1. **VHDL Summary**
   1. **Basic Language Elements**

**Comments**

Comments are preceded by two consecutive hyphens (--). Example:

-- this is a comment

**Identifiers**

VHDL identifier syntax:

* A sequence of one or more uppercase letters, lowercase letters, digits, and the underscore
* Upper and lowercase letters are treated the same (i.e. case insensitive)
* The first character must be a letter
* The last character cannot be the underscore
* Two underscores cannot be together

**Data Objects**

There are three kinds of data objects: signals, variables, and constants.

* The data object **SIGNAL** represents logic signals on a wire in the circuit. A signal does not have memory; thus, if the source of the signal is removed, the signal will not have a value
* A **VARIABLE** object remembers its content and is used for computations in a behavioral model
* A **CONSTANT** object must be initialized with a value when declared, and this value cannot be changed

Example:

SIGNAL x: BIT;

VARIABLE y: INTEGER;

CONSTANT one: STD\_LOGIC\_VECTOR(3 DOWNTO 0) := "0001";

**Data Types**

**BIT** and **BIT\_VECTOR**

The **BIT** and **BIT\_VECTOR** types are predefined in VHDL. Objects of these types can have the values 0 or 1. The **BIT\_VECTOR** type is simply a vector of type **BIT**. A vector with all bits having the same value can be obtained using the **OTHERS** keyword.

Example:

SIGNAL x: BIT;

SIGNAL y: BIT\_VECTOR(7 DOWNTO 0);

x <= '1';

y <= "00000010";

y <= (OTHERS => '0'); -- same as "00000000"

**STD\_LOGIC** and **STD\_LOGIC\_VECTOR**

The **STD\_LOGIC** and **STD\_LOGIC\_VECTOR** types provide more values than the **BIT** type for modeling a real circuit more accurately. Objects of these types can have the following values.

'0' − normal 0

'1' − normal 1

'Z' − high impedance

'−' − don’t care

'L' − weak 0

'H' − weak 1

'U' − uninitialized

'X' − unknown

'W' − weak unknown

The **STD\_LOGIC** and **STD\_LOGIC\_VECTOR** types are not predefined, and so the following two library statements must be included in order to use these types.

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

If objects of type **STD\_LOGIC\_VECTOR** are to be used as binary numbers in arithmetic manipulations, then either one of the following two **USE** statements must also be included

USE IEEE.STD\_LOGIC\_SIGNED.ALL;

for signed number arithmetic, or

USE IEEE.STD\_LOGIC\_UNSIGNED.ALL;

for unsigned number arithmetic. A vector with all bits having the same value can be obtained using the **OTHERS** keyword, as shown in the next example.

Example:

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

SIGNAL x: STD\_LOGIC;

SIGNAL y: STD\_LOGIC\_VECTOR(7 DOWNTO 0);

x <= 'Z';

y <= "0000001Z";

y <= (OTHERS => '0'); -- same as "00000000"

**INTEGER**

The predefined **INTEGER** type defines binary number objects for use with arithmetic operators. By default, an **INTEGER** signal uses 32 bits to represent a signed number. Integers using fewer bits can also be declared with the **RANGE** keyword.

Example:

SIGNAL x: INTEGER;

SIGNAL y: INTEGER RANGE –64 to 64;

**BOOLEAN**

The predefined **BOOLEAN** type defines objects having the two values **TRUE** and **FALSE**.

Example:

SIGNAL x: BOOLEAN;

**Enumeration TYPE**

An enumeration type allows the user to specify the values that the data object can have.

Syntax:

TYPE identifier IS (value1, value2, … );

Example:

TYPE state\_type IS (S1, S2, S3);

SIGNAL state: state\_type;

state <= S1;

**ARRAY**

The **ARRAY** type groups single data objects of the same type together into a one-dimensional or multidimensional array.

Syntax:

TYPE identifier IS ARRAY (range) OF type;

Example:

TYPE byte IS ARRAY(7 DOWNTO 0) OF BIT;

TYPE memory\_type IS ARRAY(1 TO 128) OF byte;

SIGNAL memory: memory\_type;

memory(3) <= "00101101";

**SUBTYPE**

A **SUBTYPE** is a subset of a type, that is, a type with a range constraint.

