VHDL2verilog™ User's Manual

May 2007

SynaptiCAD, Inc.,
PO Box 10608
Blacksburg, VA 24062-0608
Ph: (540) 953-3390 Fax: (540) 953-3078
email: sales@syncad.com

Copyright © 2007, Alternative System Concepts, Inc. All rights reserved worldwide.
May be printed or photocopied by licensed customers of ASC for internal business purposes only.

Table of Contents

About this Manual

- References

Chapter 1: Introduction and Installation

- Introduction

Chapter 2: VHDL2verilog Translation Overview

- Entities and Architectures
- Packages and Package bodies
- Data Types
- Generics
- Sequential Statements
  - Wait Statements
  - Signal Assignments
  - Variable Assignments
  - If Statements
  - Case Statements
  - Loop Statements
  - NULL Statements
  - Assert Statements
  - Procedure Call Statements

- Concurrent Statements
  - Block Statements
  - Process Statements
  - Concurrent Signal Assignments
  - Component Instantiations
  - Generate Statements

Chapter 3: Using VHDL2verilog

  - Command Summary
  - VHDL2verilog Environment
  - vhdl2v.map File
  - Function Mapping and Translation
    - Mapping Functions Existing in VHDL Packages
    - Mapping Functions in VHDL Input
    - Translating Functions (or Procedures) in VHDL Input
  - DFF Mapping in VHDL2verilog
  - Importing (Parts of) Verilog Files
  - Reserved Words

About this Manual

This manual describes how to install and use **VHDL2verilog**, a translator that produces VHDL design descriptions from Verilog HDL design descriptions.

Chapter 1 gives a brief overview of **VHDL2verilog** and explains how to install it. Chapter 2 gives an overview of how **VHDL2verilog** works and details how it translates Verilog elements to VHDL. Chapter 3 explains the command options you can use to fine tune the tool's operation.

You should use this Manual in conjunction with the **VHDL2verilog Release Notes** document, which provides the latest information on language support and usage details.

General questions and inquiries can be sent to:

SynaptiCAD, Inc.
PO Box 10608
Blacksburg, VA 24062-0608
Ph: (540) 953-3390 Fax: (540) 953-3078
email: sales@syncad.com

Bug reports or other support questions can be sent to:

email: sales@syncad.com

Conventions

In this manual, *italic text*is used to differentiate file and pathnames. **Bold print** is used for commands. `Courier type` is tool output or HDL code. **`Bold Courier type`** is text entered by the user on the command line.

References

The **VHDL2verilog Release Notes** that came with your version of the tool provide up-to-the-minute details on language support and execution command options. You can also find examples and other helpful information at the SynaptiCAD Web site.

These general references may also prove helpful as you work with VHDL and Verilog:

- IEEE Standard VHDL Language Reference Manual
- Verilog Hardware Description Language Reference Manual (LRM), version 2.0
- VHDL Designer's Reference, *Jean-Michel Berge, Alain Fonkoua, Serge Maginot, Jacques Rouillard*
- VHDL'92, *Jean-Michel Berge, Alain Fonkoua, Serge Maginot, Jacques Rouillard*

Chapter 1: Introduction and Installation

VHDL2verilog translates hierarchical IEEE Standard 1076-87 and 93 VHDL to Verilog HDL. It translates all structural VHDL constructs, as well as a large subset of RTL VHDL constructs. Verilog that is created by VHDL2verilog is functionally equivalent to input VHDL, and because of one-to-one mapping of VHDL to Verilog, is easy to understand. Syntax and semantic checking of the VHDL input is also performed during translation and output Verilog is compatible with any Verilog-XL compatible simulator.

VHDL2verilog is a valuable design reuse tool because it preserves the level of abstraction during translation. There is a one-to-one mapping of constructs in most cases. Because the abstraction level is preserved, Verilog can remain technology independent in cases where the input is behavioral or RTL.

VHDL2verilog can:

- be useful to designers working in a VHDL/Verilog environment,
- be integrated into Verilog-only based tools,
- provide library translation (VHDL library => Verilog library) capability, and
- automate the model generation procedure.

VHDL2verilog can be obtained with an object-oriented Software Procedural Interface (SPI) to access the output HDL, and is currently available Solaris, Linux and Windows platform. Future versions will support more operating platforms.

Chapter 2:
VHDL2verilog Translation Overview

Entities and Architectures | Packages and Package Bodies | Data Types | Generics
Sequential Statements: Wait | Signals | Variables | If | Case | Loop | Null | Assert | Procedure Call
Concurrent Statements: Block | Process | Concurrent Signals | Components | Generate

VHDL2verilog translates input VHDL to Verilog. This release supports a subset of VHDL (see Appendix A for comprehensive listing).

The rest of the chapter is organized as follows: For every construct supported by **VHDL2verilog**, the general translation rules are given, followed by an example illustrating the construct in both languages. Special notes are given when required. The user is advised to refer to the VHDL LRM and Verilog LRM for detailed explanations.

Entities and Architectures

Chapter Start

The basic design unit in VHDL is the `ENTITY` and `ARCHITECTURE` pair. An entity and architecture pair is translated to a Verilog `MODULE` description. An entity can have multiple architectures. Multiple architectures per entity are translated to multiple modules. **VHDL2verilog** creates module names based on the following criteria:

- Module name is the same as the entity name in input VHDL file.
- Module name is the concatenation of the entity and architecture name in input VHDL file. For e.g., if an entity *aha* has two architectures, *structural* and *behavioral*, the Verilog modules are *aha_structural* and *aha_behavioral*.

