4 Data Types

VHDL is a strongly typed language. Every constant, signal, variable, function, and parameter is declared with a type, such as BOOLEAN or INTEGER, and can hold or return only a value of that type.

VHDL predefines abstract data types, such as BOOLEAN, which are part of most programming languages, and hardware-related types, such as BIT, found in most hardware languages. VHDL predefined types are declared in the STANDARD package, which is supplied with all VHDL implementations (see Example 4–14). Data types addresses information about

Unsupported Data Types

Synopsys Data Types

Subtypes

The advantage of strong typing is that VHDL tools can catch many common design errors, such as assigning an eight-bit value to a four-bit-wide signal, or incrementing an array index out of its range.

The following code shows the definition of a new type, BYTE, as an array of eight bits, and a variable declaration, ADDEND, that uses this type.

```
type BYTE is array(7 downto 0) of BIT;

variable ADDEND: BYTE;
```

The predefined VHDL data types are built from the basic VHDL data types. Some VHDL types are not supported for synthesis, such as REAL and FILE.

The examples in this chapter show type definitions and associated object declarations. Although each constant, signal, variable, function, and parameter is declared with a type, only variable and signal declarations are shown here in the examples. Constant, function, and parameter declarations are shown in Chapter 3.

VHDL also provides *subtypes*, which are defined as subsets of other types. Anywhere a type definition can appear, a subtype definition can also appear. The difference between a type and a subtype is that a subtype is a subset of a previously defined parent *(*or base*)* type or subtype. Overlapping subtypes of a given base type can be compared against and assigned to each other. All integer types, for example, are technically subtypes of the built-in integer base type (see Integer Types, later in this chapter). Subtypes are described in the last section of this chapter.

Enumeration Types

An *enumeration type* is defined by listing (enumerating) all possible values of that type.

The syntax of an enumeration type definition is

```
type type_name is ( enumeration_literal
                    {, enumeration_literal} );
```

type_name is an identifier, and each *enumeration_literal* is either an identifier (enum_6) or a character literal ('A').

An identifier is a sequence of letters, underscores, and numbers. An identifier must start with a letter and cannot be a VHDL reserved word, such as TYPE. All VHDL reserved words are listed in Appendix C.

A character literal is any value of type CHARACTER, in single quotes.

Example 4–1 shows two enumeration type definitions and corresponding variable and signal declarations.

Example 4–1 Enumeration Type Definitions

```
type COLOR is (BLUE, GREEN, YELLOW, RED);
type MY_LOGIC is ('0', '1', 'U', 'Z');
variable HUE: COLOR;
signal SIG: MY_LOGIC;
. . .
HUE := BLUE;
SIG <= 'Z';
```

Enumeration Overloading

You can overload an enumeration literal by including it in the definition of two or more enumeration types. When you use such an overloaded enumeration literal, VHDL Compiler is usually able to determine the literal's type. However, under certain circumstances determination may be impossible. In these cases, you must qualify the literal by explicitly stating its type (see ''Qualified Expressions" in Chapter 5). Example 4–2 shows how you can qualify an overloaded enumeration literal.

Example 4–2 Enumeration Literal Overloading

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
type PRIMARY_COLOR is (RED, YELLOW, BLUE);
...
A <= COLOR'(RED);
```

Enumeration Encoding

Enumeration types are ordered by enumeration *value*. By default, the first enumeration literal is assigned the value `0`, the next enumeration literal is assigned the value `1`, and so forth.

VHDL Compiler automatically encodes enumeration values into bit vectors that are based on each value's position. The length of the encoding bit vector is the minimum number of bits required to encode the number of enumerated values. For example, an enumeration type with five values would have a three-bit encoding vector.

Example 4–3 shows the default encoding of an enumeration type with five values.

Example 4–3 Automatic Enumeration Encoding

```
type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
```

The enumeration values are encoded as follows:

```
RED    ⇒ “000”
GREEN  ⇒ “001”
YELLOW ⇒ “010”
BLUE   ⇒ “011”
VIOLET ⇒ “100”
```

The result is RED < GREEN < YELLOW < BLUE < VIOLET.

You can override the automatic enumeration encodings and specify your own enumeration encodings with the ENUM_ENCODING attribute. This interpretation is specific to VHDL Compiler.

A VHDL attribute is defined by its name and type, and is then declared with a value for the attributed type, as shown in Example 4–4 below.

Note:

Several VHDL synthesis-related attributes are declared in the ATTRIBUTES package supplied with VHDL Compiler. This package is listed in Appendix B. The section on “Synthesis Attributes and Constraints” in Chapter 11 describes how to use these VHDL attributes.

The ENUM_ENCODING attribute must be a STRING containing a series of vectors, one for each enumeration literal in the associated type. The encoding vector is specified by '0's, '1's, 'D's, 'U's, and 'Z's separated by blank spaces. The meaning of these encoding vectors is described in the next section. The first vector in the attribute string specifies the encoding for the first enumeration literal, the second vector specifies the encoding for the second enumeration literal, and so on. The ENUM_ENCODING attribute must immediately follow the type declaration.

Example 4–4 illustrates how the default encodings from Example 4–3 can be changed with the ENUM_ENCODING attribute.

Example 4–4 Using the ENUM_ENCODING Attribute

```
attribute ENUM_ENCODING: STRING;
  -- Attribute definition

type COLOR is (RED, GREEN, YELLOW, BLUE, VIOLET);
attribute ENUM_ENCODING of
  COLOR: type is "010 000 011 100 001";
  -- Attribute declaration
```

The enumeration values are encoded as follows:

```
RED    = "010"
GREEN  = "000"
YELLOW = "011"
BLUE   = "100"
VIOLET = "001"
```

The result is GREEN < VIOLET < RED < YELLOW < BLUE.

WARNING

The interpretation of the ENUM_ENCODING attribute is specific to VHDL Compiler. Other VHDL tools, such as simulators, use the standard encoding (ordering).

Enumeration Encoding Values

The possible encoding values for the ENUM_ENCODING attribute are '0's, '1's, 'D's, 'U's, and 'Z's:

'0' Bit value '0'.

'1' Bit value '1'.

'D' Don't-care (can be either '0' or '1'). To use don't-care information, see ''Don't Care Inference" in Chapter 10.

'U' Unknown. If 'U' appears in the encoding vector for an enumeration, you cannot use that enumeration literal except as an operand to the "=" and "/=" operators. You can read an enumeration literal encoded with a 'U' from a variable or signal, but you cannot assign it.

For synthesis, the "=" operator returns FALSE and "/=" returns TRUE when either of the operands is an enumeration literal whose encoding contains 'U'.

'Z' High impedance. See ''Three-State Inference" in Chapter 8 for more information.

Integer Types

The maximum range of a VHDL integer type is $-(2^{31}-1)$ to $2^{31}-1$ (`-2_147_483_647 .. 2_147_483_647`). Integer types are defined as subranges of this anonymous built-in type. Multidigit numbers in VHDL can include underscores (_) to make them easier to read.

VHDL Compiler encodes an integer value as a bit vector whose length is the minimum necessary to hold the defined range. VHDL Compiler encodes integer ranges that include negative numbers as 2's-complement bit vectors.

The syntax of an integer type definition is

type *type_name* **is range** *integer_range* **;**

type_name is the name of the new integer type, and *integer_range* is a subrange of the anonymous integer type.

Example 4–5 shows some integer type definitions.

Example 4–5 Integer Type Definitions

```
type PERCENT is range -100 to 100;
  -- Represented as an 8-bit vector
  --   (1 sign bit, 7 value bits)

type INTEGER is range -2147483647 to 2147483647;
  -- Represented as a 32-bit vector
  --   This is the definition of the INTEGER type
```

Note:

You cannot directly access the bits of an `INTEGER` or explicitly state the bit width of the type. For these reasons, Synopsys provides overloaded functions for arithmatic. These functions are defined in the `std_logic_signed` and `std_logic_unsigned` packages, listed in Appendix B.

Array Types

An array is an object that is a collection of elements of the same type. VHDL supports *N*-dimensional arrays, but VHDL Compiler supports only one-dimensional arrays. Array elements can be of any type. An array has an index whose value selects each element. The index range determines how many elements are in the array and their ordering (low `to` high, or high `downto` low). An index can be of any integer type.

You can declare multidimensional arrays by building one-dimensional arrays where the element type is another one-dimensional array, as shown in Example 4–6.

Example 4–6 Declaration of Array of Arrays