Syntax:

SUBTYPE identifier IS type RANGE range;

Example:

SUBTYPE integer4 IS INTEGER RANGE –8 TO 7;

SUBTYPE cell IS STD\_LOGIC\_VECTOR(3 DOWNTO 0);

TYPE memArray IS ARRAY(0 TO 15) OF cell;

Some standard subtypes include:

• NATURAL—an integer in the range 0 to INTEGER'HIGH

• POSITIVE—an integer in the range 1 to INTEGER'HIGH

**Data Operators**

The VHDL built-in operators are listed in the table below.

|  |  |  |
| --- | --- | --- |
| **Logical Operators** | **Operation** | **Example** |
| AND | AND | n<= a AND b |
| OR | OR | n<= a OR b |
| NOT | NOT | n<= NOT a |
| NAND | NAND | n<= a NAND b |
| NOR | NOR | n<= a NOR b |
| XOR | XOR | n<= a XOR b |
| XNOR | XNOR | n<= XNOR b |
| **Arithmetic Operators** | **Operation** | **Example** |
| + | Addition | n<= a + b |
| - | Subtraction | n<= a - b |
| \* | Multiplication (integer of floating point) | n<= a \* b |
| / | Division (integer or floating point) | n<= a / b |
| MOD | Modulus (integer) | n<= a MOD b |
| REM | Remainder (integer) | n<= a REM b |
| \*\* | Exponentiation | n<= a \*\* 2 |
| & | Concatenation | n<= ‘a’ & ‘b’ |
| ABS | Absolute | n <= abs(a) |
| **Relational Operators** | **Operation** | **Example** |
| **=** | Equal | IF (n = 10) THEN |
| **/=** | Not equal | IF (n /= 10) THEN |
| **<** | Less than | IF (n <10) THEN |
| **<=** | Less than or equal | IF (n <= 10) THEN |
| **>** | Greater than | IF (n > 10) THEN |
| **>=** | Greater than or equal | IF (n >= 10) THEN |
| **Shift Operators** | **Operation** | **Example** |
| SLL | Shift left logical | n <= “1001010” SLL 2 |
| SRL | Shift right logical | n <= “1001010” SRL 1 |
| SLA | Shift left arithmetic | n <= “1001010” SLA 2 |
| SRA | Shift right arithmetic | n <= “1001010” SRA 1 |
| ROL | Rotate left | n <= “1001010” ROL 2 |
| ROR | Rotate right | n <= “1001010” ROR 3 |

**Entity**

An **ENTITY** declaration declares the external or user interface of the module similar to the declaration of a function. It specifies the name of the entity and its interface. The interface consists of the signals to be passed into the entity or out from it using the two keywords **IN** and **OUT**, respectively.

Syntax:

ENTITY entity-name IS

PORT (list-of-port-names-and-types);

END entity-name;

Example:

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY Siren IS PORT (

M: IN STD\_LOGIC;

D: IN STD\_LOGIC;

V: IN STD\_LOGIC;

S: OUT STD\_LOGIC);

END Siren;

**Architecture**

The **ARCHITECTURE** body defines the actual implementation of the functionality of the entity. This is similar to the definition or implementation of a function. The syntax for the architecture varies, depending on the model (dataflow, behavioral, or structural) you use.

Syntax: Dataflow model

ARCHITECTURE architecture-name OF entity-name IS

Signal-declarations;

BEGIN

concurrent-statements;

END architecture-name;

The concurrent statements are executed concurrently.

Example:

ARCHITECTURE Siren\_Dataflow OF Siren IS

SIGNAL term\_1: STD\_LOGIC;

BEGIN

term\_1 <= D OR V;

S <= term\_1 AND M;

END Siren\_Dataflow;

Syntax: Behavioral model

ARCHITECTURE architecture-name OF entity-name IS

signal-declarations;

function-definitions;

procedure-definitions;

BEGIN

PROCESS-blocks;

concurrent-statements;

END architecture-name;

Statements within the PROCESS block are executed sequentially. However, the PROCESS block itself is a concurrent statement.

Example:

ARCHITECTURE Siren\_Behavioral OF Siren IS

SIGNAL term\_1: STD\_LOGIC;

BEGIN

PROCESS (D, V, M)

BEGIN

term\_1 <= D OR V;

S <= term\_1 AND M;

END PROCESS;

END Siren\_Behavioral;

Syntax: Structural model

ARCHITECTURE architecture-name OF entity-name IS

component-declarations;

signal-declarations;

BEGIN

instance-name: PORT MAP-statements;

concurrent-statements;

END architecture-name;

For each component declaration used, there must be a corresponding entity and architecture for that component. The PORT MAP statements are concurrent statements.

