VHDL Synthesis System (VSS) Release 3.0 User's Manual (February 1991 Prototype Release)

Joseph Lis
Loganath Ramachandran

Technical Report #91-19
February 25, 1991

Dept. of Information & Computer Science
University of California
Irvine, CA 92717
(714) 856-7063

ramachan@ics.uci.edu

TABLE OF CONTENTS

blank page.

1. Introduction

This distribution contains the software for a prototype version of the VHDL Synthesis System (VSS) under development in the CADLAB at the University of California, Irvine. The VSS system consists of the following modules:

(1) A *Graph Compiler* which generates a flowgraph representation from an input VHDL description

(2) A *C Graph Critic* which uses a rule-based system to optimize the graph using local pattern substitutions.

(3) A *Component Synthesis Algorithm* (*CSA*) which operates on concurrent descriptions in order to identify mutually exclusive operations that can be mapped to the same component.

(4) *CFG->DFG transformations* which restructure the representations of a behavioral (process) description into the equivalent concurrent description.

(5) A *Design Compiler* which transforms the internal flowgraph representation into a register-transfer design of generic components.

(6) A *Netlist Generator* which produces a VHDL structural netlist for the synthesized design.

All software supplied in this release is written in C and uses the UNIX yacc and lex compiler writing tools. The user interface supplied runs under the X Windows windowing system.

2. Release 3.0 Features

Release 3.0 of the VSS System contains the following enhancements made to the March 1990 release:

- A hierarchical VHDL block description is now accepted. A choice of retaining this specified hierarchy in the synthesized structure is now offered.

- Representation optimizations such as CSA and the CFG transformations have been added.

- A Control Logic Compiler is included in the release which will generate Control units from the state tables produced by VSS.

- Multiple processes are allowed in the same description. Each process is mapped to a control unit/data path.

- The Suntools dp display program has been replaced by xdp, an equivalent X windows utility.

3. Installation

The software was developed on Sun3 and Sun4 computer systems running the UNIX operating system version 4.1.

The following procedure should be used to install the software on a similar system:

(1) Select or create a subdirectory where the compiler software will reside and enter this directory with the command:

```
% cd <VHDL-directory>
```

(2) Insert the distribution tape in the 1/4-inch tape drive and execute the following command to extract the files from the tape:

```
% tar xvf /dev/rst0
```

(3) Any user of the VSS system must define environment variables which indicate the location of data files required by VSS. These variables can be defined in the *.cshrc* file in the user's home directory. Edit the .cshrc file and add the following statements:

```
setenv VSS_DEF_COMP_TABLE   <VHDL-directory>/data_tables/comp_table
setenv VSS_DEF_OP_TABLE     <VHDL-directory>/data_tables/op_table
setenv GENUS_DEF_LIB_FILE   <VHDL-directory>/data_tables/genus.comps
xrdb -merge <VHDL-directory>/bin/XVSS.ad
```

Substitute the complete pathname of the directory <VHDL-directory> selected for the installation.

(4) An executable image for the VSS system is provided with the distribution in the <VHDL-directory>/bin/$arch directory, where $arch is sun3 or sun4 depending on the machine on which VSS is run.

If it is necessary to create an executable image for the compiler, execute the following commands:

```
% cd <VHDL-directory>
% make
```

This will make the VSS executable and all display program executables. The executable images will be placed in the <VHDL-directory>/bin/$arch directory. The VSS executable has the program name **vss**.

(5) An executable images for the VSS Flowgraph X Windows Graphical Display Utility (**xdp**) and utility programs **{xvss} and {xdisp_file} are also provided in the directory < VHDL-directory> /bin/$arch.**

(6) In order to access and execute these programs from any directory in the user's file space, add the full pathname of the bin directory to the path variable definition normally found in the .cshrc or .login file in the user's home directory. For example, if the executables are in the directory /usr/joe/VHDL/bin, a *set path* command in the .login or .cshrc file should be added or modified to look like the following:

```
set path=(. /bin /usr/bin...  ... /usr/joe/VHDL/bin)
```

(7) Once the installation has been tested, the object files created during compilation can be removed using the following commands:

```
% cd <VHDL-directory>
% make clean_o
```

4. Running the VSS System

In order to process a design using VSS, the following modules are invoked:

(1) Graph Compiler
(2) Graph Critic
(3) Allocator
(4) Scheduler
(5) BIF ops based state table generator
(6) Resource Binder
(7) BIF unit based state table generator
(8) Data path netlist generator
(9) Control Logic Compiler
(10) VSS Functional synthesis of control unit

(11) CU/DP netlist merger

4.1. Behavioral Description Input File

The input file should consist of a textual VHDL behavioral description. The format of the VHDL language subset accepted by the compiler is described in Appendix A, and examples can be found on-line in the <VHDL-directory>/examples subdirectory. The input file should have the following naming convention:

<design-name>.vhdl

The design style to be used for synthesizing the description can be specified in a comment/annotation of the following format:

--VSS: design_style <style>

where <style> can be either COMBINATIONAL, FUNCTIONAL, REG_TRANSFER or BEHAVIORAL. This annotation should be placed between the entity and architecture sections of the description. See Appendices D and E for examples.

