Chapter 1

VHDL

1.1 Introduction

This section discusses some of the fundamental elements of VHDL implemented in *Warp*.

Topics include:

- identifiers
- data objects (constants, variables, signals)
- data types, including pre-defined types, user-definable types, subtypes, and composite types
- operators, including logical, relational, adding, multiplying, miscellaneous, assignment, and association operators
- entities
- architectures, for behavioral data flow and structural descriptions
- packages and libraries

Designs in VHDL are created in what are called entity and architecture pairs. Entities and architectures are discussed in Sections 1.6 Entities and 1.7 Architectures. Sections leading up to this discussion cover VHDL language basics such as identifiers, data objects, data types, operators, and syntax.

1.2 Identifiers

An identifier in VHDL is composed of a sequence of one or more alphabetic, numeric, or underscore characters.

Legal characters for identifiers in VHDL include uppercase letters (A...Z), lowercase letters (a...z), digits (0...9), and the underscore character (_).

The first character in an identifier must be a letter.

The last character in an identifier cannot be an underscore character. In addition, two underscore characters cannot appear consecutively.

Lowercase and uppercase letters are considered identical when used in an identifier; thus, SignalA, signala, and SIGNALA all refer to the same identifier.

Comments in a VHDL description begin with two consecutive hyphens (--), and extend to the end of the line. Comments can appear anywhere within a VHDL description.

VHDL defines a set of reserved words, called keywords, that cannot be used as identifiers.

Examples:

```
-- this is a comment.

-- this is the first line of
-- a three-line comment. Note the repetition
-- of the double hyphens for each line.

entity mydesign is -- comment at the end of a line
```

The following are legal identifiers in VHDL:

```
SignalA
Hen3ry
Output_Enable
C3PO
THX_1138
```

The following are **not** legal identifiers in VHDL:

```
3POC               -- identifier can not start with a digit
_Output_Enable     -- or an underscore character
My__Design         -- or have two consecutive underscores
My_Entity_         -- can't end with an underscore, either
Sig%               -- percent sign is an illegal character
Signal             -- reserved word
```

1.3 Data Objects

A data object holds a value of some specified type. In VHDL, all data objects belong to one of three classes: constants, variables, or signals.

Constant Declaration

constant identifier[,identifier...]:type:=value;

Variable Declaration

variable identifier[,identifier...]:type[:=value];

Signal Declaration

signal identifier[,identifier...]:type [:=value];

An object of class constant can hold a single value of a given type. A constant must be assigned a value upon declaration. This value cannot be changed within the design description.

An object of class variable can also hold a single value of a given type at any point in the design description. A variable, however, can take on many different values within the design description. Values are assigned to a variable by means of a variable assignment statement.

An object of class signal is similar to an object of class variable in *Warp*, with one important difference: signals can hold or pass logic values, while variables cannot. Signals can therefore be synthesized to memory elements or wires.

Variables have no such hardware analogies. Instead, variables are simply used as indexes or value holders to perform computations incidental to modeling components.

Most data objects in VHDL, whether constants, variables, or signals, must be declared before they can be used. Objects can be given a value at declaration time by means of the ":=" operator.

Exceptions to the "always-declare-before-using" rule include:

- The ports of an entity are implicitly declared as signals.
- The generics of an entity are implicitly declared as constants.
- The formal function parameters must be constants or signals, and are implicitly declared by the function declaration. The formal procedure parameters can be constants, variables, or signals, and are implicitly declared by the procedure declaration.

- The indices of a loop or generate statement are implicitly declared when the loop or generate statement begins, and disappear when it ends.

Examples:

constant bus_width:integer := 8;

This example defines an integer constant called bus_width and gives it a value of 8.

variable ctrl_bits:std_logic_vector(7 downto 0);

This example defines an eight-element bit_vector called ctrl_bits.

signal sig1, sig2, sig3:std_logic;

This example defines three signals of type std_logic, named sig1, sig2, and sig3.

1.4 Data Types

A data type is a name that specifies the set of values that a data object can hold and the operations that are permissible on those values.

Warp supports the following pre-defined VHDL types:

- integer
- boolean
- bit
- character
- string
- bit_vector
- std_logic
- std_logic_vector

Warp also gives the user the capability to define subtypes and composite types by modifying these basic types, and to define particular types by combining elements of different types.

Warp's pre-defined types, and *Warp*'s facilities for defining subtypes, composite types, and user-defined types, are all discussed in the following pages.

Note – VHDL is a strongly typed language. Data objects of one type cannot be assigned to data objects of another, and operations are not allowed on data objects of differing types. *Warp* provides functions for converting vectors to integers or integers to vectors and functions for allowing certain operations on differing data types.

1.4.1 Pre-Defined Types

Warp supports the following pre-defined VHDL types: integer, boolean, bit, character, string, bit_vector, std_logic, and std_logic_vector.

Integer

VHDL allows each implementation to specify the range of the integer type differently. However, the range must extend from at least -(2**31-1) to +(2**31-1), or -2147483648 to +2147483647. *Warp* allows data objects of type integer to take on any value in this range.

Boolean

Data objects of this type can take on the values true or false.

BitData

Objects of this type can take on the values 0 or 1.

Character

Data objects of type character can take on values consisting of any of the 128 standard ASCII characters. The non-printable ASCII characters are represented by two or three-character identifiers, as follows: NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS, HT, LF, VT, FF, CC, S0, S1, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN, EM, SUB, ESC, FSP, GSP, RSP, and USP.

String

A string is an array of characters.

Example:

```
variable greeting:string(1 to 13):="Hello, world!";
```

Bit_Vector

A bit_vector is an array of bits in ascending or descending order and provides an easy means to manipulate buses. Bit_vectors can be declared as follows:

```
signal a, b:bit_vector(0 to 7);
signal c, d:bit_vector(7 downto 0);
signal e:bit_vector(0 to 5);
```

Note – bit_vector constants are specified with double quote marks ("), whereas single bit constants are specified with single quote marks (’).

If these signals are subsequently assigned the following values:

```
a <= "00110101";
c <= "00110101";
b <= x"7A";
d <= x"7A";
e <= O"25";
```

then we can compare the individual bits of a and c to discover that a(7) is ’1’, a(6) is ’0’, a(5) is ’1’, a(4) is ’0’,..., a(0) is ’0’ whereas c(7) is ’0’, c(6) is ’0’, c(5) is ’1’, c(4) is ’1’,... c(0) is ’1’. This is because the bits of signal a are in ascending order, and the bits of signal b are in descending order, and the assignment is made simply from the left most index to the right most index.

The prefix of ’X’ or ’x’ denotes a hexadecimal value; a prefix of ’O’ or ’o’ denotes an octal value; a prefix of ’B’ or ’b’ denotes a binary value. If no prefix is included, a value of ’b’ is assumed. Underscore characters may be freely mixed in with the bit_vector value for clarity. Hexadecimal and octal designators should only be used if the hexadecimal or octal value can be directly mapped to the size of the bit_vector. For example, if ’x’ is a bit_vector(0 to 5), then the assignment a <= x"B"; cannot be made because the hexadecimal number ’B’ uses four bits and does not match the size of the bit_vector to which it is being assigned.

String Literals

A value that represents a (one-dimensional) string of characters is called a string literal. String literals are written by enclosing the characters of the string within double quotes ("..."). String literals can be assigned either to objects of type string or to objects of type bit_vector (or other types of vectors whose base type is compatible with the string contents), as long as both objects have been declared with enough elements to contain all the characters of the string:

```
variable err_msg:string(1 to 18);
err_msg := "Fatal error found!";
signal bus_a:bit_vector(7 downto 0);
bus_a<= "10011110";
signal bus_b:std_logic_vector(7 downto 0);
bus_b <= "ZZZZZZZZ" ;
```

std_logic

std_logic is similar to the basic type bit except that it is not defined within the language. The IEEE std_logic_1164 packages defines std_logic as a type which can have values ’U’, ’X’, ’0’, ’1’, ’Z’, ’W’, ’L’, ’H’, or ’-’.

For synthesis purposes, however, only ’0’, ’1’, ’Z’ and ’-’ are supported as valid values. The values ’Z’ and ’-’ also have additional restrictions on how and where they can be used.

The values ’0’ and ’1’ can be used anywhere in the design.

The ’Z’ which represents the high impedance state can only be used in an assignment to a top level output and has to map to a physical pin on the device.

The ’-’ which represents a don’t care can be used in an assignment but cannot be used to compare values of non-constant signals (in if-then-else or case statements). The std_match functions defined in the numeric packages, however, can be used to compare input don’t cares.

std_logic_vector

The std_logic_vector is simply a vector or an array of elements of type std_logic. Its use is very similar to the bit_vector type. The main difference between a bit_vector and the std_logic_vector is the type of the elements of the array.

1.4.2 Enumerated Types

An enumerated type is a type with a user-defined set of possible values.

Enumerated Type Declaration

type name is (value[,value...]);
The order in which the values are listed in an enumeration type's declaration defines the lexical ordering for that type. That is, when relational operators are used to compare two objects of an enumerated type, a given value is always less than another value that appears to its right in the type declaration. The position number of the leftmost value is 0; the position number of other values is one more than that of the value to its left in the type declaration.

Examples:

type arith_op is (add,sub,mul,div);

This example defines an enumerated type named arith_op whose possible values are add, sub, mul, and div.

type states is (state0, state1, state2, state3)

This example defines an enumerated type named states, with four possible values: state0, state1, state2, and state3.

1.4.3 Subtypes

A subtype is a subset of a larger type.

Subtype Declaration

```
subtype type is
    base_type range value {to | downto} value;
```

Subtypes are useful for range checking or for enforcing constraints upon objects of larger types.

Examples:

subtype byte is std_logic_vector(7 downto 0);
subtype single_digit is integer range 0 to 9;

These examples define subtypes called byte and single_digit. Signals or variables that are declared as byte are std_logic_vectors of eight bits in descending order. Signals or variables that are declared as single_digit are integers with possible values consisting of the integers 0 through 9, inclusive.

subtype byte is std_logic_vector(7 downto 0);
type arith_op is (add,sub,mul,div);
subtype add_op is arith_op range add to sub;
subtype mul_op is arith_op range mul to div;

This example first defines an enumerated type called arith_op, with possible values add, sub, mul, and div. It then defines two subtypes: add_op, with possible values add and sub, and mul_op, with possible values mul and div.

1.4.4 Composite Types

A composite type is a type made up of several elements from another type. There are two kinds of composite types: arrays and records.

Array Type Declaration

```
type name is array              ({low to high}|
                                 {high downto low}) of base_type;
```

Record Type Declaration

```
record type is record
      element:element_type
      [;element:element_type...];
end record;
```

An array is a data object consisting of a collection of elements of the same type. Arrays can have one or more dimensions. Individual elements of arrays can be referenced by specifying an index value into the array (see examples). Multiple elements of arrays can be referenced using aggregates.

A record is a data object consisting of a collection of elements of different types. Records in VHDL are analogous to records in Pascal and struct declarations in C. Individual fields of a record can be referenced by using selected names (see examples). Multiple elements of records can be referenced using aggregates.

Examples:

The following are examples of array type declarations:

```
type big_word is array (0 to 63) of std_logic;
      type matrix_type is array (0 to 15, 0 to 31) of std_logic;
      type values_type is array (0 to 127) of integer;
```

Possible object declarations using these types include:

```
      signal word1,word2:big_word;
      signal device_matrix:matrix_type;
      variable current_values:values_type;
```

Some possible ways of assigning values to elements of these objects include:

```
      word1(0)<=’1’; -- assigns value to 0th element in word1
      word1(5)<=; -- assigns value to 5th element in word1
      word2 <= word1; -- makes word2 identical to word1
```

```
word2(63) <= device_matrix(15,31); -- transfers value
-- of element from device_matrix to element of word2
current_values(0) := 0;
current_values(127) := 1000;
```

The following includes an example of a record type declaration:

```
type opcode is (add,sub,mul,div);
type instruction is record
    operator:opcode;
    op1:integer;
    op2:integer;
    end record;
```

Here are two object declarations using this record type declaration:

```
variable inst1, inst2:instruction;
```

Some possible ways of assigning values to elements of these objects include:

```
inst1.opcode := add; -- assigns value to opcode of inst1
inst2.opcode := sub; -- assigns value to opcode of inst2
inst1.op1 := inst2.op2; -- copies op2 of inst2
                                        -- to op1 of inst2
inst2 := inst1; -- makes inst2 identical to inst1
```

1.5 Operators

VHDL provides a number of operators used to construct expressions to compute values. VHDL also uses assignment and association operators.

VHDL's expression operators are divided into five groups. They are (in increasing order of precedence): logical, relational, adding, multiplying, and miscellaneous.

In addition, there are assignment operators that transfer values from one data object to another and association operators that associate one data object with another.

Table 1-1 lists the VHDL operators that *Warp* supports.

Table 1-1 Supported Operators

Logical Operators	**and, or, nand, nor, xor, xnor, not**
Adding Operators	**+, -, &**
Multiplying Operators	*****, **/, mod, rem**
Miscellaneous Operators	**abs,** **
Assignment Operators	**<=, :=**
Association Operator	=>
Shift Operators	**sll, srl, sla, sra, rol, ror**

1.5.1 Logical Operators

The logical operators AND, OR, NAND, NOR, XOR, XNOR, and NOT are defined for predefined types bit and boolean. These operators are also available for the type std_logic.

AND, OR, NAND, and NOR are "short-circuit" operations for bit and boolean types. The right operand is evaluated only if the value of the left operand is not sufficient to determine the result of the operation. For operations AND and NAND, the right operand is evaluated only if the value of the left operand is true. For operations OR and NOR, the right operand is evaluated only if the value of the left operand is false.

Note that there is no differentiation of precedence among the binary boolean operators. Thus, successive boolean operators in an expression must be delimited by parentheses to guarantee error-free parsing and evaluation.

This is a legal example:

a <= b AND c OR d

This is not a legal example:

a <= (b AND c) OR d

1.5.2 Relational Operators

Relational operators include tests for equality, inequality, and ordering of operands.

The operands of each relational operator must be of the same type. The result of each relational operation is of type `boolean`.

The equality operator "=" returns `true` if the two operands are equal, `false` otherwise. The inequality operator "/=" returns `false` if the two operands are equal, `true` otherwise.

Two scalar values of the same type are equal if and only if their values are the same. Two composite values of the same type (e.g., vectors) are equal if and only if for each element of the left operand there is a matching element of the right operand, and the values of matching elements are equal.

The ordering operators are defined for any scalar type and for array types (e.g., vectors). For scalar types, ordering is defined in terms of relative values (e.g., ’0’ is always less than ’1’). For array types, the relation "<" (less than) is defined such that the left operand is less than the right operand if and only if:

- the left operand is a null array and the right operand is a non-null array; otherwise
- both operands are non-null arrays, and one of the following conditions is satisfied:
- the leftmost element of the left operand is less than that of the right; or
- the leftmost element of the left operand is equal to that of the right, and the tail of the left operand is less than that of the right. The tail consists of the remaining elements to the right of the leftmost element and can be null.