Example:

```
VHDL                                                    Verilog

LIBRARY ieee;                                           module test(a,b);
USE ieee.std_logic_1164.ALL;

ENTITY test IS
PORT (a : IN std_logic;                                    input a;
      b : OUT std_logic);                                  output b;
END ENTITY;

ARCHITECTURE behave OF test IS                          // other translated
constructs
BEGIN
END behave;                                             endmodule
```

Special Note: Verilog files can be imported for VHDL Entity and Architecture pairs by using tool-specific attribute `VHDL2V_MODULE` in the entity declaration. See *Importing (parts of) Verilog files* for detailed explanation.

Packages and Package Bodies

Chapter Start

Most VHDL designs reference one or more `PACKAGE`s through `USE` statements. These packages can contain pre-defined constants, types, components, or subprograms (functions or procedures). **VHDL2verilog** will translate the packages if they are used, and includes this translation either in the output file, or in one or more separate files.

Example:

```
VHDL                                                Verilog

PACKAGE my_package is
CONSTANT Width : Integer := 16
END my_package;

USE ieee.my_package.ALL;                            module test(a,b);
ENTITY test IS                                         parameter Width = 16
PORT (a : IN std_logic;                                input a;
      b : OUT std_logic);                              output b;
END ENTITY;

ARCHITECTURE behave OF test IS                      // other translated
constructs
BEGIN
END behave;                                         endmodule
```

The translated package can also be included in a separate file using the `-Include Package files` switch. In that case **VHDL2verilog** will produce a file called:

```
<package>_package.verilog
```

If there are components in the package, and the corresponding entity - architecture pairs of the components have been read in as well, another file will be produced called:

```
<package>_modules.verilog
```

Both files are included in the Verilog code with an `` `include `` statement.

NOTE: Subprograms in the package get translated *only* when they are used in the design and when Function mapping and translation has been activated.

Data Types

Chapter Start

VHDL has three classes of data objects: SIGNAL, VARIABLE and CONSTANT. Verilog does not make any such distinction in its two data types: NETS and REGISTERS. Verilog NETS are used for driving signals, and REGISTERS are an abstraction of the hardware register. **VHDL2verilog** translates VHDL objects to Verilog NETS or REGISTERS based on the following rules:

1. If the signal is an input, it is translated to an implicitly declared WIRE.
2. Signals corresponding to output ports get translated to Verilog REGISTERS.
3. If a signal is used as the target in a VHDL concurrent assignment or an actual designator in the port mapping of a component instantiation, the Verilog translation of VHDL signal is a WIRE. Note that in this particular case, outputs get declared as WIRES.

Example:

```
VHDL                                                    Verilog

LIBRARY ieee;                                           module test(a,b,c);
USE ieee.std_logic_1164.ALL;

ENTITY test IS
PORT (a : IN std_logic;                                    input a;
      b : OUT std_logic;                                   output b;
      c : OUT std_logic);                                  output c;
END ENTITY;

ARCHITECTURE behave OF test IS                          //other translated
constructs
BEGIN                                                      wire b;
                                                           reg c;
b <= '1'; -- concurrent assignment                         assign b = 1;
END behave;                                             endmodule
```

Generics

Chapter Start

VHDL GENERICs are used to construct parameterized components. Verilog offers similar capabilities in the form of parameter declarations. VHDL generics are translated to Verilog parameters.

Example:

```
VHDL                                                    Verilog

LIBRARY ieee;                                           module test(a,b,c);
USE ieee.std_logic_1164.ALL;

ENTITY test IS
GENERIC(gen_a : IN natural := 20;                          parameter gen_a =
20;
```

```
       datapath : IN integer : 11);
PORT(a : IN std_logic;
     b : OUT std_logic_vector(datapath downto 0)
b;
     c : OUT std_logic);
END ENTITY;

ARCHITECTURE behave OF test IS
constructs
BEGIN

END behave;
```

```
parameter datapath =
11;
input a;
output [datapath:0]

output c;

// other translated

   wire [datapath:0] b;
   reg c;
endmodule
```

Sequential Statements

VHDL sequential statements occur in VHDL PROCESS or SUBPROGRAM bodies. These statements execute in the order they appear. **VHDL2verilog** translates sequential statements outlined in the following sections.

Wait Statements

The **WAIT** statement is used to suspend the execution of a process or subprogram body. The VHDL LRM defines three types of WAIT statements:

1. Wait with sensitivity clause
2. Wait with condition clause
3. Wait with timeout clause

The current version of **VHDL2verilog** supports WAITs with timeout clauses. WAITs with sensitivity clauses are supported only if appearing as the first (or last) item of the sequential body.

Example:

VHDL

```
process_1 : process
negedge enable)
CONSTANT tpd : std_logic := '1';
CONSTANT tpd1 : std_logic := '0';
begin
   wait on a, enable;
   if (enable = '0') then
        q <= '0';
   elsif a'event and a'last_value = '0' then
        q <= d;
```

Verilog

```
always @(posedge a or

begin : process_1
parameter tpd = 'b 1;
parameter tpd1 = 'b 0;

if (enable == 'b 0)
   q <= 'b 0;
else
  begin
```

```
        q <= '1' after 2 ns;
        q <= '0';
   end if;
end process process_1;
```

```
        q <= d;
        q <= #2 'b 1;
        q <= 'b 0;
      end
    end
```

Signal Assignments

SIGNAL ASSIGNMENTs are used to modify the output waveform of the target. Verilog signal assignments (known as PROCEDURAL ASSIGNMENTS) can be of two types:

1. BLOCKING procedural assignment: A blocking procedural assignment MUST be executed before the execution of the statements that follow.
2. NON-BLOCKING procedural assignment: Verilog non-blocking procedural assignments allow scheduling of assignments without blocking the procedural flow. Typically, they are used when multiple assignments need to be made in the same simulation cycle.

All VHDL signal assignments are translated to Verilog NON-BLOCKING procedural assignments.

