Operators and Attributes

Predefined Operators

VHDL provides several kinds of predefined operators:

- Assignment operators
- Logical operators
- Arithmetic operators
- Comparison (relational) operators
- Shift operators
- Concatenation operator
- Matching comparison operators

Assignment Operators

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

Operator "<=" Used to assign a value to a SIGNAL.

Operator ":=" Used to assign a value to a VARIABLE, CONSTANT, or GENERIC. Usedalso for establishing initial values.

Operator "=>" Used to assign values to individual vector elements or with OTHERS.

Example
Three object declarations ?x; y; z? are shown below, followed by several assignments. Comments follow each assignment.

```
CONSTANT x: STD_LOGIC_VECTOR(7 DOWNTO 0) := "00010001";
SIGNAL y: STD_LOGIC_VECTOR(1 TO 4);
VARIABLE z: BIT_VECTOR(3 DOWNTO 0);

y(4) <= '1'; --'1' assigned to a signal using "<="

y <= "0000"; --"0000" assigned to a signal with "<="

y <= (OTHERS=>'0') --'0' assigned to all elements of y
```

```
y <= x(3 DOWNTO 0); --part of x assigned to y

z := "1000"; --"1000" assigned to a variable with ":="

z := (0=>'1', OTHERS=>'0'); --z="0001"
```

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. The XNOR operator was introduced in VHDL93.

Examples:

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

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

y <= a NAND b; -- (a.b)'
```

Arithmetic Operators

The arithmetic operators are:

Addition (+)
Subtraction (-)
Multiplication (*)
Division (/)
Exponentiation (**)

Absolute value (ABS)
Remainder (REM)
Modulo (MOD)

x/y Returns 0 when $|x| < |y|$,

± 1 when $|y| \leq |x| < 2|y|$,

± 2 when $2|y| \leq |x| < 3$, etc

with the sign obviously negative when the signs of x and y are different

Examples 3/5=0, -3/5=0, 9/5=1, -9/5=-1, 10/5=2, -10/5=-2, 14/5=2, -14/5=-1

ABS x: Returns the absolute value of x.

Examples ABS 5 = 5, ABS -3=3.

x REM y: Returns the remainder of x/y, with the sign of x. Its equation is x REM y = x – (x/y) y, where both operands are integers.

Examples 6 REM 3 = 0, 7 REM 3 = 1, 7 REM 3 = 1, -7 REM 3 = -1, -7 REM -3 = -1.

x MOD y: Returns the remainder of x/y, with the sign of y. Its equation is x MOD y = x REM y + ay, where a = 1 when the signs of x and y are different or a = 0 otherwise.
Both operands are integers.

Examples 7 MOD 3 = 1, 7 MOD -3 = -2, -7 MOD 3 = 2, -7 MOD -3 = -1.

Comparison Operators

Also called relational operators, the comparison operators are:

Equal to (=
Not equal to (/=)
Less than (<)
Greater than (>)

Less than or equal to (<=)
Greater than or equal to (>=)

Shift Operators

Introduced in VHDL93, shift operators are used for shifting data vectors. They are:

Shift left logic (SLL): Positions on the right are filled with '0's.
Shift right logic (SRL): Positions on the left are filled with '0's.
Shift left arithmetic (SLA): Rightmost bit is replicated on the right.
Shift right arithmetic (SRA): Leftmost bit is replicated on the left.
Rotate left (ROL): Circular shift to the left.
Rotate right (ROR): Circular shift to the right.

Examples

Say that x is a BIT_VECTOR signal with value x = "01001". Then the values produced by the assignments below are those indicated in the comments (equivalent expressions, using the concatenation operator, are shown between parentheses).

```
---------------------------------------------------------------------
y <= x SLL 2; --y<="00100" (y <= x(2 DOWNTO 0) & "00";)
y <= x SLA 2; --y<="00111" (y <= x(2 DOWNTO 0) & x(0) & x(0);)
y <= x SRL 3; --y<="00001" (y <= "000" & x(4 DOWNTO 3);)
y <= x SRA 3; --y<="00001" (y <= x(4) & x(4) & x(4) & x(4 DOWNTO 3);)
y <= x ROL 2; --y<="00101" (y <= x(2 DOWNTO 0) & x(4 DOWNTO 3);)
y <= x SRL -2; --same as "x SLL 2"
---------------------------------------------------------------------
```

Concatenation Operator

Used for grouping objects and values (useful also for shifting data, as shown in the example above), the concatenation operator’s representation is &.

The synthesizable predefined data types for which the concatenation operator is intended are BIT_VECTOR, BOOLEAN_VECTOR (VHDL 2008), INTEGER_VECTOR (VHDL 2008), STD_(U)LOGIC_VECTOR, (UN)SIGNED, and STRING.

Example
Four VHDL objects (v, x, y, z) are declared below, then several assignments are made utilizing the concatenation operator (&). The use of parentheses is optional.