4.2. Allocation Specification File

For descriptions written in the behavioral style (using process and sequential statements), the user must specify the number of each type of functional units that can be used during the Scheduling phase of Design Compilation. This is accomplished via a text file with the following naming convention:

<design-name>.pd

The syntax of this Allocation Specification file is described in Appendix B, and an example file can be found in the example of Appendix E.

4.3. Command Syntax

In order to execute the program, the input file (and allocation specification file) must be in the current directory. Enter the command:

```
% vss [-bcdgt] <design-name>
```

Current command line options include:
  b - perform bitwidth consistency checks/padding of mismatched bws

```
        (the global variable 'bw_check' is set to TRUE)
    c - turn off Graph Critic (by default, it's always invoked)
        (the global variable 'invoke_gc' is set to FALSE)
    d - use default settings (eliminates a lot of prompting during runs)
        (the global variable 'default_settings' is set to TRUE)
    g - run the Graph Compiler only (diagram is generated, then VSS quits)
        (the global variable 'graph_only' is set to TRUE)
    t - invoke CFG->DFG transformations
        (the global variable 'invoke_fg_trans' is set to TRUE)
```

NOTES:
1. In the discussions that follow, all pathnames mentioned assume the home directory <VHDL-directory>.
2. See section 3 "Installation" for instructions as to how to set up a VSS user.

1. Graph Compiler

If the default settings are not used, you will be prompted to set the level of Graph Compiler debug information printing as follows:

```
Print Graph Compiler debug information (y/n)?:
```

If you respond 'y', the following prompt appears:

```
Graph Compiler debug level
   0 = print no debug information (default)
   1 = print node creation information
   2 = print node merging/deletion information
   3 = print node list modification information
   4 = print node connection information
=>
```

Options 1-4 produce report increasing levels of detail with respect to the creation of nodes, modification of global node lists, the status of the node stack, node and net connection information, etc. which may be helpful in debugging.

The Graph Compiler reads in the input VHDL description. It will process annotations of the form:

```
--VSS: <keyword> <value>
```

which direct the compiler as to what design style to use, what signal kind is to be associated with a signal, what transformations are to be performed, etc. (see the note on 'annotations').

A Control/Data Flow Graph data structure is built as the input description is parsed (see the "Flow Graph Data Structure Specification", CADLAB Internal Document #4, for details of this data structure).

2. Graph Critic

After each block (in the case of COMBINATIONAL, FUNCTIONAL or REG_TRANSFER designs) or section of straight line code (in the case of BEHAVIORAL designs) is parsed and a flow graph is created, the Graph Critic is invoked on that section of DFG.

3. Allocator

The 'Allocator' is really more of an 'allocation constraint entry' for the scheduling phase.

i. First, the GENUS generator level of the GENUS partial design hierarchy is read in and built by the 'parsers/genus' parser (a call to the function read_in_GC_table). The file 'genus.comps' in the 'data_tables' directory contains the specification of all GENUS components currently used by VSS. This builds a list of GC_GENERATOR_DEF records which store information about the name, parameters and a specification of the operation performed by each GENUS component class. The list is actually maintained as an array (GENUS_generators).

ii. Next, the GENUS partial design representation for the current design is initialized. This involves creating a main entity ENTITY_REC which represents the top level entity/architecture.

iii. Allocation data tables are then read in.

In some earlier versions of VSS, you may see the following message printed during a run:

```
reading in component table from file:
  /ch/ub/jlis/vhdl/vss_code/new_ds/data_tables/comp_table
```

Before the GENUS generator input parser was available, a function map table was generated by reading the file 'data_tables/comp_table' using the function read_comp_table(). This provided a mechanism for mapping DFG node operations to GENUS components (which is now accomplished using the GENUS generator list).

