VHDL Design Representation in the VHDL Synthesis System (VSS)

by

Joseph S. Lis
Daniel D. Gajski

Technical Report 89-15

Information and Computer Science
University of California at Irvine
Irvine, CA 92717
(714) 856 7063

Abstract: This report describes the use of the VHSIC Hardware Description Language (VHDL) for synthesis in the VHDL Synthesis System (VSS). The corresponding internal representation of VHDL used in VSS will be described. We will illustrate the use of this representation to capture characteristics of four different design models (combinational, functional, register transfer, behavioral). Algorithms for compiling the VHDL description into the design representation will be discussed.

TABLE OF CONTENTS

LIST OF FIGURES

1. Introduction

Behavioral synthesis involves the translation of a behavioral description into a structural description. A behavioral description models the design as a "black box", describing its outputs as a function of its inputs and time. A *hardware description language* (HDL) provides a method of specification for the designer so that a synthesis tool can be supplied with sufficient information about the intended functionality of a design. Usually, the HDL consists of high-level, programming language statements which allow for the specification of the design's flow of control as well as assignment statements which specify operations and data transfers to be performed. In addition, a true hardware description language should be able to describe the structure of a design implementation in terms of a set of interconnected components from a given library.

Simulation languages classified as HDLs are used to model hardware. One such language is the VHSIC Hardware Description Language (VHDL) [VHDL87], an IEEE and DoD standard. The goals of simulation are to predict accurately the voltage values on all nets of a design at any given time and to verify timing relationships between changes on these nets. The description used for simulation does not have to be minimal, elegant or implementable as long as it produces correct behavior.

Since designs can be described in several ways and at several different levels of abstraction using languages such as VHDL, a synthesis tool must know the intent or semantics of the description in order to produce hardware which performs the desired function. A consistent *modeling* practice is required if the same description is to be used for synthesis and simulation. A **structured modeling** methodology has been proposed [LisGa89b] which recommends practices for writing synthesizable descriptions using VHDL. Adherence to these standards will result in a high quality design.

A *design representation* or *data base* is the internal representation used by a synthesis tool. It organizes information extracted from the input specification necessary for synthesis. This representation is created, manipulated, and optimized by the system so that a netlist or other output specification can be produced.

One common design representation used in several synthesis systems is the **control/data flow graph** [OrGa86]. The *control flow graph* represents sequencing information. Each "state" in the behavioral description is represented as a sequence of actions to be performed, and based on the evaluation of a condition, the next state to which execution is to be advanced is indicated. Control dependencies implied in the semantics of the behavioral description (for example, loop and if-then-else constructs) are preserved in the control flow graph.

The sequence of actions to be performed (arithmetic, logical, shifting operators) is represented using *data flow graphs*. A data flow graph indicates data dependencies that exist between variable accesses in assignment statements. The data flow graph exposes the parallelism in the input description. A control flow node representing a state will have a data flow graph associated with it.

1.1. Motivation

In synthesis, we are interested in generating a structural description of components from a given library from a behavioral description. Here, we are interested in properly connecting all pins on all components instead of observing signal values on some of the pins. The behavioral description must be parsed into a design representation which can be operated on by a variety of synthesis tools. This design representation should be well defined and should capture uniquely the functionality and intention of several equivalent behavioral descriptions in a format appropriate for synthesis. The representation must allow for the transformation of behavioral information (simulatable functionality) to structural information (library components and their attributes).

In this report, we will describe a control/data flow graph representation used in the VHDL Synthesis System (VSS) [LisGa88] [LisGa89a]. We will identify how the VHDL language can be used for synthesis in VSS. Through the use of signal typing and attribute annotations, we will show how a VHDL description for simulation can be enhanced to provide necessary information for synthesis. The structural, dataflow and behavioral description styles of VHDL will be investigated.

We will show the corresponding internal representation (control and/or data flowgraph) produced as the VSS input compiler parses each VHDL statement. The various interpretations of VHDL statements used to represent characteristics of each of the design models mentioned in our structured modeling methodology (combinational, functional, register transfer, behavioral) will be illustrated. Algorithms for compiling the VHDL input description into this representation will be discussed.

2. VHDL Design Models

2.1. Design Hierarchy

The *design entity* is the primary hardware abstraction in VHDL. It represents a portion of the hardware design that has well-defined inputs and outputs and performs a well-defined function. A design entity may represent an entire system, a sub-system, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between. A *configuration* can be used to describe how design entities are put together to form a complete design as shown in Figure 1.

A design entity may be described in terms of a hierarchy of *blocks*, each of which represents a portion of the whole design. The top-level block in such a hierarchy is the design entity itself; such a block is an *external* block that resides in a library and may be used as a component of other designs. Nested blocks in the hierarchy are *internal* blocks, defined by **process** or **block** statements. A structural, dataflow or behavioral description style can be used to express the functionality of an internal block.

Successive decomposition of a design entity into components, and binding of those components to other design entities that may be decomposed in like manner, results in a hierarchy of design entities representing a complete design. Such a collection of design entities is called a *design hierarchy.* The bindings necessary to identify a design hierarchy can be specified in a configuration of the top-level entity in the hierarchy. The design hierarchy concept is illustrated in Figure 1.

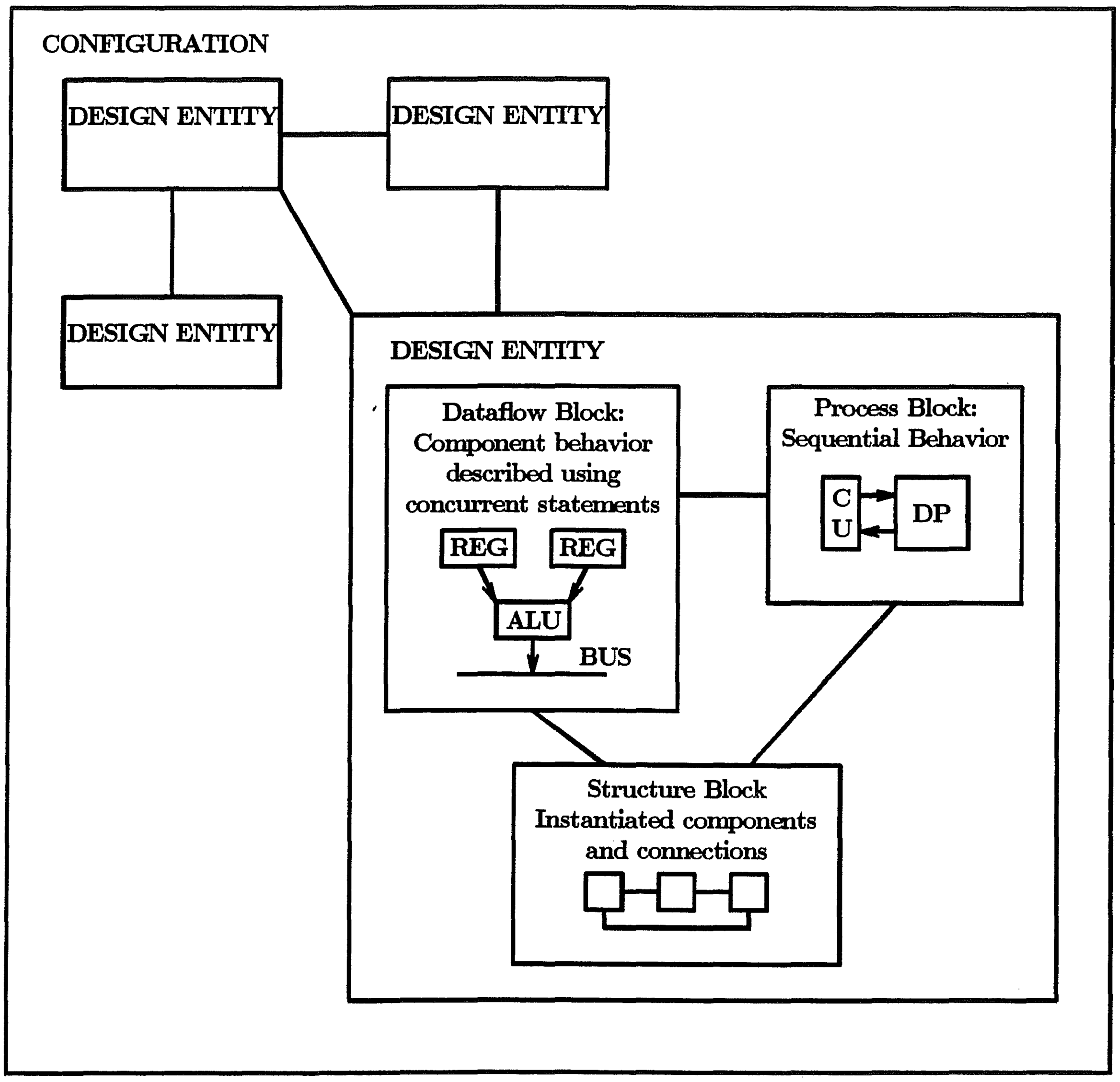

CONFIGURATION
DESIGN ENTITY
DESIGN ENTITY
DESIGN ENTITY
DESIGN ENTITY
Dataflow Block: Component behavior described using concurrent statements
REG
REG
ALU
BUS
Process Block: Sequential Behavior
C U
DP
Structure Block Instantiated components and connections

Figure 1: VHDL Design Hierarchy

A VHDL description which represents such a design hierarchy is shown in Figure 2. Each design entity description is 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 defines the body (structure and/or behavior) of a design entity. It specifies the

