Design with Hardware Description Languages (HDL)

(DHDL-94952)

Amin Mehranzadeh, Ph.D.

CE Department of
Islamic Azad University of Dezful

mehranzadeh@iaud.ac.ir
Mehran.students@gmail.com

Operators and Attributes

Operators:

- VHDL provides several kinds of pre-defined operators:
- Assignment operators
- Logical operators
- Arithmetic operators
- Relational operators
- Shift operators
- Concatenation operators

Assignment Operators:

- Are used to assign values to signals, variables, and constants. They are:

<= Used to assign a value to a SIGNAL.
:= Used to assign a value to a VARIABLE, CONSTANT, or GENERIC. Used also for establishing initial values.
=> Used to assign values to individual vector elements or with OTHERS.

Assignment Operators:

- Example: Consider the following signal and variable declarations:

```
SIGNAL x : STD_LOGIC;
VARIABLE y : STD_LOGIC_VECTOR(3 DOWNTO 0);  -- Leftmost bit is MSB
SIGNAL w: STD_LOGIC_VECTOR(0 TO 7);         -- Rightmost bit is
                                            -- MSB
```

Then the following assignments are legal:

```
x <= '1';       -- '1' is assigned to SIGNAL x using "<="
y := "0000";    -- "0000" is assigned to VARIABLE y using ":="
w <= "10000000";                -- LSB is '1', the others are '0'
w <= (0 =>'1', OTHERS =>'0');   -- LSB is '1', the others are '0'
```


Logical Operators:

- Used to perform logical operations. The data must be of type BIT, STD_LOGIC, or STD_ULOGIC (or, obviously, their respective extensions, BIT_VECTOR, STD_LOGIC_VECTOR, or STD_ULOGIC_VECTOR).
- The logical operators are:
  NOT, AND, OR, NAND, NOR , XOR, XNOR

Notes: The NOT operator has precedence over the others.

Examples:

```
y <= NOT a AND b;       -- (a'.b)
y <= NOT (a AND b);     -- (a.b)'
y <= a NAND b;          -- (a.b)'
```

Arithmetic Operators:

- Used to perform arithmetic operations. The data can be of type INTEGER, SIGNED, UNSIGNED, or REAL (recall that the last cannot be synthesized directly).
- Also, if the std_logic_signed or the std_logic_unsigned package of the ieee library is used, then STD LOGIC VECTOR can also be employed directly in addition and subtraction operations

Arithmetic Operators:

Arithmetic Operators:

no synthesis restrictions	+	Addition
	–	Subtraction
	*	Multiplication
only power of two is synthesizable	/	Division
only static values of base and exponent	**	Exponentiation
there generally is little or no synthesis support.	MOD	Modulus
	REM	Remainder
	ABS	Absolute value

Arithmetic Operators:

- There are no synthesis restrictions regarding addition and subtraction, and multiplication.
- For division, only power of two dividers (shift operation) are allowed.
- For exponentiation, only static values of base and exponent are accepted.
- Regard y mod x returns the remainder of y/x with the signal of x, while y rem x returns the remainder of y/x with the signal of y.
- Finally, abs returns the absolute value.

Comparison Operators:

- Used for making comparisons. The data can be of any of the types listed above. The relational (comparison) operators are:

=	Equal to
/=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Shift Operators:

- Used for shifting data. They were introduced in VHDL93. Their syntax is the following:

<left operand> <shift operation> <right operand>. The left operand must be of type BIT_VECTOR, while the right operand must be an INTEGER (+ or - in front of it is accepted). The shift operators are:

- sll Shift left logic – positions on the right are filled with '0's
- srl Shift right logic – positions on the left are filled with '0's

Data Attributes:

- The pre-defined, synthesizable data attributes are the following:

- d'LOW: Returns lower array index
- d'HIGH: Returns upper array index
- d'LEFT: Returns leftmost array index
- d'RIGHT: Returns rightmost array index
- d'LENGTH: Returns vector size
- d'RANGE: Returns vector range
- d'REVERSE_RANGE: Returns vector range in reverse order

Data Attributes:

- Example1:

Example: Consider the following signal:

```
SIGNAL d : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Then:

```
d'LOW=0, d'HIGH=7, d'LEFT=7, d'RIGHT=0, d'LENGTH=8,
d'RANGE=(7 downto 0), d'REVERSE_RANGE=(0 to 7).
```


Data Attributes:

- Example2:

Example: Consider the following signal:

```
SIGNAL x: STD_LOGIC_VECTOR (0 TO 7);
```

Then all four LOOP statements below are synthesizable and equivalent.