The 'operator and unit upgrade cost table' is then generated by reading in the file 'data_tables/op_table' using the function read_op_table(). This table defines the grouping of operations into OP_CLASSES, and specifies costs used in the Frequency Based Binder for adding new functionality to an ALU with a current operator list. This parser builds the op_class, operator, and upgrade_cost tables kept in integer arrays (see the op_table.h, alloc_defs.h and alloc_vars.h files in the 'include' directory for the definitions of these data structures).

iv. Next, a prompt for ALLOCATION CONSTRAINT ENTRY is given (this used to be given after the scheduler was selected, but it has been recently moved before the scheduler selection (as of 12/3/90)).

```
------------------------------------------------
ALLOCATION CONSTRAINT ENTRY

Enter allocation constraints via constraint file (y/n): y

Partial design input file [<design_name>.pd]:

------------------------------------------------
```

If a 'y' response is given to the first prompt, a prompt is issued for the name of a file which contains the unit allocation to be used during scheduling. The default name of this file appears in brackets. If this name is correct, type <RETURN>; otherwise, enter the name of the allocation file.

This file will be parsed by the parsers/pd_input parser (with the function read_pd_input(<file>)). The syntax of this file is defined in Appendix B of the "VHDL Synthesis System (VSS) Release 2.0 User's Manual". There has been one change to this syntax: the op_delay_spec has been changed from an 'unsigned_integer' to a 'real_number'.

The file will be parsed to create the GENUS component class and instance representation of the allocated components.

If a 'n' response is entered to the constraint file prompt, ALLOCATION CONSTRAINT ENTRY is placed in the interactive mode in which the user is prompted to enter the component allocation on an operation class or unit by unit basis as follows:

a. operation class - the following is an interactive session in which the user specifies the number and execution delay of components on an operation class basis:

```
------------------------------------------------
ALLOCATION CONSTRAINT ENTRY

Enter allocation constraints via constraint file (y/n): n

Enter allocation by
  o = operation class
  u = unit
=> o

Enter number of adders: 2
  operation execution delay (# cycles): 0.5

Enter number of subtractors: 1
  operation execution delay (# cycles): 0.5

Enter number of multipliers: 1
  operation execution delay (# cycles): 2.0

Enter number of dividers: 0
  operation execution delay (# cycles): 2.0

Enter number of logical units: 3
  operation execution delay (# cycles): 0.75

Enter number of relational units: 0
  operation execution delay (# cycles): 0.6

------------------------------------------------
```

b. unit by unit component allocation - the following is an interactive session in which an 8-bit ADDER/SUBTRACTOR and 8-bit COMPARATOR with a GEQ function is allocated:

```
------------------------------------------------
ALLOCATION CONSTRAINT ENTRY

Enter allocation constraints via constraint file (y/n): n

Enter allocation by
  o = operation class
  u = unit
=> u

Enter attributes for each component (enter 'q' as Operation Class to quit)

function unit 1:
Operation Class (REL,ADD,LOG,MULT,SHIFT,q)
=> ADD
bit width
=> 8

Enter component operation type(s) from the following list:
(ADD,SUB)
Enter one type at a time (enter 'q' to complete)
operation type => ADD
op_type(ADD) = -1, (+) = 6
operation type => SUB
op_type(SUB) = -1, (-) = 7
operation type => q
Execution delay (# cycles)
=> 1.0

function unit 2:
Operation Class (REL,ADD,LOG,MULT,SHIFT,q)
=> REL
bit width
=> 8

Enter component operation type(s) from the following list:
(EQ,NEQ,LT,LE,GT,GE)
Enter one type at a time (enter 'q' to complete)
operation type => GEQ
op_type(GEQ) = -1, (>=) = 3
operation type => q
Execution delay (# cycles)
=> 1.0

function unit 3:
Operation Class (REL,ADD,LOG,MULT,SHIFT,q)
=> q

-----------------------------------------------
```

4. Scheduler

i. SLICER is the primary scheduler used in VSS. This is a variant of Barry Pangrle's scheduler which calculates the as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedules in order to determine the range of machine states to which an operation can be assigned. The scheduler actually consists of two parts: a macro scheduler which traverses the CFG and assigns states to control point nodes, and the SLICER scheduler which is applied to all STMT_BLKs encountered. The first state to be assigned to the STMT_BLK is passed to the SLICER scheduler, along with the DFG nodes in the STMT_BLK, and the scheduled STMT_BLK is returned.

ii. The next prompt you will be given is to set the level of scheduler debug information printed:

```
Turn on scheduler debugging (y/n)?
```

If you respond 'y', the following prompt is issued:

```
Scheduler debug level
  0 = print no debug information (default)
  1 = print traversal information only
  2 = minimal debug information
  3 = maximal debug information
=>
```

0: By default, a message is printed every time the SLICER scheduler is invoked on a STMT_BLK. It also prints a warning message if no unit was allocated to perform the operation currently being processed. In that case, a unit is created and added to the GENUS partial design representation.