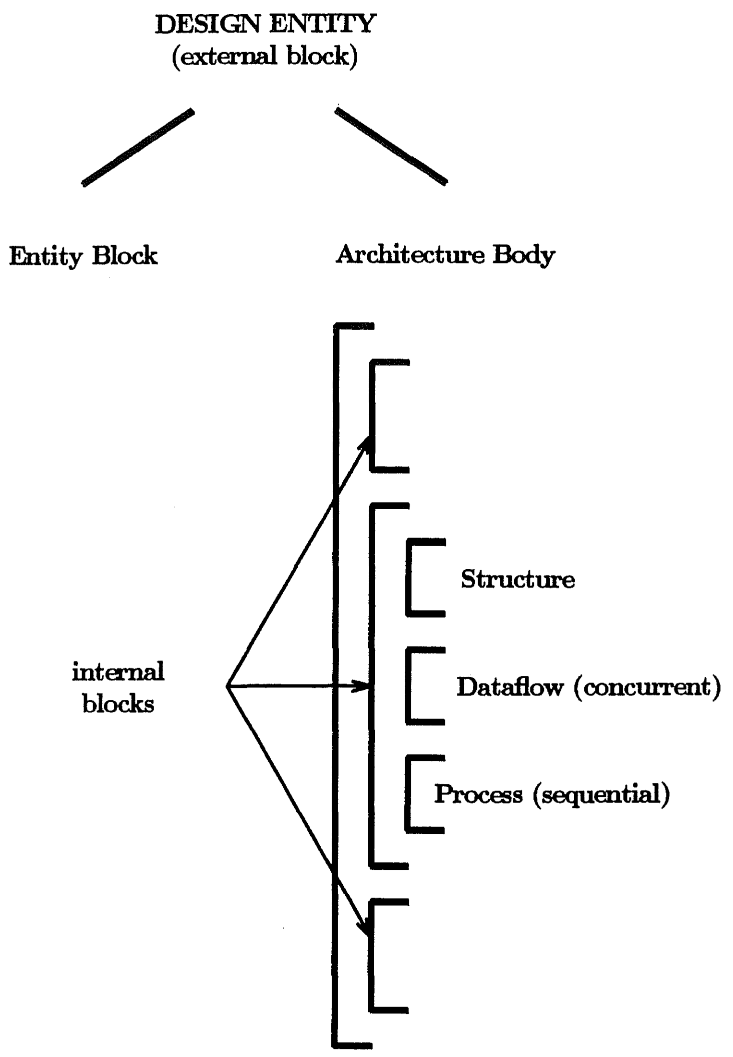


Figure 2: VHDL Design Entity Block Structure

relationships between inputs and outputs of the design entity, and may be expressed using a mixture of the three styles mentioned previously (structural, dataflow, behavioral).

2.2. Design Model

Figure 3 illustrates the underlying design model assumed for a VHDL description [Preas88]. A design is composed of communicating processing elements (PEs). Each PE consists of a Control Unit (CU) and Datapath (DP). Because a process statement may require one or several machine cycles (states) to execute the desired function, the microarchitecture implementation uses the DP to perform computations and the CU to

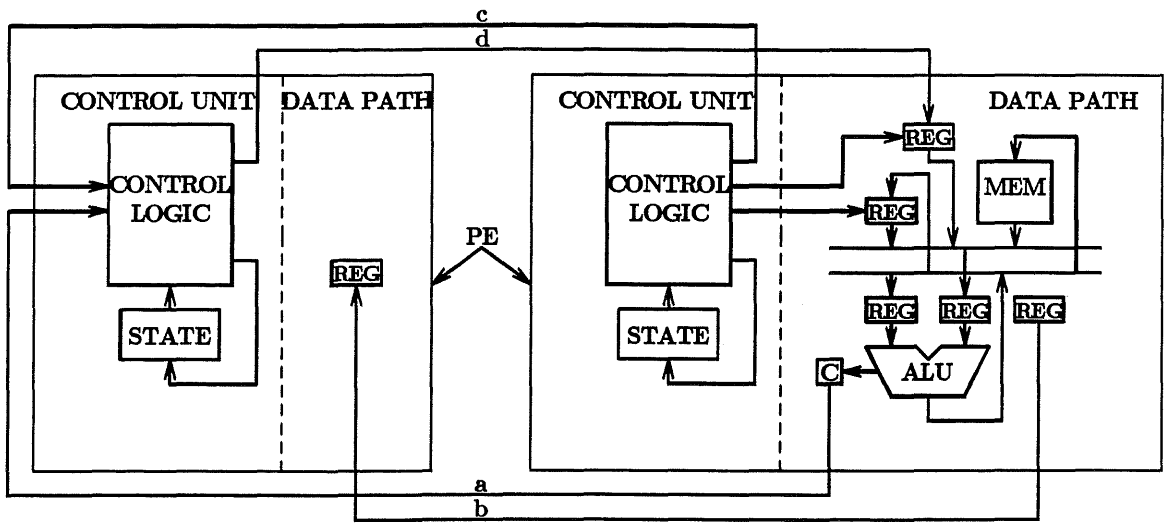


Figure 3: VHDL Design Model

sequence the machine through the necessary states and control the operations performed in the DP for each state. The CU contains a state register for storing the current state of the machine and control logic which controls the DP and communicates with other PEs. The DP consists of storage elements (registers, counters, memories) and functional units (ALUs, shifters, multiplexers) connected through sets of buses.

Access to registers, units or I/O ports is controlled by the CU. If several buses are used as sources to a storage or functional unit, a selector controlled by the CU must be added to the input. Some DP models use only point-to-point connection with selectors only and no buses. Processes also communicate via global signals. PEs communicate through DP ports to the CU or DP (nets *a* and *b* in Figure 3) or through CU ports to the CU or DP (nets *c* and *d*).

Note that in this model, an adder may be represented as a PE with no CU but with a DP (having one output port, two input ports, and no storage elements). Similarly, a flip-flop can be modeled as a DP with no functional units or as a CU with no DP and no control logic. Thus, this model is complete in the sense that it can model any synchronous digital system.

2.3. Design Model Representation

The three description styles (behavioral, dataflow, structural) use concurrent statements to describe a portion of the complete design model shown above. Each concurrent statement in a VHDL description may be used to describe a piece (one or more components) of a design. Alternatively, more than one statement can be used to describe the functionality of the same design section if the behaviors are non-overlapping (exclusive).

The design sections represented by the concurrent statements communicate via global signals. These signals are defined in the declaration section of the architecture body. A global signal may be read (input) to several blocks or processes, but should be written to (updated by) only one block or process at any given time. In the event that it is desirable to have more than one active driver for a signal simultaneously (to model a bus, for example), a *resolution function* must be written and associated with the signal to determine its proper value for simulation.

2.3.1. Behavior

A VHDL description using the behavioral style consists of *process statements* and *concurrent procedure calls*. Usually, process statements represent programs to be implemented in a microarchitecture which uses the complete control unit/data path design model. Variables within a process may represent storage components or interconnect wires. Local signals are used to communicate between the CU and DP.

Interprocess communication follows these conventions:

(1) The following subtypes are defined for descriptions to be used for synthesis:

subtype data **is** BIT;

subtype control **is** BIT;

Signals of type data are used to interface with the data path. Signals of type control interface with the CU.

(2) By default the following signal types/accesses are allowed:

Input
- signal/port reads within the data path description
- conditional bit signals input to the descriptions of control logic

Output
- constant signals output from control logic (boolean, binary, integer)
- computed signals output from DP

Timing is expressed as a part of the output signal assignments. Data computations within the process are made with variable assignment statements.

2.3.2. Dataflow

Dataflow descriptions consist of *concurrent signal assignment statements.* They describe only the data path portion of the VHDL design model. The data path is a structure of components, where each component is described by one or more statements.

2.3.3. Structure

The VHDL structural design style utilizes *component instantiation* and *generate* statements. Here, the data path portion of the design model is described through the instantiation and interconnection of component primitives or previously defined design entities.

2.4. Mixture of VHDL Design Styles

This section illustrates a mixture of the VHDL structural, dataflow and behavioral description styles in a single description. Figure 4 shows a block diagram for a controlled counter functional description adapted from [Arms89].

The operation of the controlled counter can be described as follows. On the rising edge of the STRB signal, an internal control register CONREG is loaded with the value on CON. The CONREG value is decoded to perform one of four functions: clear the counter, load a limit register, count up to a limit, or count down to a limit. The counter runs synchronously under an input clock, and the counting functions are enabled by the

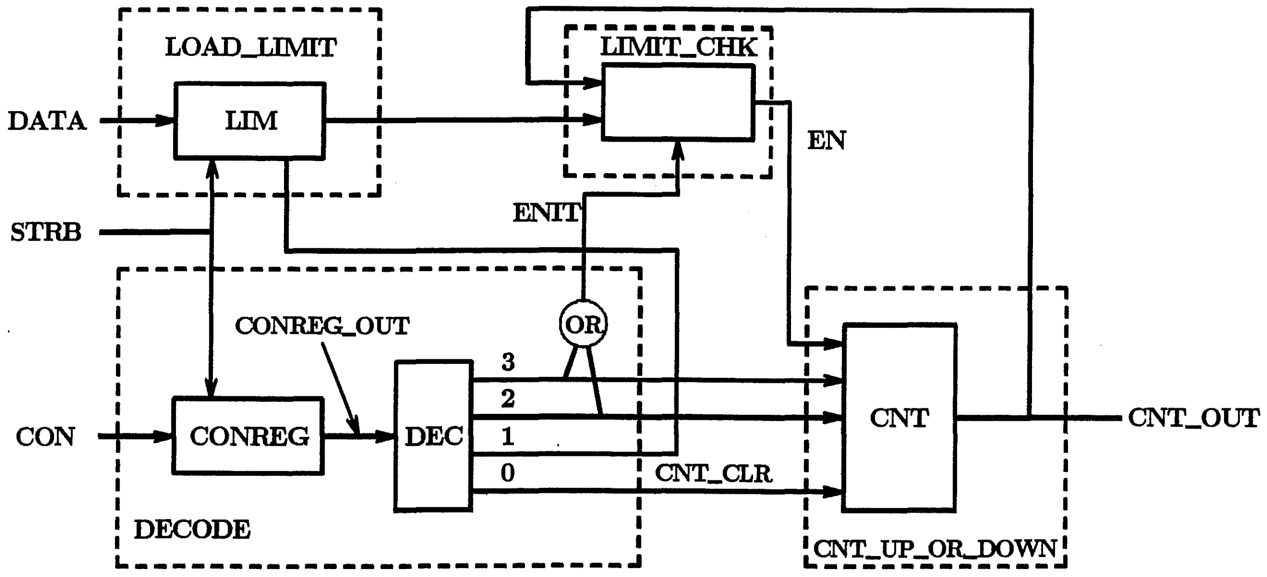


Figure 4: Controlled Counter Block Diagram

internal signal EN. The DATA value is loaded into the limit register LIM on the falling edge of STRB if the control register contains the value '00'.

The VHDL description is shown in Figure 5. This description consists of four block statements, each of which describes a portion of the design: the decoding of the CONREG value, the loading of the limit register (LIM), the asynchronous clear the synchronous up/down count of the counter (CTR), and a limit test.

The DECODE block statement describes the functionality of more than one functional block (the CONREG register and the decoder). A structural description style is used which specifies component declarations, interconnect signal declarations, component instantiations, and component interconnection (via the *port map* clause of the component instantiation statement).

