1. VHDL Basics
   1.1. Valid Names
   1.2. Comments
   1.3. Entity and Architecture
   1.4. Ports
   1.5. Signals and Variables
   1.6. Type
      1.6.1. STD_LOGIC
      1.6.2. Enumerated Type
   1.7. Libraries and Packages

1.1 VALID NAMES

A valid name in VHDL consists of a letter followed by any number of letters or numbers, without spaces. VHDL is not case sensitive. An underscore may be used within a name, but may not begin or end the name. Two consecutive underscores are not permitted.

**EXAMPLES**

Valid names:
- `decode4`
- `just_in_time`
- `What_4`

Invalid names:
- `4decode` (begins with a digit)
- `in__time` (two consecutive underscores)
- `_What_4` (begins with underscore)
- `my_design` (space inside name)
- `your_words?` (special character ? not allowed)

1.2 COMMENTS

A comment is explanatory text that is ignored by the VHDL compiler. It is indicated by two consecutive hyphens.

**EXAMPLE**

```vhdl
-- This is a comment.
```
All VHDL files require an entity declaration and an architecture body. The entity declaration indicates the input and output ports of the design. The architecture body details the internal relationship between inputs and outputs. The VHDL file name must be the same as the entity name.

Syntax:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY __entity_name IS
  GENERIC(define parameters);
  PORT(define inputs and outputs);
END __entity_name;
ARCHITECTURE a OF __entity_name IS
  SIGNAL and COMPONENT declarations;
BEGIN
  statements;
END a;
```

**EXAMPLES**

-- Majority vote circuit (majority.vhd)
```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY majority IS
  PORT(
    a, b, c: IN  STD_LOGIC;
    y      : OUT STD_LOGIC);
END majority;
ARCHITECTURE a OF majority IS
BEGIN
  y <= (a and b) or (b and c) or (a and c);
END a;
```

-- 2-line-to-4-line decoder with active-HIGH outputs (decoder.vhd)
```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY decoder IS
  PORT(
    d : IN  STD_LOGIC_VECTOR (1 downto 0);
    y : OUT STD_LOGIC_VECTOR (3 downto 0));
END decoder;
ARCHITECTURE a OF decoder IS
BEGIN
  WITH d SELECT
    y <= "0001" WHEN "00",
        "0010" WHEN "01",
        "0100" WHEN "10",
        "1000" WHEN "11",
        "0000" WHEN others;
END a;
```
A port in VHDL is a connection from a VHDL design entity to the outside world. The direction or directions in which a port may operate is called its mode. A VHDL port may have one of four modes: IN (input only), OUT (output only), INOUT (bidirectional), and BUFFER (output, with feedback from the output back into the design entity). The mode of a port is declared in the port statement of an entity declaration or component declaration.

**EXAMPLES**

```vhdl
ENTITY mux IS
  PORT(
    s1, s0      : IN STD_LOGIC;
    y0, y1, y2, y3 : OUT STD_LOGIC);
END mux;

ENTITY srg8 IS
  PORT(
    clock, reset : IN  STD_LOGIC;
    q            : BUFFER STD_LOGIC_VECTOR (7 downto 0));
END srg8;
```

### 1.5 SIGNALS AND VARIABLES

A signal is like an internal wire connecting two or more points inside an architecture body. It is declared before the BEGIN statement of an architecture body and is global to the architecture. Its value is assigned with the `<=` operator.

A variable is a piece of working memory, local to a specific process. It is declared before the BEGIN statement of a process and is assigned using the `:=` operator.

**EXAMPLE**