```
FOR i IN RANGE (0 TO 7) LOOP ...
FOR i IN x'RANGE LOOP ...
FOR i IN RANGE (x'LOW TO x'HIGH) LOOP ...
FOR i IN RANGE (0 TO x'LENGTH-1) LOOP ...
```

Data Attributes:

- If the signal is of enumerated type, then:

- d'VAL(pos): Returns value in the position specified
- d'POS(value): Returns position of the value specified
- d'LEFTOF(value): Returns value in the position to the left of the value specified
- d'VAL(row, column): Returns value in the position specified; etc.

Note: There is little or no synthesis support for enumerated data type attributes.

Data Attributes:

- Signal Attributes:

Let us consider a signal **s**. Then:

- s'EVENT: Returns true when an event occurs on s
- s'STABLE: Returns true if no event has occurred on s
- s'ACTIVE: Returns true if s = '1'
- s'QUIET ⟨time⟩: Returns true if no event has occurred during the time specified
- s'LAST_EVENT: Returns the time elapsed since last event
- s'LAST_ACTIVE: Returns the time elapsed since last s = '1'
- s'LAST_VALUE: Returns the value of s before the last event; etc.

Data Attributes:

- Signal Attributes:

Let us consider a signal **s**. Then:

- s'EVENT: Returns true when an event occurs on s
- s'STABLE: Returns true if no event has occurred on s
- s'ACTIVE: Returns true if s = '1'
- s'QUIET ⟨time⟩: Returns true if no event has occurred during the time specified
- s'LAST_EVENT: Returns the time elapsed since last event
- s'LAST_ACTIVE: Returns the time elapsed since last s = '1'
- s'LAST_VALUE: Returns the value of s before the last event; etc.

Note: Though most signal attributes are for simulation purposes only, the first two in the list above are synthesizable, s'EVENT being the most often used of them all.

Data Attributes:

- Example: All four assignments shown below are synthesizable and equivalent. They return TRUE when an event (a change) occurs on clk, AND if such event is upward (in other words, when a rising edge occurs on clk).

```
IF (clk'EVENT AND clk='1')...          -- EVENT attribute used
                                       -- with IF
IF (NOT clk'STABLE AND clk='1')...     -- STABLE attribute used
                                       -- with IF
WAIT UNTIL (clk'EVENT AND clk='1');    -- EVENT attribute used
                                       -- with WAIT
IF RISING_EDGE(clk)...                 -- call to a function
```

User-Defined Attributes:

- VHDL also allows the construction of user defined attributes. To employ a user-defined attribute, it must be declared and specified. The syntax is the following:

Attribute declaration:

```
ATTRIBUTE attribute_name: attribute_type;
```

Attribute specification:

```
ATTRIBUTE attribute_name OF target_name: class IS value;
```


User-Defined Attributes:

- where:

- attribute_type:
  any data type (BIT, INTEGER, STD_LOGIC_VECTOR, etc.)
- class: TYPE, SIGNAL, FUNCTION, etc.
- value: '0', 27, ''00 11 10 01'', etc.

Note: A user-defined attribute can be declared anywhere, except in a PACKAGE BODY. When not recognized by the synthesis tool, it is simply ignored, or a warning is issued.

User-Defined Attributes:

- where:
- attribute_type:
  any data type (BIT, INTEGER, STD_LOGIC_VECTOR, etc.)
- class: TYPE, SIGNAL, FUNCTION, etc.
- value: '0', 27, ''00 11 10 01'', etc.

Example:

```
ATTRIBUTE number_of_inputs: INTEGER;              -- declaration
ATTRIBUTE number_of_inputs OF nand3: SIGNAL IS 3; -- specification
   ...
inputs <= nand3'number_of_pins;    -- attribute call, returns 3
```


User-Defined Attributes:

- Example: Enumerated encoding.

A popular user-defined attribute, which is provided by synthesis tool vendors, is the enum_encoding attribute. By default, enumerated data types are encoded sequentially. Thus, if we consider the enumerated data type color shown below:

```
TYPE color IS (red, green, blue, white);
```

its states will be encoded as red = ''00'', green = ''01'', blue = ''10'', and white = ''11''. Enum_encoding allows the default encoding (sequential) to be changed. Thus the following encoding scheme could be employed, for example:

```
ATTRIBUTE enum_encoding OF color: TYPE IS "11 00 10 01";
```

Operator Overloading:

- We can define our own operators, using the same name as the pre-defined ones.
- For example, we could use ‘‘+’’ to indicate a new kind of addition, this time between values of type BIT_VECTOR. This technique is called operator overloading.