The relation "<=" (less than or equal to) for array types is defined to be the inclusive disjunction of the results of the "<" and "=" operators for the same two operands (i.e., it's true if either the "<" or "=" relations are true). The relations ">" (greater than) and ">=" (greater than or equal to) are defined to be the complements of "<=" and "<", respectively, for the same two operands.

1.5.3 Adding Operators

In VHDL, the "+" and "-" operators perform addition and subtraction, respectively. The ’&’ operator concatenates characters, strings, bits or bit/std_logic vectors. All three of these operators have the same precedence, and so are grouped under the category Adding Operators.

The adding operators "+" and "-" are defined for integers and retain their conventional meaning.

These operations are also supported for bit_vectors and other special packages. (See Section 1.8.1, *Predefined Packages*, for more information.)

In *Warp*, concatenation is defined for bits and arrays of bits (bit_vectors). The concatenation operator in *Warp* is "&".

If both operands are bit_vectors, the result of the concatenation is a one-dimensional array whose length is the sum of the lengths of the operands, and whose elements consist of the elements of the left operand (in left-to-right order) followed by the elements of the right operand (in left-to-right order). The left bound of this result is the left bound of the left operand, unless the left operand is a null array, in which case the result of the operation is the right operand. The direction of the result is the direction of the left operand, unless the left operand is a null array, in which case the direction of the result is that of the right operand.

If one operand is a bit_vector and the other is a bit, or if both are bits, the bit operand is replaced by an implicit one-element bit_vector having the bit operand as its only element. The left bound of the implicit bit_vector is 0, and its direction is ascending. This is in most cases an inconsequential fact if the "&" is being used during an assignment to a constrained vector (vector with known dimensions) but may become important if the concatenated vector is being assigned (or passed to a function) to a unconstrained vector.

1.5.4 Multiplying Operators

In VHDL, the "*" and "/" operators perform multiplication and division, respectively. Two other operands of the same precedence include the mod and rem operators. Mod and rem operators return the remainder when one operand is divided by another.

All the multiplication operators are defined for both operands being of the same integer or bit_vector type. The result is also of the same type as the operands.

The rem operation is defined as the following:

A rem B = A-(A/B)*B

where "/" in the above example indicates an integer division. The result has the sign of A and an absolute value less than the absolute value of B.

The mod operation is similar, except that the result has the sign of B. In addition, , for some integer N, the result satisfies the relation:

A mod B = A-B*N

Note – *Warp* predefines the "*" only. There is currently no built-in support for "/", "mod" or the "rem" operators but they can be used if the user supplies the necessary overloading.

1.5.5 Miscellaneous Operators

The two miscellaneous expression operators in VHDL are "abs" and "**".

The "abs" operator, defined for integers, returns the absolute value of its operand.

The "**" operator raises a number to a power of two. It is defined for an integer first operand and a power-of-two second operand. Its result is the same as shifting the bits in the first operand left or right by the amount specified by the second operand.

Note – *Warp* currently supports these operators for Constant integers only.

1.5.6 Assignment Operations

VHDL has two assignment operators: "<=" and ":=". The first is used for signal assignments, the second for variable assignments.

Variable Assignment

variable_name := expression;

Signal Assignment

signal_name <= expression;

Variable assignments can only occur inside a process. Signal assignments can occur anywhere inside an architecture.

Assignments to objects of composite types can be assigned values using aggregates, which is simply a way of specifying more than one value to be assigned to elements of an object with a single assignment statement. Examples of the use of aggregates are shown below.

Examples:

```
type opcode is (add,sub,mul,div);
type instruction is record
    operator:opcode;
    op1:integer;
```

```
        op2:integer;
    variable inst1,inst2:instruction;
    signal vec1, vec2 : bit_vector(0 to 3):

    vec1 <= ('1','0','1','0'); -- aggregate assignment
    vec2 <= vec1; -- another aggregate assignment
    inst1 := (add,5,10); -- aggregate assignment to record
    vec1 <= (0=>'0',others=>'1'); -- assign 0 to 0th bit,
                                            -- set others to 1
```

1.5.7 Association Operations

To instantiate a component in a *Warp* description, the user must specify the connection path(s) between the ports of the component being instantiated and the interface signals of the entity/architecture pair being defined. This is done by means of an association list within a port map or a generic map.

Warp supports both *named* and *positional* association.

In *named* association, the user uses the "=>" association operator to associate a formal (the name of the port in the component being instantiated) with an actual (the name of the signal in the entity being defined). The association operator is considered an "arrow" indicating direction. It's easy to remember which way to make the arrow point: it always points to the actual. For example, in the following instantiation of a predefined D flip-flop,

```
st0: DSRFF port map(
    d     => dat,
    s     => set,
    r     => rst,
    clk   => clk,
    q     => duh);
```

the arrow always points toward the ports of the defined component, the DSRFF in this case. Named association allows the user to associate the signals in any order he desires. In the previous example, the user could have listed the "q => duh" before "d => dat".

In positional association, the association operator is not used. Instead, the user lists the actuals (signals names) in the port map in the same order as the formals of the component being instantiated, without including the formal names at all.

For example, the jkff component is declared as follows:

```
    component jkff port(
        j     : in bit;
        k     : in bit;
        clk   : in bit;
        q     : out bit);
```

```
end component;
```

An association list for an instantiation of this component could use either named association, like this:

```
jk1:jkff port map(j=>j_in,k=>k_in,clk=>clk,q=>q_out);
```

or positional association, like this:

```
jk1:jkff port map(j_in, k_in, clk, q_out);
```

Either form maps signals j_in, k_in, clk, and q_out in the entity being defined to ports j, k, clk, and q, respectively, on the instantiated component.

1.5.8 Vector Operations

Addition, subtraction, multiplication, incrementing, decrementing, shifting inverting, and relational operators for vectors are defined in the predefined packages.

With the appropriate package, included within the user’s VHDL file, the user can gain access to the vector-vector or vector-integer operations. The specific package that is needed depends on the type of vectors that the user is using in the VHDL file. The following table associates a package with the four predefined vector types supported within *Warp*.

Table 1-2 Package to Use with Vector Type

Vector Type	Package
bit_vector	bit_arith
std_logic_vector	std_arith
unsigned (bit)	numeric_bit
signed (bit)	numeric_bit
unsigned (std_logic)	numeric_std
signed (std_logic)	numeric_std

If using bit_vectors, the user needs the following USE clause:

```
use work.bit_arith.all ;
```

If using std_logic_vectors:

```
library ieee ;
```

```
use ieee.std_logic_1164.all ;
use work.std_arith.all ;
```

If using unsigned or signed vectors which are bit based:

```
use work.numeric_bit.all ;
```

If using unsigned or unsigned vectors which are std_logic based:

```
library ieee ;
use ieee.std_logic_1164.all ;
use work.numeric_std.all ;
```

The types unsigned and unsigned are similar to the std_logic_vector or bit_vector type. This is part of an emerging standard (IEEE 1076.3) for performing numeric operations on vectored signals. The numeric_bit package defines unsigned/signed as a vector whose elements are of type bit and the numeric_std package defines the same with elements of type std_logic. This means that you cannot use both types of unsigned/signed within the same VHDL design.

In all of the above packages, the most significant bit (MSB) for a vector is considered to be the left-most bit. This means that in the following two vectors:

```
signal veca : std_logic_vector(3 downto 0) ;
signal vecb : std_logic_vector(0 to 3) ;
```

veca(3) is the MSB for veca and vecb(0) is the MSB for vecb.

All of the above four packages mentioned also provide certain other utility functions which are documented in Section 1.8 Packages.

1.6 Entities

VHDL designs consist of entity and architecture pairs, in which the entity describes the design I/O or interface and the architecture describes the content of the design.Together, entity and architecture pairs can be used as complete design descriptions or as components in a hierarchical design or both.

The syntax for an entity declaration is as follows:

```
ENTITY entity IS PORT(
    [signal][sig-name,...]:[direction] type
    [;signal[sig-name,...]:[direction] type]
    .
    .
    );
END entity-name;
```

The entity declaration specifies a name by which the entity can be referenced in a design architecture. In addition, the entity declaration specifies ports. Ports are a class of signals that define the entity interface. Each port has an associated signal name, mode, and type.

Choices for mode are in (default), out, inout and buffer. Mode in is used to describe ports that are inputs only; out is used to describe ports that are outputs only, with no feedback internal to the associated architecture; inout is used to describe bi-directional ports; buffer is used to describe ports that are outputs of the entity but are also fed back internally.

Two sample entity declarations appear below.

Examples:

```
entity cnt3bit is port(
    q:inout std_logic_vector(0 to 2);
    inc,grst,rst,clk:in std_logic;
    carry:out std_logic);
end cnt3bit;

entity Bus_Arbiter is port(
    Clk,            -- Clock
    DRAM_Refresh_Request,-- Refresh Request
    VIC_Wants_Bus,-- VIC Bus Request
    Sparc_Wants_Bus: IN std_logic;-- Sparc Bus Request
    Refresh_Control,-- DRAM Refresh Control
    VIC_Has_Bus,-- VIC Has Bus
    Sparc_Has_Bus: OUT std_logic);-- Sparc Has Bus
end Bus_Arbiter;
```

The first entity declaration shows the proper declaration for a bidirectional signal (which, in this case, is also a vector), along with several input signals and an output signal.

The second entity declaration shows how comments can be included within an entity declaration to document each signal's use within the entity.

1.7 Architectures

Architectures describe the behavior or structure of associated entities. They can be either, or a combination of the following:

- behavioral descriptions

  These descriptions provide a means to define the "behavior" of a circuit in abstract, "high level" algorithms, or in terms of "low level" boolean equations.

- structural descriptions

  These descriptions define the "structure" of the circuit in terms of components and resemble a net-list that could describe a schematic equivalent of the design. Structural descriptions contain hierarchy in which components are defined at different levels.

The architecture syntax follows:

```
ARCHITECTURE aname OF entity IS
        [type-declarations]
        [signal-declarations]
        [constant-declarations]
BEGIN
        [architecture definition]
END aname;
```

Each architecture has a name and specifies the entity it defines. Types, signals, and constants must be declared before the architecture definition. The architecture defines the concurrent signal assignments, component instantiations, and processes.

Examples:

```
library ieee ;
use work.std_logic_1164.all ;
use work.std_arith.all;
architecture archcounter of counter is
begin
proc1:  process (clk)
 begin
          if (clk'event and clk = '1') then
                   count <= count + 1;
          end if;
 end process proc1;
 x <= '1' when count = "1001" else '0';
end archcounter;
```

Archcounter is an example of a behavioral architecture description of a counter and a signal x that is asserted when count is a particular value. This design is behavioral due to the algorithmic way it is described. The details of such descriptions is covered later.

```
library ieee ;
use work.std_logic_1164.all ;
use work.rtlpkg.all;
architecture archcapture of capture is
 signal c: std_logic;
begin
                c <= a AND b;
                d1: dff port map(c, clk, x);
end archcapture;
```

Archcapture is the name of an architectural description that is both structural and behavioral in nature. It is considered structural because of the component instantiation, and it is considered behavioral because of the boolean equation. VHDL provides the flexibility to combine behavioral and structural architecture descriptions.

1.7.1 Behavioral Descriptions

Behavioral design descriptions consist of two types of statements:

- Concurrent statements which define concurrent signal assignments by way of association operators.
- Sequential statements within a process which enable an algorithmic way of describing a circuit's behavior. Sequential statements enable signal assignments to be based on relational and conditional logic.

These types of statements, as well as structural descriptions, may be combined in any architecture description.

Concurrent Statements

Concurrent statements are used outside of processes to implement boolean equations, when… else constructs, signal assignments, or generate schemes. For example:

```
u <= a;
v <= u;
w <= a XOR b;
x <= (a AND s) or (b AND NOT(s));
y <= ('1' when (a='0' and b = '1') else '0';
z <= A when (count = "0010") else b;
```

Signal u is assigned the value of signal a and is its equivalent. Likewise, v is equivalent to both signals u and a. The signal assignment order does not matter because they are outside of a process and are concurrent. The next two statements implement boolean equations, while the last statements implement when… else constructs. The assignment for signal y may be read as "y gets '1' when a is zero and b is one, otherwise y gets '0'." Likewise, "z gets a when count is "0010" otherwise z gets b."

Sequential Statements

Sequential statements must be written within a process. These statements describe signal assignments in an algorithmic fashion. The statement order is important, as statements in a process are evaluated sequentially. For example, in the process

```
proc1: process (x)
    begin
        a <= '0';
        if x = "1011" then
            a <= '1';
        end if;
    end process proc1;
```

signal a is first assigned '0'. Later in the process, if x is found to be equivalent to "1011" then signal a is assigned the value ’1’.

Final signal assignments occur at the end of the process. In other words, the VHDL compiler evaluates the code sequentially before determining the equations to be synthesized, whereas the compiler synthesizes equations for concurrent statements upon encountering them. A process taken as a whole is a concurrent statement.

The Process

In most cases, a process has a sensitivity list: a list of signals in parentheses immediately following the key word "process". Signals assigned within a process can only change value if one of the signals in the sensitivity list transitions. If the sensitivity list is omitted, then the compiler infers that signal assignments are sensitive to changes in any signal.

The user may find it helpful to think of processes in terms of simulation (VHDL is also used for simulation) in which a process is either active or inactive. A process becomes active only when a signal in the sensitivity list transitions. In the following process

```
proc1: process (rst, clk)
    begin
        if rst = ’1’ then
            q <= ’0’;
        elsif (clk’event and clk=’1’) then
            q <= d;
        end if;
    end process;
```

only transitions in rst and clk cause the process to become active. If either clk or rst transition, then the process becomes active, and the first condition is checked (if rst = ’1’). In the case that rst = ’1’ q will be assigned ’0’, otherwise the second condition is checked (if clk event and clk = ’1’). This condition looks for the rising edge of a clock. All signals within this portion of the process are sensitive to this rising edge clock, and the compiler infers a register for these signals. This process creates a D flip-flop with d as its input, q as its output, clk as the clock, and rst as an asynchronous reset.

1.7.2 Structural Descriptions

Structural descriptions are net-lists that allow the user to instantiate components in hierarchical designs. A port map is part of every instantiation and indicates how the ports of the component are connected. Structural descriptions can be combined with behavioral descriptions, as in the following example:

```
architecture archmixed of mixed is
begin
--instantiations
cntl1: motor port map(clk, ld, en, c1, chg1, start1, stop1);
```

```
cntl2: motor port map(clk, ld, en, c2, chg2, start2, stop2);
safety: mot_check port map(status, c1, c2);
--concurrent statement
en <= '1' when (status='1' and status = '1') else '0';
-- concurrent process with sequential statements
ok:  process (clk)
     begin
          if (clk'event and clk='1') then
               status <= update;
          end if;
     end process ok;
end archmixed;
```

This example shows that two motor components and one mot_check component are instantiated. The port maps are associated with inputs and outputs of the motor and mot_check components by way of positional association. Signal en is assigned by a concurrent statement, and signal status is assigned by a process that registers a signal using the common clock clk.

1.7.3 Design Methodologies

Designers can choose from multiple methods of describing designs in VHDL, depending on coding preferences. This section will discuss how to implement combinatorial logic, registered logic, counters, and state machines. The discussion of state machines will cover multiple implementations and the design and synthesis trade-offs for those implementations. Section 1.10 Additional Design Examples contains further design examples. Most of the design examples in this section can be found in the directory *c:\warp\examples*.