```vhdl
ARCHITECTURE a OF design4 IS
  SIGNAL connect : STD_LOGIC_VECTOR ( 7 downto 0);
BEGIN
  PROCESS check IS
    VARIABLE count : INTEGER RANGE 0 TO 255;
    BEGIN
      IF (clock'EVENT and clock = '1') THEN
        count := count + 1;     -- Variable assignment statement
      END IF;
    END PROCESS;
  connect <= a and b;             -- Signal assignment statement
END a;
```

### 1.6 TYPE

The type of a port, signal, or variable determines the values it can have. For example, a signal of type BIT can only have values ‘0’ and ‘1’. A signal of type INTEGER can have any integer value, up to the limits of the bit size of the
particular computer system for which the VHDL compiler is designed. Some common types are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
<th>How Written</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>‘0’, ‘1’</td>
<td>Single quotes</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Integer values</td>
<td>No quotes</td>
</tr>
<tr>
<td>BIT_VECTOR</td>
<td>Multiple instances of ‘0’ and ‘1’</td>
<td>Double quotes (e.g., “00101”)</td>
</tr>
</tbody>
</table>

### 1.6.1 STD_LOGIC

The STD_LOGIC (standard logic) type, also called IEEE Std.1164 Multi-Valued Logic, gives a broader range of output values than just ‘0’ and ‘1’. Any port, signal, or variable of type STD_LOGIC or STD_LOGIC_VECTOR can have any of the following values.

‘U’, -- Uninitialized
‘X’, -- Forcing Unknown
‘0’, -- Forcing 0
‘1’, -- Forcing 1
‘Z’, -- High Impedance
‘W’, -- Weak Unknown
‘L’, -- Weak 0
‘H’, -- Weak 1
‘-’, -- Don’t care

“Forcing” levels are deemed to be the equivalent of a gate output. “Weak” levels are specified by a pull-up or pull-down resistor. The ‘Z’ state is used as the high-impedance state of a tristate buffer.

The majority of applications can be handled by ‘X’, ‘0’, ‘1’, and ‘Z’ values. To use STD_LOGIC in a VHDL file, you must include the following reference to the VHDL library called ieee and the std_logic_1164 package before the entity declaration.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```

### 1.6.2 Enumerated Type

An enumerated type is a user-defined type that lists all possible values for a port, signal, or variable. One use of an enumerated type is to list the states of a state machine.

```vhdl
TYPE STATE_TYPE IS (idle, start, pulse, read);
SIGNAL state: STATE_TYPE;
```
A concurrent structure in VHDL acts as a separate component. A change applied to multiple concurrent structures acts on all affected structures at the same time. This is similar to a signal applied to multiple components in a circuit; a change in the signal will operate on all the components simultaneously.

2. Concurrent Structures
   2.1. Concurrent Signal Assignment Statement
   2.2. Selected Signal Assignment Statement
   2.3. Conditional Signal Assignment Statements
   2.4. Components
      2.4.1. Component Declaration
      2.4.2. Component Instantiation
      2.4.3. GENERIC Clause
   2.5. GENERATE Statement
   2.6. PROCESS Statement

A concurrent signal assignment statement assigns a port or signal the value of a Boolean expression or constant. This statement is useful for encoding a Boolean equation. Since the operators and, or, not, and xor have equal precedence in VHDL, the order of precedence must be made explicit by parentheses.

Syntax:
```
__signal <= __expression;
```
A selected signal assignment statement assigns one of several alternative values to a port or signal, based on the value of a selecting signal. It can be used to implement a truth table or a selecting circuit like a multiplexer.

**Syntax:**

```
label: WITH __expression SELECT
__signal <= __expression WHEN __constant_value,
__expression WHEN __constant_value,
__expression WHEN __constant_value,
__expression WHEN __constant_value;
```

---

A conditional signal assignment statement assigns a value to a port or signal based on a series of linked conditions. The basic structure assigns a value if the first condition is true. If not, another value is assigned if a second condition is true, and so on, until a default condition is reached. This is an ideal structure for a priority encoder.

**Syntax:**

```
__label: __signal <= __expression WHEN __boolean_expression ELSE
__expression WHEN __boolean_expression ELSE
__expression WHEN __boolean_expression ELSE
__expression;
```
A VHDL file can use another VHDL file as a component. The general form of a design entity using components is:

```vhdl
ENTITY entity_name IS
  PORT ( input and output definitions);
END entity_name;
ARCHITECTURE arch_name OF entity_name IS
  component declaration(s);
  signal declaration(s);
BEGIN
  Component instantiation(s);
  Other statements;
END arch_name;
```

### 2.4.1 Component Declaration

A component declaration is similar in form to an entity declaration, in that it includes the required ports and parameters of the component. The difference is that it refers to a design described in a separate VHDL file. The ports and parameters in the component declaration may be a subset of those in the component file, but they must have the same names.

**Syntax:**