A dataflow description style is used for the LOAD_LIMIT and CNT_UP_OR_DOWN blocks. The block guard is used to enable an update of the LIM and CNT register values. Note that these descriptions carry no information about the structure of the components to be used in the implementation, only the behavior.

The LIMIT_CHK block is described behaviorally with a process statement. This particular description involves only the data path portion of the design model.

```
entity CONTROLLED_CTR is
  port (
      CLK,STRB: in BIT;
      CON: in BIT_VECTOR(1 downto 0);
      DATA: in BIT_VECTOR(3 downto 0);
      CNT_OUT: out BIT_VECTOR(3 downto 0));
end CONTROLLED_CTR;

architecture MIXED of
  CONTROLLED_CTR is

  subtype nibble is BIT_VECTOR(3 downto 0);
  signal CONSIG: nibble := B"0000";
  signal LIM: nibble register := B"0000";
  signal ENIT: BIT := '0';
  signal EN: BIT := '0';
  signal CNT: nibble register := B"0000";
  signal CNT_CLR: BIT;

begin

DECODE: block (STRB = '1')

  component reg
    port (D: in BIT_VECTOR(1 downto 0);
        CLK: in BIT;
        Q: out BIT_VECTOR(1 downto 0));
  end component;
  component decoder
    port (D: in BIT_VECTOR(1 downto 0);
        Q: out BIT_VECTOR(3 downto 0));
  end component;
  component or2
    port (A,B: in BIT;
        O: out BIT);
  end component;
  signal CONREG_OUT: BIT_VECTOR(1 downto 0);

begin

  CONREG: register
    port map (CON,CLK, CONREG_OUT);
  DEC: decoder
    port map (CONREG_OUT,CONSIG);
  OR_1: or2
    port map (CONSIG(2),CONSIG(3),ENIT);
  CNT_CLR <= CONSIG(0);

end block DECODE;

LOAD_LIMIT: block (CONSIG(1)='1' and STRB='0'
        and not STRB'STABLE)
begin

  LIM <= guarded DATA after 10 ns;

end block LOAD_LIMIT;

CNT_UP_OR_DOWN: block ((CLK = '1' and
            not CLK'STABLE) or (CNT_CLR = '1'))
begin

  CNT <= guarded
      B"0000" after 5 ns when CNT_CLR = '1' else
      CNT when EN = '0' else
      CNT + B"0001" after 12 ns
          when CONSIG(2) = '1' else
      CNT - B"0001" after 12 ns
          when CONSIG(3) = '1' else
      CNT;

end block CNT_UP_OR_DOWN;

LIMIT_CHK: process (ENIT,CNT)
  begin

  if ((CNT /= LIM) and (ENIT = '1')) then
    EN <= '1' after 12 ns;
  else
    EN <= '0' after 5 ns;
  end if;

end process LIMIT_CHK;

CNT_OUT <= CNT;

end MIXED;
```

Figure 5: VHDL Description of Controlled Counter

3. VHDL Design Representation in VSS

This section of the report describes how each VHDL statement is processed by the VHDL Synthesis System (VSS) in order to generate and maintain an internal representation appropriate for synthesis. The **control/data flow graph (CDFG)** which is used as this internal representation is constructed as each statement is parsed. The portions of data and control flow graphs corresponding to the statements in a block or process are appropriately interconnected according to the design style used in the VHDL description.

3.1. Structural Description Style

A designer can specify an initial design, fully or partially, using a structural description mixed with behavior. When sections of the design are described using structural VHDL (for example, previously synthesized modules), these portions are copied intact to the output produced by the VSS system. The partial structural description is enhanced with additional components necessary to implement the sections of the design described using the data flow and behavioral styles.

When synthesis is completed, the VSS system produces a VHDL structural description of the design, using component declarations and instantiations derived from an Intelligent Component Data Base (ICDB) [Chen89]. VHDL behavioral models for these components are available from the data base. The use of VHDL as a netlist format in the VSS system is described in Appendix B.

3.2. Dataflow Description Style

The dataflow description style emphasizes the flow of information between storage and gating elements.

3.2.1. Concurrent Statements

Concurrent statements are used to define interconnected blocks (components, possibly of different complexity) that jointly describe the overall behavior or structure of a design. Concurrent statements execute asynchronously with respect to each other. The following concurrent statements are found in VHDL:

```
concurrent_statement ::=
    block_statement
  | process_statement
  | concurrent_procedure_call
  | concurrent_assertion_statement
  | concurrent_signal_assignment_statement
  | component_instantiation_statement
  | generate_statement
```

3.2.1.1. Block Statement

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_part will 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 [VHDL87]. The block statement groups together other concurrent statements such as signal assignments which assign values to signals. Nested blocks are flattened for synthesis to facilitate resynthesis with optimization.

The flow graph representation for a block statement in shown in Figure 6. It consists of BLK_BEG and BLK_END demarcation nodes, and a STMT_BLK node which represents the body of the block statement. The data flow graphs generated for

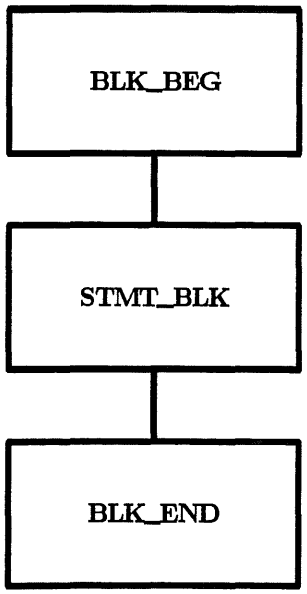


Figure 6: Block Statement Flowgraph Representation

each concurrent statement appearing in the block are associated with the STMT_BLK.

3.2.1.2. Signal Assignment

A signal assignment statement is used to assign or update values for a signal driver. The basic format of an assignment statement is the following:

target <= [**guarded**] <RHS-expression>

Each assignment made to a target or left hand side (LHS) signal/variable is represented by a WRITE node in the flow graph. Similarly, each access of a signal or variable appearing as a part of the right hand side (RHS) expression of an assignment statement is represented by a READ node.

READ and WRITE nodes for signals of can be of type PORT, REGISTER or WIRE (WIRE is the default for any variable declared as a SIGNAL). If a signal is of mode *internal* (that is, it was declared locally within some block statement) and a WRITE and READ node for that signal are connected when DFG sections are merged, the nodes can be coalesced, producing a signal net of type WIRE.

3.2.1.2.1. Conditional Signal Assignment

The *conditional signal assignment* statement has the following syntax:

```
signal <= [ guarded ] { <waveform> when <condition> else }
            <waveform> ;
  <waveform> ::= <expression> [ after <delay> ]
```

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

a) signal <= <waveform> ;

This is the simplest form of assignment statement. The VHDL simulator interprets this statement as a directive to compute the value of <expression> and schedule the activation of this driver for the signal value at time <current-simulation-time> + <delay> (if no delay is specified, the driver is activated immediately).

From the CDFG perspective, a dataflow graph is constructed for the RHS expression, and the result is input to a WRITE node for the signal. Associated with each graph arc (connection) is a **signal type** (bus, register, port, wire), **mode** (in/out/inout (for ports only), internal), **number of bits**, and **representation** (integer, floating point, 1's complement, 2's complement, sign/magnitude). The optional delay specification indicates the time which elapses between the READ of all signals/variables which appear on the RHS of the assignment statement and the appearance (WRITE) of the updated expression value at the register/port/wire represented by the signal. Figure 7 shows a typical signal assignment statement and the corresponding flowgraph with delays.

```
entity EXAMPLE is
 port (A,B: in BIT_VECTOR(0 to 3);
    ...
architecture EX of EXAMPLE
  is
  ...
  signal C: BIT_VECTOR(0 to 3);
  ...
  A <= B + C after 3 ns;
  ...
end EX;
```

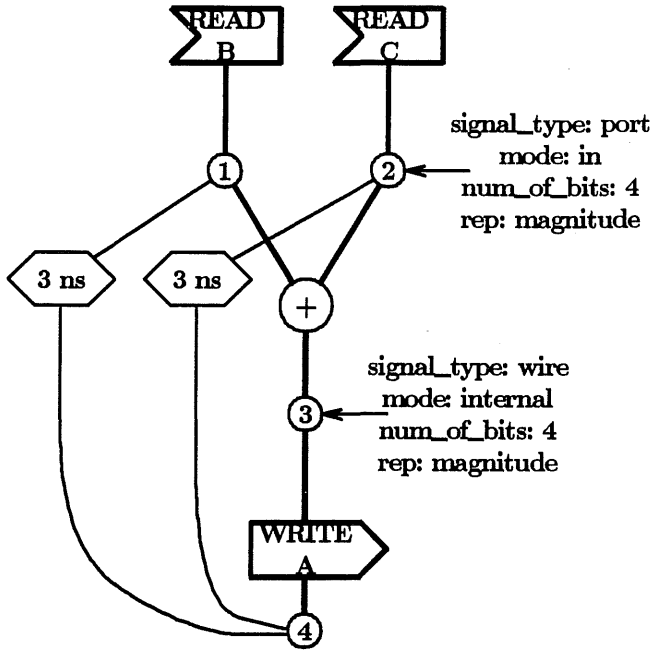


Figure 7: A Simple Conditional Signal Assignment

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. When the guard expression evaluates to TRUE, the VHDL simulator activates the signal driver and places its value on the simulator event queue so that the signal is updated at the specified simulation time.

For the purposes of CDFG generation and synthesis, a guarded signal assignment is used for signals declared with the **bus** or **register** qualifier. A data flow graph is generated for the RHS expression and is connected to the true input of a CHOOSE-VALUE node. The CHOOSE-VALUE node represents the selection of a data element based on the value of a guard (select) input. The guard input is a data flow graph representing the block guard expression. The output of the CHOOSE-VALUE node is used as the input to a WRITE node for the signal. Figure 8 shows an example of this construct.

If the signal is declared as a bus, the CHOOSE-VALUE will be mapped to a tri-state driver for the bus signal. If the signal is a register under a guard expression of type CLOCK, the CHOOSE-VALUE will be removed, and the select line will be connected to the clock input of the WRITE_REG node. The function of each signal appearing in the guard expression is determined by its signal type. In the case of multiple signals in the guard expression (clock and set, for example), an optimization step will connect each signal to the appropriate control input.

```
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 VHDL simulator will evaluate waveform/condition pairs in the order in which they appear and will schedule the assignment of the first waveform value to the signal when its associated condition evaluates to true.

