XILINX®

Chapter 3

# VHDL Support

XST supports the VHSIC Hardware Description Language (VHDL) except as otherwise noted.

- VHDL compactly describes complicated logic.
- VHDL allows you to:
  - Describe the structure of a system:
    - How the system is decomposed into subsystems.
    - How those subsystems are interconnected.
  - Specify the function of a system using familiar programming language forms.
  - Simulate a system design before it is implemented and programmed in hardware.
  - Produce a detailed, device-dependent version of a design to be synthesized from a more abstract specification.

For more information, see:

- *IEEE VHDL Language Reference Manual (LRM)*
- Chapter 9, Design Constraints, especially VHDL Attributes

## VHDL IEEE Support

The XST parsing and elaboration engine complies with VHDL IEEE 1076-1993.

XST supports non-LRM compliant constructs when the construct:

- Is supported by most synthesis and simulation tools.
- Greatly simplifies coding.
- Does not cause negatively impact synthesis.
- Does not negatively impact quality of results.

### Non-LRM Compliant Example

- The LRM does not allow instantiation with a port map if:
  - A formal port is a buffer, and
  - The corresponding effective port is an **out**.
- XST supports this non-LRM compliant construct. The construct meets the criteria stated above in *XST Support for Non-LRM Compliant Constructs*.

# VHDL Data Types

Some VHDL data types are part of predefined packages.

For information on where they are compiled, and how to load them, see VHDL Predefined Packages.

## VHDL Unsupported Data Types

VHDL supports the **real** type defined in the standard package for calculations only, such as the calculation of generics values.

You cannot define a synthesizable object of type **real**.

## VHDL Data Types

VHDL data types include:

- VHDL Predefined Enumerated Types
- VHDL User-Defined Enumerated Types
- VHDL Bit Vector Types
- VHDL Integer Types
- VHDL Multi-Dimensional Array Types
- VHDL Record Types

### VHDL Predefined Enumerated Types

XST supports the following predefined VHDL enumerated types for hardware description:

- The **bit** type, defined in the standard package.

  Allowed values are **0** (logic zero) and **1** (logic 1).
- The **boolean** type, defined in the standard package.

  Allowed values are **false** and **true.**
- The type defined in the IEEE **std_logic_1164** package.

  For allowed values, see the *std_logic Allowed Values* table below.

This information is summarized in the following table.

#### Predefined VHDL Enumerated Types Summary

| Enumerated Type | Defined In | Allowed Values |
|---|---|---|
| bit | standard package | • 0 (logic zero)<br>• 1 (logic 1) |
| boolean | standard package | • false<br>• true |
| std_logic | IEEE std_logic_1164 package | See the *std_logic Allowed Values* table below. |

### std_logic Allowed Values

| Value | Meaning | What XST does |
|---|---|---|
| U | unitialized | Not accepted by XST |
| X | unknown | Treated as don't care |
| 0 | low | Treated as logic zero |
| 1 | high | Treated as logic one |
| Z | high impedance | Treated as high impedance |
| W | weak unknown | Not accepted by XST |
| L | weak low | Treated identically to **0** |
| H | weak high | Treated identically to **1** |
| - | don't care | Treated as don't care |

### XST-Supported Overloaded Enumerated Types

| Type | Defined In IEEE Package | SubType Of | Contains Values |
|---|---|---|---|
| std_ulogic | std_logic_1164 | N/A | • Same nine values as std_logic<br>• Does not contain predefined resolution functions |
| X01 | std_logic_1164 | std_ulogic | X, 0, 1 |
| X01Z | std_logic_1164 | std_ulogic | X, 0, 1, Z |
| UX01 | std_logic_1164 | std_ulogic | U, X, 0 1 |
| UX01Z | std_logic_1164 | std_ulogic | U, X, 0, Z |

## VHDL User-Defined Enumerated Types

You can create your own enumerated types.

User-defined enumerated types usually describe the states of a Finite State Machine (FSM).

### VHDL User-Defined Enumerated Types Coding Example

```
type STATES is (START, IDLE, STATE1, STATE2, STATE3) ;
```

## VHDL Bit Vector Types

### Supported VHDL Bit Vector Types

| Type | Defined In Package | Models |
|---|---|---|
| bit_vector | Standard | Vector of bit elements |
| std_logic_vector | IEEE std_logic_1164 | Vector of std_logic elements |

### Supported VHDL Overloaded Types

| Type | Defined In IEEE Package |
|---|---|
| std_ulogic_vector | std_logic_1164 |
| unsigned | std_logic_arith |
| signed | std_logic_arith |

## VHDL Integer Types

The integer type is a predefined VHDL type.

- XST implements an integer on 32 bits by default.
- For a more compact implementation, define the exact range of applicable values.

```
type MSB is range 8 to 15
```

- You can also take advantage of the predefined natural and positive types, overloading the integer type.

## VHDL Multi-Dimensional Array Types

XST supports VHDL multi-dimensional array types.

- Although there is no restriction on the number of dimensions, Xilinx® recommends that you describe no more than three dimensions.
- Objects of multi-dimensional array type that you can describe are:
  - Signals
  - Constants
  - Variables
- Objects of multi-dimensional array type can be:
  - Passed to functions.
  - Used in component instantiations.

### Fully Constrained Array Type Coding Example

An array type must be fully constrained in all dimensions.

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (11 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB12;
```

### Array Declared as a Matrix Coding Example

You can declare an array as a matrix.

```
subtype TAB13 is array (7 downto 0,4 downto 0) of STD_LOGIC_VECTOR (8 downto 0);
```

### Multi-Dimensional Array Signals and Variables Coding Examples

These coding examples demonstrate the uses of multi-dimensional array signals and variables in assignments.

1. Make the following declarations:

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB05 is array (4 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB05;
signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB05;
signal TAB_C, TAB_D : TAB03;
constant CNST_A : TAB03 := (
("00000000","01000001","01000010","10000011","00001100"),
("00100000","00100001","00101010","10100011","00101100"),
("01000010","01000010","01000100","01000111","01000100"));
```

2. You can now specify:
   - A multi-dimensional array signal or variable

     ```
     TAB_A <= TAB_B; TAB_C <= TAB_D; TAB_C <= CNST_A;
     ```

   - An index of one array

     ```
     TAB_A (5) <= WORD_A; TAB_C (1) <= TAB_A;
     ```

   - Indexes of the maximum number of dimensions

     ```
     TAB_A (5) (0) <= '1'; TAB_C (2) (5) (0) <= '0'
     ```

   - A slice of the first array

     ```
     TAB_A (4 downto 1) <= TAB_B (3 downto 0);
     ```

   - An index of a higher level array and a slice of a lower level array

```
TAB_C (2) (5) (3 downto 0) <= TAB_B (3) (4 downto 1); TAB_D (0) (4) (2 downto 0)
\\ <= CNST_A (5 downto 3)
```

3. Add the following declaration:

```
subtype MATRIX15 is array(4 downto 0, 2 downto 0) of STD_LOGIC_VECTOR (7 downto 0);
signal MATRIX_A : MATRIX15;
```

4. You can now specify:
   - A multi-dimensional array signal or variable

     ```
     MATRIXA <= CNST_A
     ```

   - An index of one row of the array

     ```
     MATRIXA (5) <= TAB_A;
     ```

   - Indexes of the maximum number of dimensions

     ```
     MATRIXA (5,0) (0) <= '1';
     ```

     Indexes can be variable.

## VHDL Record Types

```
type mytype is record
field1 : std_logic;
field2 : std_logic_vector (3 downto 0)
end record;
```

- A field of a record type can also be of type **record**.
- Constants can be record types.
- Record types cannot contain attributes.
- XST supports aggregate assignments to record signals.

