Representing Values in VHDL

Developing code requires entering data of many types. Whether you are initializing signals or typing in operands for comparison statements, it is important that the data is entered correctly so that the compiler understands your intent and you are saved the headaches of finding both syntax and semantic errors. Below we illustrate how certain types of data should be entered into your VHDL code.

**Decimal (Base-10) Values**
Integers are represented in the usual way. For example, the integer fifty seven is represented by 57 and the integer negative five is represented by -5.

VHDL also allows the use exponential notation. For example, the number 7000 can be represented as 7E3. You might be asking yourself if you could type that number 7e3 so that you don’t have to press the SHIFT key. The answer is yes since VHDL is not case sensitive.

**BITS/CHARACTERS**
A bit or character is surrounded by single quotes. Examples include '0', '1', '$', and 'X'.

**BIT_VECTORS/STRINGS**
A BIT_VECTOR is a collection of bits that are related. A STRING is a collection of characters that are related. An example of how these would be defined in the ENTITY declaration is shown in (1) and (2).

```vhdl
ledg : OUT BIT_VECTOR(7 DOWNTO 0);
name : OUT STRING(1 TO 4);
```

Here the symbol ledg is a BIT_VECTOR that represents the 6 green leds LEDG7 through LEDG0 from the DE2 board. If you wanted to assign a the binary number 00011010 to these 8 bits, you would use the following VHDL statement.

```vhdl
ledg<="00011010";
```

If you removed the double quotation marks, you would get a VHDL type mismatch error because the compiler would think you are trying to assign the decimal value eleven thousand ten to the BIT_VECTOR.

When changing a single BIT of a BIT_VECTOR, you treat it for what it is, a single BIT. For example, to change only the BIT at index location 3 you would write

```vhdl
ledg(3)<='0';
```

**Hexadecimal Quantities**
To represent a hexadecimal quantity, place the number in double quotes and precede it with an X. Using hexadecimal can be used as a shortcut for typing in long BIT_VECTORS. The example shown in (3) above would be written as shown in (4).

```vhdl
ledg=x"1A";
```
VHDL Standard Data Types

When we want to use a SIGNAL (an input/output or internal value of a circuit) or when we want to use a variable, we have to tell the compiler what the SIGNAL or variable is. Consider the circuit below, which shows an input X to an ADDER.

![Circuit Diagram]

Is X a single bit?
Or is X a group of two bits (to be applied as the control signals for a 2-4 multiplexer)?
Or is X a group of four bits (to be applied as the input to a 4-16 decoder)?
Or is X best viewed as a Boolean expression that results in a True/False question (e.g., is smoke emanating?)

The point: We must tell the compiler what X is supposed to be. Specifically, we must tell the compiler X’s type. The allows the compiler to determine what operators are allowed and how to execute the operations.

Recall from VHDL Part I that a SIGNAL is defined using one of the two following syntaxes

```
SIGNAL portname: portmode type;  -- Used in the ENTITY declaration
SIGNAL signalname: type         -- Used in the ARCHITECTURE declaration
```

where:

- `portname` and `signalname` are user selected names meeting the standard Naming Rules
- `type` is the data type including the size for vector data types
- `portmode` is IN, OUT, INOUT, BUFFER, or LINKAGE.

Note that the keyword SIGNAL may be omitted in the PORT section of the ENTITY declaration.

The pre-defined operators for some of the more common VHDL types are shown in Table 1 followed by a number of examples illustrating how the various data types might be defined and used.