```
-------------------------------------
FUNCTION "+" (a: INTEGER, b: BIT) RETURN INTEGER IS
BEGIN
   IF (b='1') THEN RETURN a+1;
   ELSE RETURN a;
   END IF;
END "+";
-------------------------------------
```

A call to the function above could thus be the following:

```
------------------------------
SIGNAL inp1, outp: INTEGER RANGE 0 TO 15;
SIGNAL inp2: BIT;
   (...)
outp <= 3 + inp1 + inp2;
   (...)
------------------------------
```


GENERIC:

- As the name suggests, GENERIC is a way of specifying a generic parameter (that is, a static parameter that can be easily modified and adapted to different applications).
- A GENERIC statement, when employed, must be declared in the ENTITY. The specified parameter will then be truly global (that is, visible to the whole design, including the ENTITY itself). Its syntax is shown below.

```
GENERIC (parameter_name : parameter_type := parameter_value);
```

GENERIC:

- Example: whenever n is found in the ENTITY itself or in the ARCHITECTURE (one or more) that follows, its value will be assumed to be 8.

```
ENTITY my_entity IS
   GENERIC (n : INTEGER := 8);
   PORT (...);
END my_entity;
ARCHITECTURE my_architecture OF my_entity IS
   ...
END my_architecture:
```

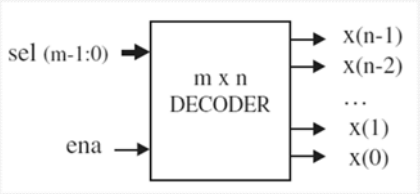
More than one GENERIC parameter can be specified in an ENTITY. For example:

```
GENERIC (n: INTEGER := 8; vector: BIT_VECTOR := "00001111");
```


Example 1- Generic Decoder:

(generic m-by-n decoder)

- The circuit has two inputs, sel (m bits) and ena (single bit), and one output, x (n bits). We assume that n is a power of two, so m = log2n. If ena = '0', then all bits of x should be high; otherwise, the output bit selected by sel should be low.



ena	sel	x
0	00	1111
1	00	1110
	01	1101
	10	1011
	11	0111

Example 1- Generic Decoder:

(generic m-by-n decoder)

```
1  ---------------------------------------------
2  LIBRARY ieee;
3  USE ieee.std_logic_1164.all;
4  ---------------------------------------------
5  ENTITY decoder IS
6     PORT ( ena : IN STD_LOGIC;
7            sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
8            x : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
9  END decoder;
10 ---------------------------------------------
11 ARCHITECTURE generic_decoder OF decoder IS
12 BEGIN
13    PROCESS (ena, sel)
14       VARIABLE temp1 : STD_LOGIC_VECTOR (x'HIGH DOWNTO 0);
15       VARIABLE temp2 : INTEGER RANGE 0 TO x'HIGH;
16    BEGIN
```

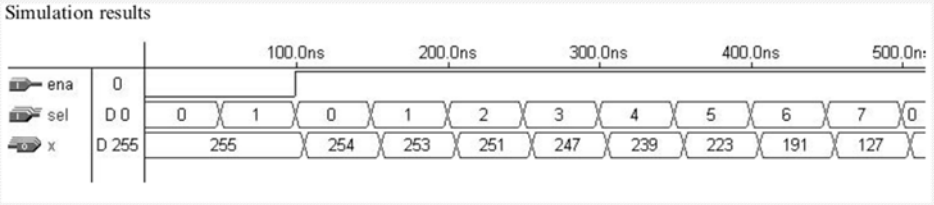

Example 1- Generic Decoder:

(generic m-by-n decoder)