# VHDL Objects

VHDL objects include:

- VHDL Signals
- VHDL Variables
- VHDL Constants

## VHDL Signals

Declare a VHDL signal in:

- An architecture declarative part

  Use the VHDL signal anywhere within that architecture.
- A block

  Use the VHDL signal within that block.

Assign the VHDL signal with the **<=** signal assignment operator.

```
signal sig1 : std_logic;
sig1 <= '1';
```

## VHDL Variables

A VHDL variable is:

- Declared in a process or a subprogram.
- Used within that process or subprogram.
- Assigned with the **:=** assignment operator.

```
variable var1 : std_logic_vector (7 downto 0); var1 := "01010011";
```

## VHDL Constants

You can declare a VHDL constant in any declarative region.

- The constant is used within that region.
- The constant values cannot be changed once declared.

```
signal sig1 : std_logic_vector (5 downto 0);constant init0 :
std_logic_vector (5 downto 0) := "010111";sig1 <= init0;
```

# VHDL Operators

XST supports VHDL operators. See VHDL Operators Support.

### Shift Operator Examples

| Operator | Example | Logically Equivalent To |
|---|---|---|
| SLL (Shift Left Logic) | sig1 <= A(4 downto 0) sll 2 | sig1 <= A(2 downto 0) & "00"; |
| SRL (Shift Right Logic) | sig1 <= A(4 downto 0) srl 2 | sig1 <= "00" & A(4 downto 2); |
| SLA (Shift Left Arithmetic) | sig1 <= A(4 downto 0) sla 2 | sig1 <= A(2 downto 0) & A(0) & A(0); |
| SRA (Shift Right Arithmetic) | sig1 <= A(4 downto 0) sra 2 | sig1 <= <= A(4) & A(4) & A(4 downto 2); |
| ROL (Rotate Left) | sig1 <= A(4 downto 0) rol 2 | sig1 <= A(2 downto 0) & A(4 downto 3); |
| ROR (Rotate Right) | A(4 downto 0) ror 2 | sig1 <= A(1 downto 0) & A(4 downto 2); |

# VHDL Entity and Architecture Descriptions

VHDL entity and architecture descriptions include:



## VHDL Circuit Descriptions

A VHDL circuit description (design unit) consists of:

- Entity declaration
  - Provides the *external* view of the circuit.
  - Describes objects visible from the outside, including the circuit interface, such as the I/O ports and generics.
- Architecture
  - Provides the *internal* view of the circuit.
  - Describes the circuit behavior or structure.

## VHDL Entity Declarations

The I/O ports of the circuit are declared in the entity.

Each port has a:

- name
- mode
  - in
  - out
  - inout
  - buffer
- type

## Constrained and Unconstrained Ports

Ports can be constrained or unconstrained.

- Ports are usually constrained.
- Ports can be left unconstrained in the entity declaration.
- If ports are left unconstrained, their width is defined at instantiation when the connection is made between formal ports and actual signals.
- Unconstrained ports allow you to create different instantiations of the same entity, defining different port widths.
- Xilinx® recommends:
  - Do not use unconstrained ports.
  - Define ports that are constrained through generics.
  - Apply different values of those generics at instantiation.
  - Do not have an unconstrained port on the top-level entity.
- Array types of more than one-dimension are not accepted as ports.
- The entity declaration can also declare VHDL generics.

## Buffer Port Mode

Xilinx recommends that you not use buffer port mode.

- VHDL allows buffer port mode when a signal is used both:
  - Internally, and
  - As an output port when there is only one internal driver.
- Buffer ports:
  - Are a potential source of errors during synthesis.
  - Complicate validation of post-synthesis results through simulation.

### NOT RECOMMENDED Coding Example WITH Buffer Port Mode

```
entity alu is
    port(
        CLK : in  STD_LOGIC;
        A   : in  STD_LOGIC_VECTOR(3 downto 0);
        B   : in  STD_LOGIC_VECTOR(3 downto 0);
        C   : buffer STD_LOGIC_VECTOR(3 downto 0));
end alu;

architecture behavioral of alu is
begin
    process begin
        if rising_edge(CLK) then
            C <= UNSIGNED(A) + UNSIGNED(B) UNSIGNED(C);
        end if;
    end process;
end behavioral;
```

## Dropping Buffer Mode

Xilinx recommends that you drop buffer port mode.

- In the coding example above, signal C:
  - Has been modeled with a buffer mode.
  - Is used both internally and as an output port.
- Every level of hierarchy that can be connected to C must also be declared as a buffer.
- To drop buffer mode:
  1. Insert a dummy signal.
  2. Declare port C as an output.

### RECOMMENDED Coding Example WITHOUT Buffer Port Mode

```
entity alu is
    port(
        CLK : in  STD_LOGIC;
        A   : in  STD_LOGIC_VECTOR(3 downto 0);
        B   : in  STD_LOGIC_VECTOR(3 downto 0);
        C   : out STD_LOGIC_VECTOR(3 downto 0));
end alu;

architecture behavioral of alu is
    -- dummy signal
    signal C_INT : STD_LOGIC_VECTOR(3 downto 0);
begin
    C <= C_INT;
    process begin
        if rising_edge(CLK) then
            C_INT <= A and B and C_INT;
        end if;
    end process;
end behavioral;
```

## VHDL Architecture Declarations

You can declare internal signals in the architecture.

Each internal signal has a:

- name
- type

### VHDL Architecture Declaration Coding Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity EXAMPLE is
    port (
        A,B,C : in std_logic;
        D,E : out std_logic );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
    signal T : std_logic;
begin
    ...
end ARCHI;
```

# VHDL Component Instantiation

Component instantiation allows you to instantiate one design unit (component) inside another design unit in order to create a hierarchically structured design description.

To perform component instantiation:

1. Create the design unit (entity and architecture) modeling the functionality to be instantiated.
2. Declare the component to be instantiated in the declarative region of the parent design unit architecture.
3. Instantiate and connect this component in the architecture body of the parent design unit.
4. Map (connect) formal ports of the component to actual signals and ports of the parent design unit.

## Elements of Component Instantiation Statement

The main elements of a component instantiation statement are:

- Label

  Identifies the instance.

- Association list
  - Introduced by the reserved **`port map`** keyword.
  - Ties formal ports of the component to actual signals or ports of the parent design unit.
- Optional association list
  - Introduced by the reserved **`generic map`** keyword.
  - Provides actual values to formal generics defined in the component.

XST supports unconstrained vectors in component declarations.

### VHDL Component Instantiation Coding Example

This coding example shows the structural description of a half-Adder composed of four nand2 components.

```
--
-- A simple component instantiation example
--   Involves a component declaration and the component instantiation itself
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/instantiation/instantiation_simple.vhd
--
entity sub is
    generic (
        WIDTH : integer := 4);
    port (
        A,B : in  BIT_VECTOR(WIDTH-1 downto 0);
        O   : out BIT_VECTOR(2*WIDTH-1 downto 0));
end sub;

architecture archi of sub is
begin
    O <= A & B;
end ARCHI;

entity top is
    generic (
        WIDTH : integer := 2);
     port (
        X, Y : in BIT_VECTOR(WIDTH-1 downto 0);
        Z    : out BIT_VECTOR(2*WIDTH-1 downto 0));
end top;

architecture ARCHI of top is

    component sub -- component declaration
        generic (
            WIDTH : integer := 2);
        port (
            A,B : in  BIT_VECTOR(WIDTH-1 downto 0);
            O   : out BIT_VECTOR(2*WIDTH-1 downto 0));
    end component;

begin

    inst_sub : sub -- component instantiation
        generic map (
   WIDTH => WIDTH
 )
 port map (
   A => X,
   B => Y,
   O => Z
 );

end ARCHI;
```

## VHDL Recursive Component Instantiation

XST supports VHDL recursive component instantiation.

- XST does not support direct instantiation for recursion.
- To prevent endless recursive calls, the number of recursions is limited by default to 64.
- Use **-recursion_iteration_limit** to specify the number of allowed recursive calls. See the following coding example.

### VHDL Recursive Component Instantiation Coding Example