1: In addition to the information printed for level 0, the nodes visited by the macro scheduler will be printed in the order in which they are visited.

2: In addition to the information printed for level 1, the ASAP and ALAP state assignments made to each node are printed.

3: In addition to the information printed for level 2, the examination of each node in the final mobility scheduler phase is printed.

5. BIF ops based state table generator

Once scheduling is completed, the OPS BASED BIF state table is generated.

6. Resource Binder

i. The Frequency Based Binder (FBB) creates input/output connection patterns for each operation in the DFG. A usage frequency (a measure of the reuse of common connection patterns) is used to establish the order in which patterns will be considered for binding to units. Binding costs consider the tradeoffs of adding functionality to existing components versus instantiating new components.

ii. The next prompt you will be given is to set the level of binding debug information printed:

```
Print allocation/binding debug info (y/n)?
```

If you respond 'y', a file <design-name>.alloc is created. Debug information generated during the resource binding phase will be written to this file.

If binding debug information is to be printed, and you have selected the 'frequency based binder', the following prompt is issued:

```
  Print pattern creation info (y/n)?
```

A response of 'y' prints information about each DFG node encountered and the pattern entry created for that DFG node to the file <design-name>.alloc.

The following prompt will then be issued:

```
Allocation/Binding debug level
  0 = print no debug information (default)
  1 = print register binding information
  2 = print unit creation and binding information
  3 = print pattern binding information
  4 = print connection binding information
=>
```

These debug levels print increasingly detailed information about the binding process of the selected Binder.

iii. Frequency Based Binder options

```
a. Reset partial design (y/n)?          (default value: 'y')
   =>
```

In the Allocation phase, component classes and instances are created in the GENUS partial design representation to be used by the Scheduler. If these are not removed from the Partial Design representation, the FBB will consider this allocation constraint as the current Partial Design and modify or add to it as needed.

By responding 'y' to this prompt, the GENUS partial design representation is reset, and the FBB will begin from scratch. The allocation constraints will be enforced via the generated schedule.

b. Treat storage elements as units (y/n)? : (default value: 'n')

This option determines whether registers are considered as sharable units for which patterns will be generated. The variable 'reg_as_unit' is set appropriately.

If you respond 'y' to this prompt, the following prompt is then issued:

Use best fit strategy for register sharing (y/n)? :

This selects a best fit selection strategy when there is more than one alternative for a register binding. By default, a first-fit strategy is used.

c. Use OPERATION sources for operators (y/n)? : (default value: 'n')

When determining frequency of patterns, this option determines whether to use the operation node which produces the input value for the current operation node or a variable access. The variable 'op_srcs' is set appropriately.

d. Restrict operator merging to same class (y/n)? (default value: 'n')
=>

If this option is set, operator merging is restricted to the same operator class for function units. This prevents unrealistic merging of '+' and '*' operators, for example.

7. BIF unit based state table generator

Once scheduling is completed, the UNIT BASED BIF state table is generated.

8. Data path netlist generator

The 'netlist' module contains a VHDL structural netlist generator.

9. Control Logic Compiler

The Control Logic Compiler (written by Tedd Hadley) uses the BIF unit based state table generated after Resource Binding and generates a VHDL functional description.

The CLC is invoked within VSS using a system call. The script 'run_clc' is executed (this script file is in the directory <VHDL-directory>/bin/$arch.

A subdirectory 'cu' is created, and the file <design-name>.ubst is copied into this subdirectory. Tedd Hadley's state_table_to_vhdl program is invoked, which generates the VHDL functional description of the control logic. This program will ask you if you want to delete intermediate files produced during execution of the CLC - if no errors occur, these files can be deleted.

10. VSS Functional synthesis of control unit

In order to generate a structural netlist for the control unit, a second pass of VSS currently has to be run using the VHDL functional description produced by the CLC. The 'run_clc' script invokes 'vss_c_gc' in FUNCTIONAL mode to generate the structure of the control logic using VSS.

When asked to select the resource binder, use the 'flattened flowgraph' option.

11. CU/DP netlist merger

The netlist generated for the control logic will be appended to the file containing the data path VHDL structural netlist. A VHDL configuration statement is then generated to associate all components which have subcomponent expansions to the lower level entity/architectures.

If a syntax error is detected in the VHDL input file, an error message of the following form is printed:

```
Error message: syntax error
  source line number: 18
  yytext = singal
  Lookahead token number: 276

Graph Compiler: syntax error in source program - compilation aborted
```

In this example, the keyword *signal* was misspelled. The source line of the error is indicated as well as the text buffer (yytext) for the current token.

NOTE: the results of previous compiler runs are overwritten by the current results. If previous results are to be saved, either the current input file should be renamed or the results to be saved should be moved to another directory.