Combinatorial Logic

Following are examples of a four-bit comparator implemented in four different ways, all yielding the same result. In all examples, the entity is the same:

```
library ieee ;
use ieee.std_logic_1164.all ;                     -- Defines std_logic
entity compare is port(
     a, b:   in std_logic_vector(0 to 3);
     aeqb:   out std_logic);
end compare;
```

The entity declaration specifies that the design has three ports: two input ports (a, b), and one output port (aeqb). The input ports are of type std_logic_vector and the output port is of type std_logic.

Using a process, the comparator can be implemented as follows:

```
use work.std_arith.all ;
architecture archcompare of compare is
begin
comp:   process (a, b)
        begin
                if a = b then
                        aeqb <= '1';
                else
                        aeqb <= '0';
                end if;
        end process comp;
end archcompare;
```

The design behavior is given in the architecture section. The architecture description consists of the process "comp". The process includes the sensitivity list (a,b) so that the process becomes active each time there is a change in one of these signals. The process permits the use of an algorithm to assert aeqb when a equals b. The std_arith package contains a tuned implementation for the "=" operator.

With one concurrent statement, making use of the when...else construct, the same comparator can be described like this:

```
use work.std_arith.all ;
architecture archcompare of compare is
begin
        aeqb <= '1' when (a = b) else '0';
end;
```

In this example, the process in the previous example has been replaced by a concurrent signal assignment for aeqb.
Using boolean equations, the comparator looks like this:

```
architecture archcompare of compare is
begin
       aeqb <= NOT(
       (a(0) XOR b(0)) OR
       (a(1) XOR b(1)) OR
       (a(2) XOR b(2)) OR
       (a(3) XOR b(3)));
end;
```

In this example, a boolean equation replaces the when... else construct.

Finally, a structural design which implements a net list of XOR gates, a 4-input OR gate, and an INV gate looks like this:

```
use work.lpmpkg.all;
```

```
architecture archcompare of compare is
begin
    c0: Mcompare
        generic map(
            lpm_width => aeqb'length,                   -- Evaluates to 4
            lpm_representation => lpm_unsigned,
            lpm_hint => speed)
        port map(
            dataa => a,         datab => b,
            alb => open,        aeb => aeqb,            agb => open,
            aleb => open,       aneb => open,           ageb => open);
end;
```

In this example, the compare architecture is described by instantiating gates much the same as one would by placing gates in a schematic diagram. The *Mcompare* component used in this architecture is the same as those available in the *Warp* LPM (Library of Parameterized Elements) library. The port map lists are associated with the inputs and outputs of the gates through named association for readability.

Many other functions or components can be implemented in multiple ways. Here is one last combinatorial example: a four-bit wide four-to-one multiplexer. In all versions, the entity is the same:

```
library ieee ;
use ieee.std_logic_1164.all ;               -- Defines std_logic
entity mux is port(
    a, b, c, d:     in std_logic_vector(3 downto 0);
    s:              in std_logic_vector(1 downto 0);
    x:              out std_logic_vector(3 downto 0));
end mux;
```

Using a process, the architecture looks like this:

```
architecture archmux of mux is
begin
mux4_1: process (a, b, c, d)
    begin
        if s = "00" then
            x <= a;
        elsif s = "01" then
            x <= b;
        elsif s = "10" then
            x <= c;
        else
            x <= d;
        end if;
    end process mux4_1;
end archmux;
```

Using a concurrent statement with a when… else construct, the architecture can be written as the following:

```
architecture archmux of mux is
begin
     x <= a when (s = "00") else
        b when (s = "01") else
        c when (s = "10") else
        d;
end archmux;
```

Using boolean equations, the architecture can be written as follows:

```
architecture archmux of mux is
begin
    x(3) <=      (a(3) and not(s(1)) and not(s(0)))
       OR  (b(3) and not(s(1)) and s(0))
       OR  (c(3) and s(1) and not(s(0)))
       OR  (d(3) and s(1) and s(0));
    x(2) <=      (a(2) and not(s(1)) and not(s(0)))
       OR  (b(2) and not(s(1)) and s(0))
       OR  (c(2) and s(1) and not(s(0)))
       OR  (d(2) and s(1) and s(0));
    x(1) <=      (a(1) and not(s(1)) and not(s(0)))
       OR  (b(1) and not(s(1)) and s(0))
       OR  (c(1) and s(1) and not(s(0)))
       OR  (d(1) and s(1) and s(0));
    x(0) <=      (a(0) and not(s(1)) and not(s(0)))
       OR  (b(0) and not(s(1)) and s(0))
       OR  (c(0) and s(1) and not(s(0)))
       OR  (d(0) and s(1) and s(0));
end archmux;
```

A structural approach can be written like this:

```
use work.lpmpkg.all ;
architecture archmux of mux is
    signal tmpBus : std_logic_vector(
            ((2**s'length * a'length) - 1) downto 0) ;
begin
    tmpBus <= d & c & b & a ;                    -- Collect all inputs
    mux_array: Mmux
        generic map(
            lpm_width => a'length,               -- Width of each input
            lpm_size => (2**s'length),           -- Number of inputs
            lpm_widths => s'length,              -- Number of selectors
```

```
                lpm_hint => speed)
            port map(
                data => tmpBus,
                sel => s,
                result => x);
end archmux;
```

This design makes use of the multiplexer in the *Warp* library. Of course, the user could build up his own multiplexers and instantiate them instead.

Registered Logic

There are two methods for implementing registered logic: instantiating a register (or other component with registers) or using a process that is sensitive to a clock edge. For example, if the user wanted to use a D register and a 4-bit counter, he could simply instantiate these components after including the appropriate packages:

```
use work.rtlpkg.all;
use work.lpmpkg.all;
...
d1: dsrff port map(d, s, r, clk, q);            -- Defined in rtlpkg
c1: Mcounter                                    -- Defined in lpmpkg
    generic map (4)
    port map(data, clk, one, one, one, count,
            zero, rst, zero, zero, zero, zero
            zero, zero, open) ;
```

Another method of using registered elements is to include a process that is sensitive to a clock edge or that waits for a clock edge. In processes that are sensitive to clock edges or that wait for clock edges, the compiler infers a register for the signals defined within that process. Four basic templates are supported; each is described below.

```
process_label: process
    begin
        wait until clk = ’1’;
        .    ..
    end process;
```

This process does not have a sensitivity list. Instead it begins with a wait statement. The process will become active when clk transitions to a one (clk—or whatever identifier you give to your clock—can also wait for zero for devices that support such clocking schemes). All signal assignments within such a process will be registered, as these signals only change values on clock edges and retain their values between clock edges.

```
my_proc: process (clk)
    begin
        if (clk’event and clk =’1’) then
```

```
                ...
            end if;
end process;
```

This process is sensitive only to changes in the clock, as the sensitivity list indicates. The first statement within the process looks for a transition from zero to one in signal clk. All signals that are assigned within this process are also registered because the assignments only occur on rising clock edges, and the signals retain their values between rising clock edges.

```
your_proc: process (rst, clk)
      begin
            if rst = '1' then
                  ...
            elsif (clk'event and clk='1') then
                  ...
            end if;
      end process;
```

This process is sensitive to changes in the clock and signal rst, as the sensitivity list indicates. This process is intended to support signals that must be registered and have an asynchronous set and/or reset. The first statement within the process checks to see if rst has been asserted. Signals that are assigned in this portion of the template are assumed to be registered with rst assigned as either the asynchronous reset or set of the register, as appropriate. If rst has not been asserted, then the remainder of this process works as does the previously described process.

```
proc1: process (rst, pst, clk)
      begin
            if rst = '1' then
                  ...
            elsif pst = '1' then
                  ...
            elsif (clk'event and clk='1') then
                  ...
            end if;
      end process;
```

This process is sensitive to changes in the clock and signals rst and pst, as the sensitivity list indicates. This process is intended to support signals that must be registered and have an asynchronous set and reset. The first statement within the process checks to see if rst has been asserted. Signals that are assigned in this portion of the template are assumed to be registered with rst used as either the asynchronous reset or set of the register, as appropriate. The second condition assigns pst as the asynchronous reset or set of the register, as appropriate. If rst and pst have not been asserted, then the remainder of this process works as does the previous process.

To register 32-bits with an asynchronous reset, the user could write the following code:

```
regs32: process (r, clk2)
    begin
        if (r = ’1’) then
            q <= x"ABC123DE";
        elsif (clk2’event and clk2=’1’) then
            q <= d;
        end if;
    end process;
```

Assuming that q and d are declared as 32-bit signals or ports, then this code example implements 32 registers with d(i) as the input, q(i) as the output, clk2 as the clock, and r as the asynchronous reset for some of the registers and r as the asynchronous preset for the others. This is because resetting the q to the value x"ABC123DE" will cause some registers to go high and other registers to go low when r is asserted.

Counters and state machines designed with processes are described in more detail in the following discussions.

Counters

This is a 4-bit loadable counter:

```
library ieee;
use ieee.std_logic_1164.all ;
use work.std_arith.all ;

entity counter is port(
    clk, load:      in std_logic;
    data:           in std_logic_vector(3 downto 0);
    count:          buffer std_logic_vector(3 downto 0));
end counter;
architecture archcounter of counter is
begin
upcount: process (clk)
    begin
        if (clk’event and clk= ’1’) then
            if load = ’1’ then
                count <= data;
            else
                count <= count + 1;
            end if;
        end if;
    end process upcount;
end archcounter;
```

The use work.std_arith.all; statement is included to make the integer/std_logic_vector math package visible to this design. The integer math package provides an addition function for adding integers to a std_logic_vector. The native VHDL addition operator applies only to integers. The architecture description is behavioral. In this design, the counter counts up or synchronously loads depending on the load control input. The counter is described by the process "upcount". The statement if (clk'event AND clk = '1') then... specifies that operation of the counter takes place on the rising edge of the signal clk. The subsequent if statement describes the loading and counting operation.

In this description, the if (clk'event AND clk = '1') then... statement (and its associated end if) could have been replaced by the statement wait until clk = '1';.

The following is a 4-bit loadable counter with synchronous reset:

```
library ieee;
use ieee.std_logic_1164.all ;

entity counter is port(
    clk, reset, load:           in std_logic;
    data:                       in std_logic_vector(3 downto 0);
    count:                      buffer std_logic_vector(3 downto 0));
end counter;


use work.std_arith.all;
architecture archcounter of counter is
begin
upcount: process (clk)
    begin
        if (clk'event and clk= '1') then
            if reset = '1' then
                count <= "0000";
            elsif load = '1' then
                count <= data;
            else
                count <= count + 1;
            end if;
        end if;
    end process upcount;
end archcounter;
```

In this design, the counter counts up, synchronously resets depending on the reset input, or synchronously loads depending on the load control input. The counter is described by the process "upcount." That the statement if (clk'event AND clk = '1') then... appears first specifies that all operations of the counter take place on the rising edge of the signal, clk. The subsequent if statement describes the synchronous reset operation; the counter is synchronously reset on the rising edge of clk. The remaining operations (load and count) are described in elsif or else clauses in this same if statement, therefore the reset takes precedence over loading or counting. If reset is not '1', then the operation of the counter depends upon the load signal. This operation is then identical to the counter in the previous example.

The following is a 4-bit loadable counter with synchronous enable and asynchronous reset:

```
library ieee;
use ieee.std_logic_1164.all ;

entity counter is port(
    clk, reset, load,counten:          in std_logic;
    data:                      in std_logic_vector(3 downto 0);
    count:                     buffer std_logic_vector(3 downto 0));
end counter;


use work.std_arith.all;
architecture archcounter of counter is
begin
upcount: process (clk, reset)
    begin
        if reset = ’1’ then
            count <= "0000";
        elsif (clk’event and clk= ’1’) then
            if load = ’1’ then
                count <= data;
            elsif counten = ’1’ then
                count <= count + 1;
            end if;
        end if;
    end process upcount;
end archcounter;
```

In this design, the counter counts up, resets depending on the reset input, or synchronously loads depending on the load control input. This counter is similar to the one in the previous example except that the reset is asynchronous. The sensitivity list for the process contains both clk and reset. This causes the process to be executed at any change in these two signals.

The first if statement, if reset = '1' then..., states that this counter will assume a value of "0000" whenever reset is ’1’. This will occur when the process is activated by a change in the signal reset. The elsif clause that is part of this if statement, elsif (clk'event AND clk = '1') then..., specifies that the subsequent statements within the if are performed synchronously (clk'event) on the rising edge (clk = '1') of the signal clk (providing that the previous if/elsif clauses were not satisfied). The synchronous operation of this process is similar to the previous example, with the exception of the counten signal enabling the counter. If counten is not asserted, then count retains its previous value.

The following is a 4-bit loadable counter with synchronous enable, asynchronous reset and preset.

```
library ieee;
use ieee.std_logic_1164.all ;

entity counter is port(
    clk, rst, pst, load,counten:                in std_logic;
    data:                   in std_logic_vector(3 downto 0);
    count:                  buffer std_logic_vector(3 downto 0));
end counter;
use work.std_arith.all;
architecture archcounter of counter is
begin
upcount: process (clk, rst, pst)
    begin
        if rst = ’1’ then
            count <= "0000";
        elsif pst = ’1’ then
            count <= "1111";
        elsif (clk’event and clk= ’1’) then
            if load = ’1’ then
                count <= data;
            elsif counten = ’1’ then
                count <= count + 1;
            end if;
        end if;
    end process upcount;
end archcounter;
```

In this design, the counter counts up, resets depending on the reset input, presets depending upon the pst signal, or synchronously loads depending on the load control input. This counter is similar to the previous example except that a preset control has been added (pst). The sensitivity list for this process contains clk, pst, and rst. This causes the process to be executed at any change in these three signals.

The first if statement if rst = '1' then. implies that this counter will assume a value of "0000" whenever rst is ’1’. This will occur when the process is activated by a change in the signal rst. The first elsif clause that is part of this if statement, elsif pst = '1' then, implies that this counter will assume a value of "1111" whenever pst is ’1’ and rst is ’0’. This will occur when the process is activated by a change in the signal pst and rst is not ’1’.

The second elsif clause that is part of this if statement, elsif (clk'event AND clk = '1') then, implies that the subsequent statements within the if are performed synchronously (clk'event) and on the rising edge (clk = ’1’) of the signal clk providing that the previous if / elsif clauses were not satisfied. In this regard the operation is identical to the counter in the previous example.

The following is an 8-bit loadable counter. The data is loaded by disabling the three-state output, and using the same I/O pins to load.

```
library ieee ;
use ieee.std_logic_1164.all ;
use work.std_arith.all ;
entity ldcnt is port (
    clk, ld, oe:            in std_logic;
    count_io:               inout std_logic_vector(7 downto 0));
end ldcnt;
architecture archldcnt of ldcnt is
    signal count, data:std_logic_vector(7 downto 0);
begin
counter: process (clk)
    begin
        if (clk’event and clk=’1’) then
            if (ld = ’1’) then
                count <= data;
            else
                count <= count + 1;
            end if;
        end if;
    end process counter;
    count_io <= count when (oe = ’1’) else "ZZZZZZZZ" ;
    data <= count_io ;
end archldcnt;
```

This design performs a synchronous counter that can be loaded. The load occurs by disabling the output pins. This allows a signal to be driven from off chip to load the counter. The three-state for I/O pins is accomplished with the use of an oe signal which specifies that if oe is high, the output of the counter is driven onto the I/O pins. Otherwise, the pin should be driven externally with data to be loaded into the counter. The signal count_io is assigned to the signal data for readability purposes only and describes the intention of the design. The signal data can be completely replaced with the count_io signal, and wherever count_io appears on the right hand side of an equation, it essentially is referring to the feedback from the three state buffer from within the I/O pad.