```
entity CONTROLLED_CTR is
 port (CLK: in CLOCK;
      DATA: in BIT_VECTOR(3 downto 0);
      ...
architecture CONCURRENT of
  CONTROLLED_CTR is
  ...
  signal CNT: BIT_VECTOR(3 downto 0) register;
  ...
CNT_UP: block (CLK = '1' and not CLK'STABLE)
  begin
    CNT <= guarded CNT + "0001" after 10 ns;
  ...
end CONCURRENT;
```

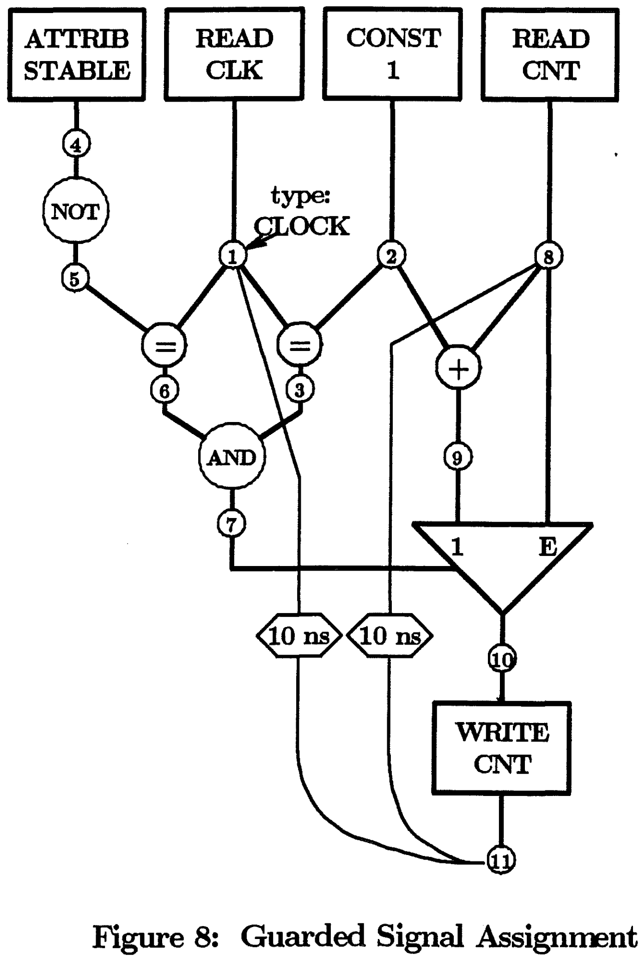


Figure 8: Guarded Signal Assignment

```
block (CLEAR = '0' or PRESET = '1' or CLK = '1')
  begin
    reg_A <= guarded
            '0' after 20 ns when CLEAR = '0' else
            '1' after 20 ns when PRESET = '1' else
            DATA after 35 ns when CLK = '1' else
            reg_A;
  end block;
```

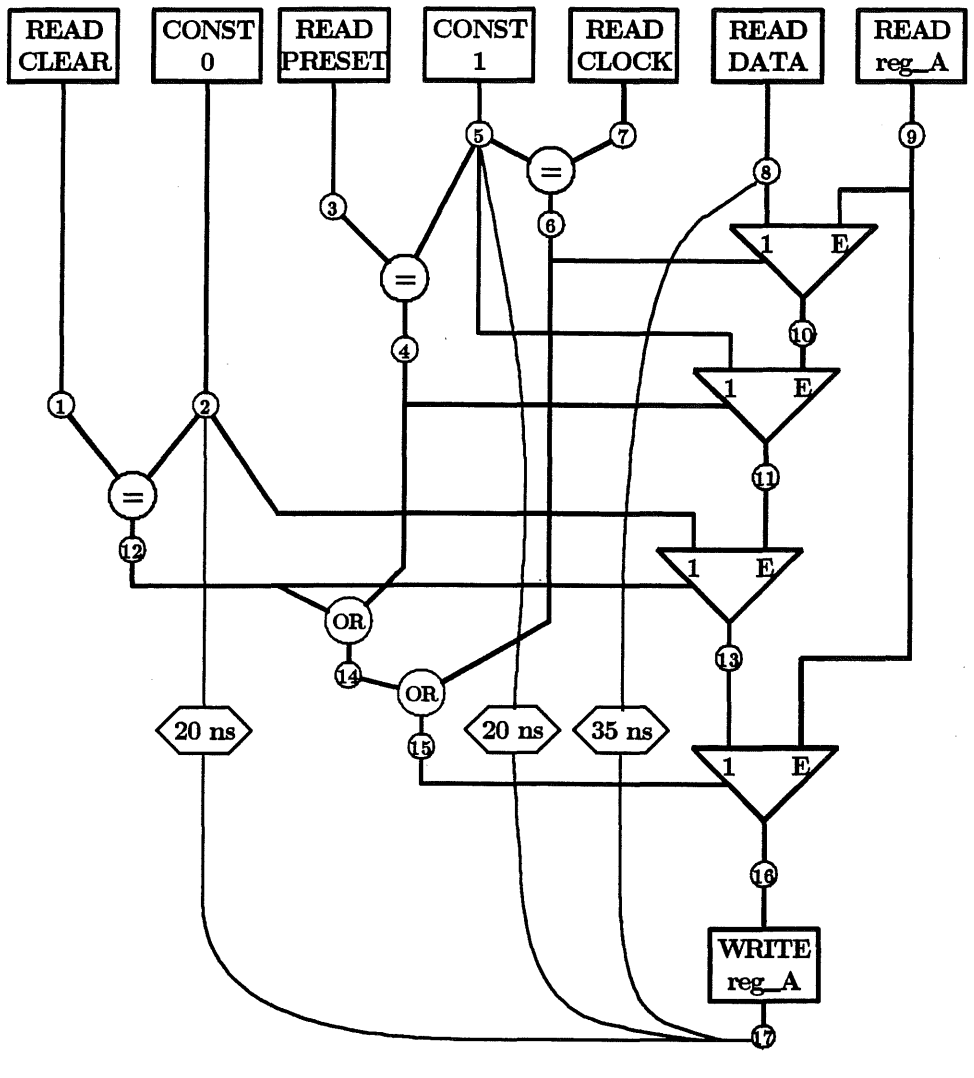


Figure 9: Conditional Signal Assignment

The conditional assignment statement can be useful in representing an assignment to a signal based on prioritized conditions. For example, the statement in Figure 9 might be used to represent a register for which the CLEAR is of highest priority, followed by PRESET and CLOCKed assignment. Figure 9 shows the flowgraph generated for the statement.

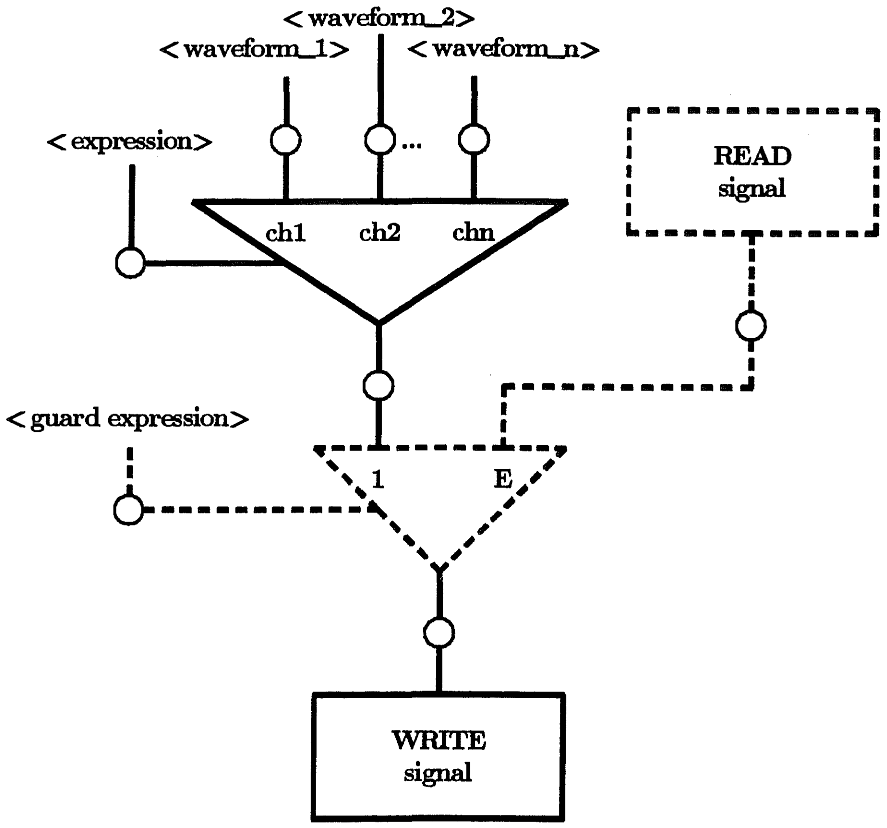
A chain of CHOOSE-VALUES is constructed to form the data flow graph for the nested if construct. The bottom most CHOOSE-VALUE is guarded by the first condition encountered, the CHOOSE-VALUE above the bottom one is guarded by the next condition, etc. The output of the bottom most CHOOSE-VALUE is connected to the WRITE node input.

3.2.1.2.2. Selected Signal Assignment

The format of the *selected signal assignment* is shown in Figure 10. This is equivalent to the case statement available as a sequential statement within the process construct. 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 by the VHDL simulator. Figure 10 shows the flowgraph generated for the general form of this statement.

The data flow graph construct associated with this statement is the multiple input CHOOSE-VALUE guarded by the <expression>. Each waveform will have a corresponding data flow graph generated for its expression value, and the guard test for each input will be stored in the input net.

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



Dashed lines represent flowgraph created for guarded selected signal assignment.

Figure 10: Selected Signal Assignment

3.3. Behavioral Description Style

A *behavioral description* is a sequentially executed, procedural style of code typical of common programming languages. A behavioral specification specifies, with any desired degree of precision, what a device does (its function) without specifying how it doe it (its structure) [CLSI87].

3.3.1. 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 [VHDL87].

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 in the next section.

The flow graph representation for an example process statement is shown in Figure 11. Note that a STMT_BLK node is a control node which has an associated data flow graph. These data flow graphs are constructed for sequential signal and variable assignment statements.