---

**Table 1**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>Logical AND</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>^</code></td>
<td>Logical XOR</td>
</tr>
</tbody>
</table>

---

2
<table>
<thead>
<tr>
<th>Operator Class</th>
<th>Operators</th>
<th>BOOLEAN</th>
<th>BIT</th>
<th>BIT_VECTOR</th>
<th>STD_LOGIC</th>
<th>STD_LOGIC_VECTOR</th>
<th>CHARACTER</th>
<th>STRING</th>
<th>SEVERITY_LEVEL</th>
<th>INTEGER</th>
<th>REAL</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>AND, OR, NAND, NOR, XOR, XNOR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>NOT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>=, /=, &lt;, &lt;=, &gt;, &gt;=</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>SLL, SRL, SLA, SRA, ROL, ROR</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adding</td>
<td>+, -</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adding</td>
<td>&amp;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sign</td>
<td>+, -</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplying</td>
<td>*, /</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplying</td>
<td>MOD, REM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>ABS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1 – Type Operators

Example ENTITY Definitions

```vhdl
ENTITY example
PORT(
    a, b : IN BIT;
    x, y : OUT BIT;
    key : IN BIT_VECTOR(3 DOWNTO 0);
    ledg : OUT BIT_VECTOR(7 DOWNTO 0);
    ledr : BUFFER BIT_VECTOR(0 TO 17);
    name: OUT STRING(1 TO 4)
);
END example;
```
Example ARCHITECTURE Definitions and signal uses.

ARCHITECTURE arc OF example IS
  SIGNAL c: BIT;
  SIGNAL d1: STD_LOGIC_VECTOR(7 DOWNTO 0);
  SIGNAL d2: STD_LOGIC_VECTOR(0 TO 7);
  SIGNAL z: BOOLEAN;
  SIGNAL initial: CHARACTER;
  -- The RANGE of an INTEGER determines the number of bits used
  -- to store the value of the integer.
  SIGNAL my4bitint: INTEGER RANGE 0 to 15;  -- Uses 4 bits
  SIGNAL my8bitint: INTEGER RANGE 0 to 255;  -- Uses 8 bits
  SIGNAL myreal: REAL;
  SIGNAL mytime: TIME;
BEGIN
  x<=NOT a;
  y<='1';
  ledg(3 downto 0)<=key;  -- Assign inputs key(3 to 0) to ledg(3 to 0)
  ledr(0)<=key(3);  -- Assign inputs key(3 to 0) to ledr(0 to 3)
  ledr(1)<=key(2);
  ledr(2)<=key(1);
  ledr(3)<=key(0);
  ledr(4 to 7)<="1001";  -- Assign 1001 to ledr(4 to 7)
  name<="USNA";
  my4bitint<=3;
  my8bitint<=2;
  d1<="00100001";
  -- Roll ledr bits 0 to 3 RIGHT 3 times and store in
  -- ledr bits 10 to 13. Note that this does not work
  -- if ledr is type OUT vice BUFFER.
  ledr(10 to 13)<=ledr(0 to 3) ROR my4bitint;
  -- Roll key bits 3 to 0 RIGHT 2 times and store in ledg
  -- bits 7 to 4. Note that the direction that the variable
  -- is defined effects how the shift functions will work.
  ledg(7 downto 4)<=key ROR my8bitint;
END arc;
Concurrent Code

VHDL is inherently concurrent. This means that the lines of code can be viewed as all being executed at the same time. If we want VHDL to NOT execute all lines at the same time—i.e., if we want VHDL to execute lines of code sequentially, one after another—we have to specifically inform VHDL of this intent. Concurrent code is also called dataflow code.

We have seen several examples of concurrent code that implements Boolean or arithmetic equations (mux, decoder, full adder, etc).

Other fundamental types of statements that can be used in concurrent code are:

- Conditional signal assignments using WHEN/ELSE statements.
- Selected signal assignments using WITH/SELECT/WHEN statements.
- Structured assignments using GENERATE statements.

WHEN/ELSE Statement

Syntax:

```
<optional_label>: <target> <=
    <value> when <condition> else
    <value> when <condition> else
    <value> when <condition> else
    ...
    <value>;
```

The conditions in a WHEN/ELSE statement are prioritized. The first output statement listed has the highest priority and will be executed first, if the condition is true. If the first condition is not true, the second output statement will be executed next, and so on.

WITH/SELECT/WHEN Statement

Syntax:

```
<optional_label>: with <expression> select
    <target> <= <value> when <choices>,
    <value> when <choices>,
    <value> when <choices>,
    ...
    <value> when others;
```

The selected signal assignment (WITH/SELECT/WHEN) examines the value of the expression and executes only the assignment statement that matches the WHEN value, and all other statements are skipped.