A utility **cleanup** has also been provided to remove all files produced during the execution of the compiler for a given design. To execute this utility, type

```
% cleanup <design-name>
```

All files associated with compilation of the design <design-name> except the input source file (and allocation specification file) will be deleted.

4.4. Output Files

Upon successful completion, the VHDL Input Compiler will produce the following statistics and data files with the naming conventions shown:

<design-name>.st	- signal symbol table
<design-name>.nodes	- flowgraph node/net information
<design-name>.alloc	- Design Compiler allocation/binding information
<design-name>.sched	- Design Compiler scheduling information
<design-name>.obst	- symbolic microcode (control) specification
<design-name>.stats	- statistics for synthesis run
GC_components	- GENUS generic component classes used in the design
GC_instances	- GENUS generic component instances used in the design
GC_nets	- GENUS net interconnections used in the design
<design-name>.nl	- VHDL structural netlist for synthesized data path
<filename>.dgm	- flowgraph diagram output used for display

where <filename> can be

<block-name>_bef_gc	- flowgraph section for block before Graph Critic is invoked
<block-name>_final	- flowgraph section for block after Graph Criticism
<design-name><num>	- flowgraph which results from the application of Graph Critic rule firing <num>
<design-name>_final	- final interconnected and optimized flowgraph

5. Flowgraph Graphical Display Utility

5.1. Input File

The <filename>.dgm files created by the VSS Graph Compiler can used as the input files for the display utility. Each file contains a textual netlist description of the flowgraph generated by the Graph Compiler. The format of this netlist output for the generated flowgraph is described in Appendix C.

Similarly, the VHDL structural netlist description can be viewed graphically with the **xdp** tool.

5.2. Running the Display Program

A utility has been included which allows for the graphic display of the flowgraph generated by the VHDL compiler. This program must be executed within a X window environment. To enter such an environment, type the command:

```
% xdp [options] <design-name>
```

See the manual page in the back of this User's Manual for a description of the command line options.

5.3. Scanning the Diagram

Sections of the diagram can be examined by placing the mouse cursor within the display window and typing a single letter command followed by <return>. The manual page for the **xdp** command describes the available options.

6. Manual Pages

Included in the release are manual pages for the **vss** and **dp** programs. The files are located in the <VHDL-directory>/doc/man/man1 directory. These manual pages can be accessed by either:

(1) Placing them in a directory where other manual pages on your system are located, or

(2) Adding the release directory to your MANPATH directory by modifying/adding the following line to a user's *.cshrc* file:

```
setenv MANPATH "<current-path>:<VHDL-directory>/doc/man"
```

APPENDIX A
VHDL Language Subset

1. Introduction

The VHDL language syntax used to develop the VHDL Dataflow Compiler was taken from the IEEE Standard VHDL Language Reference Manual, Standard 1076B.

Each VHDL behavioral description used as input to the VHDL Synthesis System (VSS) must consist of a *design entity* composed of two major sections: the *entity block* and the *architecture body*. The entity block contains the specification of external input/output port connections to the hardware to be designed. The architecture body consists of a description of the hardware to be designed using either the **data flow** or **behavioral** description styles available in VHDL. The data flow style uses *concurrent signal assignment statements* to describe the flow of information between memory and gating elements.

A VHDL description using the behavioral description style consists of *process statements* and *concurrent procedure calls*. Process statements consist of one or more sequential statements (IF-THEN-ELSE, CASE, FOR loop, WHILE loop, variable assignment) which specify programs to be implemented in a microarchitecture consisting of a control unit and a data path.

2. Signal Declarations and Types

The following VHDL standard data types are supported:

```
BIT
BIT_VECTOR
BOOLEAN
INTEGER
```

For synthesis purposes, the following special types are defined:

```
subtype CLOCK is BIT
subtype SET is BIT
subtype RESET is BIT
```

VHDL signal declarations can occur in two sections of the behavioral description: within the entity block, where external port connections are declared, and within block statements of the architectural body, where internal connections and storage elements are declared. These declarations are of the form:

{ **signal**} <signal-name> : <mode> <type>

The <mode> attribute identifies the direction in which data flows at a port (IN, OUT, INOUT). We will define a signal to be of mode **internal** if it is not declared as a port in the entity portion of the VHDL description but is declared as a local signal within an architectural body. The <type> is one of the data types defined above.

As these declarations are processed by the Graph Compiler, entries are made into the symbol table to record the signal attributes.

