APPENDIX B

VHDL Language Reference

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

```
-- This is a comment.
```

1.3 Entity and Architecture

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)
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)
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;
```

1.4 Ports

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:

```
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 an 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:

```
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:

Type	Values	How written
BIT	'0', '1'	Single quotes
STD_LOGIC (see **Section 1.6.1**)	'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'	Single quotes
INTEGER	Integer values	No quotes
BIT_VECTOR	Multiple instances of '0' and '1'	Double quotes (e.g., "00101")
STD_LOGIC_VECTOR	Multiple instances of 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'	Double quotes (e.g., "11ZZ00")

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.

```
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.

EXAMPLE:

```
TYPE STATE_TYPE IS (idle, start, pulse, read);
SIGNAL state: STATE_TYPE;
```

1.7 Libraries and Packages

A library is a collection of previously compiled VHDL constructs that can be used in a design entity. A package is an uncompiled collection of VHDL constructs that can be used in multiple design entities. Library names must be included at the beginning of a VHDL file, before the entity declaration, to use certain types or functions. The most obvious is the library **ieee,** which in the package **std_logic_1164,** defines the STD_LOGIC (standard logic) type.

Syntax:

```
LIBRARY __ library_name;
USE __library_name.__package_name.ALL;
```

EXAMPLES:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;     -- Defines STD_LOGIC type
USE ieee.std_logic_arith.ALL;    -- Defines arithmetic functions

LIBRARY lpm;                    -- Component declarations for the
USE lpm.lpm_components.ALL; -- Library of Parameterized Modules

LIBRARY altera;             -- Component declarations for
USE altera.maxplus2.ALL; -- MAX+PLUS II primitives
```

2. Concurrent Structures

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.

2.1 Concurrent Signal Assignment 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;
```

EXAMPLES:

```
sum <= (a xor b) xor c;
c_out <= ((a xor b) and c_in) or (a and b);
```

2.2 Selected Signal Assignment Statement

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;
```

EXAMPLES:

```
-- decoder implemented as a truth table (2 inputs, 4 outputs)
-- d has been defined as STD_LOGIC_VECTOR (1 downto 0)
-- y has been defined as STD_LOGIC_VECTOR (3 downto 0)
WITH d SELECT
     y <=   "0001" WHEN "00",
            "0010" WHEN "01",
            "0100" WHEN "10",
            "1000" WHEN "11",
            "0000" WHEN others;
-- multiplexer
-- input signal assigned to y, depending on states of s1, s0
M:   WITH s SELECT
     y <=   d0 WHEN "00",
            d1 WHEN "01",
            d2 WHEN "10",
            d3 WHEN "11";
```

2.3 Conditional Signal Assignment Statement

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;
```

EXAMPLE:

```
-- priority encoder
-- q defined as INTEGER RANGE 0 TO 7
-- d defined as STD_LOGIC_VECTOR (7 downto 0)
encoder:
  q   <= 7 WHEN d(7)='1' ELSE
         6 WHEN d(6)='1' ELSE
         5 WHEN d(5)='1' ELSE
         4 WHEN d(4)='1' ELSE
         3 WHEN d(3)='1' ELSE
         2 WHEN d(2)='1' ELSE
         1 WHEN d(1)='1' ELSE
         0;
```

2.4 Components

A VHDL file can use another VHDL file as a component. The general form of a design entity using components is:

```
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:

```
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;
```

EXAMPLE:

```
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
    statements
END adder;
```

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:

```
-- 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  => c(1),
               c_out => c(2),
               sum   => sum(2));
adder3: full_add
    PORT MAP ( a     => a(3),
               b     => b(3),
               c_in  => c(2),
               c_out => c(3),
               sum   => sum(3));
adder4: full_add
    PORT MAP ( a     => a(4),
               b     => b(4),
               c_in  => c(3),
               c_out => c4,
               sum   => sum (4));

-- Four component instantiations
-- Implicit port assignments
adder1: full_add PORT MAP (a(1), b(1), c0,   c(1), sum(1));
adder2: full_add PORT MAP (a(2), b(2), c(1), c(2), sum(2));
adder3: full_add PORT MAP (a(3), b(3), c(2), c(3), sum(3));
adder4: full_add PORT MAP (a(4), b(4), c(3), c4,   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;


-- 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;

(example continues)
    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;
```

2.5 Generate Statement

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

2.6 Process Statement

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;
```

EXAMPLE:

```
-- D latch
PROCESS (en)
BEGIN
    IF (en = '1') THEN
        q <= d;
    END IF;
END PROCESS;
```

A sequential structure in VHDL is one in which the order of statements affects the operation of the circuit. It can be used to implement combinational circuits, but is primarily used to implement sequential circuits such as latches, counters, shift registers, and state machines. Sequential statements must be contained within a process.

3.1 If Statement

An IF statement executes one or more statements if a Boolean condition is satisfied.

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 (reset = '1' and 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.

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;
```

EXAMPLES:

```
-- 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;
```