WHEN OTHERS is required to terminate a WITH/SELECT/WHEN statement. This is used to ensure that all possible choices of the expression are considered.

Note the use of commas in the syntax for the WITH/SELECT/WHEN statement.
WHEN/ELSE Examples
Implement a 4-to-1 mux using a WHEN/ELSE statement.

ENTITY mux_4_1_WE IS
    PORT(
        -- Input ports
        s1, s0 : IN BIT;
        f0, f1, f2, f3 : IN BIT;
    
        -- Output port
        mux_out : OUT BIT
    );
END mux_4_1_WE;
ARCHITECTURE dataflow OF mux_4_1_WE IS
BEGIN
    -- Implement 4 to 1 mux

    mux_out <= f0 WHEN (s1 = '0' AND s0 = '0') ELSE
                   f1 WHEN (s1 = '0' AND s0 = '1') ELSE
                   f2 WHEN (s1 = '1' AND s0 = '0') ELSE
                   f3;
END dataflow;

Note that if we use type BIT_VECTOR to combine S1 and S0 into one signal of 2 bits, we can write the VHDL code as:

ENTITY mux_4_1_WE_b IS
    PORT(
        -- Input ports
        s : IN BIT_VECTOR (1 downto 0);
        f0, f1, f2, f3 : IN BIT;
    
        -- Output port
        mux_out : OUT BIT
    );
END mux_4_1_WE_b;

ARCHITECTURE dataflow OF mux_4_1_WE_b IS
BEGIN
    -- Implement 4 to 1 mux

    mux_out <= f0 WHEN s = "00" ELSE
                   f1 WHEN s = "01" ELSE
                   f2 WHEN s = "10" ELSE
                   f3;
END dataflow;
WITH/SELECT/WHEN Examples
Implement a 4-to-1 mux using a WITH/SELECT/WHEN statement.

```vhdl
library IEEE;         -- Declare which VHDL library
use IEEE.std_logic_1164.all; -- and packages to use

ENTITY mux_4_1_WSW IS
  PORT(
    s : IN BIT_VECTOR (1 downto 0);
    f0,f1,f2,f3 : IN BIT;
  );
END mux_4_1_WSW;

ARCHITECTURE dataflow OF mux_4_1_WSW IS
BEGIN
  -- Implement 4 to 1 mux
  WITH s SELECT
    mux_out <= f0 WHEN "00", -- "," instead of ";"
              f1 WHEN "01",
              f2 WHEN "10",
              f3 WHEN OTHERS; -- cannot be " f3 WHEN "11" 

END dataflow;

One important keyword, UNAFFECTED, that is often used to indicate no action is to take place for some conditions. For example, let’s look at VHDL code for a 3-to-1 mux (output is assigned to one of 3 inputs).

ENTITY mux_3_1 IS
  PORT(
    s : IN BIT_VECTOR (1 downto 0);
    f0,f1,f2 : IN BIT;
  );
END mux_3_1;

ARCHITECTURE dataflow OF mux_3_1 IS
BEGIN
  -- Implement 4 to 1 mux
  WITH s SELECT
    mux_out <= f0 WHEN "00", -- "," instead of ";"
              f1 WHEN "01",
              f2 WHEN "10",
              UNAFFECTED WHEN OTHERS;

END dataflow;
```
Comparison Example 1
Implement \( F = (AB + CD) \) using the following three methods.

a. Boolean expression.
b. WHEN/ELSE statement.
c. WITH/SELECT/WHEN statement.