```
process
  begin
    while (stop = '0')
      PI := M(CR)(0 to 15);
      S := PI(3 to 15);
      case PI(0 to 2) is
        when 0 => CR := M(S);
        when 1 => Acc := Acc - M(S);
        ...
        when 6 => if (Acc < 0) then
                    CR := CR + 1;
        when 7 => stop <= '1';
      end case;
    end loop;
end process;
```

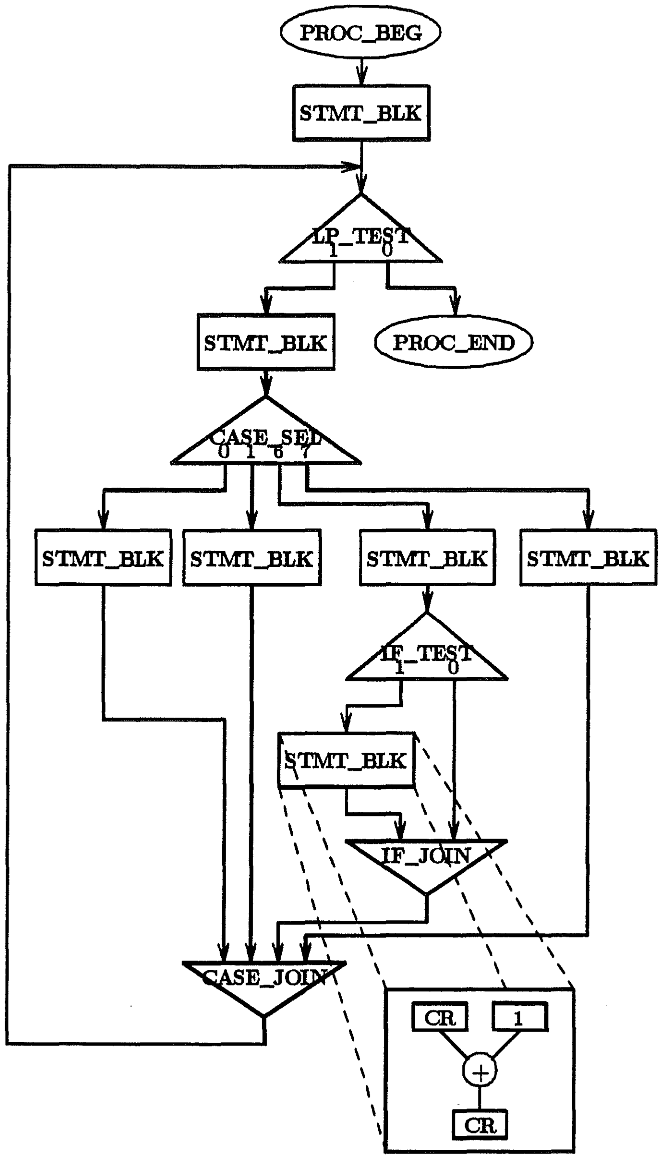


Figure 11: Process Statement Flowgraph Representation

3.3.2. Sequential Statements

The sequence of statements within a process statement may contain one or more of the following statement types:

```
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
```

As mentioned above, data flow graph sections for assignments of values to signals and variables are created as in the case of concurrent signal assignments and associated with STMT_BLK nodes. Control flow graph sections are created for each of the behavioral control constructs. These control flow graph sections are nested and interconnected to model the flow of control implicit in the sequential, behavioral description.

3.3.2.1. Signal Assignment

The syntax of the signal assignment statement for a sequential process is identical to form (a) of the conditional signal assignment in a concurrent block. A data flow graph similar to the representation generated for a concurrent signal assignment (see Figure 7) is created.

3.3.2.2. Variable Assignment

A variable assignment statement replaces the current value of a variable with a new value specified by an expression. The statement has the following syntax:

```
target := <expression> ;
```

This statement cannot use the *after* clause to specify timing relationships as in the signal assignment statement. A data flow graph is generated to represent the variable assignment.

3.3.2.3. If Statement

One construct used to model conditional execution in the VHDL process statement is the **if** statement. The if statement performs a conditional branch based on the value of a boolean signal.

The control flow graph section created to represent the if statement consists of three parts: (1) a TEST (or SELECT) node which selects the control branch to be taken based on the test signal; (2) for each control branch, one or more control nodes representing a sequence of statements to be performed in that branch; (3) a JOIN node which signifies the end of each conditional branch and connects to the flowgraph section for the next sequential statement. Figure 12 shows the control flow graph sections created for the if construct.

```
if boolean_expression
   then
      seq_of_statements_1
   else
      seq_of_statements_2
end if;
```

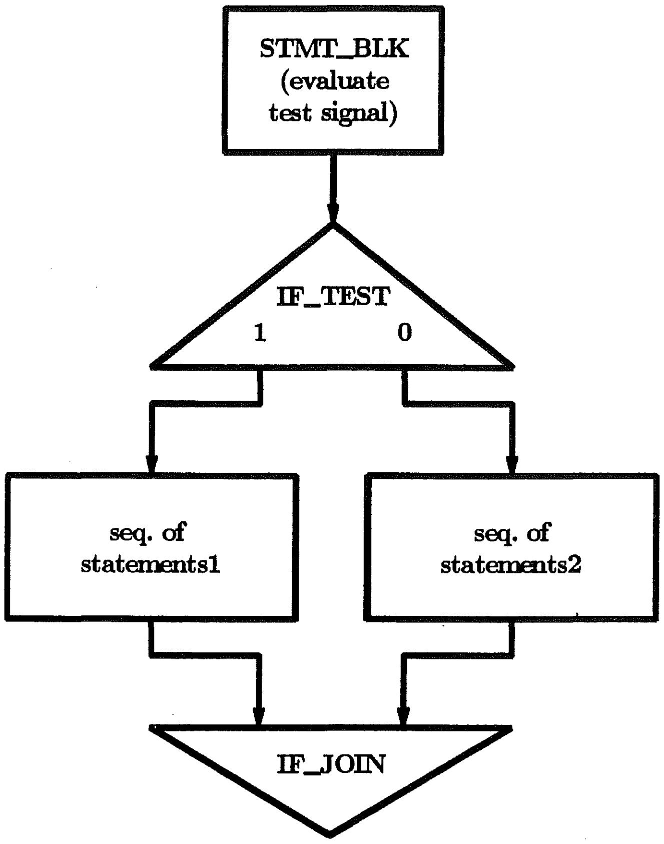


Figure 12: If Statement

3.3.2.4. Case Statement

The case statement selects between two or more conditional branches based on the value of an integer select signal. Figure 13 shows the flowgraph representation for the case statement.

```
case integer_expression is
   when choice_1 =>
      seq_of_statements_1
      ...
   when choice_n =>
      seq_of_statements_n
end case;
```

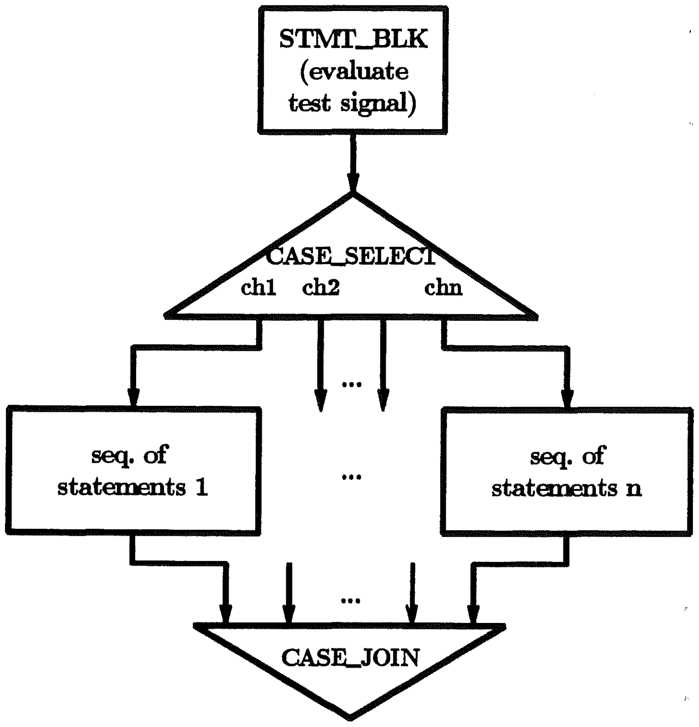


Figure 13: Case Statement

3.3.2.5. For Loop

A loop statement includes a sequence of statements that is to be executed repeatedly, zero or more times.

The **for** loop construct uses an index variable whose value steps through a a specified range for each iteration of the loop. The index variable is set to the first value in the range prior to entering the loop. A test is made to determine if the index value is within the range; if so, the loop body is entered. Once the loop body statements are executed, the index variable assumes the next value in the specified range, and control is returned to the loop entry test. If the test returns FALSE, control passes to the next sequential statement. Figure 14 shows the for loop representation.