```
---------------------------------------------------------------------
CONSTANT v: BIT :='1';
CONSTANT x: STD_LOGIC :='Z';
SIGNAL y: BIT_VECTOR(1 TO 4);
SIGNAL z: STD_LOGIC_VECTOR(7 DOWNTO 0);
y <= (v & "000"); --result: "1000"
y <= v & "000"; --same as above (parentheses are optional)
z <= (x & x & "11111" & x); --result: "ZZ11111Z"
z <= ('0' & "011111" & x); --result: "0011111Z"
---------------------------------------------------------------------
```

Example

Consider the same constants and signals above. Below is a series of individualbit assignments using the keyword OTHERS and comma instead of the regular concatenation operator. Observe the nominal and positional mapping options. Here, parentheses are required.

```
---------------------------------------------------------------------
y <= (OTHERS=>'0'); --result: "0000"
y <= (4=>'1', OTHERS=>'0'); --result: "0001" (nominal mapping)
y <= ('1', OTHERS=>'0'); --result: "1000" (positional mapping)
y <= (4=>'1', 2=>v, OTHERS=>'0'); --result: "0101" (nominal mapping)
z <= (OTHERS=>'Z'); --result: "ZZZZZZZZ"
z <= (4=>'1', OTHERS=>'0'); --result: "00010000" (nominal mapping)
z <= (4=>x, OTHERS=>'0'); --result: "000Z0000" (nominal mapping)
z <= ('1', OTHERS=>'0'); --result: "10000000" (posit. mapping)
---------------------------------------------------------------------
```

Matching Comparison Operators

Matching equality operator (?=)
Matching inequality operator (?/=)
Matching less than operator (?<)
Matching greater than operator (?>)
Matching less than or equal to operator (?<=)
Matching greater than or equal to operator (?>=)

The purpose of this operator is to allow the comparison of logic values instead of enumerated symbols in STD_ULOGIC based data.

For example, "IF 'H' = '1' . . ." returns FALSE because these symbols are different, while "IF 'H' ?= '1' . . ." returns '1' because both 'H' and '1' are interpreted as logic value '1'.

Other Operators

Other operators introduced in VHDL 2008 are:

MINIMUM and MAXIMUM operators: Return the smallest or largest value in the given set. For example, "MAXIMUM(0, 55, 23)" returns 55. These operators were defined for all VHDL types.

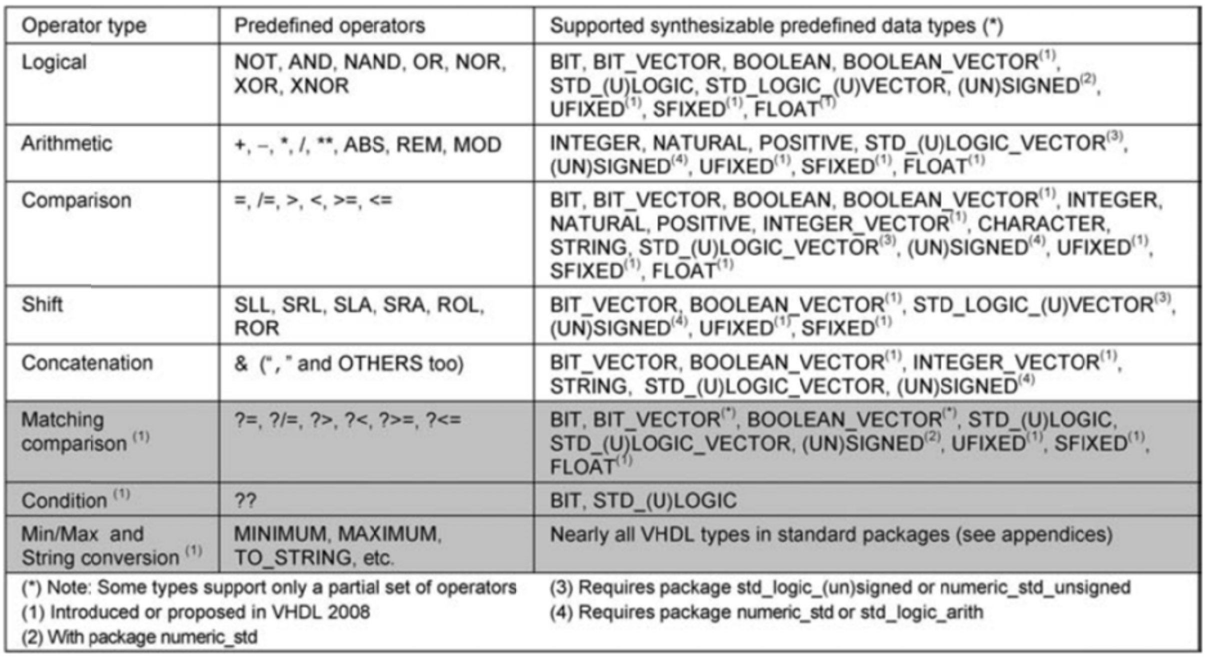
Condition operator ("??"): Converts a BIT or STD_(U)LOGIC value into a BOOLEAN value. For example, "?? a AND b" returns TRUE when a AND b = '1' or FALSE otherwise.

TO_STRING: Converts a value of type BIT, BIT_VECTOR, STD_LOGIC_VECTOR, and so on into STRING. For the types BIT_VECTOR and STD_LOGIC_VECTOR, there are also the options TO_OSTRING and TO_HSTRING, which produce an octal or hexadecimal string, respectively.

Examples

TO_STRING(58) = "58"
TO_STRING(B"1110000) = "11110000”
TO_HSTRING(B"11110000) = "F0"

Operators Summary



Operator type	Predefined operators	Supported synthesizable predefined data types (*)
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, BOOLEAN, BOOLEAN_VECTOR(1), STD_(U)LOGIC, STD_LOGIC_(U)VECTOR, (UN)SIGNED(2), UFIXED(1), SFIXED(1), FLOAT(1)
Arithmetic	+, –, *, /, **, ABS, REM, MOD	INTEGER, NATURAL, POSITIVE, STD_(U)LOGIC_VECTOR(3), (UN)SIGNED(4), UFIXED(1), SFIXED(1), FLOAT(1)
Comparison	=, /=, >, <, >=, <=	BIT, BIT_VECTOR, BOOLEAN, BOOLEAN_VECTOR(1), INTEGER, NATURAL, POSITIVE, INTEGER_VECTOR(1), CHARACTER, STRING, STD_(U)LOGIC_VECTOR(3), (UN)SIGNED(4), UFIXED(1), SFIXED(1), FLOAT(1)
Shift	SLL, SRL, SLA, SRA, ROL, ROR	BIT_VECTOR, BOOLEAN_VECTOR(1), STD_LOGIC_(U)VECTOR(3), (UN)SIGNED(4), UFIXED(1), SFIXED(1)
Concatenation	& (“ , ” and OTHERS too)	BIT_VECTOR, BOOLEAN_VECTOR(1), INTEGER_VECTOR(1), STRING, STD_(U)LOGIC_VECTOR, (UN)SIGNED(4)
Matching comparison (1)	?=, ?/=, ?>, ?<, ?>=, ?<=	BIT, BIT_VECTOR(*), BOOLEAN_VECTOR(*), STD_(U)LOGIC, STD_(U)LOGIC_VECTOR, (UN)SIGNED(2), UFIXED(1), SFIXED(1), FLOAT(1)
Condition (1)	??	BIT, STD_(U)LOGIC
Min/Max and String conversion (1)	MINIMUM, MAXIMUM, TO_STRING, etc.	Nearly all VHDL types in standard packages (see appendices)

(*) Note: Some types support only a partial set of operators
(1) Introduced or proposed in VHDL 2008
(2) With package numeric_std
(3) Requires package std_logic_(un)signed or numeric_std_unsigned
(4) Requires package numeric_std or std_logic_arith

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

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).

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 ...

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.

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 3time4: 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.
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.

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

We saw above attributes of the type HIGH, RANGE, EVENT, etc. However, 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;

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

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 = "0"’, 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";
```