Boolean Expression Solution

```vhdl
ENTITY example_1_a IS
PORT(
    a,b,c,d : IN BIT;
    f : OUT BIT
); END example_1_a;

ARCHITECTURE dataflow OF example_1_a IS
BEGIN
    f <= (a AND NOT b) NOR (NOT c AND NOT d);
END dataflow;
```

WHEN/ELSE Solution

```vhdl
ARCHITECTURE dataflow OF example_1_b IS
BEGIN
    f <= '1' WHEN ( (a = '1' AND b = '0') NOR (c = '0' AND d = '0') ) ELSE '0';
END dataflow;
```

WITH/SELECT/WHEN Solution

```vhdl
ARCHITECTURE dataflow OF example_1_c IS
BEGIN
    WITH ((a AND NOT b) NOR (NOT c AND NOT d)) SELECT
    f <=  '1' WHEN '1',
         '0' WHEN '0',
         '0' WHEN OTHERS;
END dataflow;
```
**GENERATE Statement**

GENERATE statement is another important concurrent statement that can be used to reduce number of lines of code for structured circuits such as iterative systems. By employing GENERATE statement, a section of code can be repeated for a number of times. GENERATE statement has two modes:

1. Unconditional GENERATE (FOR/GENERATE)
2. Conditional GENERATE (IF/GENERATE).

Syntax for unconditional GENERATE statement (FOR/GENERATE):

```plaintext
<generate_label>:
  for <loop_id> in <range> generate
      -- Concurrent Statement(s)
  end generate;
```

Syntax for conditional GENERATE statement (IF/GENERATE):

```plaintext
<generate_label>:
  if <condition> generate
      -- Concurrent Statement(s)
  end generate;
```

Notes:
- A label is required for a GENERATE statement.
- IF/GENERATE is used within a FOR/GENERATE loop.
- loop_id is generally of type INTEGER.

**GENERATE Example**

Implement the following circuit using GENERATE statements. Assume that Switches 7 down to 0 are the A inputs and Switches 17 down to 10 are the B inputs. The output C is displayed on the eight green LEDs. The inputs A and B should be displayed on the corresponding red LEDs.

![Circuit Diagram](image.png)
Unconditional GENERATE Solution

ENTITY unconditional_generate IS
  PORT(
    sw : IN BIT_VECTOR (17 DOWNTO 0);
    ledr : OUT BIT_VECTOR (17 DOWNTO 0);
  -- Note that Switches and Red LEDs 8 and 9
  -- are defined but not rea
    ledg : OUT BIT_VECTOR (7 DOWNTO 0)
  );
END unconditional_generate;

ARCHITECTURE dataflow OF unconditional_generate IS
BEGIN
  ledr<=sw; -- Display value of switch on Red LEDs

  -- Take care of two NOR cases
  ledg(0) <= sw(0) NOR sw(10);
  ledg(7) <= sw(7) NOR sw(17);

  -- Use GENERATE function to address 6 OR cases.
  ugen: FOR i IN 1 TO 6 GENERATE
    ledg(i) <= sw(i) OR sw(i+10);
  END GENERATE;
END dataflow;

Conditional GENERATE Solution

ENTITY conditional_generate IS
  PORT(
    sw : IN BIT_VECTOR (17 DOWNTO 0);
    ledr : OUT BIT_VECTOR (17 DOWNTO 0);
  -- Note that Switches and Red LEDs 8 and 9
  -- are defined but not rea
    ledg : OUT BIT_VECTOR (7 DOWNTO 0)
  );
END conditional_generate;

ARCHITECTURE dataflow OF conditional_generate IS
BEGIN
  ledr<=sw; -- Display value of switch on Red LEDs

  ugen: FOR i IN 0 TO 7 GENERATE
    -- If i equals 0 or 7, use NOR
    ugen1: IF (i=0) OR (i=7) GENERATE
      ledg(i) <= sw(i) NOR sw(i+10);
    END GENERATE;
    -- If i is not equal to 0 or 7 use OR
    ugen2: IF (i/=0) AND (i/=7) GENERATE
      ledg(i) <= sw(i) OR sw(i+10);
    END GENERATE;
  END GENERATE;
END dataflow;
Comparison Example 2
Implement a 2-to-4 decoder using
a. WHEN/ELSE statement.
b. WITH/SELECT/WHEN statement.
c. GENERATE statement.