```
--
-- Recursive component instantiation
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/instantiation/instantiation_recursive.vhd
--
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity single_stage is
    generic (
        sh_st: integer:=4);
    port (
        CLK : in std_logic;
 DI : in std_logic;
 DO : out std_logic );
end entity single_stage;

architecture recursive of single_stage is
    component single_stage
        generic (
            sh_st: integer);
        port (
            CLK : in std_logic;
            DI : in std_logic;
            DO : out std_logic );
    end component;
 signal tmp : std_logic;
begin
    GEN_FD_LAST: if sh_st=1 generate
        inst_fd: FD port map (D=>DI, C=>CLK, Q=>DO);
    end generate;
    GEN_FD_INTERM: if sh_st /= 1 generate
        inst_fd: FD port map (D=>DI, C=>CLK, Q=>tmp);
        inst_sstage: single_stage
            generic map (sh_st => sh_st-1)
            port map (DI=>tmp, CLK=>CLK, DO=>DO);
    end generate;
end recursive;
```

## VHDL Component Configuration

A component configuration explicitly links a component with the appropriate model.

- A model is an entity and architecture pair.
- XST supports component configuration in the declarative part of the architecture.

```
for instantiation_list :  component_name use
LibName.entity_Name(Architecture_Name);
```

- The statement below indicates that:
  - All NAND2 components use the design unit consisting of entity NAND2 and architecture ARCHI.
  - The design unit is compiled in the work library.

```
For all : NAND2 use entity work.NAND2(ARCHI);
```

- If the configuration clause is missing for a component instantiation:
  - XST links the component to the entity with the same name (and same interface).
  - XST links the selected architecture to the most recently compiled architecture.
- XST generates a Black Box during synthesis if no entity or architecture is found.
- In command line mode, you may use a dedicated configuration declaration to link component instantiations to design entities and architectures.
- The value of the mandatory Top Module Name (**-top**) option in the run command is the *configuration name* instead of the *top level entity name*.

## VHDL Generics

VHDL generics:

- Are the equivalent of Verilog parameters.
- Help you create scalable design modelizations.
- Allow you to write compact, factorized VHDL code.
- Allow you to parameterize functionality such as:
  - Bus sizes
  - The amount of certain repetitive elements in the design unit

### Parameterize Functionality Example

For the same functionality that must be instantiated multiple times, but with different bus sizes, you need describe only one design unit with generics. See *VHDL Generic Parameters Coding Example* below.

### Declaring Generics

You can declare generic parameters in the entity declaration part.

- XST supports all types for generics including:
  - integer
  - boolean
  - string
  - real
  - std_logic_vector
- Declare a generic with a default value.

## VHDL Generic Parameters Coding Example

```
--
-- VHDL generic parameters example
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/generics/generics_1.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity addern is
    generic (
        width : integer := 8);
    port (
        A,B : in std_logic_vector (width-1 downto 0);
        Y : out std_logic_vector (width-1 downto 0) );
end addern;

architecture bhv of addern is
begin
    Y <= A + B;
end bhv;

Library IEEE;
use IEEE.std_logic_1164.all;

entity top is
    port (
        X, Y, Z : in std_logic_vector (12 downto 0);
 A, B : in std_logic_vector (4 downto 0);
 S :out std_logic_vector (17 downto 0) );
end top;

architecture bhv of top is
    component addern
        generic (width : integer := 8);
        port (
            A,B : in std_logic_vector (width-1 downto 0);
            Y : out std_logic_vector (width-1 downto 0) );
    end component;
 for all : addern use entity work.addern(bhv);

    signal C1 : std_logic_vector (12 downto 0);
    signal C2, C3 : std_logic_vector (17 downto 0);
begin
    U1 : addern generic map (width=>13) port map (X,Y,C1);
    C2 <= C1 & A;
    C3 <= Z & B;
    U2 : addern generic map (width=>18) port map (C2,C3,S);
end bhv;
```

# Conflicts Among VHDL Generics and Attributes

Conflicts can arise among VHDL generics and attributes because:

- You can apply VHDL generics and attributes to both *instances* and *components* in the HDL source code.

  AND

- You can specify *attributes* in a constraints file.

## Rules for Conflict Resolution

XST resolves the conflicts among VHDL generics and attributes as follows:

- Specifications on an *instance* (lower level) take precedence over specifications on a *component* (higher level).
- If a generic and an attribute are applicable to the same instance or the same component, the attribute takes precedence over the generic, regardless of where the generic was specified.

  Do not use both mechanisms to define the same constraint. XST flags such occurrences.
- An attribute specified in the XST Constraint File (XCF) takes precedence over attributes or generics specified in the VHDL code.
- Security attributes on the block definition take precedence over any other attribute or generic.

This information is summarized in the following table.

### Rules for Conflict Resolution Summary

| Item | Takes Precedence Over |
|---|---|
| Specifications on an instance (lower level) | Specifications on a component (higher level) |
| Attribute applied to an instance or component | Generic applied to the same instance or the same component |
| Attribute specified in the XST Constraint File (XCF) | Attributes or generics specified in the VHDL code |
| Security attributes on the block definition | Any other attribute or generic |

# VHDL Combinatorial Circuits

XST supports the following VHDL combinatorial circuits:

- VHDL Concurrent Signal Assignments
- VHDL Generate Statements
- VHDL Combinatorial Processes

## VHDL Concurrent Signal Assignments

Combinatorial logic is described using concurrent signal assignments.

- Concurrent signal assignments are specified in the body of an architecture.
- VHDL supports three types of concurrent signal assignments:
  - Simple
  - Selected (with-select-when)
  - Conditional (when-else)
- You can describe as many concurrent signal assignments as are necessary.
- The order of appearance of the concurrent signal assignments in the architecture is irrelevant.
- All concurrent signal assignments are concurrently active.
- A concurrent signal assignment is re-evaluated when any signal on the right side of the assignment changes value.
- The re-evaluated result is assigned to the signal on the left-hand side.

### Simple Signal Assignment VHDL Coding Example

```
T <= A and B;
```

### Concurrent Selection Assignment VHDL Coding Example

```
--
-- Concurrent selection assignment in VHDL
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/concurrent_selected_assignment.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity concurrent_selected_assignment is
    generic (
        width: integer := 8);
    port (
        a, b, c, d : in std_logic_vector (width-1 downto 0);
        sel : in std_logic_vector (1 downto 0);
        T : out std_logic_vector (width-1 downto 0) );
end concurrent_selected_assignment;

architecture bhv of concurrent_selected_assignment is
begin
    with sel select
        T <= a when "00",
             b when "01",
             c when "10",
             d when others;
end bhv;
```

### Concurrent Conditional Assignment (When-Else) VHDL Coding Example

```
--
-- A concurrent conditional assignment (when-else)
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/concurrent_conditional_assignment.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity concurrent_conditional_assignment is
    generic (
     width: integer := 8);
    port (
        a, b, c, d : in std_logic_vector (width-1 downto 0);
        sel : in std_logic_vector (1 downto 0);
        T : out std_logic_vector (width-1 downto 0) );
end concurrent_conditional_assignment;

architecture bhv of concurrent_conditional_assignment is
begin
    T <= a when sel = "00" else
         b when sel = "01" else
         c when sel = "10" else
         d;
end bhv;
```

# VHDL Generate Statements

VHDL generate statements include:

- VHDL For-Generate Statements
- VHDL If-Generate Statements

## VHDL For-Generate Statements

VHDL **for-generate** statements describe repetitive structures.

### For-Generate Statement VHDL Coding Example

In this coding example, the **for-generate** statement describes the calculation of the result and carry out for each bit position of this 8-bit Adder.