Conceptually, the above VHDL implements the following circuit:

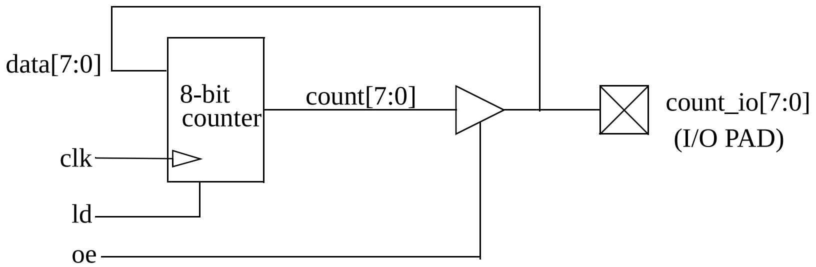

Figure 1-1 8-bit counter using IOPAD for input and output

State Machines

VHDL provides constructs that are well-suited for coding state machines. VHDL also provides multiple ways to describe state machines. This section will describe some coding implementations and how the implementation affects synthesis (the way in which the design description is realized in terms of logic and the architectural resources of the target device).

The implementation that is chosen during coding may depend on which considerations are important: fast time-to-market or squeezing all the possible capacity and performance out of a device. Often times, however, choosing one coding style over another will not result in much difference and will meet performance and capacity requirements while achieving fast time-to-market.

This discussion will include Moore and Mealy state machines, discussing Moore machines first. Moore machines are characterized by the outputs changing only with a change in state. Moore machines can be implemented in multiple ways:

- Outputs are decoded from state bits combinatorially.
- Outputs are decoded in parallel using output registers.
- Outputs are encoded within the state bits. A state encoding is chosen such that a set of the state bits are the required outputs for the given states.
- One-hot encoded. One register is asserted "hot" per state. This encoding scheme often reduces the amount of logic required to transition to the next state at the expense of more registers.
- Truth Tables. A truth table maps the current state and inputs to a next state and outputs.

In the following examples, the same state machine is implemented five different ways as a Moore machine in order to illustrate the design and synthesis issues. Figure 1-2 shows the state diagram.

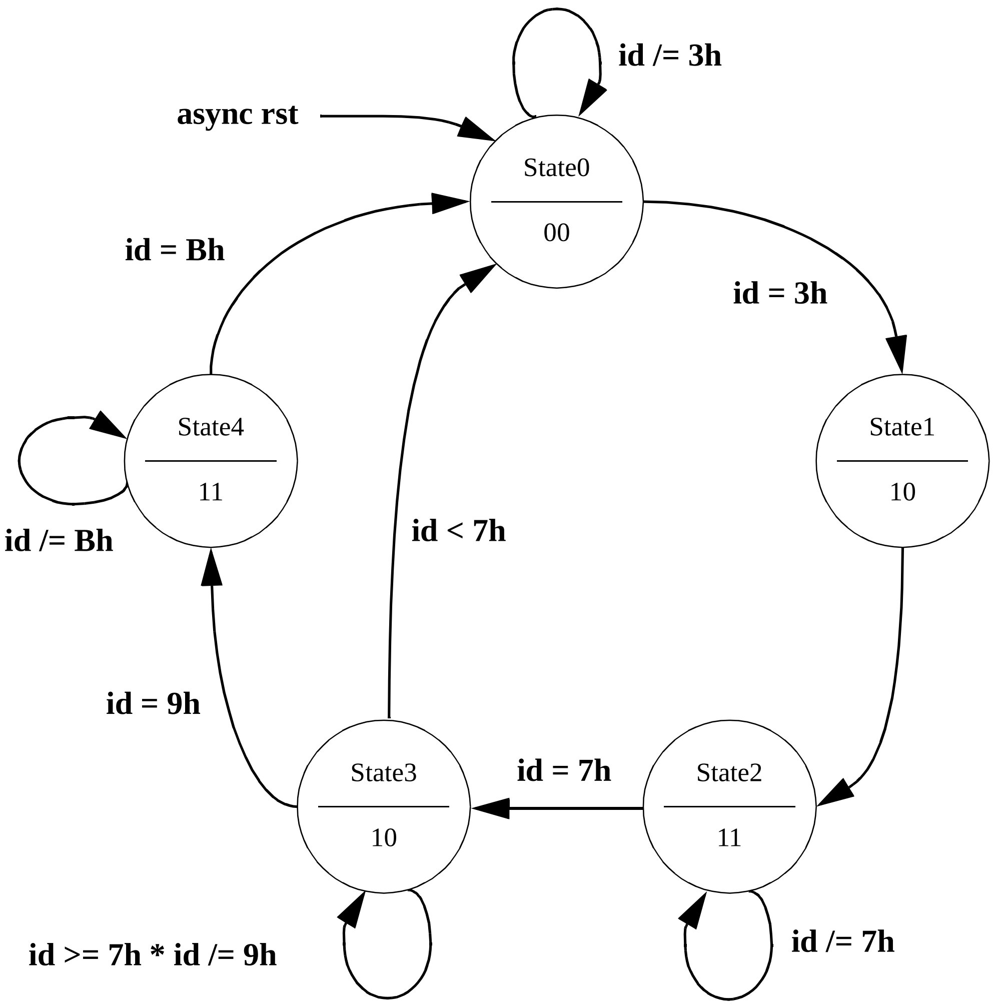

Figure 1-2 Moore State machine

Outputs decoded combinatorially

Figure 1-3 shows a block diagram of an implementation in which the state machine outputs are decoded combinatorially. The code follows:

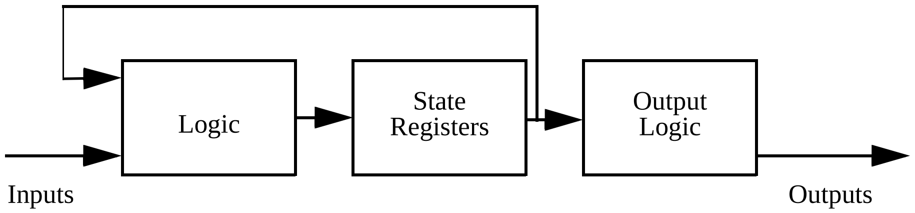

Figure 1-3 Outputs Decoded Combinatorially

```
library ieee ;
use ieee.std_logic_1164.all ;
entity moore1 is port(
        clk, rst:in std_logic;
        id:         in std_logic_vector(3 downto 0);
        y:          out std_logic_vector(1 downto 0));
end moore1;

architecture archmoore1 of moore1 is
        type states is (state0, state1, state2, state3, state4);
        signal state: states;
begin
moore: process (clk, rst)
        begin
                if rst='1' then
                        state <= state0;
                elsif (clk'event and clk='1') then
                        case state is
                        when state0 =>
                            if id = x"3" then
                                state <= state1;
                            else
                                state <= state0;
                            end if;
                        when state1 =>
                            state <= state2;
                        when state2 =>
                            if id = x"7" then
                                state <= state3;
                            else
                                state <= state2;
                            end if;
                        when state3 =>
                            if id < x"7" then
                                state <= state0;
                            elsif id = x"9" then
                                state <= state4;
                            else
                                state <= state3;
                            end if;
                        when state4 =>
                            if id = x"b" then
                                state <= state0;
                            else
                                state <= state4;
                            end if;
                    end case;
                end if;
            end process;

--assign state outputs;
y <= "00" when (state=state0) else
   "10" when (state=state1 or state=state3) else
   "11";
end archmoore1;
```

The architecture description begins with a type declaration, called an enumerated type, for states which defines five states labeled state0 through state4. A signal, state, is then declared to be of type states. This means that the signal called state can take on values of state0, state1, state2, state3, or state4.

The state machine itself is described within a process. The first condition of this process defines the asynchronous reset condition which puts the state machine in state0 whenever the signal rst is a ’1’. If the rst signal is not a ’1’ and the clock transitions to a ’1’-- elsif (clk'event and clk='1') --then the state machine algorithm is sequenced. The design can be rising edge triggered, as it is in this example, or falling edge triggered by specifying clk=’0’.

On a rising edge of the clock, the case statement (which contains all of the state transitions for the Moore machine) is evaluated. The when statements define the state transitions which are based on the input ID. For example, in the case when the current state is state0, the state machine will transition to state1 if id=x"3", otherwise the state machine will remain in state0. In a concurrent statement outside of the process, the output vector y is assigned a value based on the current state.

This implementation demonstrates the algorithmic and intuitive fashion which VHDL permits in the description of state machines. Simple case... when statements enable the user to define the states and their transitions. There are two design and synthesis issues with this implementation which some designers may wish to consider: clock-to-out times for the combinatorially decoded state machine outputs and an alternative state encoding to use minimal product terms.

The clock-to-out times for the state machine outputs are determined by the time it takes for the state bits to be combinatorially decoded. For designs that require minimal clock-to-out times, an implementation similar to the one above can be used with a design modification: a second process could register the outputs after combinatorial decode. This would introduce a one clock-cycle latency, however. If this latency is not acceptable, then the user will need to choose from the second implementation (outputs decoded in parallel registers) or the third implementation (outputs encoded within state bits).

For designs in which the number product terms must be minimized, the user can implement a design similar to the on described above, with one exception: rather than using the enumerated encoding, the user will want to implement his own encoding scheme. The third implementation shows how to do this.

Outputs Decoded in Parallel Output Registers

Figure 1-4 shows a block diagram of an implementation using output registers. The state machine outputs are determined at the same time as the next state. The code follows.

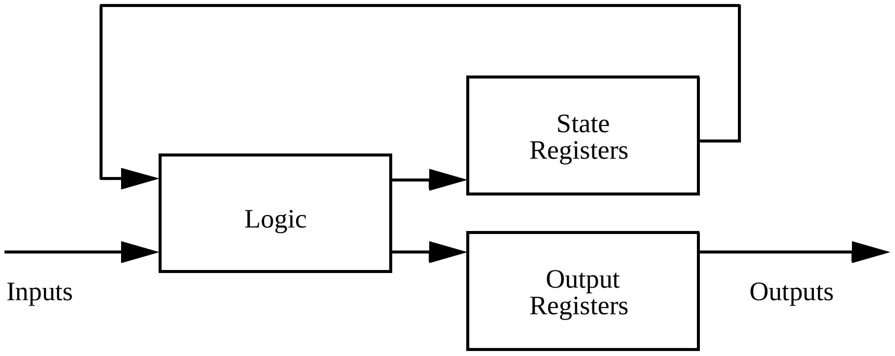

Figure 1-4 Outputs decoded in parallel

```
library ieee ;
use ieee.std_logic_1164.all ;
entity moore2 is port(
      clk, rst:in std_logic;
      id:        in std_logic_vector(3 downto 0);
      y:         out std_logic_vector(1 downto 0));
end moore2;

architecture archmoore2 of moore2 is
      type states is (state0, state1, state2, state3, state4);
      signal state: states;
begin
moore: process (clk, rst)
      begin
            if rst='1' then
                  state <= state0;
                  y <= "00";
            elsif (clk'event and clk='1') then
                  case state is
```

```
                    when state0 =>
                        if id = x"3" then
                            state <= state1;
                            y <= "10";
                        else
                            state <= state0;
                            y <= "00";
                        end if;
                    when state1 =>
                        state <= state2;
                        y <= "11";
                    when state2 =>
                        if id = x"7" then
                            state <= state3;
                            y <= "10";
                        else
                            state <= state2;
                            y <= "11";
                        end if;
                    when state3 =>
                        if id < x"7" then
                            state <= state0;
                            y <= "00";
                        elsif id = x"9" then
                            state <= state4;
                            y <= "11";
                        else
                            state <= state3;
                            y <= "10";
                        end if;
                    when state4 =>
                        if id = x"b" then
                            state <= state0;
                            y <= "00";
                        else
                            state <= state4;
                            y <= "11";
                        end if;
                end case;
            end if;
        end process;
end archmoore2;
```

This implementation requires that the user specify--in addition to the state transitions--the state machine outputs for every state and every input condition because the outputs must be determined in parallel with the next state.

Assigning the state machine outputs in the synchronous portion of the process causes the compiler to infer registers for the output bits. Having output registers rather than decoding the outputs combinatorially results in a smaller clock-to-out time. This implementation has one design/synthesis issue which some may wish to consider: while this implementation achieves a better clock-to-out time for the state machine outputs (as compared to the first implementation), it uses more registers (and possibly more product terms) than the first implementation. The next implementation (outputs encoded within state bits) achieves the fastest clock-to-out times while at the same time using the fewest total number of macrocells in a PLD/CPLD.

Outputs Encoded Within State Bits

Table 1-3 and Figure 1-5 show the state encoding table and a block diagram of an implementation in which the outputs are encoded within the state registers--the two least significant state bits are the outputs. No decoding is required for the outputs, so the output signals can be directed from the state registers to output pins. The code follows:

Table 1-3 Outputs Encoded Within State Registers

State	Output	State Encoding
s0	00	000
s1	10	010
s2	11	011
s3	10	110
s4	11	111

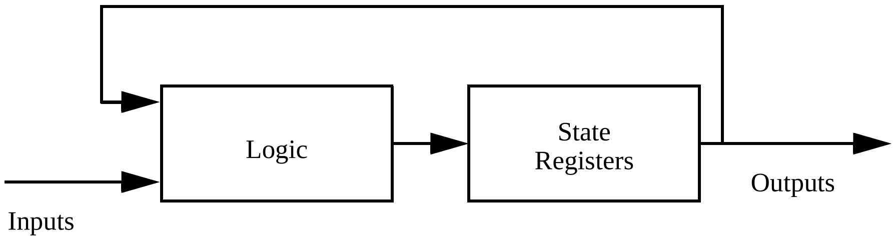

Figure 1-5 Outputs Encoded Within State Bits

```
library ieee ;
use ieee.std_logic_1164.all ;
entity moore1 is port(
    clk, rst:in std_logic;
    id:      in std_logic_vector(3 downto 0);
    y:       out std_logic_vector(1 downto 0));
end moore1;

architecture archmoore1 of moore1 is
    signal state: std_logic_vector(2 downto 0);
-- State assignment is such that 2 LSBs are outputs
constant state0: std_logic_vector(2 downto 0) := "000";
constant state1: std_logic_vector(2 downto 0) := "010";
constant state2: std_logic_vector(2 downto 0) := "011";
constant state3: std_logic_vector(2 downto 0) := "110";
constant state4: std_logic_vector(2 downto 0) := "111";
begin
moore: process (clk, rst)
    begin
        if rst='1' then
            state <= state0;
        elsif (clk'event and clk='1') then
            case state is
                when state0 =>
                    if id = x"3" then
                        state <= state1;
                    else
                        state <= state0;
                    end if;
                when state1 =>
                    state <= state2;
                when state2 =>
                    if id = x"7" then
                        state <= state3;
                    else
                        state <= state2;
                    end if;
                when state3 =>
                    if id < x"7" then
                        state <= state0;
                    elsif id = x"9" then
                        state <= state4;
                    else
                        state <= state3;
                    end if;
                when state4 =>
                    if id = x"b" then
                        state <= state0;
                    else
                        state <= state4;
                    end if;
                when others =>
                    state <= "---";
            end case;
        end if;
    end process;

--assign state outputs (equal to state bits)
y <= state(1 downto 0);
end archmoore1;
```

A state encoding was chosen for this design so that the last two bits were equivalent to the state machine outputs for that state. By using constants, the state machine could be encoded and the transitions specified as in the first implementation. The output was specified in a concurrent statement. This statement shows that the outputs are a set of the state bits. One synthesis issue is highlighted in this example: the use of "when others =>".