Example:

ARCHITECTURE Siren\_Structural OF Siren IS

COMPONENT myOR PORT (

in1, in2: IN STD\_LOGIC;

out1: OUT STD\_LOGIC);

END COMPONENT;

SIGNAL term1: STD\_LOGIC;

BEGIN

U0: myOR PORT MAP (D, V, term1);

S <= term1 AND M;

END Siren\_Structural;

* 1. **Dataflow Model – Concurrent Statements**

Concurrent statements used in the dataflow model are executed concurrently. Hence, the ordering of these statements does not affect the resulting output.

**Concurrent signal assignment**

The concurrent signal assignment statement assigns a value or the result of evaluating an expression to a signal. This statement is executed whenever a signal in its expression changes value. However, the actual assignment of the value to the signal takes place after a certain delay and not instantaneously as for variable assignments. The expression can be any logical or arithmetical expressions.

Syntax:

signal <= expression;

Example:

y <= '1';

z <= y AND (NOT x);

A vector with all bits having the same value can be obtained using the OTHERS keyword as shown here.

SIGNAL x: STD\_LOGIC\_VECTOR(7 DOWNTO 0);

x <= (OTHERS => '0'); -- 8-bit vector of 0, same as "00000000"

**Conditional signal assignment**

The conditional signal assignment statement selects one of several different values to assign to a signal based on different conditions. This statement is executed whenever a signal in any one of the value or condition changes.

Syntax:

signal <= value1 WHEN condition ELSE

value2 WHEN condition ELSE

…

value3;

Example:

z <= in0 WHEN sel = "00" ELSE

in1 WHEN sel = "01" ELSE

in2 WHEN sel = "10" ELSE

in3;

**Selected signal assignment**

The selected signal assignment statement selects one of several different values to assign to a signal based on the value of a select expression. All possible choices for the expression must be given. The keyword **OTHERS** can be used to denote all remaining choices. This statement is executed whenever a signal in the expression or any one of the value changes.

Syntax:

WITH expression SELECT

signal <= value1 WHEN choice1,

value2 WHEN choice2 | choice3,

…

value4 WHEN OTHERS;

In the above syntax, if *expression* is equal to *choice1*, then *value1* is assigned to *signal*. Otherwise, if *expression* is equal to *choice2* or *choice3*, then *value2* is assigned to *signal*. If *expression* does not match any of the above choices, then *value4* in the optional **WHEN** **OTHERS** clause is assigned to *signal*.

Example:

WITH sel SELECT

z <= in0 WHEN "00",

in1 WHEN "01",

in2 WHEN "10",

in3 WHEN OTHERS;

**Dataflow model sample**

-- outputs a 1 if the 4-bit input is a prime number, 0 otherwise

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY Prime IS PORT (

number: IN STD\_LOGIC\_VECTOR(3 DOWNTO 0);

yes: OUT STD\_LOGIC);

END Prime;

ARCHITECTURE Prime\_Dataflow OF Prime IS

BEGIN

WITH number SELECT

yes <= '1' WHEN "0001" | "0010",

'1' WHEN "0011" | "0101" | "0111" | "1011" | "1101",

'0' WHEN OTHERS;

END Prime\_Dataflow;

* 1. **Behavioral Model – Sequential Statements**

The behavioral model allows statements to be executed sequentially just like in a regular computer program. Sequential statements include many of the standard constructs, such as variable assignments, if-then-else statements, and loops.

**Process**

The PROCESS block contains statements that are executed sequentially. However, the PROCESS statement itself is a concurrent statement. Multiple process blocks in an architecture will be executed simultaneously. These process blocks can be combined together with other concurrent statements.

Syntax:

process-name: PROCESS (sensitivity-list)

variable-declarations;

BEGIN

sequential-statements;

END PROCESS process-name;

The sensitivity list is a comma-separated list of signals, which the process is sensitive to. In other words, whenever a signal in the list changes value, the process will be executed (i.e., all of the statements in the sequential order listed). After the last statement has been executed, the process will be suspended until the next time that a signal in the sensitivity list changes value before it is executed again.