```
type BYTE   is array (7 downto 0) of BIT;
type VECTOR is array (3 downto 0) of BYTE;
```

VHDL provides both constrained arrays and unconstrained arrays. The difference between these two comes from the index range in the array type definition.

Constrained Array

A constrained array's index range is explicitly defined; for example, the integer range `(1 to 4)`. When you declare a variable or signal of this type, it has the same index range.

The syntax of a constrained array type definition is

```
type array_type_name is
    array ( integer_range ) of type_name ;
```

array_type_name is the name of the new constrained array type, *integer_range* is a subrange of another integer type, and *type_name* is the type of each array element.

Example 4–7 shows a constrained array definition.

Example 4–7 Constrained Array Type Definition

```
type BYTE is array (7 downto 0) of BIT;
  -- A constrained array whose index range is
  -- (7, 6, 5, 4, 3, 2, 1, 0)
```

Unconstrained Array

You define an unconstrained array's index range as a *type*; for example, INTEGER. This definition implies that the index range can be any contiguous subset of that type's values. When you declare an array variable or signal of this type, you also define its actual index range. Different declarations can have different index ranges.

The syntax of an unconstrained array type definition is

```
type array_type_name is
    array (range_type_name range <>)
        of element_type_name ;
```

array_type_name is the name of the new unconstrained array type, *range_type_name* is the name of an integer type or subtype, and *element_type_name* is the type of each array element.

Example 4–8 shows an unconstrained array type definition and a declaration that uses it.

Example 4–8 Unconstrained Array Type Definition

```
type BIT_VECTOR is array(INTEGER range <>) of BIT;
  -- An unconstrained array definition
. . .
variable MY_VECTOR : BIT_VECTOR(5 downto -5);
```

The advantage of using unconstrained arrays is that a VHDL tool remembers the index range of each declaration. You can use *array attributes* to determine the range (bounds) of a signal or variable of an unconstrained array type. With this information, you can write routines that use variables or signals of an unconstrained array type, independently of any one array variable's or signal's bounds. The next section describes array attributes and how they are used.

Array Attributes

VHDL Compiler supports the following predefined VHDL attributes for use with arrays:

- left
- right
- high
- low
- length
- range
- reverse_range

These attributes all return a value corresponding to part of an array's range. Table 4–1 shows the values of the array attributes for Example 4–8's variable `MY_VECTOR`.

Table 4–1 Array Index Attributes

Attribute Expression	Value
MY_VECTOR'left	5
MY_VECTOR'right	–5
MY_VECTOR'high	5
MY_VECTOR'low	–5
MY_VECTOR'length	11
MY_VECTOR'range	(5 down to –5)
MY_VECTOR'reverse_range	(–5 to 5)

Example 4–9 shows the use of array attributes in a function that ORs together all elements of a given BIT_VECTOR (declared in Example 4–8) and returns that value.

Example 4–9 Use of Array Attributes

```
function OR_ALL (X: in BIT_VECTOR) return BIT is
  variable OR_BIT: BIT;
  begin
    OR_BIT := '0';
    for I in X'range loop
      OR_BIT := OR_BIT or X(I);
    end loop;

    return OR_BIT;
  end;
```

Note that this function works for any size BIT_VECTOR.

State Vector Attributes

When writing a state machine in VHDL, you can use a *state vector attribute* to provide information to Design Compiler. Put a STATE_VECTOR attribute on the architecture, where the attribute value is the name of the state signal. Use only one STATE_VECTOR attribute for an architecture; for example:

```
entity FSM1_ST is
  port(clk : in BIT; toggle : in BIT; op1 : out BIT);
end FSM1_ST;

architecture STATE_MACHINE_VIEW of FSM1_ST is
  -- Declare an enum type for the state
  type STATE_TYPE is (ZERO, ONE);
  signal STATE : STATE_TYPE;
  signal NEXT_STATE : STATE_TYPE;
  -- Set the state vector attribute
  attribute STATE_VECTOR : STRING;
  attribute STATE_VECTOR of STATE_MACHINE_VIEW :
            architecture is "STATE";
```

When you input the design into dc_shell, you can extract the state machine. See Example 4–10. Encodings may then be modified from those specified in the original VHDL description.

Example 4–10 Extracting a State Table

```
dc_shell> analyze -f vhdl state_vector.vhdl

dc_shell> elaborate STATE_MACHINE

dc_shell> replace_synthetic

dc_shell> extract

dc_shell> report -fsm
```

If your entity contains memory devices in addition to the state vectors, you must group the state vectors by using the dc_shell command group -fsm. Grouping the state vectors pulls the state machine into a separate level of hierarchy that you can then extract, as shown in Example 4–11.

Example 4–11 Extracting a State Table by Using the group –fsm Command

```
dc_shell> analyze -f vhdl state_vector.vhdl

dc_shell> elaborate FSM1_ST -arch STATE_MACHINE_VIEW

dc_shell> group -fsm -design_name STATE_MACHINE

dc_shell> current_design = STATE_MACHINE

dc_shell> replace_synthetic

dc_shell> extract

dc_shell> report -fsm
```

A STATE_VECTOR attribute inserted in your VHDL source does not work with flip-flops that are instantiated rather than inferred. To create a state machine with instantiated flip-flops, you can use an embedded dc_shell script like

```
-- synopsys dc_script_begin

-- set_fsm_state_vector { U1, U2 }
-- set_fsm_encoding { "S0=2#00", "S1=2#01", \
--                    "S2=2#10", "S3=2#11" }

-- synopsys dc_script_end

U1:FLIP_FLOP port map (NEXT_STATE[0], CLK, STATE[0]);
U2:FLIP_FLOP port map (NEXT_STATE[1], CLK, STATE[1]);
...
```

See Chapter 11 for information about embedded `dc_shell` scripts. See Chapter 10, “Finite State Machines,” in the *Design Compiler Family Reference Manual* for more information on specifying and encoding state machines.

Example 4–12 shows an example of linking a vector to a finite state machine.

Example 4–12 State Vector Attribute

```
library synopsys; use synopsys.attributes.all;

entity FSM1_ST is
  port(clk : in BIT; toggle : in BIT; op1 : out BIT);
end FSM1_ST;

architecture STATE_MACHINE_VIEW of FSM1_ST is
  -- Declare an enum type for the state
  type STATE_TYPE is (ZERO, ONE);
  signal STATE : STATE_TYPE;
  signal NEXT_STATE : STATE_TYPE;
  -- Set the state vector attribute
  attribute STATE_VECTOR of STATE_MACHINE_VIEW : architecture
            is ”STATE”;
begin
  -- This process sets the next state on the clock edge.
  SET_STATE: process(clk, NEXT_STATE)
  begin
    if (clk’event and clk = ’1’) then
      STATE <= NEXT_STATE;
    end if;
  end process SET_STATE;
```

```
  -- This process determines the next state and output
  -- values based on the current state and input values.
  SET_NEXT_STATE: process(STATE, toggle)
  begin
    -- SET defaults for NEXT_STATE and all outputs.
    op1 <= '0';
    NEXT_STATE <= ONE;
    case STATE is
      when ONE =>
        if (toggle = '0') then
          op1 <= '1';
          NEXT_STATE <= ONE;
        elsif (toggle = '1') then
          op1 <= '0';
          NEXT_STATE <= ZERO;
        end if;
      when ZERO =>
        if (toggle = '1') then
          op1 <= '1';
          NEXT_STATE <= ONE;
        elsif (toggle = '0') then
          op1 <= '0';
          NEXT_STATE <= ZERO;
        end if;
    end case;
  end process SET_NEXT_STATE;
end STATE_MACHINE_VIEW;
```

Record Types

A record is a set of named fields of various types, unlike an array, which is composed of identical anonymous entries. A record's field can be of any previously defined type, including another record type.

Note:

Constants in VHDL of type record are not supported for synthesis (initialization of records is not supported).

Example 4–13 shows a record type declaration (BYTE_AND_IX), three signals of that type, and some assignments.

Example 4–13 Record Type Declaration and Use