3. Entity Block

The entity block is used to define external port connections for the hardware component to be synthesized. It has the following form:

entity *entity_name* **is**
  port (
    interface declaration section
)
end *entity_name*;

4. Architectural Body

The architectural body of the VHDL description has the following form:

architecture *design_name* **of** *entity_name* **is**
  declaration section
begin
  concurrent_statements
end *design_name*;

The architectural body may consist of one or more *concurrent_statements*. Concurrent statements are used to define interconnected blocks that jointly describe the overall behavior and/or structure of a design. The following VHDL concurrent statements are supported:

1) block statements
2) concurrent signal assignments
3) process statements

4.1. Block Statements

The primary VHDL construct used for the dataflow description style is the **block** statement. A block statement defines an internal block representing a portion of a design. It has the following syntax:

```
block_statement ::=
  block [(guard_expression)]
    block_header
    block_declarative_part
  begin
    block_statement_part
  end block;

block_header ::=
  [ generic_clause
  [ generic_map_aspect; ] ]
  [ port_clause
  [ port_map_aspect; ] ]

block_declarative_part ::=
  { block_declarative_item }

block_statement_part ::=
  { concurrent_statement }
```

The optional *guard_expression* defines an implicit signal GUARD of time BOOLEAN for simulation. If the guard_expression evaluates to TRUE, all signal assignments with a **guarded** qualifier appearing in the block_statement_partwill have their RHS evaluated, and a driver is placed on the event queue to update the signal values at the appropriate time. For synthesis, the guard_expression is used to specify a synchronous or asynchronous event which results in a signal update.

The *block_header* explicitly identifies certain values or signals that are to be imported from the enclosing environment into the block and associated with formal generics or ports.

The *block_declarative_part* defines all local signals, types and subtypes, constants, components and attributes.

One or more concurrent statements constitute the *block_statement_part*. Blocks may be hierarchically nested to support design decomposition. The block statement groups together other concurrent statements such as signal assignments which assign values to signals.

4.2. Concurrent Signal Assignments

4.2.1. Conditional Signal Assignment

The *conditional signal assignment* can occur in one of the following forms:

a) signal <= <waveform> ;

This is the simplest form of assignment statement where

<waveform> ::= <expression> { **after** <delay> }

b) signal <= **guarded** <waveform> ;

The *guarded assignment* involves the conditional assignment of the evaluated <waveform> to the signal based on the value of the **guard expression** which appears at the beginning of the enclosing VHDL block statement. For the purposes of CDFG generation and synthesis, a guarded signal assignment is used for signals declared with the **bus** or **register** qualifier.

```
c) signal <= { guarded }
                waveform1 when condition1 else
                waveform2 when condition2 else
                              .
                              .
                              .
                waveformN when conditionN else
                waveformN;
```

This statement corresponds to a nested if arrangement of assignments to the same signal based on different boolean conditions. The statement can be useful in representing an assignment to a signal based on prioritized conditions. For example, the statement

```
reg_A <=
        '0' after 20 ns when CLEAR = '0' else
        '1' after 20 ns when PRESET = '1' else
        DATA after 35 ns;
```

might be used to represent a register for which the CLEAR is of highest priority, followed by PRESET and CLOCKed assignment.

4.2.2. Selected Signal Assignment

The format of the *selected signal assignment* is as follows:

```
with <expression> select
  signal <= { guarded }
              waveform1 when choice1 ,
              waveform2 when choice2 ,
           .
           .
           .
              waveformN when choiceN ;
```

The choices are exclusive conditions (either integer or boolean values) such that only the waveform matching the value of the <expression> is evaluated and scheduled for assignment to the signal value.

4.3. Process Statement

The primary VHDL construct used for the behavioral description style is the **process** statement. A process statement defines an independent sequential process representing the behavior of some portion of the design. It has the following syntax:

```
process_statement ::=
  process [(sensitivity_list)]
    process_declarative_part
  begin
    process_statement_part
  end process;

process_declarative_part ::=
  { process_declarative_item }

process_statement_part ::=
  { sequential_statement }
```

The execution of a process statement consists of the repetitive execution of its sequence of sequential statements. After the last statement in the sequence of statements of a process statement is executed, execution will immediately continue with the first statement in the sequence of statements.

A *sensitivity list* may be specified for each process. By specifying a sensitivity list of one or more signals, the process statement is assumed to contain an implicit wait statement as the last in the sequence of statements. This wait statement will suspend execution of the process statement until an event (change) occurs involving one of the signals in the sensitivity list. The sensitivity list is ignored by the VSS synthesis tool.

The *process_declarative_part* defines all local signals, variables, types and subtypes, constants and attributes.