Example:

VHDL

```
architecture diffeq of diffeq is

begin

p0 : process
-- initialize some values
begin
  Xoutport <= 0;
  Youtport <= 0;
  Uoutport <= 0;
  WAIT;
end process p0;

P1 : process (Aport, DXport, Xinport,
Xinport or
     Yinport, Uinport)

   variable x_var,y_var,u_var
        : std_logic ;

begin

  Xoutport <= x_var after 2ns;
//will always occur
  Youtport <= transport y_var;
non-blocking
```

Verilog

```
module diffeq;


initial
begin : p0
   Xoutport <= 0;
   Youtport <= 0;
   Uoutport <= 0;
end


always @(Aport or DXport or

Yinport or Uinport)

 begin : P1
   reg  x_var;
   reg  y_var;
   reg  u_var;

   Xoutport <= # 2 x_var;

   Youtport <= y_var; //fully
```

```
  Xoutport <= u_var after 5ns;              Xoutport <= # 5 u_var;
//will not cancel
  -- will cancel previous assignment
//previous assignment
                                          end
end process P1;
```

NOTE: Verilog non-blocking assignments differ from VHDL signal assignments in one important respect. Verilog non-blocking assignments ALWAYS occur. However, VHDL signal assignments with INERTIAL delay generate projected events which can be cancelled by future events. Verilog non-blocking assignments execute without cancelling a previous non-blocking assignment to the same register. Therefore, delays in Verilog non-blocking assignments are handled as TRANSPORT delays. If the delay type in VHDL signal assignments were specified to be TRANSPORT, then it would be equivalent to Verilog non-blocking assignment.

Variable Assignments

Sequential Statements: Wait | Signals | Variables | If | Case | Loop | Null | Assert | Procedure Call ||| Chapter Start

VARIABLE ASSIGNMENTS are used to replace the current value of the variable with the new value specified on the right hand side of the assignment without any delay. They are mapped to Verilog non-blocking register assignments.

Example:

VHDL

```
process_2 : process(dummy)
variable q_1 : std_logic_vector(3 downto 0);
variable q_2 : std_logic_vector(3 downto 0);
begin
  if (dummy = '1') then
     q_1 := "0001";
  end if;
end process process_2;
```

Verilog

```
always @(dummy)


begin : process_2
reg     [3:0]  q_1;
reg     [3:0]  q_2;
if (dummy == 'b 1)
  q_1 = 'b 0001;
end
```

If Statements

Sequential Statements: Wait | Signals | Variables | If | Case | Loop | Null | Assert | Procedure Call ||| Chapter Start

The IF statement chooses to execute one (or none) of many sequence of statements based on the value of corresponding condition. The semantics of the Verilog IF statement are the same, barring the VHDL requirement that an IF (ELSIF) expression be boolean. VHDL IFs are mapped to Verilog IFs.

Example:

VHDL

Verilog

```
process_1 : process
CONSTANT tpd : std_logic := '1';
CONSTANT tpd1 : std_logic := '0';
begin
   wait on a,enable;
   if (enable = '0') then
        q <= '0';
   elsif a = '0' then
        q <= d;
        q <= '1' after 2 ns;
        q <= '0';
   end if;
end process process_1;
```

```
always @(a or enable)
begin : process_1
parameter tpd = 'b 1;
parameter tpd1 = 'b 0;

if (enable == 'b 0)
   q <= 'b 0;
else if (a == 'b 0)
  begin
   q <= d;
   q <= #2 'b 1;
   q <= 'b 0;
  end
end
```

Case Statements

CASE statements in VHDL have the same semantics as CASE statements in Verilog and are mapped to the same.

Example:

VHDL

```
PROCESS
BEGIN
   WAIT ON sel_output;
   V2V_d_en <= '0';
   CASE sel_output IS
      WHEN (B"000") =>
      V2V_d_out <= B"00000001" AFTER 2 ns;
00000001;
      WHEN (B"001") =>
      V2V_d_out <= B"00000010";
00000010;
      WHEN (B"0_1_0") =>
      V2V_d_out <= B"00000011";
00000011;
      WHEN ("0Z1") =>
      V2V_d_out <= B"00000100";
00000100;
      WHEN (B"100") =>
      V2V_d_out <= B"00000101";
00000101;
      WHEN (B"101") =>
      V2V_d_out <= B"00000110";
00000110;
      WHEN (B"110") =>
      V2V_d_out <= B"00000111";
00000111;
      WHEN (B"111") =>
      V2V_d_out <= B"00001000";
00001000;
```

Verilog

```
always @(sel_output)
  begin

  V2V_d_en <= 'b 0;
  case (sel_output)
  'b 000:
     V2V_d_out <= #2 'b

  'b 001:
     V2V_d_out <= 'b

  'b 0_1_0:
     V2V_d_out <= 'b

  'b 0Z1:
     V2V_d_out <= 'b

  'b 100:
     V2V_d_out <= 'b

  'b 101:
     V2V_d_out <= 'b

  'b 110:
     V2V_d_out <= 'b

  'b 111:
     V2V_d_out <= 'b
```

```
      WHEN OTHERS =>
      V2V_d_out <= "ZZZZZZZZ";
ZZZZZZZZ;
   END CASE;
END PROCESS;
```

```
      default:
         V2V_d_out <= 'b

      endcase
      end
```

Loop Statements

VHDL defines LOOP statements, that are a collection of sequential statements. The execution is specified by the iteration scheme (e.g FOR, WHILE) used. Loop statements without iteration schemes are equivalent to Verilog FOREVER statements, iteration scheme FOR is translated to Verilog FOR loop and VHDL WHILE iteration scheme is translated to Verilog WHILE loop.