Example:

Siren: PROCESS (D, V, M)

BEGIN

term\_1 <= D OR V;

S <= term\_1 AND M;

END PROCESS;

**Sequential signal assignment**

The sequential signal assignment statement assigns a value to a signal. This statement is just like its concurrent counterpart, except that it is executed sequentially (i.e., only when execution reaches it).

Syntax:

signal <= expression;

Example:

y <= '1';

z <= y AND (NOT x);

**Variable assignment**

The variable assignment statement assigns a value or the result of evaluating an expression to a variable. The value always is assigned to the variable instantaneously whenever this statement is executed. Variables are only declared within a process block.

Syntax:

signal := expression;

Example:

y := '1';

yn := NOT y;

**Wait**

When a process has a sensitivity list, the process always suspends after executing the last statement. An alternative to using a sensitivity list to suspend a process is to use a WAIT statement, which must also be the first statement in a process.

Syntax:

WAIT UNTIL condition;

Example:

-- suspend until a rising clock edge

WAIT UNTIL clock'EVENT AND clock = '1';

**If-Then-Else**

Syntax:

IF condition THEN

sequential-statements1;

ELSE

sequential-statements2;

END IF;

Or

IF condition1 THEN

sequential-statements1;

ELSIF condition2 THEN

sequential-statements2;

…

ELSE

sequential-statements3;

END IF;

Example:

IF count /= 10 THEN -- not equal

count := count + 1;

ELSE

count := 0;

END IF;

**Case**

Syntax:

CASE expression IS

WHEN choices => sequential-statements;

WHEN choices => sequential-statements;

…

WHEN OTHERS => sequential-statements;

END CASE;

Example:

CASE sel IS

WHEN "00" => z <= in0;

WHEN "01" => z <= in1;

WHEN "10" => z <= in2;

WHEN OTHERS => z <= in3;

END CASE;

**Null**

The **NULL** statement does not perform any actions.

Syntax:

NULL;

**For**

Syntax:

FOR identifier IN start [TO | DOWNTO] stop LOOP

sequential-statements;

END LOOP;

Loop statements must have locally static bounds. The identifier is implicitly declared, so no explicit declaration of the variable is needed.

Example:

sum := 0;

FOR count IN 1 TO 10 LOOP

sum := sum + count;

END LOOP;

**While**

Syntax:

WHILE condition LOOP

sequential-statements;

END LOOP;

**Loop**

Syntax:

LOOP

sequential-statements;

EXIT WHEN condition;

END LOOP;

**Exit**

The **EXIT** statement can only be used inside a loop. It causes execution to jump out of the innermost loop and is usually used in conjunction with the **LOOP** statement.

Syntax:

EXIT WHEN condition;

**Next**

The **NEXT** statement can be used only inside a loop. It causes execution to skip to the end of the current iteration and continue with the beginning of the next iteration. It usually is used in conjunction with the **FOR** statement.

Syntax:

NEXT WHEN condition;

Example:

sum := 0;

FOR count IN 1 TO 10 LOOP

NEXT WHEN count = 3;

sum := sum + count;

END LOOP;

**Function**

Syntax: Function declaration

FUNCTION function-name (parameter-list) RETURN return-type;

Syntax: Function definition

FUNCTION function-name (parameter-list) RETURN return-type IS

BEGIN

sequential-statements;

END function-name;

Syntax: Function call

function-name (actuals);

Parameters in the parameter list can be either signals or variables of mode **IN** only.

Example:

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY test\_function IS PORT (

x: IN STD\_LOGIC\_VECTOR(3 DOWNTO 0);

z: OUT STD\_LOGIC\_VECTOR(3 DOWNTO 0));

END test\_function;

ARCHITECTURE Behavioral OF test\_function IS

SUBTYPE bit4 IS STD\_LOGIC\_VECTOR(3 DOWNTO 0);

FUNCTION Shiftright (input: IN bit4) RETURN bit4 IS

BEGIN

RETURN '0' & input(3 DOWNTO 1);

END shiftright;

SIGNAL mysignal: bit4;

BEGIN

PROCESS

BEGIN

mysignal <= x;

z <= Shiftright(mysignal);

END PROCESS;

END Behavioral;

**Procedure**

Syntax: Procedure declaration

PROCEDURE procedure-name (parameter-list);

Syntax: Procedure definition

PROCEDURE procedure-name (parameter-list) IS

BEGIN

sequential-statements;

END procedure-name;

Syntax: Procedure call

procedure-name (actuals);

Parameters in the parameter-list are variables of modes **IN**, **OUT**, or **INOUT**.