```
--
-- A for-generate example
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/for_generate.vhd
--
entity for_generate is
    port (
        A,B  : in  BIT_VECTOR (0 to 7);
        CIN  : in  BIT;
        SUM  : out BIT_VECTOR (0 to 7);
        COUT : out BIT );
end for_generate;

architecture archi of for_generate is
    signal C : BIT_VECTOR (0 to 8);
begin
    C(0) <= CIN;
    COUT <= C(8);
    LOOP_ADD : for I in 0 to 7 generate
        SUM(I) <= A(I) xor B(I) xor C(I);
        C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
    end generate;
end archi;
```

## VHDL If-Generate Statements

- An **if-generate** statement activates specific parts of the HDL source code based on a test result.
- The **if-generate** statement is supported for static (non-dynamic) conditions.

### If-Generate Example

- A generic indicates which device family is being targeted.
- The **if-generate** statement:
  - Tests the value of the generic against a specific device family.
  - Activates a section of the HDL source code written specifically for that device family.

### For-Generate Nested in an If-Generate Statement VHDL Coding Example

In this coding example, a generic **N-bit** Adder with a width ranging between **4**and **32** is described with an **if-generate** and a **for-generate** statement.

```
--
-- A for-generate nested in a if-generate
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/if_for_generate.vhd
--
entity if_for_generate is
    generic (
        N : INTEGER := 8);
    port (
        A,B : in BIT_VECTOR (N downto 0);
        CIN : in BIT;
        SUM : out BIT_VECTOR (N downto 0);
        COUT : out BIT );
end if_for_generate;

architecture archi of if_for_generate is
    signal C : BIT_VECTOR (N+1 downto 0);
begin
    IF_N: if (N>=4 and N<=32) generate
        C(0) <= CIN;
        COUT <= C(N+1);
        LOOP_ADD : for I in 0 to N generate
            SUM(I) <= A(I) xor B(I) xor C(I);
            C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
        end generate;
    end generate;
end archi;
```

# VHDL Combinatorial Processes

VHDL combinatorial logic can be modeled with a process.

- A process is combinatorial when 1) signals assigned in the process, 2) are explicitly assigned a new value, 3) every time the process is executed.
- No such signal should implicitly retain its current value.
- A process can contain local variables.

## Memory Elements

Hardware inferred from a combinatorial process does not involve any memory elements.

- A process is combinatorial when 1) all assigned signals in a process 2) are always explicitly assigned 3) in all possible paths within a process block.
- A signal that is not explicitly assigned in all branches of an **if** or **case** statement typically leads to a Latch inference.
- If XST infers unexpected Latches, review the HDL source code for a signal that is not explicitly assigned.

## Sensitivity List

A combinatorial process has a sensitivity list.

- The sensitivity list appears within parentheses after the **`process`** keyword.
- A process is activated if an event (value change) appears on one of the sensitivity list signals.
- For a combinatorial process, this sensitivity list must contain:
  - All signals in conditions (for example, **if** and **case**).
  - All signals on the right-hand side of an assignment.

## Missing Signals

Signals may be missing from the sensitivity list.

- If one or more signals is missing from the sensitivity list:
  - The synthesis results can differ from the initial design specification.
  - XST issues a warning message.
  - XST adds the missing signals to the sensitivity list.
- To avoid problems during simulation:
  - Explicitly add all missing signals in the HDL source code.
  - Re-run synthesis.

## VHDL Variable and Signal Assignments

XST supports VHDL variable and signal assignments.

- A process can contain local variables.
- Local variables are:
  - Declared and used within a process.
  - Generally not visible outside the process.

### Signal Assignment in a Process VHDL Coding Example

```
--
-- Signal assignment in a process
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/signals_variables/signal_in_process.vhd
--
entity signal_in_process is
    port (
        A, B : in BIT;
        S : out BIT );
end signal_in_process;

architecture archi of signal_in_process is
begin
    process (A, B)
    begin
        S <= '0' ;
        if ((A and B) = '1') then
            S <= '1' ;
        end if;
    end process;
end archi;
```

### Variable and Signal Assignment in a Process VHDL Coding Example

```
--
-- Variable and signal assignment in a process
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/signals_variables/variable_in_process.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity variable_in_process is
    port (
        A,B     : in  std_logic_vector (3 downto 0);
        ADD_SUB : in  std_logic;
        S       : out std_logic_vector (3 downto 0) );
end variable_in_process;

architecture archi of variable_in_process is
begin
    process (A, B, ADD_SUB)
        variable AUX : std_logic_vector (3 downto 0);
    begin
        if ADD_SUB = '1' then
            AUX := A + B ;
        else
            AUX := A - B ;
        end if;
        S <= AUX;
    end process;
end archi;
```

## VHDL If-Else Statements

**if-else** and **if-elsif-else** statements use **true-false** conditions to execute statements.

- If the expression evaluates to **true**, the **if** branch is executed.
- If the expression evaluates to **false**, **x**, or **z**, the **else** branch is executed.
- A block of multiple statements is executed in an **if** or **else** branch.
- **begin** and **end** keywords are required.
- **if-else** statements can be nested.

### If-Else Statement VHDL Coding Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
    port (
        a, b, c, d : in std_logic_vector (7 downto 0);
        sel1, sel2 : in std_logic;
        outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is
begin
    process (a, b, c, d, sel1, sel2)
    begin
        if (sel1 = '1') then
            if (sel2 = '1') then
                outmux <= a;
            else
                outmux <= b;
            end if;
        else
            if (sel2 = '1') then
                outmux <= c;
            else
                outmux <= d;
            end if;
        end if;
    end process;
end behavior;
```

## VHDL Case Statements

A VHDL **case** statement:

- Performs a comparison to an expression in order to evaluate one of several parallel branches.
- Evaluates the branches in the order in which they are written.
- Executes the first branch that evaluates to **true**.
- Executes the default branch if none of the branches match.

### Case Statement VHDL Coding Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
    port (
        a, b, c, d : in std_logic_vector (7 downto 0);
        sel : in std_logic_vector (1 downto 0);
        outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is
begin
    process (a, b, c, d, sel)
    begin
        case sel is
            when "00" => outmux <= a;
            when "01" => outmux <= b;
            when "10" => outmux <= c;
            when others => outmux <= d; -- case statement must be complete
        end case;
    end process;
end behavior;
```

## VHDL For-Loop Statements

XST supports VHDL **for-loop** statements for:

- Constant bounds
- Stop test condition using any of the following operators:
  - <
  - <=
  - >
  - >=
- Next step computation falling within one of the following specifications:
  - *var* = *var* + step
  - *var* = *var* - step
    - *var* is the loop variable
    - **step** is a constant value
- **Next** and **exit** statements

### For-Loop VHDL Coding Example