Example:

VHDL

```
PROCESS
BEGIN
 WAIT UNTIL a'event;
 i1 <= 0;
 WAIT FOR 0 NS;

 WHILE i1 < reg_dummy + 1 LOOP
1) begin
   V2V_b(3 DOWNTO 0) <= a(3 DOWNTO 0);
   i1 <= i1 + 1;
   WAIT FOR 0 NS;
 END LOOP;

 FOR cnt IN 4 DOWNTO 1 LOOP
cnt = cnt - 1)
   V2V_b(i - "00000001") <= a(i);
<= a[i];
 END LOOP;
 len <= '8';
 WAIT FOR 0 NS;

 LOOP
  V2V_b(i - "00000001") <= a;
<= a;
 END LOOP;
END PROCESS;
```

Verilog

```
always @(a)
  begin
i1 <= 0;
#0;


while ( i1 < reg_dummy +

 V2V_b[3:0] <= a[3:0];
 i1 <= i1 + 1;
 #0;
end

for (cnt = 4; cnt >= 1;

 V2V_b[i - 'b 00000001]


len <= 'b 8;
#0;

forever
 V2V_b[i - 'b 00000001]


end
```

NULL Statements

A NULL statement performs no action and is equivalent to the Verilog ``;" statement.

Example:

VHDL

```
process_2 : process(dummy)
variable q_1 : std_logic;
variable q_2 : std_logic;
begin
   if (dummy = '1') then
        q_1 := '1';
   else
        null;
   end if;

end process process_2;
```

Verilog

```
always @(dummy)
  begin : process_2
  reg      q_1;
  reg      q_2;
  if (dummy == 'b 1)
    q_1 = 'b 1;
  else
    ; //null statement


 end
```

Assert Statements

Sequential Statements: Wait | Signals | Variables | If | Case | Loop | Null | Assert | Procedure Call ||| Chapter Start

The Assert statement shows a pre-defined message when the assertion fails. The severity is made part of the message. Furthermore, in the Verilog translation the time when the assertion was activated is shown.

Example:

VHDL

```
VARIABLE status : boolean;

-- status gets value

ASSERT status = FALSE REPORT
`false))
   "Some message" SEVERITY note;


message");

$time);
```

Verilog

```
reg      status;

// status gets value

if (! ( status ==

    begin
    $write("note: ");
    $display("Some

    $display("Time: ",

    end
```

Procedure Call Statements

Sequential Statements: Wait | Signals | Variables | If | Case | Loop | Null | Assert | Procedure Call ||| Chapter Start

A VHDL Procedure call is translated into a Verilog task enable when the -Support Subprogram Calls switch has been selected.

Example:

VHDL

Verilog

```
architecture cpu of cpu                          module cpu
use my_package.all; -- contains ProcCall
begin               -- procedure

p0 : process(s)                                  always @(s)
begin                                               begin : p0
        ProcCall(s,s_out);                          ProcCall(s,s_out);
end process;                                        end

                                                 // including file with
called functions and tasks
                                                 `include
"cpu_cpu.verilog"

end;                                             endmodule
```

Concurrent Statements

VHDL concurrent statements occur in VHDL Architectures. These statements execute in concurrently, as there name suggests. **VHDL2verilog** translates concurrent statements outlined in the following sections.

Block Statements

In VHDL it is possible to group statements together in a `BLOCK`. Declarations made in the Block have a local scope. In verilog there is no concepts of Blocks. **VHDL2verilog** translates blocks by moving all its contents into the enclosing scope. Local declarations are renamed (the label of the block is prepended) and are also moved into the enclosing scope.

Example:

```
VHDL                                               Verilog

                                                     module aha

                                                     parameter B0_a =
`true;
ARCHITECTURE foo OF aha IS                           parameter B1_a =
`false;
BEGIN
B0: Block
CONSTANT a : Boolean := TRUE;
begin                                                // begin BLOCK B0
   s <= a AND b AND c;                               assign s = B0_a & b
& c;
end block;                                           // end BLOCK B0
```

```
B1: Block
CONSTANT a : Boolean := FALSE;
begin                                                  // begin BLOCK B1
   co <= (a AND b) OR (a AND c) OR (b AND c);          assign co = B1_a &
b | B1_a & c | b & c;
end block;                                             // end BLOCK B1

END foo;                                               endmodule foo
```

Process Statements

VHDL and Verilog have constructs that allow blocks of sequential statements to execute concurrently. In VHDL, these statements are PROCESS statements. PROCESS statements in VHDL are mapped to Verilog ALWAYS statements. If the VHDL process will execute only once during the simulation run (possible if the process sequential body has an infinite WAIT as its last item), then the process is mapped to the Verilog INITIAL statement.

VHDL2verilog follows specific rules for mapping signal edges (RISING EDGE, FALLING EDGE) to Verilog. These rules apply *only* to processes with the following characteristics:

Process should have

1. Sensitivity signals or wait statement, and
2. Single IF (optional ELSIF) statement, and
3. IF (optional ELSIF) expression equivalent to Verilog POSEDGE or NEGEDGE. The IEEE Std_logic_1164 package functions RISING_EDGE and FALLING_EDGE are identified by **VHDL2verilog** as POSEDGE and NEGEDGE respectively.

In this case, the edge controls are moved to the ALWAYS sensitivity list, and the IF (ELSIF) expressions are modified to be equivalent to the VHDL description. Process PROCESS_DFF in the example below illustrates this concept.