Example:

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY test\_procedure IS PORT (

x: IN STD\_LOGIC\_VECTOR(3 DOWNTO 0);

z: OUT STD\_LOGIC\_VECTOR(3 DOWNTO 0));

END test\_procedure;

ARCHITECTURE Behavioral OF test\_procedure IS

SUBTYPE bit4 IS STD\_LOGIC\_VECTOR(3 DOWNTO 0);

PROCEDURE Shiftright (input: IN bit4; output: OUT bit4) IS

BEGIN

output := '0' & input(3 DOWNTO 1);

END shiftright;

BEGIN

PROCESS

VARIABLE mysignal: bit4;

BEGIN

Shiftright(x, mysignal);

z <= mysignal;

END PROCESS;

END Behavioral;

**Behavioral Model Sample**

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY bcd IS PORT (

I: IN STD\_LOGIC\_VECTOR(3 DOWNTO 0);

Segs: OUT STD\_LOGIC\_VECTOR(1 TO 7));

END bcd;

ARCHITECTURE Behavioral OF bcd IS

BEGIN

PROCESS(I)

BEGIN

CASE I IS

WHEN "0000" => Segs <= "1111110";

WHEN "0001" => Segs <= "0110000";

WHEN "0010" => Segs <= "1101101";

WHEN "0011" => Segs <= "1111001";

WHEN "0100" => Segs <= "0110011";

WHEN "0101" => Segs <= "1011011";

WHEN "0110" => Segs <= "1011111";

WHEN "0111" => Segs <= "1110000";

WHEN "1000" => Segs <= "1111111";

WHEN "1001" => Segs <= "1110011";

WHEN OTHERS => Segs <= "0000000";

END CASE;

END PROCESS;

END Behavioral;

* 1. **Structural Model – Concurrent Statements**

The structural model allows the manual connection of several components together using signals. All components used must first be defined with their respective **ENTITY** and **ARCHITECTURE** sections, which can be in the same file or can be in separate files.

In the topmost module, each component used in the *netlist* is first declared using the **COMPONENT** statement. The declared components are then instantiated with the actual components in the circuit using the **PORT MAP** statement. **SIGNAL**s are then used to connect the components together according to the *netlist*.

**Component** Declaration

The **COMPONENT** declaration statement declares the name and the interface of a component that is used in the circuit description. For each **COMPONENT** declaration used, there must be a corresponding **ENTITY** and **ARCHITECTURE** for that component. The declaration name and the interface must match exactly the name and interface that is specified in the **ENTITY** section for that component.

Syntax:

COMPONENT component-name IS

PORT (list-of-port-names-and-types);

END COMPONENT;

or

COMPONENT component-name IS

GENERIC (identifier: type := constant);

PORT (list-of-port-names-and-types);

END COMPONENT;

Example:

COMPONENT half\_adder IS PORT (

xi, yi, cin: IN STD\_LOGIC;

cout, si: OUT STD\_LOGIC);

END COMPONENT;

**Port Map**

The **PORT MAP** statement instantiates a declared component with an actual component in the circuit by specifying how the connections to this instance of the component are to be made.

Syntax:

label: component-name PORT MAP (association-list);

or

label: component-name GENERIC MAP (constant) PORT MAP (association-list);

The association list can be specified using either the positional or named method.

Example: Positional association

SIGNAL x0, x1, y0, y1, c0, c1, c2, s0, s1: STD\_LOGIC;

U1: half\_adder PORT MAP (x0, y0, c0, c1, s0);

U2: half\_adder PORT MAP (x1, y1, c1, c2, s1);

Example: Named association

SIGNAL x0, x1, y0, y1, c0, c1, c2, s0, s1: STD\_LOGIC;

U1: half\_adder PORT MAP (cout=>c1, si=>s0, cin=>c0, xi=>x0, yi=>y0);

U2: half\_adder PORT MAP (cin=>c1, xi=>x1, yi=>y1, cout=>c2, si=>s1);

**Open**

The **OPEN** keyword is used in the **PORT MAP** association list to signify that a particular output port is not connected or used. It cannot be used for an input port.