```
--
-- For-loop example
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/combinatorial/for_loop.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity countzeros is
    port (
        a : in std_logic_vector (7 downto 0);
        Count : out std_logic_vector (2 downto 0) );
end countzeros;

architecture behavior of countzeros is
    signal Count_Aux: std_logic_vector (2 downto 0);
begin
    process (a, Count_Aux)
    begin
        Count_Aux <= "000";
        for i in a'range loop
            if (a(i) = '0') then
                Count_Aux <= Count_Aux + 1;
            end if;
        end loop;
        Count <= Count_Aux;
    end process;
end behavior;
```

# VHDL Sequential Logic

VHDL sequential logic includes:

- VHDL Sequential Processes With a Sensitivity List
- VHDL Sequential Processes Without a Sensitivity List
- VHDL Initial Values and Operational Set/Reset
- VHDL Default Initial Values on Memory Elements

## VHDL Sequential Processes With a Sensitivity List

A VHDL process is sequential (as opposed to combinatorial) when 1) some assigned signals, 2) are not explicitly assigned, 3) in all paths within the process.

The hardware generated has an internal state or memory (Flip-Flops or Latches).

Xilinx® recommends that you use the sensitivity-list based description style to describe sequential logic.

For more information, see Chapter 7, HDL Coding Techniques.

### Describing Sequential Logic

Describing sequential logic using a process with a sensitivity list includes:

- A sensitivity list containing:
  - The clock signal.
  - Any optional signal controlling the sequential element asynchronously (asynchronous set/reset).
- An **if** statement that models the clock event.

### Asynchronous Control Logic Modelization

- Modelization of any asynchronous control logic (asynchronous set/reset) is done before the clock event statement.
- Modelization of the synchronous logic (data, optional synchronous set/reset, optional clock enable) is done in the clock event **if** branch.

  This information is summarized in the following table.

**Asynchronous Control Logic Modelization Summary**

| Modelization of | Contains | Performed |
|---|---|---|
| Asynchronous control logic | Asynchronous set/reset | Before the clock event statement |
| Synchronous logic | • Data<br>• Optional synchronous set/reset<br>• Optional clock enable | In the clock event **if** branch. |

### Sequential Process With a Sensitivity List Syntax

```
process (<sensitivity list>)
begin
    <asynchronous part>
    <clock event>
    <synchronous part>
end;
```

### Clock Event Statements

- Describe the clock event statement as:
  - **rising edge** clock

    ```
    If clk'event and clk = '1' then
    ```

  - **falling edge** clock

    ```
    If clk'event and clk = '0' then
    ```

- For greater clarity, use the VHDL'93 IEEE standard **rising_edge** and **falling_edge** functions.
  - **rising edge** clock

    ```
    If rising_edge(clk) then
    ```

  - **falling edge** clock

    ```
    If falling_edge(clk) then
    ```

### Missing Signals

Signals may be missing from the sensitivity list.

- If one or more signals is missing from the sensitivity list:
  - The synthesis results can differ from the initial design specification.
  - XST issues a warning message.
  - XST adds the missing signals to the sensitivity list.
- To avoid problems during simulation:
  - Explicitly add all missing signals in the HDL source code.
  - Re-run synthesis.

## VHDL Sequential Processes Without a Sensitivity List

XST allows the description of a sequential process using a **wait** statement.

- The sequential process is described without a sensitivity list.
- The same sequential process cannot have both a sensitivity list and a **wait** statement.

Only one **wait** statement is allowed.

- The **wait** statement is the first statement.
- The condition in the **wait** statement describes the sequential logic clock.

### VHDL Sequential Process Using a Wait Statement Coding Example

```
process
begin
    wait until rising_edge(clk);
    q <= d;
end process;
```

### Describing a Clock Enable in the Wait Statement Coding Example

A **clock enable** can be described in the **wait** statement together with the **clock**.

```
process
begin
    wait until rising_edge(clk) and clken = '1';
    q <= d;
end process;
```

### Describing a Clock Enable After the Wait Statement Coding Example

You can describe the **clock enable** separately.

```
process
begin
    wait until rising_edge(clk);
    if clken = '1' then
        q <= d;
    end if;
end process;
```

## Describing Synchronous Control Logic

- Besides the **clock enable**, this coding method also allows you to describe synchronous control logic, such as a synchronous reset or set.
- You cannot describe a sequential element with asynchronous control logic using a process *without* a sensitivity list. Only a process *with* a sensitivity list allows such functionality.
- XST does not allow the description of a Latch based on a **wait** statement.
- For greater flexibility, Xilinx® recommends that you describe synchronous logic using a process with a sensitivity list.

## VHDL Initial Values and Operational Set/Reset

You can initialize Registers when you declare them.

The initialization value:

- Is a constant.
- May be generated from a function call. For example, loading initial values from an external data file.
- Cannot depend on earlier initial values.
- Can be a parameter value propagated to a Register.

### Initializing Registers VHDL Coding Example One

This coding example specifies a power-up value in which:

- The sequential element is initialized when the circuit goes live.
- The circuit global reset is applied.

```
signal arb_onebit   : std_logic := '0';
signal arb_priority : std_logic_vector(3 downto 0) := "1011";
```

### Initializing Sequential Elements Operationally

- To initialize sequential elements operationally, describe:
  - Set/reset values
  - Local control logic
- Assign a value to a Register when the Register reset line goes to the appropriate value.
- For an example, see the following coding example.
- See Flip-Flops and Registers for more information about the advantages and disadvantages of:
  - Operational set/reset
  - Asynchronous versus synchronous set/reset

### Initializing Registers VHDL Coding Example Two

```
process (clk, rst)
begin
    if rst='1' then
        arb_onebit <= '0';
    end if;
end process;
```

### Initializing Registers VHDL Coding Example Three

This coding example combines power-up initialization and operational reset.

```
--
-- Register initialization
--   Specifying initial contents at circuit powes-up
--   Specifying an operational set/reset
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/initial/initial_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity initial_1 is
    Port (
        clk, rst : in std_logic;
        din : in std_logic;
        dout : out std_logic);
end initial_1;

architecture behavioral of initial_1 is
    signal arb_onebit : std_logic := '1'; -- power-up to vcc
begin

    process (clk)
    begin
        if (rising_edge(clk)) then
            if rst='1' then -- local synchronous reset
                arb_onebit <= '0';
            else
                arb_onebit <= din;
            end if;
        end if;
    end process;

    dout <= arb_onebit;

end behavioral;
```

## VHDL Default Initial Values on Memory Elements

Every memory element must come up in a known state.

- Since every memory element must come up in a known state, XST does not apply IEEE standards for initial values in some cases.
- For example:
  - In the previous coding example, if **arb_onebit** is not initialized to **1** (one), XST assigns it a default of **0** (zero) as its initial state.
  - XST does not follow the IEEE standard, where **U** is the default for **std_logic**.

### Initialization

Initialization is the same for both Registers and RAM components.

- XST adheres whenever possible to the IEEE VHDL standard when initializing signal values.
- If no initial values are supplied in the VHDL code, XST uses the default values (where possible) shown in the XST column in the *VHDL Initial Values* table below.

## Unconnected Ports

Unconnected *output* ports default to the values shown in the XST column in the *VHDL Initial Values* table below.

- If the output port has an initial condition, XST ties the unconnected output port to the explicitly-defined initial condition.
- The IEEE VHDL specification does not allow unconnected *input* ports.
  - XST issues an error message for an unconnected input port.
  - Even the **open** keyword is not sufficient for an unconnected input port.

## VHDL Initial Values

| Type | IEEE | XST |
|---|---|---|
| bit | 0 | 0 |
| std_logic | U | 0 |
| bit_vector (3 downto 0) | 0 | 0 |
| std_logic_vector (3 downto 0) | 0 | 0 |
| integer (unconstrained) | integer'left | integer'left |
| integer range 7 downto 0 | integer'left = 7 | integer'left = 7 (coded as 111) |
| integer range 0 to 7 | integer'left = 0 | integer'left = 0 (coded as 000) |
| Boolean | FALSE | FALSE (coded as 0) |
| enum (S0,S1,S2,S3) | type'left = S0 | type'left = S0 (coded as 000) |

# VHDL Functions and Procedures

Use VHDL functions and procedures for blocks that are used multiple times in a design.

- Functions and procedures are declared in:
  - The declarative part of an entity
  - An architecture
  - A package
- A function or procedure consists of:
  - A declarative part
  - A body
- The declarative part specifies:
  - Input parameters
  - Output and inout parameters (procedures only)
  - Output and inout parameters (procedures only)
- These parameters can be unconstrained. They are not constrained to a given bound.
- The content is similar to the combinatorial process content.
- Resolution functions are not supported except the function defined in the IEEE **std_logic_1164** package.

### Function Declared Within a Package VHDL Coding Example

This coding example declares an ADD function within a package.

- The ADD function is a single-bit Adder.
- The ADD function is called four times to create a 4-bit Adder.