Example:

```
VHDL                                                   Verilog

architecture diffeq of diffeq is                       module diffeq;

begin

p0 : process                                           initial
-- initialize some values                              begin : p0
begin                                                    Xoutport <= 0;
  Xoutport <= 0;                                         Youtport <= 0;
```

```
  Youtport <= 0;
  Uoutport <= 0;
  WAIT;
end process p0;

P1 : process (Aport, DXport, Xinport,
or Xinport or
        Yinport, Uinport)
Uinport)

   variable x_var,y_var,u_var,
        a_var, dx_var: integer ;

begin

  Xoutport <= x_var;
  Youtport <= y_var;
  Uoutport <= u_var;


end process P1;

process_dff : process(a,b,enable)
posedge b or negedge enable)
begin
 if (enable = '0') then
    q <= '0';
 elsif b = '1' then
    q <= '1';
 elsif a'event and a'last_value = '1' then
    q <= d;
 end if;
end process process_dff  ;
```

```
  Uoutport <= 0;
end


always @(Aport or DXport

          Yinport or


begin : P1
  integer  x_var;
  integer  y_var;
  integer  u_var;
  integer  a_var;
integer  dx_var;
Xoutport <= x_var;
Youtport <= y_var;
  Uoutport <= u_var;
end


always @(negedge a or

begin : process_dff
  if (enable == 'b 0)
     q <= 'b 0;
  else if (b == 'b 1 )
     q <= 'b 1;
  else
     q <= d;
  end

endmodule
```

Concurrent Signal Assignments

Concurrent Signal Assignments are used to drive signals continuously throughout the simulation run. Verilog CONTINUOUS ASSIGNMENTS provide the same capability. All VHDL concurrent signal assignments are translated to Verilog continuous assignments.

Example:

```
VHDL

ARCHITECTURE VeriArch OF aha IS
aha(a,b,c,s,co);
SIGNAL V2V_s : std_logic;
SIGNAL V2V_co : std_logic;
BEGIN
```

```
Verilog

module

wire    V2V_s;
wire    V2V_co;
```

```
V2V_s <= a AND b AND c AFTER 2 ns;                assign #2 V2V_s = a &
b & c;
V2V_co <= (a AND V2V_s) OR (a AND c)              assign V2V_co = a &
V2V_s | a & c
                OR (b AND
c);                                             | b & c;
s <= V2V_s;                                       assign s = V2V_s;
co <= V2V_co;                                     assign co = V2V_co;
END VeriArch;                                     endmodule // module
aha
```

Component Instantiations

Both VHDL and Verilog are hierarchical languages and provide module instantiation capabilities. Additionally, values of generics can be changed during the process of module instantiation. Generic maps in VHDL component instantiations are translated to Verilog DEFPARAM statements.

Example:

```
VHDL                                                    Verilog

component inbuf -- described previously                 module
inbuf(pad,y);
  generic (len : integer := 2);                         parameter len =
2;
  port( pad : in std_logic_vector(len DOWNTO 1);        input [len:1]
pad;
        y : out std_logic_vector(len DOWNTO 1));        output [len:0] y;
end component;                                          assign y = pad;
                                                        endmodule
for all : inbuf use entity work.inbuf;

begin
   comp_1: inbuf                                        inbuf comp_1
defparam comp_1.len = 2;
        generic map (len=>2)                            inbuf comp_1
(.pad({a[2], a[1]}),
        port map ( pad(1)=>a(1), y(1) => z(1),             .y({z[2],
z[1]}));
                 pad(2)=>a(2), y(2)=>z(2));
   comp_2: inbuf                                        inbuf comp_2
defparam comp_2.len = 2;
        generic map (len=>2) port map                   inbuf comp_2
(.pad(a), .y(z));
                 ( pad=>a, y=>z);
   comp_3: inbuf                                        inbuf comp_3
defparam comp_3.len = 2;
        generic map (len=>2)
        port map ( a, z);                               inbuf comp_3 (a,
z);
end rtl;
```

Generate Statements

VHDL `GENERATE` statements are useful when regular structures need to be modeled. In Verilog however, there is no comparable construct. **VHDL2verilog** therefore *elaborates* the generate statement, and then translates the result into Verilog.

Example (VHDL):

```
Bit2 : BLOCK
CONSTANT ks : INTEGER := BitWidth + BitWidth;
CONSTANT kc : INTEGER := BitWidth-1;
CONSTANT i : INTEGER := 2;
SIGNAL in1ANDin2 : std_ulogic_vector (0 TO BitWidth-2);
BEGIN
    NextRow : FOR j IN 0 TO BitWidth-2 GENERATE
        in1ANDin2(j) <= ARR_MUL1in1(i) AND ARR_MUL1in2(j);
        NextRow1 : IF j < (BitWidth-2) GENERATE
            xx : fullxxx PORT MAP
                (in1ANDin2(j),
                carry(kc-BitWidth+1+j),
                carry(kc+j));
        END GENERATE NextRow1;
        NextRow2 : IF j = (BitWidth-2) GENERATE
            pxx : partialfullxxx PORT MAP
                (in1ANDin2(j),
                carry(kc-BitWidth+1+j),
                carry(kc+j));
        END GENERATE NextRow2;
    END GENERATE NextRow;
END BLOCK Bit2;
```

The above VHDL code is transformed into the following Verilog code:

```
// begin BLOCK Bit2
parameter Bit2_ks = BitWidth + BitWidth;
parameter Bit2_kc = BitWidth - 1;
parameter Bit2_i = 2;
wire    [0:BitWidth - 2] Bit2_in1ANDin2;
assign Bit2_in1ANDin2[0] = ARR_MUL1in1[Bit2_i] & ARR_MUL1in2[0];
assign Bit2_in1ANDin2[1] = ARR_MUL1in1[Bit2_i] & ARR_MUL1in2[1];
assign Bit2_in1ANDin2[2] = ARR_MUL1in1[Bit2_i] & ARR_MUL1in2[2];
fullxxx xx_Bit2_NextRow1_0 (Bit2_in1ANDin2[0],
          carry[Bit2_kc - BitWidth + 1 + 0],
          carry[Bit2_kc + 0]);
fullxxx xx_Bit2_NextRow1_1 (Bit2_in1ANDin2[1],
          carry[Bit2_kc - BitWidth + 1 + 1],
          carry[Bit2_kc + 1]);
partialfullxxx pxx_Bit2_NextRow2_2 (Bit2_in1ANDin2[2],
          carry[Bit2_kc - BitWidth + 1 + 2],
          carry[Bit2_kc + 2]);
// end BLOCK Bit2
```

Chapter 3:
Using VHDL2verilog

This chapter summarizes the options you can use to tailor **VHDL2verilog** to your specific needs.