"When others" is used when not all possible combinations of a bit sequence have been specified in other when clauses. In this, the states "001," "100," and "101" are not defined, and no transitions are specified for these states. If when others is not used, then next state logic must be synthesized, assuming that if the machine gets in one of these states, then it will remain in that state. This has the effect of utilizing more logic (product terms in the case of a PLD/CPLD). Supplying a simple "when others" is a quick solution to this design issue.

One-Hot-One State Machines

In a one-hot-one state machine, there is one register for each state. Only one register is asserted, or "hot," at a time, corresponding to one distinct state. Figure 1-6 shows three states of a state machine and how one of the state bits would be implemented. This implementation demonstrates that the next state logic is quite simple. The trade-off is the number of registers that is required. For example, a state machine with eight states could be coded in three registers. The equivalent one-hot coded state machine would require eight registers. The code follows.

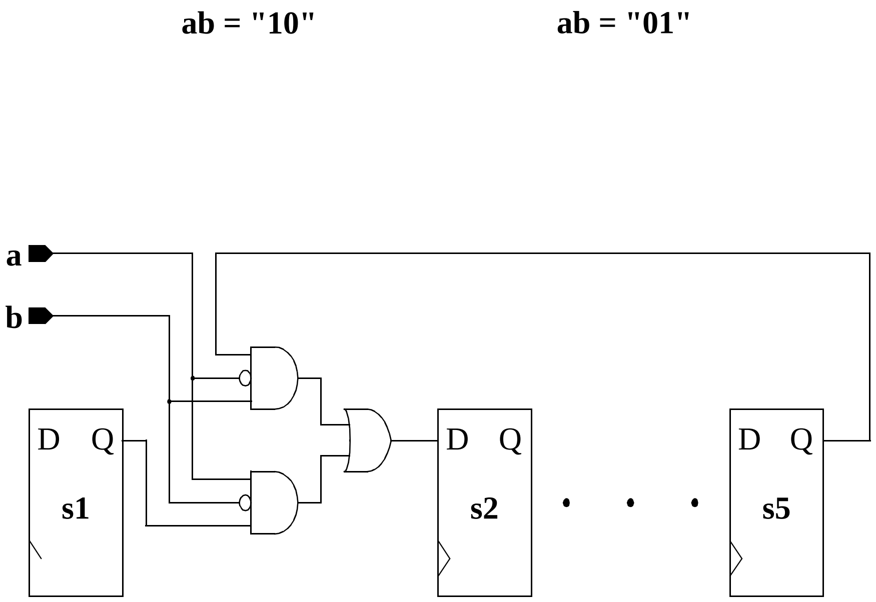

Figure 1-6 Implementation of one-hot state machine bits

```
library ieee ;
use ieee.std_logic_1164.all ;
entity one_hot is port(
        clk, rst:in std_logic;
        id:         in std_logic_vector(3 downto 0);
        y:          out std_logic_vector(1 downto 0));
end one_hot;
```

```
architecture archone_hot of one_hot is
    type states is (state0, state1, state2, state3, state4);
    attribute state_encoding of states:type is one_hot_one;
    signal state: states;
begin
machine: process (clk, rst)
    begin
        if rst='1' then
            state <= state0;
        elsif (clk'event and clk='1') then
            case state is
                when state0 =>
                    if id = x"3" then
                        state <= state1;
                    else
                        state <= state0;
                    end if;
                when state1 =>
                    state <= state2;
                when state2 =>
                    if id = x"7" then
                        state <= state3;
                    else
                        state <= state2;
                    end if;
                when state3 =>
                    if id < x"7" then
                        state <= state0;
                    elsif id = x"9" then
                        state <= state4;
                    else
                        state <= state3;
                    end if;
                when state4 =>
                    if id = x"b" then
                        state <= state0;
                    else
                        state <= state4;
                    end if;
            end case;
        end if;
    end process;
```

```
--assign state outputs;
y <= "00" when (state=state0) else
     "10" when (state=state1 or state=state3) else
     "11";
end archone_hot;
```

This implementation is similar to the first implementation, the only difference being the additional attribute which causes the state encoding to use one register for each state.

State Transition Tables

The final Moore implementation of this state machine uses a truth table. The state transition table can be found in the VHDL code. The code follows.

```
library ieee ;
use ieee.std_logic_1164.all ;
entity ttf_fsm is port(
        clk, rst:in std_logic;
        id:         in std_logic_vector(0 to 3);
        y:          out std_logic_vector(0 to 1));
end ttf_fsm;

use work.table_std.all;
architecture archttf_fsm of ttf_fsm is
        signal table_out: std_logic_vector(0 to 4);
        signal state: std_logic_vector(0 to 2);
constant state0: std_logic_vector(0 to 2) := "000";
constant state1: std_logic_vector(0 to 2) := "001";
constant state2: std_logic_vector(0 to 2) := "010";
constant state3: std_logic_vector(0 to 2) := "011";
constant state4: std_logic_vector(0 to 2) := "100";

constant table: ttf_table(0 to 21, 0 to 11) := (
--    present state  inputs  nextstate  output
--    -------------  ------  ---------  ------
          state0  &  "--0-" &  state0 &  "00",
          state0  &  "---0" &  state0 &  "00",
          state0  &  "0011" &  state1 &  "10",
          state1  &  "----" &  state2 &  "11",
          state2  &  "1---" &  state2 &  "11",
          state2  &  "-0--" &  state2 &  "11",
          state2  &  "--0-" &  state2 &  "11",
          state2  &  "---0" &  state2 &  "11",
          state2  &  "0111" &  state3 &  "10",
          state3  &  "0111" &  state3 &  "10",
          state3  &  "1000" &  state3 &  "10",
            state3  &  "11--" &  state3 &  "10",
            state3  &  "101-" &  state3 &  "10",
            state3  &  "0110" &  state0 &  "00",
            state3  &  "010-" &  state0 &  "00",
            state3  &  "00--" &  state0 &  "00",
            state3  &  "1001" &  state4 &  "11",
            state4  &  "0---" &  state3 &  "10",
            state4  &  "100-" &  state3 &  "10",
            state4  &  "11--" &  state4 &  "11",
            state4  &  "1010" &  state4 &  "11",
            state4  &  "1011" &  state0 &  "00");

begin
machine: process (clk, rst)
    begin
        if rst ='1' then
            table_out <= "00000";
        elsif (clk'event and clk='1') then
            table_out <= ttf(table,state & id);
        end if;
    end process;
state <= table_out(0 to 2);

--assign state outputs;
y <= table_out(3 to 4);
end archttf_fsm;
```

This implementation uses the `ttf` function (truth table function) which enables you to create a state transition table that lists the inputs, the current state, the next state, and the associated outputs. Within the architecture statement, a few signals and constants are defined. The signal called `table_out` is the vector which will contain the output from the state table. The signal called `state` is the state variable itself. Six constants are defined which contain the state encoding - `state0`, `state1`, `state2`, `state3`, and `state4`, and `table` - which contains the entire state transition table. The table itself is created as an array with a certain number of rows designating the number of transitions, and a certain number of columns designating the number of input bits, present state bits, next state bits, and output bits.

Since the ttf function is not a standard part of VHDL, it has been defined in a separate package and provided as part of the *Warp* software. This package is located in the *work* library and is called `table_std`. To allow a design to have access to the ttf function, the user must add the statement `use work.table_std.all`; to his VHDL description immediately above his architecture definition.

Most of the work lies in creating the truth table, and the process becomes fairly simple. The first portion of the process defines the asynchronous reset. Next, the synchronous portion of the process (elsif clk’event and clk=’1’) is defined in which the signal table_out is assigned the returned value of the ttf function. The function is called with two parameters: the name of the state transition table, and the set of bits which contain the inputs and the present state information. The value that is returned is the remainder of the columns in the table (total number of columns - second parameter). These bits will contain the next state value and the associated outputs. The only task remaining is to split the state information from the output information and assign them to the appropriate signal names. Both of these assignments must occur outside of the process, otherwise another level of registers will be created, as this portion of the process defines synchronous assignments.

This design, as implemented, uses more registers than required but could easily be modified. Registers must be created for both the state registers and the output registers, as in the second implementation (outputs decoded in parallel). The truth table can be modified so that the outputs are encoded in the state bits, as in the third example. Thus, rather than specifying both next state values and outputs, the user can simply specify next state values in which the outputs are encoded.

Mealy state machines are characterized by the outputs which can change depending on the current inputs. This example implements the state machine shown in Figure 1-7, which has Moore outputs and one Mealy output. Figure 1-8 shows a block diagram of a Mealy machine. The code follows.

```
library ieee ;
use ieee.std_logic_1164.all ;
entity mealy1 is port(
    clk, rst:in std_logic;
    id:     in std_logic_vector(3 downto 0);
    w:      out std_logic;
    y:      out std_logic_vector(1 downto 0));
end mealy1;
architecture archmealy1 of mealy1 is
    type states is (state0, state1, state2, state3, state4);
    signal state: states;
begin
moore: process (clk, rst)
    begin
        if rst=’1’ then
            state <= state0;
        elsif (clk’event and clk=’1’) then
            case state is
                when state0 =>
                    if id = x"3" then
```

```
                              state <= state1;
                         else
                              state <= state0;
                         end if;
                    when state1 =>
                         state <= state2;
                    when state2 =>
                         if id = x"7" then
                              state <= state3;
                         else
                              state <= state2;
                         end if;
                    when state3 =>
                         if id < x"7" then
                              state <= state0;
                         elsif id = x"9" then
                              state <= state4;
                         else
                              state <= state3;
                         end if;
                    when state4 =>
                         if id = x"b" then
                              state <= state0;
                         else
                              state <= state4;
                         end if;
               end case;
          end if;
     end process;

--assign moore state outputs;
y <= "00" when (state=state0) else
     "10" when (state=state1 or state=state3) else
     "11";
--assign mealy output;
w <= '0' when (state=state3 and id < x"7") else
     '1';
end archmealy1;
```

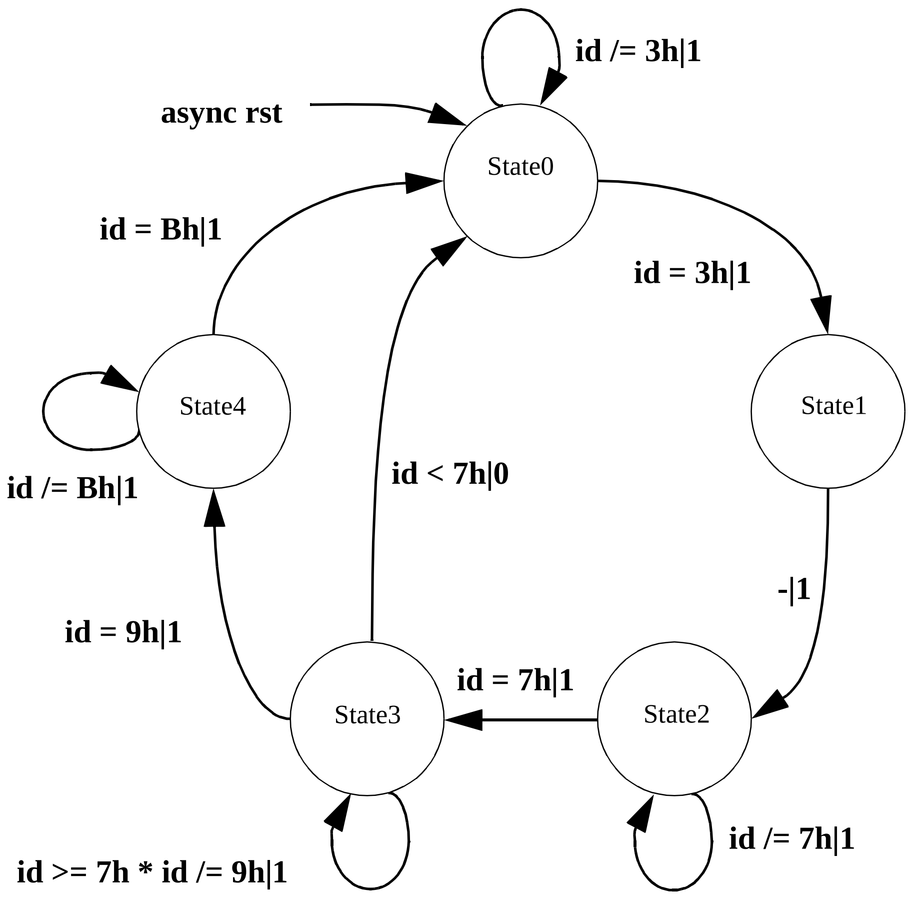

Figure 1-7 State Diagram for Combination Moore-Mealy State Machine

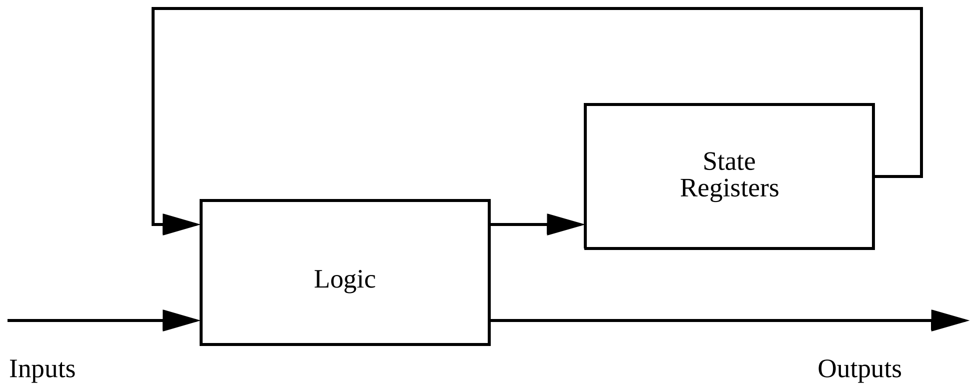

Figure 1-8 Block Diagram of Mealy State Machine

This implementation is almost identical to the first Moore implementation. The only difference is the additional Mealy output defined at the end of the architecture. The next example will examine a Mealy state machine, with all Mealy outputs.

Figure 1-9 is the state diagram for this new Mealy state machine. Two implementations follow.