```
--
-- Declaration of a function in a package
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/functions_procedures/function_package_1.vhd
--
package PKG is
    function ADD (A,B, CIN : BIT )
    return BIT_VECTOR;
end PKG;

package body PKG is
    function ADD (A,B, CIN : BIT )
    return BIT_VECTOR is
        variable S, COUT : BIT;
        variable RESULT : BIT_VECTOR (1 downto 0);
    begin
        S := A xor B xor CIN;
        COUT := (A and B) or (A and CIN) or (B and CIN);
        RESULT := COUT & S;
        return RESULT;
    end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
    port (
        A,B : in BIT_VECTOR (3 downto 0);
        CIN : in BIT;
        S : out BIT_VECTOR (3 downto 0);
        COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
    signal S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
begin
    S0 <= ADD (A(0), B(0), CIN);
    S1 <= ADD (A(1), B(1), S0(1));
    S2 <= ADD (A(2), B(2), S1(1));
    S3 <= ADD (A(3), B(3), S2(1));
    S <= S3(0) & S2(0) & S1(0) & S0(0);
    COUT <= S3(1);
end ARCHI;
```

### Procedure Declared Within a Package VHDL Coding Example

Following is the same example using a procedure.

```
--
-- Declaration of a procedure in a package
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/functions_procedures/procedure_package_1.vhd
--
package PKG is
    procedure ADD (
        A, B, CIN : in BIT;
        C : out BIT_VECTOR (1 downto 0) );
end PKG;

package body PKG is
    procedure ADD (
        A, B, CIN : in BIT;
        C : out BIT_VECTOR (1 downto 0)
 ) is
        variable S, COUT : BIT;
    begin
        S := A xor B xor CIN;
        COUT := (A and B) or (A and CIN) or (B and CIN);
        C := COUT & S;
    end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
    port (
        A,B : in BIT_VECTOR (3 downto 0);
        CIN : in BIT;
        S : out BIT_VECTOR (3 downto 0);
        COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
    process (A,B,CIN)
        variable S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
    begin
        ADD (A(0), B(0), CIN, S0);
        ADD (A(1), B(1), S0(1), S1);
        ADD (A(2), B(2), S1(1), S2);
        ADD (A(3), B(3), S2(1), S3);
        S <= S3(0) & S2(0) & S1(0) & S0(0);
        COUT <= S3(1);
    end process;
end ARCHI;
```

### Recursive Functions VHDL Coding Example

XST supports recursive functions. This coding example models an **n!** function.

```
function my_func(x : integer) return integer is
begin
    if x = 1 then
        return x;
    else
        return (x*my_func(x-1));
    end if;
end function my_func;
```

# VHDL Assert Statements

VHDL assert statements.

- Help you debug your design.
- Enable you to detect undesirable conditions, such as bad values for:
  - Generics, constants, and generate conditions.
  - Parameters in called functions.

For any failed condition in an assert statement, depending on the severity level, XST either:

- Issues a warning message, or
- Rejects the design and issues an error message.

XST supports the assert statement only with static condition.

## Using an Assert Statement for Design Rule Checking

The coding example below contains a block (SINGLE_SRL) that describes a Shift Register.

- The size of the Shift Register depends on the SRL_WIDTH generic value.
- The assert statement ensures that the implementation of a single Shift Register does not exceed the size of a single Shift Register LUT (SRL).
- The maximum size of the Shift Register cannot exceed 17 bits, since:
  - The size of the SRL is 16 bit, and
  - XST implements the last stage of the Shift Register using a Flip-Flop in a slice.
- The SINGLE_SRL block is instantiated twice in the entity named TOP:
  - First instantiation

    SRL_WIDTH = 13
  - Second instantiation

    SRL_WIDTH = 18

### Using an Assert Statement for Design Rule Checking VHDL Coding Example

```
--
-- Use of an assert statement for design rule checking
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/asserts/asserts_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity SINGLE_SRL is
    generic (SRL_WIDTH : integer := 24);
    port (
        clk : in std_logic;
        inp : in std_logic;
        outp : out std_logic);
end SINGLE_SRL;

architecture beh of SINGLE_SRL is
    signal shift_reg : std_logic_vector (SRL_WIDTH-1 downto 0);
begin
    assert SRL_WIDTH <= 17
    report "The size of Shift Register exceeds the size of a single SRL"
    severity FAILURE;

    process (clk)
    begin
        if rising_edge(clk) then
            shift_reg <= shift_reg (SRL_WIDTH-2 downto 0) & inp;
        end if;
    end process;

    outp <= shift_reg(SRL_WIDTH-1);
end beh;

library ieee;
use ieee.std_logic_1164.all;

entity TOP is
    port (
        clk : in std_logic;
        inp1, inp2 : in std_logic;
        outp1, outp2 : out std_logic);
end TOP;

architecture beh of TOP is
    component SINGLE_SRL is
        generic (SRL_WIDTH : integer := 16);
        port(
            clk : in std_logic;
            inp : in std_logic;
            outp : out std_logic);
     end component;
begin
    inst1: SINGLE_SRL
        generic map (SRL_WIDTH => 13)
        port map(
            clk => clk,
            inp => inp1,
            outp => outp1 );
    inst2: SINGLE_SRL
        generic map (SRL_WIDTH => 18)
        port map(
            clk => clk,
            inp => inp2,
            outp => outp2 );
end beh;
```

## Using an Assert Statement for Design Rule Checking Error Message

```
HDL Elaboration *
=========================================================
Elaborating entity <TOP> (architecture <beh>) from library
<work>.  Elaborating entity <SINGLE_SRL> (architecture
<beh>) with generics from library <work>.  Elaborating
entity <SINGLE_SRL> (architecture <beh>) with generics
from library <work>.  ERROR:HDLCompiler:1242 -
"VHDL_Language_Support/asserts/asserts_1.vhd" Line 15:  "The size
of Shift Register exceeds the size of a single SRL":  exiting
elaboration "VHDL_Language_Support/asserts/asserts_1.vhd" Line
4.  netlist SINGLE_SRL(18)(beh) remains a blackbox,
due to errors in its contents
```

# VHDL Libraries and Packages

VHDL libraries and packages include:

- VHDL Libraries
- VHDL Predefined Packages

## VHDL Libraries

A VHDL library is a directory in which design units are compiled.

- Design units are entity or architectures and packages.
- Each VHDL and Verilog source file is compiled in a designated library.
- See Creating an HDL Synthesis Project for information on:
  - The syntax of the HDL synthesis project file.
  - How to specify the library in which an HDL source file is compiled.
- Invoke a design unit compiled in a library from any VHDL source file. Reference it through a library clause.