Command Summary

VHDL2verilog provides several options that allow you to tailor the operation of the tool. This section summarizes the commands you use to control these options.

```
Usage: vhdl2v input_file [[input_files] output_file] [options] [-Help]
[-Usage] [-Version]

where option can be one or all of

[-File {file_name}] [-Replace] [-Silent] [-No_Comments] [-Debug] [-
No_Default_defines]
[-No_Package_translation {package_file_names}] [No_Component_Check] [-
Include_Package_files]
[-Function_Map {files}] [-Ignore_Subprogram_Calls] [-
Translate_Subprogram_Bodies] [Preserve_Generate] [Support_Multi-
dimensional array] [Ignore_Integer_Range] [Support_Directives] [-
Force_Lower_Case] [-Time_Scale {time}] [-Log {logfile_name}]
[-SYNTHesis] [-87] [-No_header] [-No_timescale]

OPTIONS

-File {file_name} [-f {file_name}]:
        Read command line arguments from the specified file

-Replace [-r]:
        Replace the existing output_file with the new output_file ;
default
        is to backup the output_file to <output_file_name>.old

-Silent [-s]:
        Suppress printing out of messages indicating translator actions

-No_Comments [-nc]:
        Supress extraction of comments from input HDL file

-Debug [-d]:
        Prints debug messages from the tool

-No_Default_defines [-nd]:
        Verilog define directives for TRUE and FALSE are present in all
output files;
        use of this switch suppresses printing of default defines
```

```
-No_Package_translation {package_names} [-np {package_names}]:
        Packages with <package_name> or having <package_name> as prefix
are not translated;
        this can be used to suppress translation of specific packages
-No_Component_Check [-ncc]:
        Translation will proceed even if some component definiations or
entity
        declarations are missing; this can be used to force translation
of individual
        components without all its components; the translation result
might not compile
        because of the absence of its conpoment;

-Include_Package_files [-ip]:
        Use of this switch will direct the translator to write the
translation of packages
        into files '<package_name>_package.verilog' and
'<package_name>_modules.verilog';
        default is to include them in the output file

-Function_Map {files} [-fm {files}]:
        Read the function mapping files specified for translation

-Ignore_Subprogram_Calls [-ssc]:
        Do not translate the headers of function and procedure calls

-Translate_Subprogram_Bodies [-tsb]:
        Translate the bodies of function and procedures into a separate
file
        (override -isc switch). Translation is done on unconditional
basis: Output is
        not guaranteed to work!
-Preserve_Generate [-pg]:
        Preserve generate statement. Generate Statements will not be
elaborated if the
        concurrent body contains only concurrent assignments
-Support_Multi-dimensional_array [-sm]:
        Bit-wise access of multi-dimensional array will be translated.
Additional signals
        or subprograms may be created for translating multi-dimensional
arrays
-Ignore_Integer_Range [-iir]:
        Ignore the integer range and translate VHDL integers to 32 bit
Verilog integers
-Support_Directives [-sd]:
        Support translation directives, such as
translate_off/translate_on
-Force_Lower_Case [-flc]:
        Changes all identifiers to lower case

-Time_Scale {timescale} [-ts {timescale}]:
        Set the Verilog 'timescale

-Log {logfile_name} [-l {logfile_name}]:
        Logs translator messages into file <logfile_name>; default log
file is 'vhdl2v.log'
```

```
-SYNTHesis [-synth]:
        Produces synthesizable code; presently restricted to
suppressing initial statements
        and using '==' Verilog operators instead of '===' operators

-87 [-87]:
        Disable support for VHDL-93; enable VHDL-87 support instead
-No_header [-nh]
        Do not print ASC header in the output file.
-No_timescale [-nt]
        Do not print timescale directive in the output file. Default is
OFF(timescale is printed in the output file).
```

VHDL2verilog Environment

Chapter Start

If **VHDL2verilog** was not installed in the default directory `/usr/local/ASC` or `C:\Program Files\ASC`, the environment variable `ASC_HOME` needs to be initialized to the location of the software. For example, if the software was installed in `/home/users/ASC`, the variable `ASC_HOME` needs to be set to `/home/users/ASC`. To initialize `ASC_HOME`, type the following at the operating system prompt (if using `csh`):

```
% setenv ASC_HOME /home/users/ASC
```

On NT the setup program will set environment variables for you. It will also update your `PATH` environment variable. This allows the user to run the translator from any location in the file system.

In the `ASC_HOME` directory, there should be a subdirectory `v2v`. If not, please make sure the `ASC_HOME` is set up to a correct directory.

Additionally, the user would also need to specify the packages being used in the input VHDL file. By default, **VHDL2verilog** looks in the current working directory for the filename `<package_name>.hdl`, and if the file is not found there, in `$ASC_HOME/v2v/lib/<package_name>.hdl`.