```
for ident in discrete_range loop
   seq_of_statements
end loop;
```

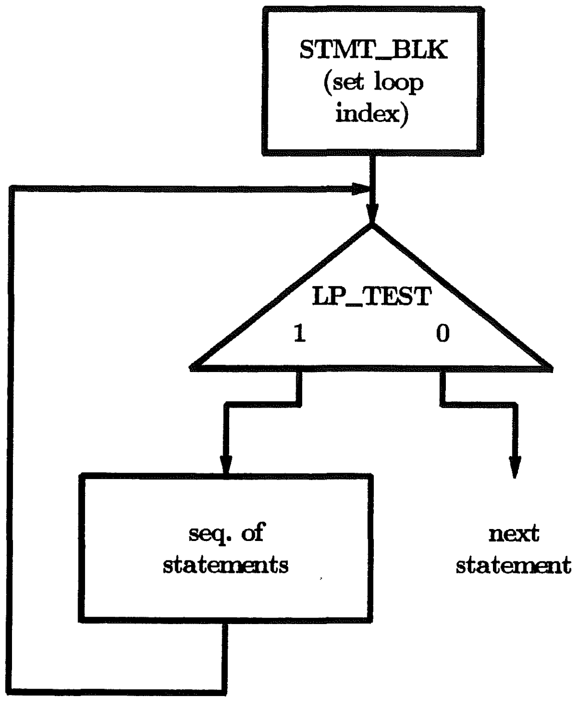


Figure 14: For Loop Statement

3.3.2.6. While Loop

The **while** loop construct tests a boolean condition, and if it is TRUE, passes control to the first control node of the flowgraph section implementing the sequence of statements for the loop body. Once the loop body statements are executed, control returns to the condition test which is repeated. If the condition evaluates to FALSE, control passes to the sequential statement following the while loop. Figure 15 shows the representation corresponding to a while loop.

while *boolean_expression* **loop**
  seq_of_statements
end loop;

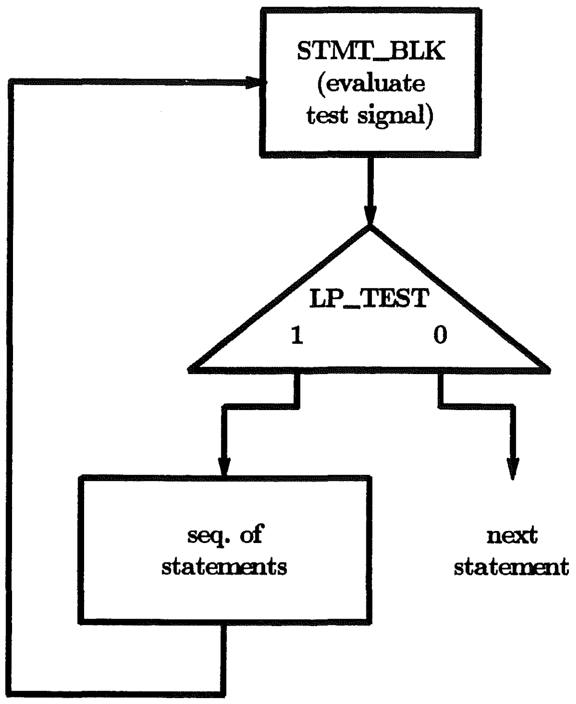


Figure 15: While Loop Statement

3.3.2.7. Procedure Call

The procedure call has the following syntax:

procedure_name (<parameter_list>);

Procedure calls are used in a VHDL description to invoke a procedure body consisting of sequential statements which are used one or more times in the description. Figure 16 shows the flow graph representation for a procedure call.

proc_name (<parameter_list>);

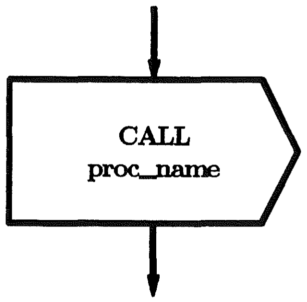


Figure 16: Procedure Call Statement

The procedure call may be processed in one of two ways:

(1) In-line expansion of each call may be performed, where the statements of the procedure body are substituted for the procedure call statement. A template flowgraph created for the procedure body is inserted, with actual parameters replacing occurrences of formal parameters. When this description is synthesized, each procedure call invocation can be mapped to available hardware in the data path, or a microcode implementation in control can be implemented. Annotations in the VHDL description will determine the implementation style.

(2) The procedure body is treated as a description of a block in the design. A flowgraph is created for the procedure body. Hardware is synthesized for this description, and each procedure call supplies the values of actual parameters as inputs to the procedure body hardware.

3.3.2.8. Wait Statement

The *wait* statement has the following syntax:

> **wait** [<condition_clause>] [<timeout_clause>] ;
>
> <condition_clause> ::= **until** <boolean_expression>
>
> <timeout_clause> ::= **for** <time_expression>

A wait statement is used to suspend the execution of a process statement until a specified condition is TRUE, or a timeout period elapses. Figure 17 shows the control flow graph sections created for a wait statement with condition and timeout clauses. This statement is implemented in control and is synchronized with the system clock; time is measured in multiples of the clock period.

wait until *cond_expr* **for** *time*;

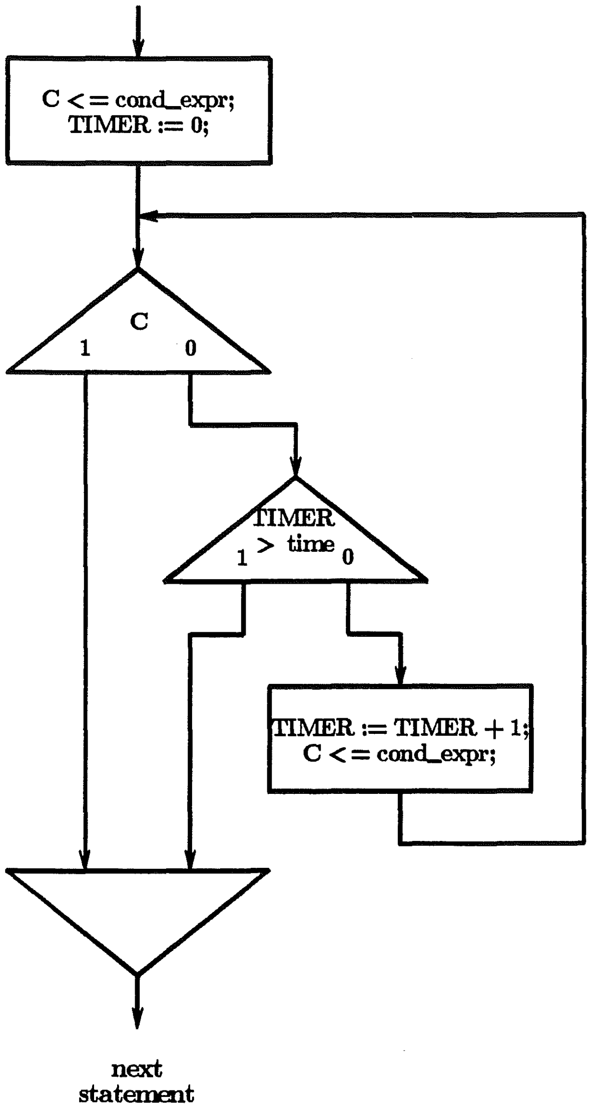


Figure 17: Wait Statement

3.4. Graph Construction Algorithm

3.4.1. Block Statement Compilation

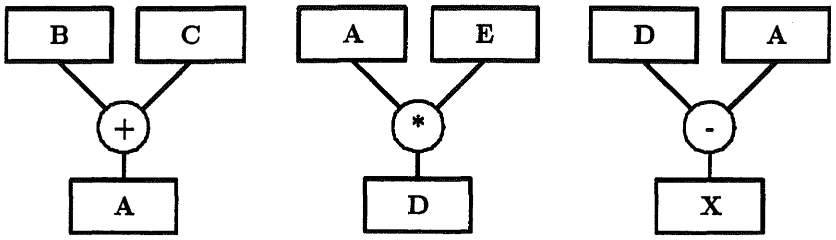
For signal assignments appearing in a block statement, flowgraph sections generated for each statement are interconnected once all statements have been processed. This corresponds to the concurrent data flow style where all operations are assumed to be executed in parallel. Variables appearing on the LHS of an assignment statement are assigned the value of the variable prior to the execution of the block statement. Figure 18 shows a VHDL code fragment consisting of several concurrent assignment statements, the flow graphs created for each statement, and the final interconnected flow graph.

The sections of DFG representing each signal assignment will be appropriately interconnected based on the signal type. It is the signal type that will define whether a VHDL signal (container) represents a memory element, port or wire. The signal type will also determine the interconnect protocol (wired-or, bus) which results when multiple sources for the same VHDL signal are encountered.

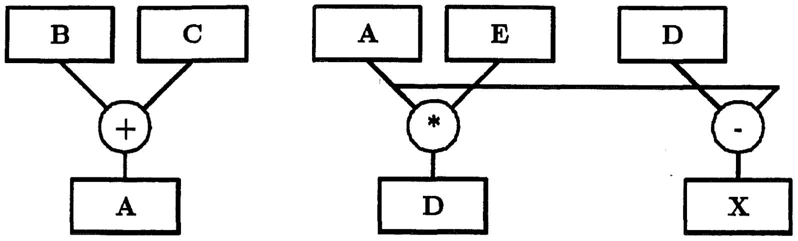
Multiple WRITEs (sources) to a signal of type WIRE indicate that a WIRED-OR node should be created with each WRITE node as an input. Any READ nodes for this signal should be connected to the output of the WIRED-OR node. This DFG construct will be mapped to a wired-or connection during design compilation.

```
A <= B + C;
D <= A * E;
X <= D - A;
```

VHDL Concurrent Statements



Individual Statement Flow Graphs



Interconnected Flow Graphs

Figure 18: Block Statement Compilation

Similarly, recognition of multiple WRITEs to a signal of type BUS should produce a BUS node to which all WRITE nodes are connected. Since each WRITE node for a signal of type bus was created when a guarded signal assignment was made, each input is controlled by some guard. This flow graph pattern will be mapped to a bus connection, where each CHOOSE-VALUE controlling a WRITE input becomes a tri-state bus driver.

Accesses to signals of type register are merged into a single WRITE access node. The inputs are muxed on the data input if they are synchronous, or are applied to different inputs (e.g., load and clear) if they are asynchronous.

The compilation algorithm for concurrent statements is summarized in the procedure **interconnect_concur_stmts** shown in Figure 19 below.

```
interconnect_concur_stmnts ()
{
merge duplicate READ nodes

merge duplicate READ_CONST nodes

for (each WRITE node)
   switch (signal type)
      case WIRE    : if (WIRED-OR node does not exist)
        create WIRED-OR node
     attach data input of WRITE node as input
to WIRED-OR
      case BUS     : if (BUS node does not exist)
create BUS node
     attach data input of WRITE node (from a CH-VALUE
        node) as input to BUS node
      case REGISTER: if (another WRT_REG node for the same var exists)
merge WRT_REG nodes, connecting appropriate
   control lines

look at all WRITE nodes and appropriately connect them to READ
nodes for the same signal

}
```

Figure 19: Block Compilation Algorithm

3.4.2. Process Statement Compilation

Unlike concurrent statements which are interconnected once all statements in the block have been processed, sequential statements appearing within a process statement are interconnected as they are encountered. Each control flow graph section corresponding to a sequential statement (STMT_BLK, if, case, loop, wait and procedure call) has a single entry point and single exit point. As these statements are processed, the exit point of the previous statement is connected to the entry point of the current statement. Since the control flow graph sections of most sequential statements are hierarchically constructed from other sequential statements, a stack is used to maintain the control flow node to which the current control flow node is to be attached.

Assignment statements are associated with the current STMT_BLK. Thus, a sequence of assignment statements is grouped initially into the same STMT_BLK control node until state binding is performed by the synthesis tool. A STMT_BLK is created if no current STMT_BLK exists when an assignment statement is encountered.

The compilation algorithm for sequential statements appearing in a process is summarized in Figure 20.