  ```
  library library_name;
  ```

- The `work` library:
  - Is the default library.
  - Does not require a library clause.
- To change the name of the default library, use:

  ```
  run -work_lib
  ```

- The physical location of the default library, and of any other user-defined library, is a subdirectory with the same name located under a directory defined by the Work Directory constraint.

## VHDL Predefined Packages

XST supports the following VHDL predefined packages:

- VHDL Predefined Standard Packages
- VHDL Predefined IEEE Packages

VHDL predefined packages:

- Are defined in the **std** and **ieee standard** libraries.
- Are pre-compiled.
- Need not be user-compiled.
- Can be directly included in the HDL source code.

## VHDL Predefined Standard Packages

VHDL predefined standard packages:

- Are included by default.
- Define basic VHDL types:
  - bit
  - bit_vector
  - integer
  - natural
  - real
  - boolean

## VHDL Predefined IEEE Packages

XST supports *some* VHDL predefined IEEE packages.

VHDL predefined IEEE packages:

- Are pre-compiled in the IEEE library.
- Define common data types, functions, and procedures.

XST supports the following IEEE packages:

- numeric_bit
  - Unsigned and signed vector types based on **bit**.
  - Overloaded arithmetic operators, conversion functions, and extended functions for these types.
- std_logic_1164
  - std_logic, std_ulogic, std_logic_vector, and std_ulogic_vector types.
  - Conversion functions based on these types.
- std_logic_arith (Synopsys)
  - Unsigned and signed vector types based on std_logic.
  - Overloaded arithmetic operators, conversion functions, and extended functions for these types.
- numeric_std
  - Unsigned and signed vector types based on std_logic.
  - Overloaded arithmetic operators, conversion functions, and extended functions for these types. Equivalent to std_logic_arith.
- std_logic_unsigned (Synopsys)

  Unsigned arithmetic operators for std_logic and std_logic_vector
- std_logic_signed (Synopsys)

  Signed arithmetic operators for std_logic and std_logic_vector
- std_logic_misc (Synopsys)

  Supplemental types, subtypes, constants, and functions for the std_logic_1164 package, such as and_reduce and or_reduce

## Defining Your Own VHDL Packages

You can define your own VHDL packages to specify:

- Types and subtypes
- Constants
- Functions and procedures
- Component declarations

Defining a VHDL package permits access to shared definitions and models from other parts of your project.

Defining a VHDL package requires a:

- Package declaration

  Declares each of the elements listed above.
- Package body

  Describes the functions and procedures declared in the package declaration.

### Package Declaration Syntax

```
package mypackage is

  type mytype is
    record
      first : integer;
      second : integer;
    end record;

  constant myzero : mytype := (first => 0, second => 0);

  function getfirst (x : mytype) return integer;

end mypackage;
```

### Package Body Syntax

```
package body mypackage is

  function getfirst (x : mytype) return integer is
  begin
    return x.first;
  end function;

end mypackage;
```

## Accessing VHDL Packages

To access a VHDL package:

1. Use a library clause to include the library in which the package has been compiled.

   **library** *library_name*;
2. Designate the package, or a specific definition contained in the package, with a use clause.

   **use** *library_name*.*package_name*.**all**;
3. Insert these lines immediately before the entity or architecture in which you use the package definitions.

Because the work library is the default library, you can omit the library clause if the designated package has been compiled into this library.

# VHDL File Type Support

| Function | Package |
|---|---|
| file (type text only) | standard |
| access (type line only) | standard |
| file_open (file, name, open_kind) | standard |
| file_close (file) | standard |
| endfile (file) | standard |
| text | std.textio |
| line | std.textio |
| width | std.textio |
| readline (text, line) | std.textio |
| readline (line, bit, boolean) | std.textio |
| read (line, bit) | std.textio |
| readline (line, bit_vector, boolean) | std.textio |
| read (line, bit_vector) | std.textio |
| read (line, boolean, boolean) | std.textio |
| read (line, boolean) | std.textio |
| read (line, character, boolean) | std.textio |
| read (line, character) | std.textio |
| read (line, string, boolean) | std.textio |
| read (line, string) | std.textio |
| write (file, line) | std.textio |
| write (line, bit, boolean) | std.textio |
| write (line, bit) | std.textio |
| write (line, bit_vector, boolean) | std.textio |
| write (line, bit_vector) | std.textio |
| write (line, boolean, boolean) | std.textio |
| write (line, boolean) | std.textio |
| write (line, character, boolean) | std.textio |
| write (line, character) | std.textio |
| write (line, integer, boolean) | std.textio |
| write (line, integer) | std.textio |
| write (line, string, boolean) | std.textio |
| write (line, string) | std.textio |
| read (line, std_ulogic, boolean) | ieee.std_logic_textio |
| read (line, std_ulogic) | ieee.std_logic_textio |
| read (line, std_ulogic_vector), boolean | ieee.std_logic_textio |
| read (line, std_ulogic_vector) | ieee.std_logic_textio |
| read (line, std_logic_vector, boolean) | ieee.std_logic_textio |

| Function | Package |
|---|---|
| read (line, std_logic_vector) | ieee.std_logic_textio |
| write (line, std_ulogic, boolean) | ieee.std_logic_textio |
| write (line, std_ulogic) | ieee.std_logic_textio |
| write (line, std_ulogic_vector, boolean) | ieee.std_logic_textio |
| write (line, std_ulogic_vector) | ieee.std_logic_textio |
| write (line, std_logic_vector, boolean) | ieee.std_logic_textio |
| write (line, std_logic_vector) | ieee.std_logic_textio |
| hread | ieee.std_logic_textio |

## VHDL File Read and File Write Capability

XST supports a limited VHDL File Read and File Write capability.

### File Read Capability

Use File Read capability to initialize memories from an external data file. For more information, see Specifying Initial Contents in an External Data File.

### File Write Capability

Use File Write capability for:

- Debugging
- Writing a specific constant or generic value to an external file

### Required Packages

The following packages are required.

- The **std.textio** package:
  - Is available in the **std** library.
  - Provides basic text-based File I/O capabilities.
  - Defines the following procedures for file I/O operations:
    - readline
    - read
    - writeline
    - write
- The **ieee.std_logic_textio** package:
  - Is available in the IEEE library.
  - Provides extended text I/O support for other data types.
  - Overloads the **read** and **write** procedures shown in VHDL File Type Support.

### Implicit and Explicit File Open and Close Operations

XST supports both implicit and explicit file open and close operations.

A file is implicitly opened when declared as follows:

```
file myfile :  text open write_mode is "myfilename.dat"; --
declaration and implicit open
```

Explicitly open and close an external file as follows:

```
file myfile :  text; -- declaration
variable file_status :  file_open_status;
…
file_open (file_status, myfile, "myfilename.dat", write_mode);
-- explicit open
…
file_close(myfile); -- explicit close
```

## Loading Memory Contents from an External File

See Specifying RAM Initial Contents in an External Data File.

## Writing to a File for Debugging

For update information, see "Coding Examples" in the Introduction.

### Writing to a File (Explicit Open/Close) VHDL Coding Example

File write capability is often used for debugging. In this coding example, write operations are performed to a file that has been explicitly opened.

```
--
-- Writing to a file
-- Explicit open/close with the VHDL'93 FILE_OPEN and FILE_CLOSE procedures
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/file_type_support/filewrite_explicitopen.vhd
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_TEXTIO.all;
use STD.TEXTIO.all;

entity filewrite_explicitopen is
    generic (data_width: integer:= 4);
    port (  clk : in  std_logic;
            di  : in  std_logic_vector (data_width - 1 downto 0);
            do  : out std_logic_vector (data_width - 1 downto 0));
end filewrite_explicitopen;

architecture behavioral of filewrite_explicitopen is
    file results : text;
    constant base_const: std_logic_vector(data_width - 1 downto 0):= conv_std_logic_vector(3,data_width);
    constant new_const:  std_logic_vector(data_width - 1 downto 0):= base_const + "0100";
begin

    process(clk)
        variable txtline : line;
        variable file_status : file_open_status;
    begin
        file_open (file_status, results, "explicit.dat", write_mode);
        write(txtline,string'("--------------------"));
        writeline(results, txtline);
        write(txtline,string'("Base Const: "));
        write(txtline, base_const);
        writeline(results, txtline);
        write(txtline,string'("New  Const: "));
        write(txtline,new_const);
        writeline(results, txtline);
        write(txtline,string'("--------------------"));
        writeline(results, txtline);
        file_close(results);
 if rising_edge(clk) then
     do <= di + new_const;
 end if;
    end process;

end behavioral;
```

### Writing to a File (Implicit Open/Close) VHDL Coding Example

You can also use an implicit file open.