```vhdl
COMPONENT __component_name
  GENERIC(__parameter_name : string := __default_value;
          __parameter_name : integer := __default_value);
  PORT(
    __input name, __input_name        : IN STD_LOGIC;
    __bidir name, __bidir_name        : INOUT STD_LOGIC;
    __output name, __output_name      : OUT STD_LOGIC);
END COMPONENT;
```
2.4.2 Component Instantiation

Each instance of a component requires a component instantiation statement. Ports can be assigned explicitly with the => operator, or implicitly by inserting the user port name in the position of the corresponding port name within the component declaration.

Syntax:

```
_instance_name: _component_name
GENERIC MAP (_parameter_name => _parameter_value,
   _parameter_name => _parameter_value)
PORT MAP (_component_port => _connect_port,
   _component_port => _connect_port);
```

Examples

```vhdl
ARCHITECTURE adder OF add4pa IS
   COMPONENT full_add
      PORT(
         a, b, c_in : IN BIT;
         c_out, sum : OUT BIT);
   END COMPONENT;
   SIGNAL c : BIT_VECTOR (3 downto 1);
BEGIN
   c : BIT_VECTOR (3 downto 1);
BEGIN
   statements
END adder;

-- Four Component Instantiation Statements
-- Explicit port assignments
adder1: full_add
   PORT MAP ( a => a(1),
     b => b(1),
     c_in => c0,
     c_out => c(1),
     sum => sum(1));
adder2: full_add
   PORT MAP ( a => a(2),
     b => b(2),
     c_in => c1,
     c_out => c(2),
     sum => sum(2));
adder3: full_add
   PORT MAP ( a => a(3),
     b => b(3),
     c_in => c2,
     c_out => c(3),
     sum => sum(3));
adder4: full_add
   PORT MAP ( a => a(4),
     b => b(4),
     c_in => c3,
     c_out => c4,
     sum => sum(4));
```
2.4.3 GENERIC Clause

A GENERIC clause allows a component to be designed with one or more unspecified properties (“parameters”) that are specified when the component is instantiated. A parameter specified in a GENERIC clause must be given a default value with the := operator.

Syntax:

- parameters defined in entity declaration of component file

**ENTITY entity_name IS**

  GENERIC(__parameter_name : type := __default_value;
             __parameter_name : type := __default_value);

  PORT (port declarations);

END entity name;

- Component declaration in top-level file also has GENERIC clause.
- Default values of parameters not specified.

**COMPONENT component_name IS**

  GENERIC (__parameter_name : type;
             __parameter_name : type);

  PORT (port declarations);

END COMPONENT;

- Parameters specified in GENERIC MAP in component instantiation

**__instance_name: __component_name**

  GENERIC MAP (__parameter_name => __parameter_value,
                __parameter_name => __parameter_value)

  PORT MAP (port instantiations);

---

**EXAMPLE**

- Component: behaviorally defined shift register
  - with default width of 4.

**ENTITY srt_bhv IS**

  GENERIC (width : POSITIVE := 4);

  PORT(
    serial_in, clk : IN     STD_LOGIC;
    q              : BUFFER STD_LOGIC_VECTOR(width-1 downto 0));

END srt_bhv;

**ARCHITECTURE right_shift of srt_bhv IS**

**BEGIN**

  **PROCESS (clk)**

  **BEGIN**

  **IF (clk’EVENT and clk = ‘1’) THEN**

  **q(width-1 downto 0) <= serial_in & q(width-1 downto 1);**

  **END IF;**

  **END PROCESS;**

**END right_shift;**

(continued)
-- srt8_bhv.vhd
-- 8-bit shift register that instantiates srt_bhv
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY srt8_bhv IS
  PORT(
    data_in, clock : IN     STD_LOGIC;
    qo             : BUFFER STD_LOGIC_VECTOR(7 downto 0));
END srt8_bhv;
ARCHITECTURE right shift of srt8_bhv IS
  -- component declaration
  COMPONENT srt_bhv
    GENERIC (width : POSITIVE);
    PORT(
      serial_in, clk : IN  STD_LOGIC;
      q              : OUT STD_LOGIC_VECTOR(7 downto 0));
  END COMPONENT;
BEGIN
  -- component instantiation
  Shift_right_8: srt_bhv
    GENERIC MAP  (width=> 8)
    PORT MAP  (serial_in => data_in,
               clk        => clock,
               q          => qo);
END right_shift;

A GENERATE statement is used to create multiple instances of a particular hardware structure. It relies on the value of one or more index variables to create the required number of repetitions.

Syntax:

__generate_label:
FOR __index_variable IN __range GENERATE
  __statement;
  __statement;
END GENERATE;

EXAMPLES

-- Instantiate four full adders
adders:
FOR i IN 1 to 4 GENERATE
  adder: full_add PORT MAP (a(i), b(i), c(i-1), c(i), sum(i));
END GENERATE;
-- Instantiate four latches from MAX+PLUS II primitives
-- Requires the statements LIBRARY altera; and
-- USE altera.maxplus.ALL;
latch4:
3. Sequential Structures

3.1. IF Statement

3.1.1. Evaluating Clock Functions

3.2. CASE Statement

An IF statement executes one or more statements if a Boolean condition is satisfied. First, the condition in the IF clause is evaluated. If it is true, the statements immediately following the IF are executed. If the condition is not true, then the condition in the first ELSIF clause is checked. If this condition is true, then the statements immediately following the ELSIF are executed. If the condition is false, then further ELSIF statements are evaluated, if present. If none of the conditions are true, then the statements immediately following the ELSE clause are executed.

FOR i IN 3 downto 0 GENERATE
latch_primitive: latch
    PORT MAP (d => d_in(i), ena => enable, q => q_out (i));
END GENERATE;

A process is a concurrent statement, but the statements inside the process are sequential. For example, a process can define a flip-flop, a separate component whose ports are affected concurrently, but the inside of the flip-flop acts sequentially. A process executes all statements inside it when there is a change of a signal in its sensitivity list. The process label is optional.

Syntax:

```
__process_label:
PROCESS (sensitivity list)
    variable declarations
BEGIN
    sequential statements
END PROCESS __process_label;
```

```
-- D latch
PROCESS (en)
BEGIN
    IF (en = '1') THEN
        q <= d;
    END IF;
END PROCESS;
```
In an IF statement, the ELSIF and ELSE clauses are optional. There may be more than one ELSIF clause. Only one set of statements, those following the IF, ELSIF, or ELSE clause, will be executed in any given IF statement.

Syntax:

```
IF __expression THEN
__statement;
__statement;
ELSIF __expression THEN
__statement;
__statement;
ELSE
__statement;
__statement;
END IF;
```

**EXAMPLE**

```
PROCESS (reset, load, clock)
  VARIABLE count INTEGER RANGE 0 TO 255;
BEGIN
  IF (reset = '0') THEN
    q <= 0;
  ELSIF (load = '0') THEN
    q <= p;
  ELSIF (clock'EVENT and clock = '1') THEN
    count := count + 1;
    q <= count;
  END IF;
END PROCESS;
```

### 3.1.1 Evaluating Clock Functions

As implied in previous examples, the state of a system clock can be checked with an IF statement using the predefined attribute called EVENT. The clause `clock'EVENT` ("clock tick EVENT") is true if there has been activity on the signal called `clock`. Thus `(clock'EVENT and clock = '1')` is true just after a positive edge on `clock`. The clock can be evaluated only once in a PROCESS.

### 3.2 CASE STATEMENT

A CASE statement is used to execute one of several sets of statements, based on the evaluation of a signal.

**Syntax:**

```
CASE __expression IS
  WHEN __constant_value =>
    __statement;
    __statement;
  WHEN __constant_value =>
    __statement;
    __statement;
  WHEN OTHERS =>
    __statement;
    __statement;
END CASE;
```
-- Case evaluates 2-bit value of s and assigns
-- 4-bit values of x and y accordingly
-- Default case (others) required if using STD_LOGIC
CASE s IS
  WHEN "00" =>
    y <= "0001";
    x <= "1110";
  WHEN "01" =>
    y => "0010";
    x => "1101";
  WHEN "10" =>
    y <= "0100";
    x <= "1011";
  WHEN "11" =>
    y <= "1000";
    x <= "0111";
  WHEN others =>
    y <= "0000";
    x <= "1111";
END CASE;

-- This case evaluates the state variable "sequence"
-- that can have two possible values; "start" and "continue"
-- Values of out1 and out2 are also assigned for each case.
CASE sequence IS
  WHEN start =>
    IF in1 = '1' THEN
      sequence <= start;
      out1 <= '0';
      out2 <= '0';
    ELSE
      sequence <= continue;
      out1 <= '1';
      out2 <= '0';
    END IF;
  WHEN continue =>
    sequence <= start;
    out1 <= '0';
    out2 <= '1';
END CASE;