```
compile_process ()
{
curr_node = create_node(PROC_BEG);
process_seq_stmnts();
curr_node = create_node(PROC_END);
}

process_seq_stmnts ()
{
token = get_input_token();
switch (token) {
  case IF       :
  case ELSIF    : if (top_node()->type != STMT_BLK)
      curr_node = create_node(STMT_BLK);
   create data flow graph for condition
   curr_node = create_node(IF_TEST);
   token = get_input_token();/* THEN keyword */
   process_seq_stmnts();
   if (top_node->type == STMT_BLK)
      associate current data flow nodes with STMT_BLK;
   curr_node = create_node(IF_JOIN);
   else_branch = FALSE;/* global flag */
  case ELSE     : process_seq_stmnts();
   if (top_node->type == STMT_BLK)
      associate current data flow nodes with STMT_BLK;
  case CASE     : if (top_node()->type != STMT_BLK)
      curr_node = create_node(STMT_BLK);
   create data flow graph for case select
   curr_node = create_node(CASE_SELECT);
   token = get_input_token();/* IS keyword */
  case WHEN     : /* case statement alternative */
   associate guard value with current CASE alternative
   token = get_input_token();/* => keyword */
   process_seq_stmnts();
   if (top_node->type == STMT_BLK)
      associate current data flow nodes with STMT_BLK;
   set ptr in CASE_SELECT node so that last node of
      this alternative can be attached to the CASE_JOIN
  case FOR      :
  case WHILE    : if (top_node()->type != STMT_BLK)
      curr_node = create_node(STMT_BLK);
   create data flow graph for condition test
   curr_node = create_node(LOOP_TEST);
   token = get_input_token();/* LOOP keyword */
   loop_type = token;/* global flag */
```

```
case WAIT      : token = get_input_token();
 if (top_node()->type != STMT_BLK)
    node2 = create_node(STMT_BLK);
 if (token == UNTIL)
    {
    create dfg for cond_expr;
    cond_expr = TRUE;/* local flag */
    }
 token = get_input_token();
 if (token == FOR)
    {
    create dfg for TIMER initialization;
    time_expr = TRUE;/* local flag */
    }
 associate dfg with STMT_BLK;
 wait_test = create_node(IF_TEST);
 if (cond_expr && time_expr)
    {
    node2 = create_node(IF_JOIN);
    curr_node = create_node(IF_TEST);
    connect_nodes(curr_node,node2);
    }
 node2 = create_node(STMT_BLK);
 if (cond_expr)
    create dfg to evaluate condition;
 if (time_expr)
    create dfg to increment TIMER;
 connect_nodes(node2,wait_test);
 curr_node = pop_node();/* STMT_BLK */
case identifier: /* signal or variable assignments */
 if (top_node()->type != STMT_BLK)
    curr_node = create_node(STMT_BLK);
 create data flow graph for assignment statement
case END      : token = get_input_token();
 switch (token) {
    case IF: if (else_branch == TRUE)
{
if (top_node()->type == STMT_BLK)
   associate current df nodes
curr_node = pop_node();
if (top_node()->type == IF_JOIN)
   {
   connect_nodes(curr_node,top_node());
   curr_node = pop_node();
   }
```

```
    }
   else
    {
    curr_node = pop_node();/* IF_JOIN */
    if (top_node()->type == IF_TEST)
            {
      node2 = pop_node();/* IF_TEST */
      connect_nodes(node2,curr_node);
      }
    }
   while (top_node()->type == IF_JOIN)
    {
    node2 = pop_node();
    connect_nodes(curr_node,node2);
    curr_node = node2;
    }
  push_node(curr_node);
       case CASE: curr_node = create_node(CASE_JOIN);
   for (all CASE alternative branches)
     connect_node(alt_last_node,curr_node);
   node2 = pop_node();/* CASE_SELECT */
   push_node(curr_node);
       case LOOP: if (top_node()->type != STMT_BLK)
     curr_node = create_node(STMT_BLK);
   create data flow graph to update FOR
     index, evaluate WHILE condition
   associate current df nodes
   node2 = pop_node();/* STMT_BLK */

   /* make loop back connection */
   connect_nodes (node2,top_node());
       }
   }
   if (token != PROCESS)
     {
     if (token != ';')
       token = get_input_token();/* ; keyword */
     process_seq_stmts();
     }
}
```

create_node (type)

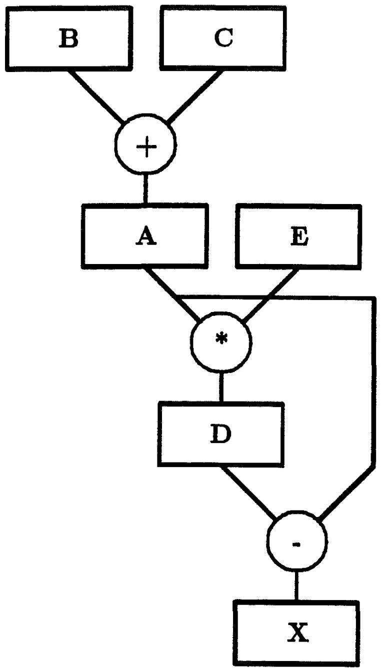
```
{
curr_node = create_node_structure();
prev_node = top_node();
switch (prev_node->type) {
  case IF_JOIN   : if (prev_node->num_inputs == 2)
     prev_node = pop_node();
   else
     {
     prev_node = pop_node();
     if (top_node()->type == IF_TEST)
 {
 node2 = pop_node();
 push_node(prev_node);
 }
     }
  case STMT_BLK  : if (prev_node->type != BLK_START)
     {
     associate current df nodes;
     prev_node = pop_node();
     }
  case PROC_BEG  :
  case LOOP_TEST :
  case IF_TEST   : if (prev_node->fg_num_outputs == 1)
     prev_node = pop_node();
  case CF_START  : if (type == BLK_END)
     prev_node = pop_node();
  case CASE_SELECT: break;
  default        : prev_node = pop_node();
  }
connect_nodes(prev_node,curr_node);
push_node(curr_node);
}
```

Figure 20: Process Compilation Algorithm

For signal or variable assignments appearing in a process statement, data flow graph sections are generated for each statement. The location of the last update (WRITE) of all signals and variables is maintained. Variables appearing on the LHS of an assignment statement are assigned this last update value. If a value is updated and subsequently accessed within the same STMT_BLK, the data flow WRITE and READ nodes, respectively, are interconnected. Figure 21 shows the same VHDL code fragment from the previous section as it would appear in a process statement consisting of several variable assignment statements. Notice that the sequential nature of the process imposes data dependencies on the variable accesses, resulting in a different interconnected flow graph.

```
A := B + C;
D := A * E;
X := D - A;
```

VHDL Variable Assignment Statements



Interconnected Sequential Statements

Figure 21: Compilation of Variable Assignments in a Process

3.5. Annotations

In some instances, it is necessary to indicate to the VSS system which design process should be used for a given VHDL description. This is accomplished through the use of *annotations* in the form of special VHDL comments as shown below:

--VSS: functional description

Annotations are used in the following situations:

(1) To indicate the structured modeling style used in the VHDL description.

(2) To indicate loop unwinding, where iterations are flattened into a sequence of assignments, rather than implementing indexing or conditional tests in control.

(3) To denote a next state in process descriptions. This can be used to define state boundaries for a register transfer description consisting of a sequence of assignment statements.

4. Conclusion

We have described an intermediate representation which is used for synthesis from VHDL. The representation generated for the VHDL constructs has been presented, along with algorithms for construction of the complete description. This intermediate representation can be used by a variety of synthesis tools and allows for the description of several design styles. The use of signal typing and attribute annotations incorporates the necessary design information for synthesis into a simple and complete representation which can be easily maintained and manipulated by synthesis tools.

5. References

[Arms89] Armstrong, J., *Chip Level Modeling with VHDL*, Prentice-Hall, 1989.

[Chen89] Chen, G.D., *Intelligent Component Data Base (ICDB)*, Technical Report (in preparation), University of California at Irvine, June 1989.

[CLSI87] *VHDL Tutorial for IEEE Standard* 1076 *VHDL*, CAD Language Systems Inc., 1987.

[LisGa88] Lis, J. and Gajski, D., *Synthesis from VHDL*, ICCD88, 1988.

[LisGa89a] Lis, J. and Gajski, D., *VHDL Synthesis Using Structured Modeling*, 26th DAC, 1989.

[LisGa89b] Lis, J. and Gajski, D., *Structured Modeling for VHDL Synthesis*, Technical Report (in preparation), University of California at Irvine, June 1989.

[OrGa86] Orailoglu, A., Gajski, D., *Flow Graph Representation*, 23rd DAC, 1986.

[Preas88] Preas, B. and Lorenzetti, M., *Physical Design Automation of VLSI Systems*, Benjamin/Cummings, 1988.

[VHDL87] *VHDL Language Reference Manual, Draft Standard* 1076/*B*, IEEE, June 1987.

APPENDIX A
VHDL Statement Syntax

1. Concurrent Statements

1.1 Signal Assignment

1.1.1 Conditional Signal Assignment

```
signal <= { <waveform> when <condition> else }
            <waveform> ;
   <waveform> ::= <expression> { after <delay> }
```

1.1.2 Selected Signal Assignment

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

1.1.3 Guarded Signal Assignment

```
signal <= guarded <waveform> ;
```

2. Sequential Statements

2.1 Signal Assignment

```
target <= <waveform> ;
```

2.2 Variable Assignment

```
target := <expression> ;
```

2.3 If Statement

```
if <expression>
   then
      sequence_of_statements
   else
      sequence_of_statements
end if;
```

2.4 Case Statement

```
case <discrete_expression> is
   when choice_1 =>
      sequence_of_statements
         ...
   when choice_N =>
      sequence_of_statements
end case;
```

2.5 For Loop

```
for <ident> in <discrete_range> loop
   sequence_of_statements
end loop;
```

2.6 While Loop

```
while <boolean_expression> loop
   sequence_of_statements
end loop;
```

2.6 Procedure Call