```
constant LEN:  INTEGER := 8;

subtype BYTE_VEC is BIT_VECTOR(LEN–1 downto 0);

type BYTE_AND_IX is
  record
    BYTE: BYTE_VEC;
    IX:   INTEGER range 0 to LEN;
  end record;

signal X, Y, Z: BYTE_AND_IX;

signal DATA: BYTE_VEC;
signal NUM:  INTEGER;
. . .

X.BYTE <= ”11110000”;
X.IX   <= 2;

DATA <= Y.BYTE;
NUM  <= Y.IX;

Z <= X;
```

As shown in Example 4–13, you can read values from or assign values to records in two ways:

- By individual field name

```
X.BYTE <= DATA;
X.IX   <= LEN;
```

- From another record object of the same type

```
Z <= X;
```

Note:

A record type object's individual fields are accessed by the object name, a period, and a field name: `X.BYTE` or `X.IX`. To access an element of the `BYTE` field's array, use array notation: `X.BYTE(2)`.

Predefined VHDL Data Types

IEEE VHDL describes two site-specific packages, each containing a standard set of types and operations: the `STANDARD` package and the `TEXTIO` package.

The `STANDARD` package of data types is included in all VHDL source files by an implicit `use` clause. The `TEXTIO` package defines types and operations for communication with a standard programming environment (terminal and file I/O). This package is not needed for synthesis, and therefore VHDL Compiler does not support it.

The VHDL Compiler implementation of the `STANDARD` package is listed in Example 4–14. This `STANDARD` package is a subset of the IEEE VHDL `STANDARD` package. Differences are described in ‘‘Unsupported Types” later in this chapter.

Example 4–14 VHDL Compiler STANDARD Package

```
package STANDARD is

  type BOOLEAN is (FALSE, TRUE);

  type BIT is ('0', '1');

  type CHARACTER is (
    NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
    DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
    CAN, EM,  SUB, ESC, FSP, GSP, RSP, USP,

    ' ', '!', '"', '#', '$', '%', '&', ''',
    '(', ')', '*', '+', ',', '-', '.', '/',
    '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', ':', ';', '<', '=', '>', '?',

    '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
    'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
    'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
    'X', 'Y', 'Z', '[', '\', ']', '^', '_',

    '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
    'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
    'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
    'x', 'y', 'z', '{', '|', '}', '~', DEL);

  type INTEGER is range -2147483647 to 2147483647;

  subtype NATURAL is INTEGER range 0 to 2147483647;

  subtype POSITIVE is INTEGER range 1 to 2147483647;

  type STRING is array (POSITIVE range <>)
       of CHARACTER;

  type BIT_VECTOR is array (NATURAL range <>)
       of BIT;

end STANDARD;
```

Data Type BOOLEAN

The BOOLEAN data type is actually an enumerated type with two values, FALSE and TRUE, where FALSE < TRUE. Logical functions such as equality (=) and comparison (<) functions return a BOOLEAN value.

Convert a BIT value to a BOOLEAN value as follows:

```
BOOLEAN_VAR := (BIT_VAR = '1');
```

Data Type *BIT*

The BIT data type represents a binary value as one of two characters, '0' or '1'. Logical operations such as and can take and return BIT values.

Convert a BOOLEAN value to a BIT value as follows:

```
if (BOOLEAN_VAR) then
  BIT_VAR := '1';
else
  BIT_VAR := '0';
end if;
```

Data Type *CHARACTER*

The CHARACTER data type enumerates the ASCII character set. Nonprinting characters are represented by a three-letter name, such as NUL for the null character. Printable characters are represented by themselves, in single quotation marks, as follows:

```
variable CHARACTER_VAR: CHARACTER;
. . .
CHARACTER_VAR := 'A';
```

Data Type *INTEGER*

The INTEGER data type represents positive and negative whole numbers.

Data Type *NATURAL*

The NATURAL data type is a subtype of INTEGER, used for representing natural (nonnegative) numbers.

Data Type *POSITIVE*

The POSITIVE data type is a subtype of INTEGER that is used for representing positive (nonzero, nonnegative) numbers.

Data Type *STRING*

The STRING data type is an unconstrained array of CHARACTERS. A STRING value is enclosed in double quotation marks, as follows:

```
variable STRING_VAR: STRING(1 to 7);
. . .
STRING_VAR := ”Rosebud”;
```

Data Type *BIT_VECTOR*

The BIT_VECTOR data type represents an array of BIT values.

Unsupported Data Types

Some data types are either not useful for synthesis, or are not supported. Unsupported types are parsed but ignored by VHDL Compiler. They are listed and described below.

Appendix C describes the level of VHDL Compiler support for each VHDL construct.

Physical Types

VHDL Compiler does not support physical types, such as units of measure (for example, `nS`). Since physical types are relevant to the simulation process, VHDL Compiler allows but ignores physical type declarations.

Floating Point Types

VHDL Compiler does not support floating point types, such as `REAL`. Floating point *literals*, such as `1.34`, are allowed in the definitions of VHDL Compiler-recognized attributes.

Access Types

VHDL Compiler does not support access (pointer) types because no equivalent hardware construct exists.

File Types

VHDL Compiler does not support file (disk file) types. A hardware file is a RAM or ROM.

SYNOPSYS Data Types

The `std_logic_arith` package provides arithmetic operations and numeric comparisons on array data types. The package also defines two major data types: `UNSIGNED` and `SIGNED`. These data types, unlike the predefined `INTEGER` type, provide access to the individual bits (wires) of a numeric value. For more information, see Appendix B.

Subtypes

A *subtype* is defined as a subset of a previously defined type or subtype. A subtype definition can appear anywhere a type definition is allowed.

Subtypes are a powerful way to use VHDL type checking to ensure valid assignments and meaningful data handling. Subtypes inherit all operators and subprograms defined for their parent (base) types.

Subtypes are also used for resolved signals to associate a resolution function with the signal type. (See "Signal Declarations" in Chapter 3 for more information.)

For example, note in Example 4–14 that NATURAL and POSITIVE are subtypes of INTEGER and they can be used with any INTEGER function. They can be added, multiplied, compared, and assigned to each other, so long as the values are within the appropriate subtype's range. All INTEGER types and subtypes are actually subtypes of an anonymous predefined numeric type.

Example 4–15 shows some valid and invalid assignments between NATURAL and POSITIVE values.

Example 4–15 Valid and Invalid Assignments between INTEGER Subtypes

```
variable NAT:  NATURAL;
variable POS:  POSITIVE;
. . .
POS := 5;
NAT := POS + 2;
. . .
NAT := 0;
POS := NAT;      -- Invalid; out of range
```

For example, the type BIT_VECTOR is defined as

```
type BIT_VECTOR is array(NATURAL range <>) of BIT;
```

If your design uses only 16–bit vectors, you can define a subtype MY_VECTOR as

```
subtype MY_VECTOR is BIT_VECTOR(0 to 15);
```

Example 4–16 shows that all functions and attributes that operate on BIT_VECTOR also operate on MY_VECTOR.

Example 4–16 Attributes and Functions Operating on a Subtype

```
type BIT_VECTOR is array(NATURAL range <>) of BIT;
subtype MY_VECTOR is BIT_VECTOR(0 to 15);
. . .
signal   VEC1, VEC2:  MY_VECTOR;
signal   S_BIT:  BIT;
variable UPPER_BOUND: INTEGER;
. . .
if (VEC1 = VEC2)
. . .
VEC1(4) <= S_BIT;
VEC2 <= ”0000111100001111”;
. . .
RIGHT_INDEX := VEC1’high;
```