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).

The purpose is to confer the code more flexibility and reusability.
A GENERIC statement, when employed, must be declared in the ENTITY.

Its syntax is shown below.

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

Example:

The GENERIC statement below specifies a parameter called n, of type INTEGER, whose default value is 8.

Therefore, 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:
```

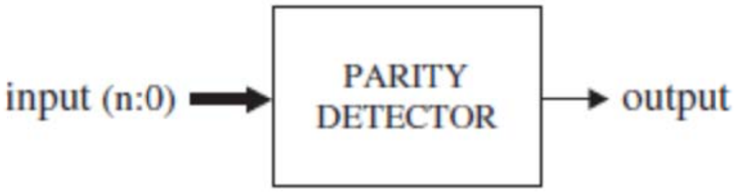
More than one GENERIC parameter can be specified in an ENTITY.

For example:

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

Example: Generic Parity Detector

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

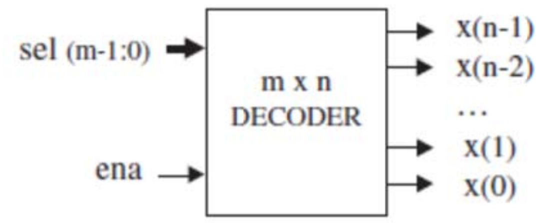


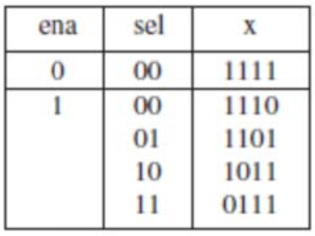
```
---------------------------------------------
ENTITY parity_det IS
    GENERIC (n : INTEGER := 7);
    PORT ( input: IN BIT_VECTOR (n DOWNTO 0);
    output: OUT BIT);
END parity_det;