However, the user can create a file *vhdl2v.map* in the current directory, and specify the mapping of VHDL packages to the physical file location.

vhdl2v.map File

Chapter Start

vhdl2v.map is an ASCII file with format as shown below:

```
// This is a comment
// Package name   Name of file containing package
```

```
functions           $HOME/ASC/v2v/lib/functions_header.vhd
numeric_std         $HOME/ASC/v2v/lib/numeric_std_header.vhd
my_package          mine.vhd
bscmp               ibuf_pack.hdl
```

`//' or `#' as the first character(s) of a line denotes a comment. Everything following the comment will be ignored. A mapping line should contain two fields. The first field is the package name *exactly* as used in the VHDL file, and the second field is the *complete path* to the physical location of the VHDL file containing that particular package.

The delimiter between the two fields can be a space, tab or a combination of the above. Previously defined environment variables can be used to specify pathnames.

A default map file can be found in the `$ASC_HOME/v2v/lib` directory. Usually, the user would copy this default map file to the current directory and modify it as necessary.

On NT the following syntax is valid as well:

```
functions           %HOME%\ASC\v2v\lib\functions_header.vhd
```

Function Mapping and Translation

Mapping Functions Existing in VHDL Packages | Mapping Functions in VHDL Input | Translating Functions (or Procedures) in VHDL Input ||| Chapter Start

The primary purpose of the function mapping file is to direct **VHDL2verilog** on how it should process SUBPROGRAMs present in input VHDL.

It can be used to suppress printing of type conversion function calls widely used in VHDL. Most of the type conversions are not needed in Verilog, and **VHDL2verilog** can read in a user-supplied function mapping file(s), and utilize information therein to create optimized Verilog output.

In addition, a function map file can be used to specify a replacement string that will replace all occurences of a function in the VHDL input. Each function can have its own string. It is assumed that the object specified by the replacement string exists.

Mapping Functions Existing in VHDL Packages

Mapping Functions Existing in VHDL Packages | Mapping Functions in VHDL Input | Translating Functions (or Procedures) in VHDL Input ||| Section Start

Typically, subprograms are encapsulated in VHDL packages. An example of such a package is the IEEE Standard STD_LOGIC_1164 package. Information about mapping of functions declared inside packages is supplied via the function mapping file. The function mapping file is an ASCII file with format as shown below:

```
// Comments and empty lines are ignored
// Each entry in the file should have 5 fields
//
```

```
//    1           2            3                    4           5
// Library     Package     Function              Argument     Action

IEEE        std_logic_1164  To_bit                  1           ignore
IEEE        std_logic_1164  To_StdULogicVector      1           -
IEEE        standard        now                     $time       replace
```

The fields are:

1. Library :
    - SYNTAX: identifier
    - This field states the library.

1. Package :
    - SYNTAX: identifier
    - This field states the package name; if the file in which the package resides is not <package name>.hdl, the mapping of package name to file needs to be specified in the 'vhdl2v.map' file.
2. Function :
    - SYNTAX: identifier
    - This is the name of the function that needs to be ignored or replaced.
3. Argument :
    - SYNTAX: integer_number | string | "-"
    - The argument can be any character or character string. Based on the Action field entry (Field 5), the character string is interpreted as an integer (Action entry is "ignore" or "-") or a character string (Action entry is "replace").

      If the argument is interpreted as an integer, this denotes the function parameter that needs to be printed in the output Verilog. The parameter to be printed is specified by giving the argument number. If this field contains '-', the argument defaults to '1'. It is an error if the function does not have the argument specified.
      If the argument is interpreted as a string, this string will replace ALL occurences of the function in the output Verilog.

4. Action :
    - SYNTAX: "ignore" | "replace" | "change" | "-"
    - VHDL2verilog will ignore the function, and print only the specified argument in the output file if the entry is ``ignore" or ``-". If the entry is ``replace", the specified string will replace all occurences of the function and dropping all the arguments. If the entry is ``change", the specified string will replace all occurences of the function and keep all the arguments.

Mapping files for popular packages are available in the software distribution in the `$ASC_HOME/v2v/lib` directory. If a function map file is not found in the current directory, **VHDL2verilog** will look in the `$ASC_HOME/v2v/lib` directory.

Mapping Functions in VHDL Input

Mapping Functions Existing in VHDL Packages | Mapping Functions in VHDL Input | Translating Functions (or Procedures) in VHDL Input ||| Section Start

Functions not declared in packages can be mapped by attaching **VHDL2verilog** specific attributes in the architecture. The two attributes and attribute specification that need to be defined in an architecture block declaration are as follows:

For printing a specified argument use:

```
 attribute vhdl2v_function_action : string;
 attribute vhdl2v_function_argument : integer;

 attribute vhdl2v_function_action of function_name : function is
"ignore";
 attribute vhdl2v_function_argument of function_name : function is 1;
```

In the above case, **VHDL2verilog** will ignore the printing of the function name, and only print the 1st argument of the function call.

For replacing a specific function use:

```
  attribute vhdl2v_function_action : string;
  attribute vhdl2v_function_argument : string;

  attribute vhdl2v_function_action of function_name : function is
"replace";
  attribute vhdl2v_function_argument of function_name : function is
"whatever";
```

In the above case, string ``whatever'' will replace all occurences of functions_name.

Translating Functions (or Procedures) in VHDL Input

Mapping Functions Existing in VHDL Packages | Mapping Functions in VHDL Input | Translating Functions (or Procedures) in VHDL Input ||| Section Start

With the new 2.7 release of **VHDL2verilog** functions (and procedure calls) can be translated with the `-ssc` switch. The subprogram that gets called will be translated in the following manner:

```
...

FUNCTION foo (vec : std_logic_vector(7 DOWNTO 0)) return BOOLEAN;
...
```

If the above function is called in an architecture that is being translated, **VHDL2verilog** will also create a file that contains the following Verilog code:

```
function  foo;
```

```
input   [7:0] vec;

begin

//
// *** INSERT THE TRANSLATED BODY HERE ***
//

end
endfunction
```
If the body of the subprogram was available to the translator (e.g. in a `PACKAGE BODY`), the local signals, variables, constants, and local types will be translated as well.