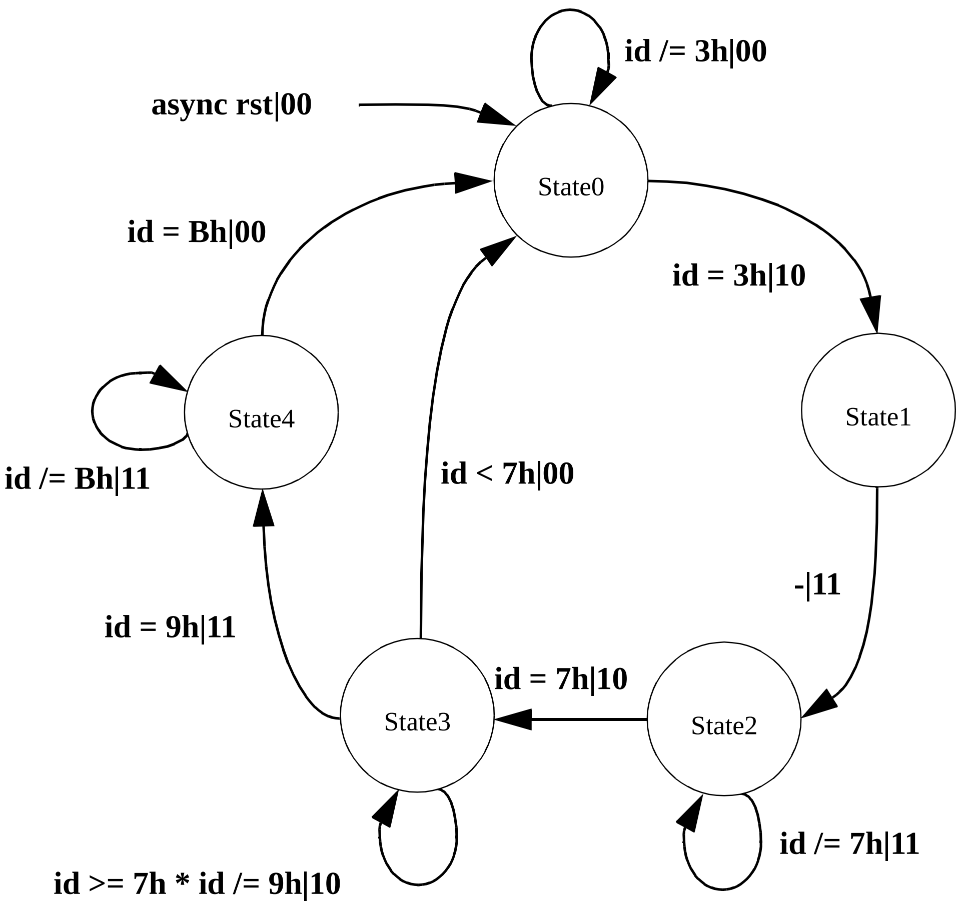

Figure 1-9 State Diagram for Second Mealy Machine

The following implementation specifies the state transitions in a synchronous process and the Mealy outputs with a concurrent statement.

```
library ieee ;
use ieee.std_logic_1164.all ;
entity mealy1 is port(
        clk, rst:               in std_logic;
        id:                     in std_logic_vector(3 downto 0);
        y:                      out std_logic_vector(1 downto 0));
end mealy1;

architecture archmealy1 of mealy1 is
        type states is (state0, state1, state2, state3, state4);
        signal state: states;
begin
machine: process (clk, rst)
        begin
                if rst='1' then
                        state <= state0;
                elsif (clk'event and clk='1') then
                        case state is
                                when state0 =>
                                        if id = x"3" then
                                                state <= state1;
                                        else
                                                state <= state0;
                                        end if;
                                when state1 =>
                                        state <= state2;
                                when state2 =>
                                        if id = x"7" then
                                                state <= state3;
                                        else
                                                state <= state2;
                                        end if;
                                when state3 =>
                                        if id < x"7" then
                                                state <= state0;
                                        elsif id = x"9" then
                                                state <= state4;
                                        else
                                                state <= state3;
                                        end if;
                                when state4 =>
                                        if id = x"b" then
                                                state <= state0;
```

```
                    else
                        state <= state4;
                    end if;
                end case;
            end if;
        end process;

--assign mealy output;
y <=        "00" when     ((state=state0 and id /= x"3") or
                        (state=state3 and id < x"7") or
                        (state=state4 and id = x"B")) else
            "10" when ((state=state0 and id = x"3") or
                        (state=state2 and id = x"7") or
                        (state=state3 and (id >= x"7") and
                        (id /= x"9"))) else
    "11";
end archmealy1;
```

This implementation of the Mealy state machine uses a synchronous process in much the same way as all of the other examples. An enumerated type is used to define the states. As in all but the one_hot coding implementation, the user can choose his own state assignment, as in the third Moore implementation. The Mealy outputs in this implementation are defined in a concurrent `when...else` construct. Thus, the output y is a function of the current state and the present inputs.

A second implementation of the same state machine follows. This implementation uses one synchronous process (in which the next state is captured by the state registers) and one combinatorial process in which the state transitions and Mealy outputs are defined.

```
library ieee ;
use ieee.std_logic_1164.all ;
entity mealy1 is port(
        clk, rst:           in std_logic;
        id:                 in std_logic_vector(3 downto 0);
        y:                  out std_logic_vector(1 downto 0));
end mealy1;
architecture archmealy1 of mealy1 is
        type states is (state0, state1, state2, state3, state4);
        signal state, next_state: states;
begin
st_regs: process (clk, rst)
        begin
            if rst='1' then
                state <= state0;
            elsif (clk'event and clk='1') then
                state <= next_state;
                end if;
        end process;
mealy: process (id)
        begin
                case state is
                        when state0 =>
                                if id = x"3" then
                                        next_state <= state1;
                                        y <= "10";
                                else
                                        next_state <= state0;
                                        y <= "00";
                                end if;
                        when state1 =>
                                next_state <= state2;
                                y <= "11";
                        when state2 =>
                                if id = x"7" then
                                        next_state <= state3;
                                        y <= "10";
                                else
                                         next_state <= state2;
                                        y <= "11";
                                end if;
                        when state3 =>
                                if id < x"7" then
                                        next_state <= state0;
                                        y <= "00";
                                elsif id = x"9" then
                                        next_state <= state4;
                                        y <= "11";
                                else
                                        next_state <= state3;
                                        y <= "10";
                                end if;
                        when state4 =>
                                if id = x"b" then
                                        next_state <= state0;
                                        y <= "00";
                                else
                                        next_state <= state4;
                                        y <= "11";
                                end if;
                        end case;
                end process;
end archmealy1;
```

In this implementation, the first process, `st_regs`, captures the next state value. The second process, `mealy`, defines the state transitions and the Mealy outputs. This second process is not synchronous and is activated each time the signal id transitions. Because the second process is not synchronous, the outputs can change even if the state doesn't, as would be expected in a Mealy state machine.

This concludes the discussion of state machines. Additional state machine, counter, and logic examples are documented in Section 1.10 Additional Design Examples. The next section discusses hierarchical design and addresses the concept of packages.

1.8 Packages

A package can declare components (which are entity and architecture pairs), types, constants, or functions as a way to make these items visible in other designs.

The form of a package declaration is as follows:

```
PACKAGE package_name IS
      declarations

END package_name;
```

Package declarations are typically used in *Warp* to declare types, constants, and components to be used by other VHDL descriptions. Most commonly, the user places a package declaration (containing component declarations) at the beginning of a design file (before the entity and architecture pair definitions) in order to use the components in a subsequent or hierarchical design.

Packages which contain only components do not need a package body. If the user wishes to write VHDL functions to be used in multiple designs, however, then these functions must be declared in the package declaration as well as defined in a package body:

```
PACKAGE BODY package_name IS
      declarations
END package_name;
```

A package body always has the same name as its corresponding package declaration and is preceded by the reserved words PACKAGE BODY. A package body contains the function bodies whose declarations occur in the package declaration, as well as declarations that are not intended to be used by other VHDL descriptions.

The following example shows a package that declares a component named demo, whose design (entity and architecture pair) follows the package declaration:

```
package demo_package is
      component demo
      port(x:out std_logic; clk, y, z:in std_logic);
      end component;
end package;

entity demo is
      port(x:out std_logic; clk, y, z:in std_logic);
end demo;

architecture fsm of demo is
.
.
.
end fsm;
```

If this description were in the file *demofile.vhd*, the user could analyze the package and add it to the current *work* library with the following commands:

```
warp -a demofile
```

Items declared inside a package declaration are not automatically visible to another VHDL description. A USE clause within a VHDL description makes items analyzed as part of a separate package visible within that VHDL design unit.

USE clauses may take one of three forms:

- USE library_name.package_name;
- USE package_name.object;
- USE library_name.package_name.object;

The portion of the USE clause argument preceding the final period is called the prefix; that after the final period is called the suffix.

Some examples of use clauses are:

```
LIBRARY project_lib;
USE project_lib.special_pkg;
USE project_lib.special_pkg.comp1;
```

The LIBRARY statement makes the library *project_lib* visible within the current VHDL design unit. The first USE clause makes a package called "special_pkg" contained within library *project_lib* visible within the current VHDL design unit. The second USE clause makes a component called "comp1," contained within "special_pkg," visible within the current VHDL design unit.

The suffix of the name in the USE clause may also be the reserved word ALL. The use of this reserved word means that all packages within a specified library, or all declarations within a specified package, are to be visible within the current VHDL design unit. Some examples are:

```
USE project_lib.all;
```

This example makes all packages contained within library *project_lib* visible within the current VHDL design unit.

```
USE project_lib.special_pkg.all;
```

This example makes all declarations contained within package "special_pkg," itself contained within library *project_lib*, visible within the current VHDL design unit.

Note the important difference between these two USE clauses:

```
USE project_lib.special_pkg;

USE project_lib.special_pkg.all;
```

The first USE clause just makes the package named special_pkg within library project_lib visible within the current VHDL design unit. While the package name may be visible, however, **its contents are not**. The second USE clause makes all contents of package special_pkg visible to the current VHDL design unit.

Example:

The following code defines a four bit counter:

```
library ieee ;
use ieee.std_logic_1164.all ;
package counter_pkg is
        subtype nibble is std_logic_vector(3 downto 0);
        component upcnt port(
                clk:    in std_logic;
                count:  buffer nibble);
        end component;
end counter_pkg;

library ieee ;
use ieee.std_logic_1164.all ;                   -- Defines std_logic
use work.std_arith.all;                         -- Defines "+"
use work.counter_pkg.all;                       -- My package

entity upcnt is port(
        clk: in std_logic;
        count: buffer nibble);
end upcnt;

architecture archupcnt of upcnt is
begin
counter:process (clk)
        begin
                if (clk'event and clk='1') then
                        count <= count + 1;
                end if;
        end process counter;
end archupcnt;
```

The package declaration allows the user to use the `upcnt` component and the type `nibble` in other designs. For example, suppose the user needed five of these counters but did not want to write five separate process. They might prefer to simply instantiate the `upcnt` counter defined above in a new design, creating a level of hierarchy. The code follows.

```
use work.counter_pkg.all;

entity counters is port(
        clk1, clk2:             in std_logic;
        acnt, bcnt, ccnt:       buffer nibble;
                deqe:                           out std_logic);
end counters;

architecture archcounters of counters is
signal dcnt, ecnt: nibble;
begin
counter1: upcnt port map (clk1, acnt);
counter2: upcnt port map (clk2, bcnt);
counter3: upcnt port map (clk => clk1, count => ccnt);
counter4: upcnt port map (clk2, dcnt);
counter5: upcnt port map (count => ecnt, clk => clk2);
deqe <= ’1’ when (dcnt = ecnt) else ’0’;
end archcounters;
```

The initial USE clause makes the counter_pkg available to this design. Counter_pkg is required for the nibble definition used in the entity and the upcnt component used in the architecture. Five counters are then instantiated by using the port map to associate the component I/O with the appropriate entity ports or architecture signals. Three of the instantiations use positional association in which the position of the signals in the port map determines which I/O of the component the signal is associated with. In counter3 and counter5, named association is used to explicitly define the signal to component I/O connections. In named association, the order of the signal assignment is not important.

When using *Warp* to compile and synthesize the counters design, the design file that contains upcnt must be compiled first, before counters can be compiled and synthesized. This is because the contents of the counter_pkg must first be added to the *work* library. Therefore, when selecting (in Galaxy) the *Warp* input files to be compiled and synthesized, select the upcnt design as the first file and the counters design as the second. This will ensure that upcnt is compiled first. Once the upcnt design has been added to the current *work* library, the design does not need to recompile it when synthesizing the top-level design unless the user makes changes to it or target the design to a different device.

1.8.1 Predefined Packages

Special purpose packages are provided with the *Warp* compiler to simplify description and synthesis of frequently used and useful structures and operations not native to VHDL. Additionally the math packages also contain very highly tuned designs for datapath components used during operator inferencing.

The following packages are supplied standard with the *Warp* compiler. Synthesis versions of these packages are found in *c:\warp\lib\common*. Original versions of these packages are in *c:\warp\lib\prim\presynth* and are likely to be more readable than the synthesis versions.

Table 1-4 Package Name and Purpose

Package	Purpose
std_logic_1164	This package defines the IEEE 1164 specification for std_logic.
std_arith/bit_arith	These two packages define math operations for std_logic_vectors and bit_vectors. They support vector-vector operations as well as vector, integer, integer-vector operations.
int_arith	For designs using integer signals, VHDL predefines operations for integers. However, using this package in the design will give the user hand tuned implementations for datapath operators.
numeric_bit / numeric_std	These two packages implement the upcoming IEEE standard for numeric operations. They are similar to 'std_arith' and 'bit_arith'. According to the standard, these packages support SIGNED and UNSIGNED operations.
table_std	This package provides a state transition table format for description of state machines.
rtlpkg	This package provides a set of simple logic functions useful for creation hierarchical structural VHDL design files.
math34x8_pkg	This package defines 8-bit arithmetic components for use in designs targeted to the MAX340 family CPLDs implemented using MAX340 architecture primitive elements.The math34x12_pkg, math34x16_pkg, and math34x24_pkg packages provide similar arithmetic components of widths 12, 16, and 24 respectively.
std_logic_arith/ std_logic_signed/ std_logic_unsigned	These three packages implement the Synopsys libraries for Standard Logic Arithmetic, Standard Logic Signed and Standard Logic Unsigned. Enables the user to access many of the arithmetic operators for use with many VHDL standard types.

The USE statement is required in a design file to make the package "visible" to the design file. The USE statement should immediately precede the entity and architecture pair and appear as follows:

USE work.package_name.all;
where package_name refers to one of the package names listed above. In a file defines more than one entity and architecture pair, the necessary USE clauses for each of the architectures have to be listed specifically. This means that the scope of the USE clause is limited to one entity/architecture pair.

Package Contents and Usage of std_arith, bit_arith, numeric_std and numeric_bit.

These packages contain functions which allow arithmetic operations on vectors. VHDL is a strongly typed language which does not recognize the arbitrary data types as types compatible with arithmetic manipulation. These packages contain functions which when invoked in the design file allow arithmetic operations on vectors.