Example:

U1: half\_adder PORT MAP (x0, y0, c0, OPEN, s0);

**Generate**

The **GENERATE** statement works like a macro expansion. It provides a simple way to duplicate similar components.

Syntax:

label: FOR identifier IN start [TO | DOWNTO] stop GENERATE

port-map-statements;

END GENERATE label;

Example:

-- using a FOR-GENERATE statement to generate four instances of the full adder

-- component for a 4-bit adder

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY Adder4 IS PORT (

Cin: IN STD\_LOGIC;

A, B: IN STD\_LOGIC\_VECTOR(3 DOWNTO 0);

Cout: OUT STD\_LOGIC;

SUM: OUT STD\_LOGIC\_VECTOR(3 DOWNTO 0));

END Adder4;

ARCHITECTURE Structural OF Adder4 IS

COMPONENT FA PORT (

ci, xi, yi: IN STD\_LOGIC;

co, si: OUT STD\_LOGIC);

END COMPONENT;

SIGNAL Carryv: STD\_LOGIC\_VECTOR(4 DOWNTO 0);

BEGIN

Carryv(0) <= Cin;

Adder: FOR k IN 3 DOWNTO 0 GENERATE

FullAdder: FA PORT MAP (Carryv(k), A(k), B(k), Carryv(k+1), SUM(k));

END GENERATE Adder;

Cout <= Carryv(4);

END Structural;

**Structural Model Sample**

This example is based on the following circuit:

**

-- declare and define the 2-input OR gate

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY myOR IS PORT (

in1, in2: IN STD\_LOGIC;

out1: OUT STD\_LOGIC);

END myOR;

ARCHITECTURE OR\_Dataflow OF myOR IS

BEGIN

out1 <= in1 OR in2; -- performs the OR operation

END OR\_Dataflow;

-- declare and define the 2-input AND gate

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY myAND IS PORT (

in1, in2: IN STD\_LOGIC;

out1: OUT STD\_LOGIC);

END myAND;

ARCHITECTURE AND\_Dataflow OF myAND IS

BEGIN

out1 <= in1 AND in2; -- performs the AND operation

END AND\_Dataflow;

-- topmost module for the siren

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_1164.ALL;

ENTITY Siren IS PORT (

M: IN STD\_LOGIC;

D: IN STD\_LOGIC;

V: IN STD\_LOGIC;

S: OUT STD\_LOGIC);

END Siren;

ARCHITECTURE Siren\_Structural OF Siren IS

-- declaration of the needed OR gate

COMPONENT myOR PORT (

in1, in2: IN STD\_LOGIC;

out1: OUT STD\_LOGIC);

END COMPONENT;

-- declaration of the needed AND gate

COMPONENT myAND PORT (

in1, in2: IN STD\_LOGIC;

out1: OUT STD\_LOGIC);

END COMPONENT;

-- signal for connecting the output of the OR gate

-- with the input to the AND gate

SIGNAL term1: STD\_LOGIC;

BEGIN

U0: myOR PORT MAP (D, V, term1);

U1: myAND PORT MAP (term1, M, S);

END Siren\_Structural;

* 1. **Conversion Routines**

**CONV\_INTEGER( )**

The **CONV\_INTEGER**( ) routine converts a **STD\_LOGIC\_VECTOR** type to an **INTEGER** type. Its use requires the inclusion of the following library.

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_UNSIGNED.ALL;

Syntax:

CONV\_INTEGER(std\_logic\_vector)

Example:

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_UNSIGNED.ALL;

SIGNAL four\_bit: STD\_LOGIC\_VECTOR(3 DOWNTO 0);

SIGNAL n: INTEGER;

n := CONV\_INTEGER(four\_bit);

**CONV\_STD\_LOGIC\_VECTOR( , )**

The **CONV\_STD\_LOGIC\_VECTOR( , )** routine converts an **INTEGER** type to a **STD\_LOGIC\_VECTOR** type. Its use requires the inclusion of the following library.

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_ARITH.ALL;

Syntax:

CONV\_STD\_LOGIC\_VECTOR (integer, number\_of\_bits)

Example:

LIBRARY IEEE;

USE IEEE.STD\_LOGIC\_ARITH.ALL;

SIGNAL four\_bit: STD\_LOGIC\_VECTOR(3 DOWNTO 0);

SIGNAL n: INTEGER;

four\_bit := CONV\_STD\_LOGIC\_VECTOR(n, 4);