```
--
-- Writing to a file. Implicit open/close
--
-- Download: http://www.xilinx.com/txpatches/pub/documentation/misc/xstug_examples.zip
-- File: VHDL_Language_Support/file_type_support/filewrite_implicitopen.vhd
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_TEXTIO.all;
use STD.TEXTIO.all;

entity filewrite_implicitopen is
    generic (data_width: integer:= 4);
    port (  clk : in  std_logic;
            di  : in  std_logic_vector (data_width - 1 downto 0);
            do  : out std_logic_vector (data_width - 1 downto 0));
end filewrite_implicitopen;

architecture behavioral of filewrite_implicitopen is
    file results : text open write_mode is "implicit.dat";
    constant base_const: std_logic_vector(data_width - 1 downto 0):= conv_std_logic_vector(3,data_width);
    constant new_const:  std_logic_vector(data_width - 1 downto 0):= base_const + "0100";
begin

    process(clk)
        variable txtline : LINE;
    begin
        write(txtline,string'("--------------------"));
        writeline(results, txtline);
        write(txtline,string'("Base Const: "));
        write(txtline,base_const);
        writeline(results, txtline);
        write(txtline,string'("New  Const: "));
        write(txtline,new_const);
        writeline(results, txtline);
        write(txtline,string'("--------------------"));
        writeline(results, txtline);
 if rising_edge(clk) then
     do <= di + new_const;
 end if;
    end process;

end behavioral;
```

## Debugging Using Write Operations

Follow these rules for debugging using **write** operations.

- During a **read** operation in **std_logic**:
  - The only allowed characters are **0**, **1**, and a blank space character.
  - Other values such as **X** and **Z** are not allowed.
  - XST rejects the design if the file includes characters other the allowed characters.
- Do not use identical names for files in different directories.
- Do not use conditional calls to read procedures.

```
if SEL = '1' then
    read (MY_LINE, A(3 downto 0));
else
    read (MY_LINE, A(1 downto 0));
end if;
```

# VHDL Constructs

VHDL constructs include:

- VHDL Design Entities and Configurations
- VHDL Expressions
- VHDL Statements

## VHDL Design Entities and Configurations

XST supports VHDL design entities and configurations except as noted below.

### VHDL Entity Headers

- Generics

  Supported
- Ports

  Supported, including unconstrained ports
- Entity Statement Part

  Unsupported

### VHDL Packages

- STANDARD
- Type TIME is not supported

### VHDL Physical Types

- TIME

  Ignored
- REAL

  Supported, but only in functions for constant calculations

### VHDL Modes

- Linkage
- Unsupported

### VHDL Declarations

- Type
- Supported for
  - Enumerated types
  - Types with positive range having constant bounds
  - Bit vector types
  - Multi-dimensional arrays

## VHDL Objects

- Constant Declaration

  Supported except for deferred constant
- Signal Declaration

  Supported except for register and bus type signals
- Attribute Declaration

  Supported for some attributes, otherwise skipped.

For more information, see Chapter 9, Design Constraints

## VHDL Specifications

- Supported for some predefined attributes only:
  - HIGHLOW
  - LEFT
  - RIGHT
  - RANGE
  - REVERSE_RANGE
  - LENGTH
  - POS
  - ASCENDING
  - EVENT
  - LAST_VALUE
- Configuration

  Supported only with the all clause for instances list. If no clause is added, XST looks for the entity or architecture compiled in the default library
- Disconnection

  Unsupported
- Object names can contain underscores in general (DATA_1), but XST does not allow signal names with leading underscores ( _DATA_1).

# VHDL Expressions

VHDL expressions include:

- VHDL Operators
- VHDL Operands

## VHDL Operators

| Operator | Status |
|---|---|
| Logical Operators: and, or, nand, nor, xor, xnor, not | Supported |
| Relational Operators: =, /=, <, <=, >, >= | Supported |
| & (concatenation) | Supported |
| Adding Operators: +, - | Supported |
| * | Supported |
| / | Supported if the right operand is a constant power of 2, or if both operands are constant |
| rem | Supported if the right operand is a constant power of 2 |
| mod | Supported if the right operand is a constant power of 2 |
| Shift Operators: sll, srl, sla, sra, rol, ror | Supported |
| abs | Supported |
| ** | Supported if the left operand is 2 |
| Sign: +, - | Supported |

## VHDL Operands

| Operand | Status |
|---|---|
| Abstract Literals | Only integer literals are supported |
| Physical Literals | Ignored |
| Enumeration Literals | Supported |
| String Literals | Supported |
| Bit String Literals | Supported |
| Record Aggregates | Supported |
| Array Aggregates | Supported |
| Function Call | Supported |
| Qualified Expressions | Supported for accepted predefined attributes |
| Types Conversions | Supported |
| Allocators | Unsupported |
| Static Expressions | Supported |

## VHDL Statements

VHDL statements include:

- VHDL Wait Statements
- VHDL Loop Statements
- VHDL Concurrent Statements

### VHDL Wait Statements

| Wait Statement | Status |
|---|---|
| • Wait on **sensitivity_list** until **boolean_expression**.<br>• See VHDL Combinatorial Circuits. | • Supported with one signal in the sensitivity list and in the Boolean expression.<br>• Multiple **wait** statements are not supported.<br>• **wait** statements for Latch descriptions are not supported. |
| • Wait for **time_expression**.<br>• See VHDL Combinatorial Circuits. | Unsupported |
| Assertion Statement | Supported for static conditions only. |
| Signal Assignment Statement | • Supported<br>• Delay is ignored. |
| Variable Assignment Statement | Supported |
| Procedure Call Statement | Supported |
| If Statement | Supported |
| Case Statement | Supported |

### VHDL Loop Statements

| Loop Statement | Status |
|---|---|
| for... loop... end loop | • Supported for constant bounds only.<br>• Disable statements are not supported. |
| while... loop... end loop | Supported |
| loop ... end loop | Supported for multiple **wait** statements only. |
| Next Statement | Supported |
| Exit Statement | Supported |
| Return Statement | Supported |
| Null Statement | Supported |

### VHDL Concurrent Statements

| Concurrent Statement | Status |
|---|---|
| Process Statement | Supported |
| Concurrent Procedure Call | Supported |
| Concurrent Assertion Statement | Ignored |
| Concurrent Signal Assignment Statement | • Supported<br>• No after clause, no transport or guarded options, no waveforms<br>• UNAFFECTED is supported. |
| Component Instantiation Statement | Supported |
| for-generate | Statement supported for constant bounds only |
| if-generate | Statement supported for static condition only |

## VHDL Reserved Words

| | | | |
|---|---|---|---|
| abs | access | after | alias |
| all | and | architecture | array |
| assert | attribute | begin | block |
| body | buffer | bus | case |
| component | configuration | constant | disconnect |
| downto | else | elsif | end |
| entity | exit | file | for |
| function | generate | generic | group |
| guarded | if | impure | in |
| inertial | inout | is | label |
| library | linkage | literal | loop |
| map | mod | nand | new |
| next | nor | not | null |
| of | on | open | or |
| others | out | package | port |
| postponed | procedure | process | pure |
| range | record | register | reject |
| rem | report | return | rol |
| ror | select | severity | signal |
| shared | sla | sll | sra |
| srl | subtype | then | to |
| transport | type | unaffected | units |
| until | use | variable | wait |
| when | while | with | xnor |
| xor | | | |