One or more sequential statements comprise the *process_statement_part*. The sequential statements which may appear in the description are listed below:

```
sequential_statement ::=
    wait_statement
  | signal_assignment_statement
  | variable_assignment_statement
  | procedure_call_statement
  | if_statement
  | case_statement
  | loop_statement
  | next_statement
  | exit_statement
  | return_statement
  | null_statement
```

5. Structured Modeling

Structured Modeling is a set of guidelines developed in conjunction with VSS system implementation for VHDL modeling to support synthesis. The quality of a design as well as the complexity of the synthesis process are directly related to the style of description chosen to represent a particular design model. Certain VHDL constructs or description styles are better suited to describe a particular design model than others. Because VHDL allows the designer several ways of describing the same functionality, it is important to set standard modeling practices for designers using VHDL. These standards should guarantee high quality of synthesized design, while divergence from the standard will result in simulatable but not optimal design.

5.1. Design Models

Our synthesis system supports four design models: *combinational logic*, *functional* descriptions (involving clocked components such as counters), *register transfer* (data path) descriptions, and *behavioral* (instruction set or processor) designs. These design models must be described using the *structural*, *dataflow*, and *behavioral* description styles provided by VHDL.

5.1.1. Combinational Logic

The design model for **combinational logic** consists of a network of logic gates. The most common method used to describe combinational logic designs is boolean equations. In this model, concurrent evaluation of all signal values is assumed. The VHDL dataflow model is used for the description of combinational logic. The VHDL **after** clause is used only for assignments made to output signals. This delay represents the maximum allowed delay from any input to the next particular output, and it will be used as a constraint during synthesis.

5.1.2. Functional Descriptions

The **functional** design model consists of combinational logic as well as storage elements (registers, counters). It may include a mixture of synchronous and asynchronous events for loading storage elements. It cannot be guaranteed that these events are mutually exclusive; an asynchronous event such as a register reset can occur concurrently with a synchronous load of the same register. The functional design can be described in VHDL using block and process statements. When modeling such a design, one or more functional blocks can be described with one block or process. Furthermore, several block statements could be used to describe exclusive functionality (synchronous and asynchronous behavior of the same functional block). The guard expression should contain only typed signals such as the clock signal or an asynchronous reset/set.

5.1.3. Register Transfer Designs

Register transfer descriptions involve the specification of operations to be performed within a processor for each machine state of a design. A common method for describing this behavior uses a **state table**. For each state, one or more triplets specify actions to be performed. Each triplet is composed of a *condition*, a *next state* specification, and a set of *operations*. The condition tests a boolean expression. Within each state, one or more conditions may evaluate to true. The actions corresponding to each true condition are performed in the state. If the result of the test is true, a specified set of operations or register transfers is performed. Finally, control is transferred to the specified next state upon completion of the current state operations.

In VHDL, block statements may be used to represent the state table using the following conventions:

(1) Every block represents a different state.

(2) The block guard specifies the clock, while the body of the block sets the state variable to the appropriate next state and performs operations under the desired conditions.

5.1.4. Behavioral Descriptions

An **behavioral** description allows the designer to describe the design as a black box with well defined interfaces. Variables within a description can be allocated storage by default, or the synthesis system can determine which variables require storage. As in the combinational model, input to output timing is expressed. Algorithmic design is modeled by VHDL process statements. Signal assignments are used only to represent output port assignments.

5.2. Modeling Guidelines

The following modeling practices are recommended when using VHDL for synthesis in the VSS system:

(1) Use the dataflow model for synthesis of combinational logic.

(2) Use an **after** clause only for assignments made to output signals. This delay represents the maximum allowed delay from any input to the next particular

output, and it will be used as a constraint during synthesis.

(3) One or more functional blocks should be described by one VHDL block statement. Several block statements could be used to describe exclusive behavior (synchronous and asynchronous behavior of the same functional block).

(4) The guard expression should contain only signals of type clock, set or reset.

(5) All signals should be typed. Signal types include clock, reset, set, test, data and control.

(6) Each state of a register transfer design should be described with block statements containing condition, next state assignment and all register transfers with the clock specified in the guard expression. Alternatively, a single process with a case statement can be used.

(7) Behavioral designs are modeled by VHDL process statements. Signal assignments are used to represent output port assignments. Signals may also be used to hold temporary values (for example, the swapping of register contents) in order to model concurrent events within the sequential process.