Several of the functions in this package are implemented by "overloading" the native VHDL operators for arithmetic operations on vectors and integers. Overloading is a scheme by which one or more functions can be called by use of the same conventional arithmetic operator symbol. The compiler will call the correct function by determining the data types of the arguments of the function call. Multiple functions can be represented by the same symbol as long as no two functions accept the same combination of argument data types. This means the "+" sign can be used, for example, to call a vector addition routine since the data type of the arguments (both of type bit_vector) will signal that "+" should call the bit_vector addition function.

The contents of all of these packages contain identical support for all arithmetic operations. The main difference between each of these packages is the **type** of vector that they support. The package needed in the design depends on what kind of vectors are in the design. Table 1-5 illustrates the **type** support by these packages and the necessary use clauses.

Table 1-5 Use Clause Needed for Each Package

Package	Vector	Base	USE clause(s)

std_arith	std_logic_vector	std_logic	library ieee;
			use ieee.std_logic_1164.all;
			use work.std_arith.all;

bit_arith	bit_vector	bit	use work.bit_arith.all;

numeric_std	unsigned, signed	std_logic	library ieee;
			use ieee.std_logic_1164.all;
			use work.numeric_std.all;

numeric_bit	unsigned, signed	bit	use work.numeric_bit.all;
std_logic_arith	signed, unsigned	std_ulogic	use synopsys.std_logic_arith.all;
std_logic_signed	std_logic_vector	std_logic	use synopsys.std_logic_signed.all;
std_logic_unsigned	std_logic_vector	std_logic	use synopsys.std_logic_unsigned.all;

In Table 1-5, the VHDL definition for the vector type each of these packages support is as follows:

TYPE <Vector> is ARRAY (NATURAL RANGE <>) of <Base> ;
Where <Vector> is the name of the vector and the <Base> is the name of the type of the elements of the vector.

Note – Even though these packages support Vector to Integer operations, *Warp* implementations allow the Integers to be Integer constants only. This means that these packages do not support operations between Vectors and Integers where both of the operands are signals.

The common convention followed in all of these packages is that the left most bit is the MSB (most significant bit) regardless of the direction of the vector.

The following describes each of the operators supported. The conventions here use:

"a" and "b" for vector signals

"i" for integer constants

.op. for operators.

Support is provided for addition, multiplication, relational, shift and boolean operators.

Addition Operators (+, -)

(a .op. b) Operands are both vectors. The resultant vector is the size of the larger of the two vectors.

(a .op. i) (i .op. a)
One operand is a vector and the other an integer constant. The resulant vector is the size of "a". Integers larger than "a" lose their MSBs.

Multiplication Operators (*)

(a .op. b) Operands are both vectors. The resultant vector is the sum of the sizes of the two vectors.

(a .op. i) (i .op. a)
One operand is a vector and the other an integer constant. The resulant vector is twice the size of "a". Integers larger than "a" lose their MSBs.

Relational Operators (=, /=, <=, >=, <, >)

(a .op. b) Operands are both vectors. The resultant is of type boolean.

(a .op. i) (i .op. a)
One operand is a vector; the other, an integer constant. The result is of type boolean. The current implementations require that the number of bits required to represent "i" be less than or equal to "a"s size.

Shift Operators (sll, srl, sla, sra, ror, rol)

(a .op. i) For the shift operators, the second operand must be an integer constant.

Boolean Operators (ALL)

(a .op. b) All boolean operators are supported. For binary operators, the length of the two vectors has to be the same.

Miscellaneous Functions

std_match (a, "string")

This function is provided in the numeric_std and std_arith packages only.

This function can be used to compare a vector to a string containing ’0’, ’1’ and ’-’. Normally in VHDL, if a is compared to "00-", where the intention is not to compare the LSB, the result will always be false unless a(0) is really set to the value ’-’. In synthesis, this will always evaluate to false. To avoid this problem, the std_match function is provided. The size of the string has to match the size of the vector.

Usage:
```
if (std_match(a, "10-11")) then
x <= ’1’ ;
```

resize (a, size)

This function resizes an array (a) to size ’size’, padding with ’0’s if necessary.

to_unsigned (i,s)

This function is provided in the numeric_std and the numeric_bit packages only.

The argument is an integer that is to be converted, and the size of the resultant vector and the result is an unsigned vector of size "s".

to_std_logic_vector(i,s)

This function is provided in the std_arith package only.

The argument is an integer that is to be converted, and the size of the resultant vector and the result is an std_logic vector of size "s".

to_bit_vector (i,s)

This function is provided in the bit_arith package only.

The argument is an integer that is to be converted, and the size of the resultant vector and the result is an bit_vector of size "s".

to_integer (a)

This function is provided in all the packages and is given a vector, returns an integer.

Package Contents and Usage of int_arith

The int_arith package should be used when both operands of an operator are integer signals. If this package is not used, the design will not produce the best results even though the results will be logically correct. Without this package, the design will use the VHDL language's implementations. If this package is included, it overloads all the operators and provides tuned implementations for the datapath operators.

Package Contents and Usage of bv_math

This package will be omitted in the next release and is provided in this release only for compatibility purposes. This package should be substituted by one of the above packages. There is one important difference, however, between this package and the arithmetic packages described previously. In this package, the highest index is considered the most significant bit.

The operators for which functions are provided are:

inc_bv(a)	increment bit_vector a. If function is assigned to a signal within a clocked process, the synthesized result will be an up counter. Equivalent to a <= a + 1; Usage: a <= inc_bv(a);
dec_bv(a)	decrement a bit_vector a. If function is assigned to a signal within a clocked process, the synthesized result will be a down counter. Equivalent to a <= a - 1;. Usage: a <= dec_bv(a);

+(a; b)	regular addition function for two bit_vectors a and b. The "+" operator overloads the existing "+" operator for definition of arithmetic operations on integers. The output vector is the same length as the input vector, so that there is no carry output. If a carry out is required, the user should increase the length of the input vectors and use the MSB as the carry out.
	
	Usage for two vectors of length 8 with carry out:
	signal a: bit_vector(0 to 8);
	signal b: bit_vector(0 to 8);
	signal q: bit_vector(0 to 8);
	q <= a + b;

+(a; b) regular addition function for adding to bit_vector a the object b of type bit. This is the equivalent of a conditional incrementing of bit_vector a. The "+" operator overloads the existing "+" operator for definition for arithmetic operations on integers. The output vector is the same length as the input vector so there is no carry output. If a carry out is required the user should increase the length of the input vector and use the MSB as the carry out.

Usage for 16 bit_vector with no carry out:

```
signal a: bit_vector(0 to 15);
signal b: bit;
signal q: bit_vector(0 to 15);
q <= a + b;
```

-(a; b) regular subtraction function for two bit_vectors. The "-" operator overloads the existing "-" operator definition for arithmetic operations on integers.

Usage:

```
signal a: bit_vector(0 to 7);
signal b: bit_vector(0 to 7);
signal q: bit_vector(0 to 7);
q <= a - b;
```

-(a; b) regular subtraction function for subtracting from bit_vector a the object b of type bit. This is equivalent to the conditional decrementing of only bit_vector a. The "-" operator overloads the existing "-" operator definition for arithmetic operations on integers.

Usage:

```
signal a: bit_vector(0 to 7);
signal b: bit;
signal q: bit_vector(0 to 7);
q <= a - b;
```

inv(b) unary invert function for object b of type bit. For use in port maps and sequential assignments.

Usage:

```
signal b: bit;
signal z: bit;
z <= inv(b);
```

inv(a)	invert function which inverts each bit of bit_vector a and returns resulting bit_vector.
	
	Usage:
	signal a: bit_vector(0 to 15);
	signal q: bit_vector(0 to 15);
	q <= inv(a);

Package Contents and Usage of int_math

This package is being provided only for compatibility purposes.

This package contains functions which allow mixed arithmetic operations on bit_vectors and integers.

VHDL is a strongly typed language which does not recognize the bit data type as a type compatible with arithmetic manipulation. This package contains functions, which when invoked in the design file, allow arithmetic operations which mix integers and bit_vectors.

Several of the functions in this package are implemented by "overloading" the native VHDL operators for arithmetic operations on integers. Overloading is a scheme by which one or more functions can be called by use of the same conventional arithmetic operator symbol. The compiler will call the correct function by noting the data types of the arguments of the function call. Multiple functions can be represented by the same symbol as long as no two functions accept the same combination of argument data types. This means the "+" sign can be used, for example, to call a routine for adding mixed data types (bit_vector and integer) since the data type of the arguments will signal that "+" should call the package function for mixed addition rather than the native function for integer addition. The operators for which functions are provided are:

bv2i(a)	converts bit_vector a to an integer.
	
	Usage:
	variable z: integer range 0 to 15;
	signal a: bit_vector(0 to 3);
	z := bv2i(a);

i2bv(i; w)	converts integer i to binary equivalent and expresses as a bit_vector of length w.
	
	Usage:
	variable i: integer range 0 to 31;
	signal a: bit_vector(0 to 4);
	a <= i2bv(i, 5);

i2bvd(i; w)	converts integer i to a binary coded decimal bit_vector of length w.
	
	Usage:
	variable i: integer range 0 to 31;
	signal a: bit_vector(0 to 7);
	a <= i2bv(i, 8);

=(a; b)	converts bit_vector a to integer and checks for equality with integer b. Returns boolean value true if equal, false if not equal. Overloads native operator for integer arithmetic.
	
	Usage:
	signal a: bit_vector(0 to 15);
	variable b: range 0 to 64;
	variable z: boolean;
	z := (a = b);

/=(a; b)	converts bit_vector a to integer and checks for equality with integer b. Returns boolean value true if not equal, false if equal.Overloads native operator for integer arithmetic.
	
	Usage:
	signal a: bit_vector(0 to 15);
	variable b: range 0 to 128;
	variable z: boolean;
	z := (a /= b);

>(a; b)	converts bit_vector a to integer and compares with integer b. If a is > b, returns boolean value true, otherwise returns false. Overloads native operator for integer arithmetic.
	
	Usage:
	signal a: bit_vector(0 to 15);
	variable b: range 0 to 128;
	variable z: boolean;
	z := (a > b);

<(a; b)	converts bit_vector a to integer and compares with integer b. If a is < b, returns boolean value true, otherwise returns false. Overloads native operator for integer arithmetic.
	
	Usage:
	signal a: bit_vector(0 to 15);
	variable b: range 0 to 128;
	variable z: boolean;
	z:= (a < b);

>=(a; b) converts bit_vector a to integer and compares with integer b. If a is >= b, returns boolean value true, otherwise returns false. Overloads native operator for integer arithmetic.

Usage:
signal a: bit_vector(0 to 15);
variable b: range 0 to 128;
variable z: boolean;
z:= (a >= b);

<=(a; b) converts bit_vector a to integer and compares with integer b. If a is <= b, returns boolean value true, otherwise returns false. Overloads native operator for integer arithmetic.

Usage:
signal a: bit_vector(0 to 3);
variable b: range 0 to 128;
variable z: boolean;
z:= (a <= b);

+(a; b) increments bit_vector a the number of times indicated by the value of integer b and returns bit_vector result. Implemented by conversion of integer b to bit_vector and adding to bit_vector a. Overloads native operator for integer arithmetic.

Usage:
signal a: bit_vector(0 to 15);
variable b: range 0 to 128;
signal z: bit_vector(0 to 15);
z <= a + b;

-(a; b) decrements bit_vector a the number of times indicated by the value of integer b and returns bit_vector result. Implemented by conversion of integer b to bit_vector and subtracting from bit_vector a. Overloads native operator for integer arithmetic.

Usage:
signal a: bit_vector(0 to 15);
variable b: range 0 to 128;
signal z: bit_vector(0 to 15);
z <= a - b;

Package Contents and Usage of table_std

The table_std package describes a truth table function, ttf, which can be used to implement state transiting tables or other truth tables. A description and example of the ttf function can be found in Section 1.7.3 Design Methodologies.

Package Contents and Usage of rtlpkg

The package rtlpkg contains VHDL component declarations for basic VHDL components which can be used to construct structural design files of more complex logic circuits. These components are useful for controlling implementation of the design by the *Warp* compiler to ensure that specific performance or architecture choices are preserved in the final synthesized design. These components are generic components which can be used to describe retargetable designs, which can be synthesized and fit to any desired Cypress device. The compiler makes the appropriate synthesis choices based on the target device's architectural resources to achieve the best possible utilization of the device. By preserving the specified interconnection of the declared components, the compiler maintains the specific circuit implementation intended by the designer.

Components contained in the package rtlpkg are:

Name	Function
bufoe	bidirectional I/O with three state output driver y with type bit feedback to logic array (yfb)
dlatch	transparent latch with active low latch enable (e) (transparent high)
dff	positive edge triggered D-Type flip flop
xdff	positive edge triggered D-Type flip flop with XOR of two inputs (x1 & x2) feeding D input
jkff	positive edge triggered jk flip flop
buf	signal buffer to represent a signal not to be removed during synthesis to enable visibility during simulation and to create a manual factoring point
srl	set/reset latch with reset dominant, set and reset active high
srff	positive edge triggered set/reset flip flop, reset dominant, set and reset active high
dsrff	positive edge triggered D-Type flip flop without asynchronous set and reset, reset dominant, set and reset active high
tff	toggle flip flop
xbuf	two input exclusive OR gate
triout	three state buffer with active high output enable input

Package Contents and Usage of math34x8_pkg

The packages math34x8_pkg, math34x12_pkg, math34x16_pkg, and math34x24_pkg contain VHDL component declarations for optimal arithmetic components to be implemented in the MAX340 family of devices. These components are useful for controlling implementation of arithmetic functions by the *Warp* compiler to ensure that specific performance or architecture choices are preserved in the final synthesized design. There are four packages containing components that are functionally the same but operate on signals of different widths. The usage for these components is the same. These packages are only available with the 3.5 library. The components included in the math34x8_pkg are as follows:

Name	Function
add_8	eight-bit adder
sub_8	eight-bit subtracter
gt_8	eight-bit greater than comparator
lt_8	eight-bit less than comparator
ge_8	eight-bit greater than or equal to comparator
le_8	eight-bit less than or equal to comparator

1.9 Libraries

If all information about a design description had to appear in one file, many VHDL files would be huge and cumbersome, and information re-use would be impossible. Fortunately, VHDL allows the user to share information between files by means of two constructs: libraries and packages.

In VHDL, a library is a collection of previously analyzed design elements (packages, components, entities, architectures) that can be referenced by other VHDL descriptions. In *Warp*, a library is implemented as a directory, containing one or more VHDL files and an index to the design elements they contain.

Note – In VHDL, analysis is the examination of a VHDL description to guarantee compliance with VHDL syntax, and the extraction of design elements (packages, components, entities, architectures) from that description. Synthesis is the production of a file (to be written onto a physical chip) that embodies the design elements extracted from the VHDL descriptions by analysis.

To make the contents of a library accessible to a VHDL description, use a library clause. A library clause takes the form:

LIBRARY library_name [, library name...];
For example, the statement

LIBRARY gates, my_lib;
makes the contents of two libraries called *gates* and *my_lib* accessible in the VHDL description in which the **library** clause is contained.