DFF mapping in VHDL2verilog

Chapter Start

The translation of Dff-equivalent processes is restricted to a few templates.

NOTE: edges can be rising or falling; for example purposes, the rising edge has been chosen.

The main templates are:

1. An equivalent `PROCESS` in Verilog will be created if the *first* or *last* statement in the `PROCESS` sequential body is found to match the `WAIT` statements below. The process should have NO sensitivity signals.

   ```
    process
   begin
     -- one of the following wait statements:
     wait until rising_edge(clk);
     wait until (clk'event AND clk'last_value = '0');
     wait until (clk'event AND clk'last_value = '0' AND clk = '1');
     wait until (clk'event AND clk = '1' );
     wait until (NOT clk'STABLE  AND clk'last_value = '0');
     wait until (NOT clk'STABLE  AND clk'last_value = '0' AND clk =
   '1');
     wait until (NOT clk'STABLE  AND clk = '1');
     -- can be any legal VHDL construct in the sequential body
   end process;
   ```
   For all these cases, output Verilog is:
   ```
   always @(posedge clk)
   begin
     // Verilog statement
   end
   ```

2. If the process has two sensitivity signals, and one sensitivity signal corresponds to asynchronous reset, and one to clock, and there is ONLY one `IF-ELSIF-END-IF` statement in the `PROCESS` sequential body, an equivalent Verilog process is created as follows:

```
process (reset, clk)
begin
  if reset = '1' then
    q <= '0';
  -- one of the following elsif expressions:
  elsif rising_edge(clk) then
  elsif (clk'event AND clk'last_value = '0') then
  elsif (clk'event AND clk = '1' ) then
  elsif (NOT clk'STABLE  AND clk'last_value = '0' AND clk = '1')
then
  elsif (NOT clk'STABLE  AND clk'last_value = '0') then
  elsif (NOT clk'STABLE  AND clk = '1') then
    q <= d;
  end if;
end process;
```

Output Verilog is:

```
always @(posedge clk or posedge reset)
begin
  if (reset == 1'b 1)
    q <= 'b 0
  else
    q <= d;
end
```

3. If the process has one sensitivity signal corresponding to clock and there is ONLY one `IF-ELSIF-END-IF` statement in the `PROCESS` sequential body, the `PROCESS` is mapped to an equivalent process in Verilog.

```
process (clk)
begin
  -- one of the following if expressions:
  if rising_edge(clk) then
  if (clk'event AND clk'last_value = '0') then
  if (clk'event AND clk'last_value = '0' AND clk = '1') then
  if (clk'event AND clk = '1' ) then
  if (NOT clk'STABLE  AND clk'last_value = '0') then
  if (NOT clk'STABLE  AND clk'last_value = '0' AND clk = '1')
then
  if (NOT clk'STABLE  AND clk = '1') then
    q <= d;
  end if;
end process;
```

Output Verilog is:

```
 always @(posedge clk)
begin
    q <= d;
end
```

NOTE: All the above templates are also supported for Verilog `negedge` expressions (the `'clk'` value being `'0'` instead of `'1'`)

Importing (parts of) Verilog Files

Chapter Start

Tool-specific attributes can be defined in the VHDL entity declaration to import pre-existing Verilog files. Currently, the framework of attributes provides support for full Verilog module import.

To import Verilog files in **VHDL2verilog**, the user can define the attribute `VHDL2V_MODULE` in the entity declaration. The type mark of this attribute needs to be of type `STRING`. Next, the attribute needs to be specified in the entity declaration. The expression in the attribute specification should also be a string. The pre-defined format of the string is explained below. Optional parts of the string in attribute specification are in italics.

```
( <file_to_read_from> ) (<begin_line>, <end_line>)
```

Spaces are ignored. It is an error to define multiple `VHDL2V_MODULE` attribute specifications. If begin and end line numbers for the file are not specified, the whole file is imported. If begin and end line numbers are specified, the line(s) starting from the specified begin line number till the end line number are read in.

For example, consider the following attribute definition and specification:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY aha IS
PORT (SIGNAL s : OUT std_logic;
      SIGNAL co : OUT std_logic;
      SIGNAL a : IN std_logic;
      SIGNAL b : IN std_logic;
      SIGNAL c : IN std_logic);

attribute vhdl2v_module : string;
attribute vhdl2v_module of aha : entity is
        "(/home/users/foo/verilog/demo.v) (1,20)";
END aha;
```

In the above example, because of the specification of attribute `VHDL2V_MODULE`, **VHDL2verilog** will paste line 1 to 20 of file `/home/users/foo/verilog/demo.v` in the output file, instead of translating the given ENTITY and ARCHITECTURE description of `aha` to Verilog.

NOTE: No sematic, syntax or error checking is performed on the imported file. The same version that exists on disk is pasted in the output file.

Reserved Words

Chapter Start

The following words are reserved in **VHDL2verilog** and cannot be used in any of the input files. If one or more of the words below does occur, an error is flagged.

```
always assign
buf bufif0 bufif1
casex casez cmos
deassign default defparam disable
edge else end endattribute endcase endmodule endfunction endprimitive
endspecify endtable endtask event
force forever fork function
highz0 highz1
initial inout input
join
large
macromodule medium module
negedge nmos notif0 notif1
output
parameter pmos posedge primitive pull0 pull1 pullup pulldown
rcmos reg release repeat rnmos rpmos rtran rtranif0 rtranif1
scalared small specify specparam strength strong0 strong1 supply0
supply1
table task tran tranif0 tranif1 tri tri0 tri1 triand trior trireg
vectored
wand weak0 weak1 while wire wor
```