APPENDIX B
Allocation Specification File Format

```
pd_input_file ::=
     component_spec
     | component_spec
       pd_input

component_spec ::=
     op_class_spec
     op_type_spec
     bw_spec
     op_delay_spec
     num_inst_spec

op_class_spec ::=
     op_class : op_class

op_class ::=
     REL | ADD | LOG | MULT | SHIFT

op_type_spec ::=
     op_types : op_type_list

op_type_list ::= op_type
     | op_type, op_type_list

op_type ::=
     EQ | NEQ | LT | LEQ | GT | GEQ
     | ADD | SUB | OR | NOT | NAND | NOR | XOR | MULT
     | DIV | SHL0 | SHL1 | SHR0 | SHR1 | SHL | SHR

bw_spec     ::=
     bit_width : unsigned_integer

op_delay_spec ::=
     op_delay : real_number

num_inst_spec ::= empty
     | num_inst : unsigned_integer
```

Example

```
op_class: ADD
op_types: ADD
bit_width: 8
op_delay: 1.0

op_class: ADD
op_types: ADD, SUB
bit_width: 8
op_delay: 1.0

op_class: MULT
op_types: MULT
bit_width: 8
op_delay: 1.0

op_class: MULT
op_types: DIV
bit_width: 8
op_delay: 1.0

op_class: LOG
op_types: AND, OR
bit_width: 8
op_delay: 1.0
num_inst: 2
```

APPENDIX C
Flowgraph Netlist Specification

File Format

```
dgm_file ::=
    main_graph
    sub_blocks

main_graph ::=
    node_records

sub_blocks ::=
    empty
    | sub_block
      sub_blocks

sub_block ::=
    sub_block identifier block_id_num
    node_records

node_records ::=
    node_record
    | node_record
      node_records
```

Node Record Format

```
node_record ::=
    node_info_line
    node_inputs
    node_outputs

node_info_line ::=
    * node_num block_id_num num_top_inps num_l_inps num_bot_outps num_r_outps name
```

where

block_id_num identifies the type of node as follows:

1-400	constants, DF_START, DF_END	3601-4000	SWITCH_BOX
401-800	READ_PORT	4001-4400	IF_TEST
801-1200	READ_REGISTER	4401-4800	IF_JOIN
1201-1600	WRITE_PORT	4801-5200	input, output ports
1601-2000	WRITE_REGISTER	5201-5600	memories
2001-2400	CHOOSE_VALUE	5601-6000	inverter gate
2401-2800	operator nodes	6001-6400	and gate
2801-3200	DELAY nodes	6401-6800	or gate
3201-3600	READ_SIGNAL	6801-7200	tri-state
901-1000	WRITE_SIGNAL	7201-7600	nand gate

Input Record Format

```
node_inputs ::=
    empty
    | node_input
      node_inputs
```

node_input ::=
    input_port_num block_id net_num bit_width net_name

Output Record Format

```
node_outputs ::=
    empty
    | node_output
      node_outputs
```

node_output ::=
    net_num block_id num_outputs bit_width net_name dest_list

```
dest_list ::=
    dest_rec
    | dest_rec
      dest_list
```

dest_rec ::=
    dest_block_id dest_port_number bit_width

Example

```
* 7    7  1  0  1  0 STMT_BLK
 12    7  11 1   7-ctrl0
 13    7 1 1   7-out13  8  14  1
* 8    8  1  0  1  0 BLK_END
 14    8  13 1   8-ctrl0
 16    8 1 1   8-out16  2  15  1
* 9    9  1  0  1  0 CNT_UP
 18    9  17 1   9-ctrl0
 19    9 1 1   9-out19  10  20  1
* 10   10  1  0  1  0 STMT_BLK
 20   10  19 1  10-ctrl0
 21   10 1 1  10-out21  11  22  1
   ...
* 1    1  0  0  4  0 CF_START
  1    1 1 1   1-out1  3  2  1
  9    1 1 1   1-out9  6  10  1
 17    1 1 1   1-out17  9  18  1
 25    1 1 1   1-out25  12  26  1

sub_block STMT_BLK 7
* 42   15  0  0  1  0 DF_BLK_START
 42   15  3   1 42-dep_out 403  169  1  3202  175  1  404  176  1
* 40  403  1  0  1  0 DATA
169  403  42   1 40-dep0
 41  403  1   4 40-out 1602  163  4
* 41 1602  1  1  1  0 LIM
163 1602  41   4 41-data0
165 1602  40   1 41-clock1
 43 1602  1   1 41-dep_out  16  173  1
   ...

sub_block STMT_BLK 10
* 65 3203  1  0  1  0 EN
356 3203  86   1 65-dep0
 65 3203  1   1 65-out 2404  261  1
* 73 4005  1  0  1  0 SWITCH_BOX
291 4005  79   4 73-data0
 74 4005  1   1 73-out 1603  352  1
* 79 3204  1  0  1  0 CONSIG
358 3204  86   1 79-dep0
 79 3204  2   4 79-out 4006  321  4 4005  291  4
   ...
```

blank page.