes of node values within the simulator. When the node associated with an event has its value updated as specified by an event and the event is removed from the event queue, the event is said to have *fired*.

The simulator has three main modes of operation:

Initialization –	read in the netlist and initialize the nodes,
Waiting for Command –	at the command line waiting for a command from the user, and
Firing Events –	updating nodes from events on the event list which are supposed to occur up to the current time.

After the Initialization state, the simulator goes into a Wait-for-Command and Fire-Event loop. The simulator transitions from the Wait-for-Command state to the Fire-Event state when a command is issued which modifies the current time. The simulator remains in the Fire-Event state until all events which should fire up to the current time have fired or until the user interrupts event firing by typing an interrupt key; either of these conditions returns the simulator to the Wait-for-Command state.

6 VHDL Design System Summary

This report has provided an introduction to the VHDL Design System. We have discussed various implementation issues which have determined the behavior and capabilities of the VHDL compiler and simulator. For a design system such as this to be useful, the ability to specify the design is not sufficient by itself; a means of exercising the design must also be provided. We have addressed both of these needs by providing both a VHDL compiler for system description and a simulator for testing. This project has been an attempt at implementing a robust, academic and research oriented design system for describing and testing designs with VHDL.

References

[1] David R. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, Boston, MA, 1989.

[2] Roger Lipsett, Carl Schaefer & Cary Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Boston, MA, 1989.

[3] James Armstrong, *Chip-Level Modelling With VHDL*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[4] IEEE Press, *IEEE Standard VHDL Language Reference Manual, IEEE Standard 1076-1987*, Montvale, NJ, 1988.

[5] Lyn Ann Mears, "Tickle: An Integrated Design Capture and Simulation Tool," Elec. Eng. Department, Univ. Of Pittsburgh, TR-CE-88-002, Pittsburgh, PA, Jan., 1988.

[6] D. J. Frauenfelder, "An Implementation of a Language Analyzer for the Very High Speed Integrated Circuit Hardware Description Language," School of Engineering, Air Force Institute of Technology, unpublished M. S. Thesis, 1986.

[7] A. T. Schreiner & H. G. Friedman, Jr., *Introduction to Compiler Construction with UNIX*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[8] C. J. Terman, "Simulation Tools for Digital LSI Design," MIT Laboratory of Technology for Computer Science, MIT/LCS/TR-304, Sept., 1983.

[9] S.P. Levitan, A.R. Martello, R.M. Owens & M.J. Irwin, "Using VHDL As A Language For Synthesis Of CMOS VLSI Circuits," *Proc. 9th Inter. Symposium on Comp. Hard. Description Lang.* (June, 1989).

[10] CAD Language Systems, Inc., *VHDL Language Reference Manual, Draft Standard 1076/B*, Rockville, MD, 1987.

[11] Charles Donnelly & Richard Stallman, *BISON Reference Manual*, Free Software Foundation, Cambridge, MA, 1988.

[12] Stephen T. Frezza & Steven P. Levitan, *SPAR: A Schematics Place and Route System*, Dept. of Elect. Eng., University of Pittsburgh, Aug., 1990.