---------------------------------------------
ARCHITECTURE parity OF parity_det IS
    BEGIN
        PROCESS (input)
        VARIABLE temp: BIT;
    BEGIN
        temp := '0';
        FOR i IN input'RANGE LOOP
t           emp := temp XOR input(i);
        END LOOP;
        output <= temp;
    END PROCESS;
END parity;
---------------------------------------------
```

Solution-2

```
---------------------------------------------
ENTITY parity_gen IS
    GENERIC (n : INTEGER := 7);
    PORT ( input: IN BIT_VECTOR (n-1 DOWNTO 0);
        output: OUT BIT_VECTOR (n DOWNTO 0));
END parity_gen;
---------------------------------------------
ARCHITECTURE parity OF parity_gen IS
    BEGIN
        PROCESS (input)
            VARIABLE temp1: BIT;
            VARIABLE temp2: BIT_VECTOR (output'RANGE);
        BEGIN
            temp1 := '0';
            FOR i IN input'RANGE LOOP
                temp1 := temp1 XOR input(i);
                temp2(i) := input(i);
            END LOOP;
            temp2(output'HIGH) := temp1;
            output <= temp2;
        END PROCESS;
END parity;
---------------------------------------------
```

Example: Generic Decoder

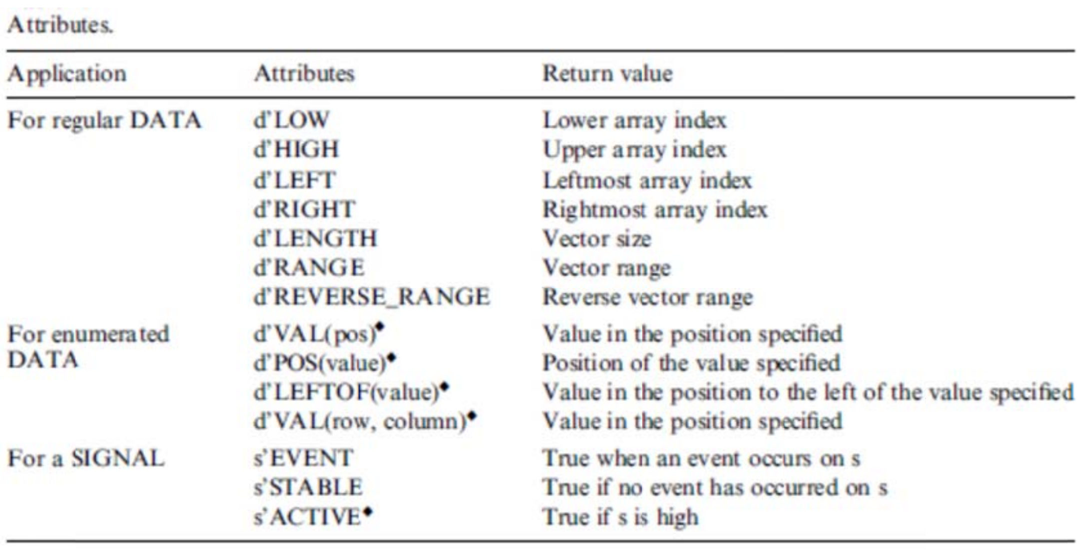




ena	sel	x
0	00	1111
1	00	1110
	01	1101
	10	1011
	11	0111

```
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------------
ENTITY decoder IS
        PORT ( ena : IN STD_LOGIC;
        sel : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        x : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END decoder;
---------------------------------------------
ARCHITECTURE generic_decoder OF decoder IS
        BEGIN
        PROCESS (ena, sel)
                VARIABLE temp1 : STD_LOGIC_VECTOR (x'HIGH DOWNTO 0);
                VARIABLE temp2 : INTEGER RANGE 0 TO x'HIGH;

        BEGIN
                temp1 := (OTHERS => '1');
                temp2 := 0;
        IF (ena='1') THEN

                FOR i IN sel'RANGE LOOP -- sel range is 2 downto 0
                        IF (sel(i)='1') THEN -- Bin-to-Integer conversion
                                temp2:=2*temp2+1;
                        ELSE
                                 := 2*temp2;
                        END IF;
                END LOOP;

                temp1(temp2):='0';

        END IF;
                x <= temp1;
        END PROCESS;

END generic_decoder;
---------------------------------------------
```

Summary:



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