WHEN/ELSE Solution
ENTITY example_2a IS
   PORT(
      -- Input ports
         sel       : IN BIT_VECTOR (1 downto 0);
      
      -- Output ports
         f        : OUT BIT_VECTOR (3 downto 0)
   );
END example_2a;
ARCHITECTURE dataflow OF example_2a IS
BEGIN
   -- Implement function 2-to-4 decoder;
   
   f <=  "0001" WHEN sel = "00" ELSE  
         "0010" WHEN sel = "01" ELSE  
         "0100" WHEN sel = "10" ELSE  
         "1000" ;
END dataflow;

WITH/SELECT/WHEN Solution
(Uses Same ENTITY Section as WHEN/ELSE Solution)
ARCHITECTURE dataflow OF example_2b IS
BEGIN
   WITH sel SELECT
      f <=  "0001" WHEN "00",  
            "0010" WHEN "01",  
            "0100" WHEN "10",  
            "1000" WHEN OTHERS;
END dataflow;

GENERATE Solution
ENTITY example_2c IS
   PORT(
      sw        : IN INTEGER RANGE 0 TO 3;
      ledg : OUT BIT_VECTOR (3 DOWNTO 0)
   );
END example_2c;
ARCHITECTURE dataflow OF example_2c IS
BEGIN
   gen: FOR i IN 0 TO 3 GENERATE
      ledg(i) <= '1' WHEN (i=sw) ELSE '0';
END GENERATE;
END dataflow;

Note: type INTEGER
**STD_LOGIC Type**

Although declaring a signal to be a BIT or a BIT VECTOR is intuitive and easy to understand, it is actually frowned upon! It is common practice to use the STD_LOGIC and STD_LOGIC_VECTOR types in place of the BIT and BIT VECTOR types. The reasons why could be debated but the primary benefit is it can have a value of more than just ‘1’, or ‘0’. Specifically it allows for conditions such as the High Impedance condition necessary for a buffer and other special case conditions. The values that the STD_LOGIC type may have are shown in Table 2.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Uninitialized</td>
</tr>
<tr>
<td>X</td>
<td>Forcing Unknown</td>
</tr>
<tr>
<td>0</td>
<td>Forcing 0</td>
</tr>
<tr>
<td>1</td>
<td>Forcing 1</td>
</tr>
<tr>
<td>Z</td>
<td>High Impedance</td>
</tr>
<tr>
<td>W</td>
<td>Weak Unknown</td>
</tr>
<tr>
<td>L</td>
<td>Weak 0</td>
</tr>
<tr>
<td>H</td>
<td>Weak 1</td>
</tr>
<tr>
<td>-</td>
<td>Don’t Care</td>
</tr>
</tbody>
</table>

Table 2 – STD_LOGIC Values
Seemingly Complicated Example
An ALU.

-- Library declaration
library IEEE;
-- Declare which VHDL library
use IEEE.std_logic_1164.all;
-- and packages to use
use IEEE.std_logic_unsigned.all; -- for +, - operators

-- Entity declaration
ENTITY alu IS
PORT(
    a, b : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    cin : IN STD_LOGIC;
    opcode : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
    y : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
);
END alu;

-- Architecture body
ARCHITECTURE alu_arch OF alu IS

SIGNAL y_logical : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL y_arith : STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN

    WITH opcode(2 DOWNTO 0) SELECT
    y_logical <=
    NOT a WHEN "000",
    NOT b WHEN "001",
    a AND b WHEN "010",
    a OR b WHEN "011",
    a NAND b WHEN "100",
    a NOR b WHEN "101",
    a XOR b WHEN "110",
    a XNOR b WHEN OTHERS;

    WITH opcode(2 DOWNTO 0) SELECT
    y_arith <=
    a WHEN "000",
    b WHEN "001",
    a + 1 WHEN "010",
    b + 1 WHEN "011",
    a - 1 WHEN "100",
    b - 1 WHEN "101",
    a + b WHEN "110",
    a + b + cin WHEN OTHERS;

    WITH opcode(3) SELECT
    y <=
    y_logical WHEN '0',
    y_arith WHEN OTHERS;

END alu_arch;