```
procedure_name (<parameter_list>);
```

2.7 Wait Statement

```
wait [<condition_clause>] [<timeout_clause>] ;
    <condition_clause> ::= until <boolean_expression>
    <timeout_clause> ::= for <time_expression>
```

APPENDIX B
VHDL Structural Netlist Specification

1. Introduction

The types of information contained in the design representation after the Graph Compilation, Graph Critic, and Node Compilation phases have completed can be classified as follows: **component instances, connectivity, instance parameters,** and **timing.** The *generic component netlist* representation which is the output of the synthesis system must express this information as well. A standard format for this netlist is desirable so that interfacing to other design tools can be more easily accomplished.

The VHDL structural style of description seems suited to this purpose. The general form of this netlist is shown in Figure 22. A simple circuit schematic is shown in Figure 23. Figure 24 shows an example VHDL structural description using this format. Note that this netlist conforms to IEEE Standard VHDL 1076B [VHDL88] language. The following sections describe how the various types of design information are represented in this netlist format.

-- interface portion

entity *<entity-name>* **is**
  <port-declarations>
  <external-timing-assertions>
end *<entity-name>*;

-- architectural body (structural description style)

architecture *Structure_View* **of** *<entity-name>* **is**
  <component-declarations>
  <component-attributes>
  <internal-signal-declarations>
  <internal-timing-assertions>
begin
  <component-instantion-statements>
end Structure_View;

Figure 22: VHDL Generic Component Netlist Format

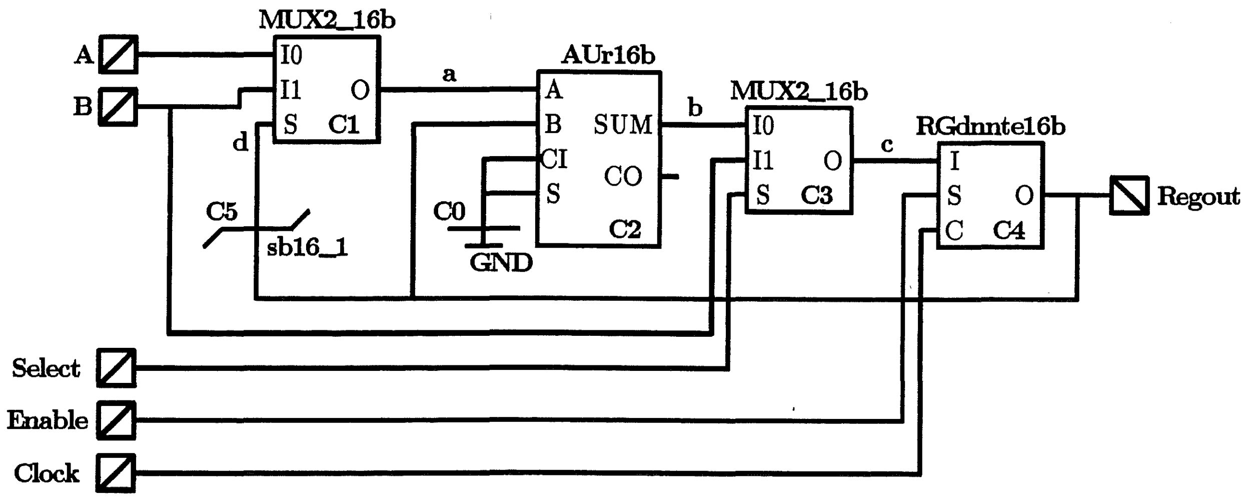


Figure 23: Example Circuit Schematic

```
-- interface portion

entity Example1 is
  port (A,B: in: BIT_VECTOR(0 to 15);
        Select,Enable: in BIT;
 Clock: in CLOCK;
 Regout: out BIT_VECTOR(0 to 15));

-- external timing assertions

--T ↑↓ A to ↑↓ Regout: 20 ns average
--T ↑ Clock to ↑↓ Regout: 5,10 ns

end Example1;

-- architectural body (structural description style)

architecture Structure_View of Example1 is

-- component declarations

  Component MUX2_16b
    port (I0,I1: in: BIT_VECTOR(0 to 15);
          S0: in BIT;
   O0: out BIT_VECTOR(0 to 15));

  Component AUr16b
    port (A,B: in: BIT_VECTOR(0 to 15);
   CI: in BIT;
   S: in BIT_VECTOR(0 to 1);
   SUM: out BIT_VECTOR(0 to 15);
   CO: out BIT);

  Component RGdnnte16b
    port (I: in: BIT_VECTOR(0 to 15);
   S: in BIT;
   C: in CLOCK;
   O: out BIT_VECTOR(0 to 15));
```

Figure 24: Example VHDL Structural Description

```
    Component sb16_1
      port (I: in: BIT_VECTOR(0 to 15);
     O: out BIT);

    Component GND
      port ( O: out BIT);

 -- component attributes

    type FUNC_TYPE is (ADD,SUB,INC,DEC);
    type CARRY is (RIPPLE,LOOKAHEAD);
    attribute FUNCTION: FUNC_TYPE;
    attribute ADDER_TYPE: CARRY;
    attribute ENBL: BOOLEAN;

    attribute FUNCTION of AUr16b: component is ADD;
    attribute ADDER_TYPE of AUr16b: component is RIPPLE;
    attribute ENBL of C4: label is TRUE;

 -- internal signal declarations

    signal a,b,c: BIT_VECTOR(0 to 15);
    signal d,Gnd: BIT;

 -- internal timing assertions

 --T ↑↓ a to ↑↓ b: 20,25,35 ns
 --T ↓ A to ↑↓ b: 40 ns max

 -- component instantiations

 begin
   C0: GND port map (Gnd);
   C1: MUX2_16b port map (A,B,d,a);
   C2: AUr16b port map (a => A,Regout => B,Gnd => CI,Gnd => S(0),
 Gnd => S(1),b => SUM);
   C3: MUX2_16b port map (b,B,Select,c);
   C4: RGdnnte16b port map (c,Enable,Clock,Regout);
   C5: sb16_1 port map (Regout,d);
       d <= Regout(15);
 end Structure_View;
```

Figure 25: Example VHDL Structural Description (cont'd)

2. Netlist Format

2.1. Entity Declaration

The entity declaration portion of the VHDL structural description specifies the design name and defines the design's interface to the outside world. Port declarations are used to define input and output connections. VHDL **assertion** statements are used to specify timing constraints from input to output ports of the design. See the section on timing assertions below for the format of these statements.

2.2. Component Declarations

For each unique component in the netlist, a component declaration must exist. This declaration defines a template containing input and output pin specifications via port declarations. The type and bit width of the signals (nets) to be attached to the component ports is specified in these declaration statements.

In order to generate a generic netlist using a set of generic components, a table of available components and their component declarations must exist. This table should identify the function of each input and output pin and the pin naming conventions for each component. It should also specify the operand port mappings for multiple operation units. If this component declaration table is available to interface programs which accept the netlist as input, it would not be necessary to include component declarations in the netlist.

2.3. Component Attributes

In order to specify parameters particular to a component such as ALU functions, control input codes, etc., the VHDL *attribute* declaration and specification features can be used. Enumeration types can be used to specify the allowable values of an attribute. Attributes may be associated with the template component declaration, or with specific labelled instances of a component. For example, the statements

```
type FUNC_TYPE is (ADD,SUB,INC,DEC);
attribute FUNCTION: FUNC_TYPE;
attribute FUNCTION of AUr16b: component is ADD;
```

will associate the FUNCTION attribute ADD with every instance of an AUr16b component, while the attribute specification

```
attribute ENBL of C4: label is TRUE;
```

will associate the ENBL attribute with RGdnnte16b instance C4 only.

2.4. Internal Signal Declarations

Internal connection of components is accomplished by defining each internal net of the generic component netlist using signal declaration statements. These signal (net) names are used in the port map specification of component instantiations described below in order to identify uniquely the net connections between component ports.

2.5. Timing Assertions

It is often necessary and useful when specifying timing constraints of a circuit to have the capability of specifying relationships between signals. For example, a common requirement is that the data input to a clocked register be stable a duration of time prior to the clock transition that strobes the data into the register (sometimes known as set up time) [Arms87]. The following signal transitions should be representable:

1. ↑ Stransition from 0 to 1 of signal S (rising)
2. ↓ Stransition from 1 to 0 of signal S (falling)
3. ↑↓ Sany transition of signal S (change)

The timing relationship is expressed as follows:

<transition1> to <transition2> : <duration>

where <transition1> and <transition2> are of the form specified above. The <duration> specification is used to specify the minimum, maximum and/or average time interval(s) between two events. A single time period specification must be followed by a qualifier (**max, min,** or **average**). For example:

--T ↓ A to ↑↓ b: 40 ns max

A list of two time intervals specifies a minimum/maximum timing specification, such as:

--T ↑ Clock to ↑↓ Regout: 5,10 ns

A triplet of time intervals denotes minimum, average, and maximum, as in:

--T ↑↓ a to ↑↓ b: 20,25,35 ns

VHDL uses the *assertion* mechanism to represent this information with the following syntax:

> **assert** <*boolean-expression*>
> **report** <*error-message*>;

When the assertion is executed during simulation, the boolean expression is evaluated. If it evaluates to FALSE (indicating that the timing specification is not satisfied), the error message (a text string) is printed. The VHDL STABLE and DELAYED signal attributes would be used to express the signal transitions as follows:

1. S and not S'STABLE
2. not S and not S'STABLE
3. not S'STABLE

The timing relationship **↑S1 to ↓S2: 20 ns** would be expressed as follows:

```
(S1 and not S1'STABLE) and
(not S2'DELAYED(20 ns) and not S2'DELAYED(20 ns)'STABLE))
```

The VHDL assertion statement is considered a sequential statement; consequently, it would not appear in a valid VHDL structural description. However, the language can be extended so that assertion statements may appear in the entity and block declaration parts. The entity assertions could specify input to output port timing, while the block assertions denote internal signal timing relationships.

A better method of expressing this information which conforms to the VHDL language definition would be to use comments. For example, the statements

```
--T ↑↓ A to ↑↓ Regout: 20 ns average
--T ↑ Clock to ↑↓ Regout: 5,10 ns
```

would be parsed as comments by the VHDL Analyzer, but the netlist parser could recognize the *--T* timing assertion delimiter and record the specified timing information.

2.6. Component Instantiations

A component is instantiated through the use of a VHDL *component instantiation statement* within the block of the architectural body. This statement has the form:

<label>: *<component-name>* *<port-map>*;

The <label> is a unique id for the component. A component declaration statement for <component-name> ,must exist, defining the ports (mode, bit width) to be found in the <port-map> list. The <port-map> is a list of previously defined port or internal signal names which defines the interconnection of components. This list may be of *named* or *positional* format. Named format is an unordered list of association of signals to ports. For example, if net N1 is attached to port P1 (as defined in the port list of the component declaration), N1 => P1 would appear in the <port-map> list. Positional format assigns elements of the <port-map> to ports with the corresponding position in the port list of the component declaration.

Concurrent assignment statements may be used to specify necessary behavior characteristics of a component. Examples of this type of specification include the concatenation of input signals to form output signals for the switchbox component of Figure 24, or the specification of the functionality of a random logic component using boolean equations.