```
17       temp1 := (OTHERS => '1');
18       temp2 := 0;
19       IF (ena='1') THEN
20          FOR i IN sel'RANGE LOOP  -- sel range is 2 downto 0
21             IF (sel(i)='1') THEN  -- Bin-to-Integer conversion
22                temp2:=2*temp2+1;
23             ELSE
24                temp2 := 2*temp2;
25             END IF;
26          END LOOP;
27          temp1(temp2):='0';
28       END IF;
29       x <= temp1;
30    END PROCESS;
31 END generic_decoder;
32 ---------------------------------------------
```

Example 1- Generic Decoder:

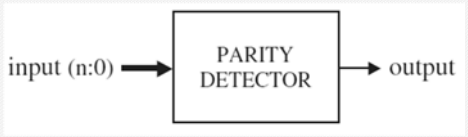
(generic m-by-n decoder)

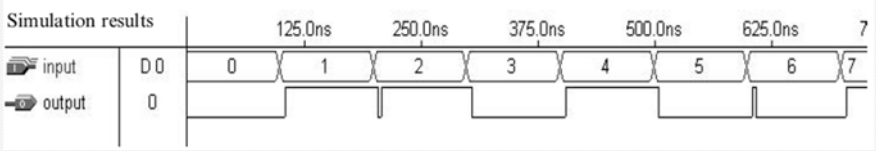


As can be seen, all outputs are high, that is, x ¼ "11111111" (decimal 255), when ena = '0'. After ena has been asserted, only one output bit (that selected by sel) is turned low. For example, when sel = "000" (decimal 0), x = "11111110" (decimal 254); when sel = "001" (decimal 1), x = "11111101" (decimal 253); when sel = "010" (decimal 2), x = "11111011" (decimal 251); and so on.

Example 2- Generic Parity Detector:

- The circuit must provide output = '0' when the number of '1's in the input vector is even, or output = '1' otherwise.



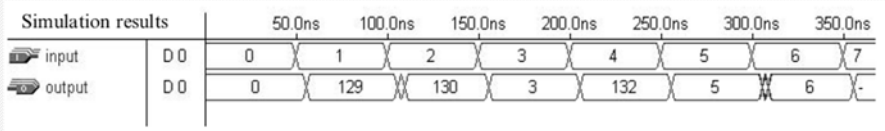


Example 2:
(Generic Parity Detector)

```
1  ----------------------------------------
2  ENTITY parity_det IS
3     GENERIC (n : INTEGER := 7);
4     PORT ( input: IN BIT_VECTOR (n DOWNTO 0);
5            output: OUT BIT);
6  END parity_det;
7  ----------------------------------------
8  ARCHITECTURE parity OF parity_det IS
9  BEGIN
10    PROCESS (input)
11       VARIABLE temp: BIT;
12    BEGIN
13       temp := '0';
14       FOR i IN input'RANGE LOOP
15          temp := temp XOR input(i);
16       END LOOP;
17       output <= temp;
18    END PROCESS;
19 END parity;
20 ----------------------------------------
```


Example3-Generic Parity Generator:

- The circuit must add one bit to the input vector (on its left). Such bit must be a '0' if the number of '1's in the input vector is even, or a '1' if it is odd, such that the resulting vector will always contain an even number of '1's (even parity).



when input ="0000000" (decimal 0, with seven bits), output = "00000000" (decimal 0, with eight bits); when input = "0000001" (decimal 1, with seven bits), output = "10000001" (decimal 129, with eight bits); and so on.

Example 3:
(Generic Parity Generator)


```
1  ------------------------------------------------
2  ENTITY parity_gen IS
3     GENERIC (n : INTEGER := 7);
4     PORT ( input: IN BIT_VECTOR (n-1 DOWNTO 0);
5            output: OUT BIT_VECTOR (n DOWNTO 0));
6  END parity_gen;
7  ------------------------------------------------
8  ARCHITECTURE parity OF parity_gen IS
9  BEGIN
10    PROCESS (input)
11       VARIABLE temp1: BIT;
12       VARIABLE temp2: BIT_VECTOR (output'RANGE);
13    BEGIN
```


Example 3:
(Generic Parity Generator)

```
13    BEGIN
14       temp1 := '0';
15       FOR i IN input'RANGE LOOP
16          temp1 := temp1 XOR input(i);
17          temp2(i) := input(i);
18       END LOOP;
19       temp2(output'HIGH) := temp1;
20       output <= temp2;
21    END PROCESS;
22 END parity;
23 ------------------------------------------------
```

Summary:

- A summary of VHDL operators and attributes is presented in tables 4.1 and 4.2, respectively. The constructs that are not synthesizable (or have little synthesis support) are marked with the "◆" symbol.

Table 4.1
Operators.

Operator type	Operators	Data types
Assignment	<=, :=, =>	Any
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	+, –, *, /, ** (mod, rem, abs)◆	INTEGER, SIGNED, UNSIGNED
Comparison	=, /=, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (,,,)	Same as for logical operators, plus SIGNED and UNSIGNED

Summary:

Table 4.2
Attributes.

Application	Attributes	Return value
For regular DATA	d'LOW	Lower array index
	d'HIGH	Upper array index
	d'LEFT	Leftmost array index
	d'RIGHT	Rightmost array index
	d'LENGTH	Vector size
	d'RANGE	Vector range
	d'REVERSE_RANGE	Reverse vector range
For enumerated DATA	d'VAL(pos)◆	Value in the position specified
	d'POS(value)◆	Position of the value specified
	d'LEFTOF(value)◆	Value in the position to the left of the value specified
	d'VAL(row, column)◆	Value in the position specified
For a SIGNAL	s'EVENT	True when an event occurs on s
	s'STABLE	True if no event has occurred on s
	s'ACTIVE◆	True if s is high