Library clauses are seldom needed in *Warp* VHDL descriptions. This is because all VHDL implementations include a special library, named *work*. *Work* is the symbolic name given to the current working library during analysis. The results of analyzing a VHDL description are placed by default into the *work* library for use by other analyses. (In other words, a **library** clause is not necessary to make the *work* library accessible).

1.9.1 Synopsys Library Support

There are three additional libraries to provide compatibility with designs targeted for Synopsys. These libraries are Standard Logic Arithmetic, Standard Logic Signed, and Standard Logic Unsigned.

Both Standard Logic Signed and Standard Logic Unsigned will be fully supported. The Standard Logic Arithmetic library will support all functions that return signed or unsigned values along with those that return boolean values. Those functions that return std_logic_vector values are not supported.

To enable the use of these packages in a design, the user must include the appropriate "use" clauses:

Standard Logic Arithmetic	use ieee.std_logic_arith.all;
Standard Logic Signed	use ieee.std_logic_signed.all;
Standard Logic Unsigned	use ieee.std_logic_unsigned.all;

The inclusion of these libraries enables the user to access many of the arithmetic operators (e.g. "+", "-", "*", "<", ">", ">=", "<=", "=", "/=") for use with many of the VHDL standard types.

The Synopsys compatible libraries can be used in place of the earlier libraries supplied with Warp.

1.10 Additional Design Examples

Many examples demonstrating design methodologies can be found in Section 1.7.3 Design Methodologies. Most of these examples can be found in the *c:\warp\examples* directory. This section provides a discussion for additional design examples found in the *c:\warp\examples* directory but not discussed earlier in this chapter. These designs include:

Logic

- DEC24: a two-to-four bit decoder.
- PINS: shows how to use the `part_name` and `pin_numbers` attributes to map signals to pins.
- NAND2_TS: a two-input NAND gate with three-state output.

Counters

- CNT4_EXP: Four bit counter with synchronous reset. The counter uses expressions for clocks and resets.
- CNT4_REC: Four bit counter with load on the bidirectional pins. Demonstrates use of a record.

State Machines

- DRINK: a behavioral description of a mythical drink machine (the drinks only cost 30 cents!).
- TRAFFIC: a traffic-light controller.
- SECURITY: a simple security system.

1.10.1 DEC24

This example demonstrates a two-to-four decoder.

```
library ieee ;
use ieee.std_logic_1164.all ;

-- two to four demultiplexer/decoder

ENTITY demux2_4 IS
      PORT(in0, in1: IN std_logic;
       d0, d1, d2, d3: OUT std_logic);
END demux2_4;
```

```
ARCHITECTURE behavior OF demux2_4 IS
BEGIN
    d0 <= (NOT(in1) AND NOT(in0));
    d1 <= (NOT(in1) AND in0);
    d2 <= (in1 AND NOT(in0));
    d3 <= (in1 AND in0);
END behavior;
```

The entity declaration specifies two input ports, in0 and in1, and four output ports, d0, d1, d2, and d3, all of type std_logic.

The architecture specifies the various ways that the two inputs are combined to determine the outputs. This is one of several ways that a two-to-four decoder can be implemented.

1.10.2 PINS

This example shows how to use the part_name and pin_numbers attributes to map signals to pins.

```
library ieee ;
use ieee.std_logic_1164.all ;

--Signals that are not assigned to pins can be automatically
--assigned pins by Warp. This design uses the C22V10-25DMB.
ENTITY and5Gate IS
    PORT (a: IN std_logic_vector(0 TO 4);
          f: OUT std_logic);

ATTRIBUTE part_name of and5Gate:ENTITY IS "C22V10";
ATTRIBUTE pin_numbers of and5Gate:ENTITY IS
     "a(0):2 a(1):3 " --The spaces after 3 and 5 are necessary
     & "a(2):4 a(3):5 " --for concatenation (& operator)
     & "f:23";  --signal a(4) will be assigned a pin by warp
END and5Gate;

ARCHITECTURE see OF and5Gate IS
BEGIN
    f <= a(0) AND a(1) AND a(2) AND a(3) AND a(4);
END see;
```

Of particular importance in this example is the space just before the closing right-quote of each portion of the attribute value to be concatenated. As shown, this value resolves to

```
a(0):2 a(1):3 a(2):4 a(3):5 f:23
```

Had the spaces not been included, this value would have been

```
a(0):2 a(1):3a(0):4 a(1):5f:23
```

which is an unrecognizable string.

1.10.3 NAND2_TS

This example is a two-input NAND gate with three-state output.

```
library ieee ;
use ieee.std_logic_1164.all ;

--Two input NAND gate with three-state output
--This design is DEVICE DEPENDENT.

USE work.rtlpkg.all;                    --needed for triout

ENTITY threeStateNand2 IS
    PORT (a, b, outen: IN std_logic;
        c: INOUT std_logic);
END threeStateNand2;

ARCHITECTURE show OF threeStateNand2 IS
SIGNAL temp: std_logic;

BEGIN
    temp <= a NAND b;
    tri1: triout PORT MAP (temp, outen, c);
END show;
```

This design is implemented by instantiating one triout component from rtlpkg. Temp is a signal created to be the input to the three-state buffer. Outen is the output enable, and c is the output (the NAND of signals a and b).

1.10.4 CNT4_EXP

This example is a counter that uses expressions for clocks and resets.

```
-- Fits to a c344

library ieee ;
use ieee.std_logic_1164.all ;
USE work.std_arith.all;

ENTITY testExpressions IS
    PORT (clk1, clk2, res1, res2, in1, in2: IN std_logic;
         count: BUFFER std_logic_vector(0 TO 3));
END testExpressions;

ARCHITECTURE cool OF testExpressions IS
```

```
SIGNAL clk, reset: std_logic;

BEGIN
    clk <= clk1 AND clk2; --both clocks must be asserted;
    reset <= res1 OR res2; --either reset

proc1:PROCESS
    BEGIN
    WAIT UNTIL clk = ’1’;
        IF reset = ’1’ THEN
            count <= x"0";
        ELSE
            count <= count + 1 ;
        END IF;
    END PROCESS;
END cool;
```

The entity declaration specifies two clock signals and two reset signals as external interfaces, as well as two input data ports and a four-bit_vector for output.

The architecture declares two new signals, clk and reset, which are later defined to be the AND of clk1 and clk2 and the OR of reset1 and reset2, respectively. Both clocks must be asserted to detect a clock pulse and trigger the execution of the process. If either reset is asserted when a clock pulse is detected, the counter resets itself, else it increments by one and waits for the next clock pulse.

1.10.5 CNT4_REC

This example uses a record with a four bit counter with load on the bidirectional pins.

```
library ieee ;
use ieee.std_logic_1164.all ;
USE work.std_arith.all;
-- loads on the i/o pins
-- temp is a RECORD used to simplify instantiating bufoe
USE work.rtlpkg.all;
ENTITY counter IS
    PORT (clk, reset, load, outen: IN std_logic;
        count: INOUT std_logic_vector(0 TO 3));
END counter;
ARCHITECTURE behavior OF counter IS
TYPE bufRec IS      -- record for bufoe
    RECORD              -- inputs and feedback
        cnt: std_logic_vector(0 TO 3);
        dat: std_logic_vector(0 TO 3);
END RECORD;
```

```
SIGNAL temp: bufRec;
CONSTANT counterSize: INTEGER:= 3;
BEGIN
g1:  FOR i IN 0 TO counterSize GENERATE
          bx: bufoe PORT MAP(temp.cnt(i), outen, count(i), temp.dat(i));
     END GENERATE;
proc1:PROCESS
     BEGIN
     WAIT UNTIL (clk = '1');
          IF reset = '1' THEN
               temp.cnt <= "0000";
          ELSIF load = '1' THEN
               temp.cnt <= temp.dat;
          ELSE
               temp.cnt <= temp.cnt + 1; -- increment vector
          END IF;
     END process;
END behavior;
```

The entity declaration specifies that the design has four input bits (clk, reset, load, and outen) and a four-bit_vector for output.

The architecture implements a counter with synchronous reset and load, and also demonstrates the use of RECORD types.

1.10.6 Drink

This example a behavioral description of a mythical drink dispensing machine (the drinks only cost 30 cents!):

```
library ieee ;
use ieee.std_logic_1164.all ;

--In keeping with the fact that this is a mythical drink
--machine, the cost of the drink is 30 cents!

entity drink is port (
     nickel,dime,quarter,clock : in std_logic;
     returnDime,returnNickel,giveDrink: out std_logic);
end drink;

architecture fsm of drink is
     type drinkState is (zero,five,ten,fifteen,twenty,twentyfive,owedime);
     signal drinkStatus: drinkState;
begin
     process begin
```

```
wait until clock = ’1’;
-- set up default values
giveDrink <= ’0’;
returnDime <= ’0’;
returnNickel <= ’0’;
case drinkStatus is
    when zero =>
        IF (nickel = ’1’) then
            drinkStatus <= Five;
        elsif (dime = ’1’) then
            drinkStatus <= Ten;
        elsif (quarter = ’1’) then
            drinkStatus <= TwentyFive;
        end if;
    when Five =>
        IF (nickel = ’1’) then
            drinkStatus <= Ten;
        elsif (dime = ’1’) then
            drinkStatus <= Fifteen;
        elsif (quarter = ’1’) then
            giveDrink <= ’1’;
            drinkStatus <= zero;
        end if;
    when Ten =>
        IF (nickel = ’1’) then
            drinkStatus <= Fifteen;
        elsif (dime = ’1’) then
            drinkStatus <= Twenty;
        elsif (quarter = ’1’) then
            giveDrink <= ’1’;
            returnNickel <= ’1’;
            drinkStatus <= zero;
        end if;
    when Fifteen =>
        IF (nickel = ’1’) then
            drinkStatus <= Twenty;
        elsif (dime = ’1’) then
            drinkStatus <= TwentyFive;
        elsif (quarter = ’1’) then
            giveDrink <= ’1’;
            returnDime <= ’1’;
            drinkStatus <= zero;
        end if;
    when Twenty =>
        IF (nickel = ’1’) then
```

```
                    drinkStatus <= TwentyFive;
               elsif (dime = '1') then
                    giveDrink <= '1';
                    drinkStatus <= zero;
               elsif (quarter = '1') then
                    giveDrink <= '1';
                    returnNickel <= '1';
                    returnDime <= '1';
                    drinkStatus <= zero;
               end if;
          when TwentyFive =>
               IF (nickel = '1') then
                    giveDrink <= '1';
                    drinkStatus <= zero;
               elsif (dime = '1') then
                    returnNickel <= '1';
                    giveDrink <= '1';
                    drinkStatus <= zero;
               elsif (quarter = '1') then
                    giveDrink <= '1';
                    returnDime <= '1';
                    drinkStatus <= oweDime;
               end if;
          when oweDime =>
               returnDime <= '1';
               drinkStatus <= zero;
-- The following WHEN makes sure that the state machine
-- resets itself if it somehow gets into an undefined state.
          when others =>
               drinkStatus <= zero;
          end case;
     end process;
end fsm;
```

The entity declaration specifies that the design has four inputs: nickel, dime, quarter, and clock. The outputs are giveDrink, returnNickel, and returnDime. The last two outputs tell the design when to give change after the 30-cent price of the drink has been satisfied.

The architecture then defines an enumerated type with one value for each possible state of the machine, i.e., each possible amount of money deposited. Thus, the initial state of the machine is zero, while other states include five, ten, fifteen, etc.

After some initialization statements, the major part of the architecture consists of a large case statement, containing a when clause for each possible state of the machine. Each when clause contains an if...then...elsif statement to handle each possible input and change of state.

1.10.7 Traffic

This example is a traffic-light controller.

```
library ieee ;
use ieee.std_logic_1164.all ;

-- This state machine implements a simple traffic light.
-- The N - S light is usually green, and remains green
-- for a minimum of five clocks after being red. If a
-- car is travelling E-W, the E-W light turns green for
-- only one clock.

ENTITY traffic_light IS
    PORT(clk, car: IN std_logic;--car is E-W travelling
        lights: BUFFER std_logic_vector(0 TO 5));
END traffic_light;

ARCHITECTURE moore1 OF traffic_light IS
-- The lights (outputs) are encoded in the following states.
-- For example, the
-- state green_red indicates the N-S light is green and the
-- E-W light is red.
-- "001" indicates green light, "010" yellow, "100" red;
-- "&" concatenates
    CONSTANT green_red : std_logic_vector(0 TO 5) := "001" & "100";
    CONSTANT yellow_red : std_logic_vector(0 TO 5) := "010" & "100";
    CONSTANT red_green : std_logic_vector(0 TO 5) := "100" & "001";
    CONSTANT red_yellow : std_logic_vector(0 TO 5) := "100" & "010";

-- nscount to verify five consecutive N-S greens
    SIGNAL nscount: INTEGER RANGE 0 TO 5;

BEGIN
    PROCESS
    BEGIN
        WAIT UNTIL clk = ’1’;
        CASE lights IS
            WHEN green_red =>
                IF nscount < 5 THEN
                    lights <= green_red;
                    nscount <= nscount + 1;
                ELSIF car = ’1’ THEN
                    lights <= yellow_red;
                    nscount <= 0;
                ELSE
```

```
                    lights <= green_red;
                END IF;
            WHEN yellow_red =>
                lights <= red_green;
            WHEN red_green =>
                lights <= red_yellow;
            WHEN red_yellow =>
                lights <= green_red;
            WHEN others =>
                lights <= green_red;
        END CASE;
    END PROCESS;
END moore1;
```

The states in this example are defined such that the outputs are encoded in the state, using red/yellow/green triplets for each of the north-south and east-west light. For example, if the north-south light is red and the east-west light is green, then the state encoding is "100001".

In this design, the north-south light remains green for a minimum of five clock cycles, while the east-west light only remains green for one clock cycle. Note the use of signal nscount to keep track of the number of clock cycles the north-south light has remained green. This is less confusing than creating five extra states that do basically nothing.

1.10.8 Security

This example is a simple security system.

```
ENTITY securitySystem IS
    PORT (set, intruder, clk: IN std_logic;
        horn: OUT std_logic);
END securitySystem;

ARCHITECTURE behavior OF securitySystem IS
TYPE states IS (securityOff, securityOn, securityBreach);
SIGNAL state, nextState: states;


BEGIN
PROC1:PROCESS (set, intruder)
    BEGIN
        CASE state IS
        WHEN securityOff =>
            IF set = ’1’ THEN
                nextState <= securityOn;
            END IF;
        WHEN securityOn =>
```

```
                        IF intruder = ’1’ THEN
                              horn <= ’1’;  --Mealy output
                              nextState <= securityBreach;
                        ELSIF set = ’0’ THEN
                              horn <= ’0’;
                              nextState <= securityOff;
                        END IF;
                  WHEN securityBreach =>
                        IF set = ’0’ THEN
                              horn <= ’0’;
                              nextState <= securityOff;
                        END IF;
                  WHEN others =>
                        nextState <= securityOff;
                  END CASE;
            END PROCESS;
      proc2:PROCESS
            BEGIN
                  WAIT UNTIL clk =’1’;
                  state <= nextState;
            END PROCESS;
      END behavior;
```

The entity declaration specifies that the design has three inputs (set, intruder, and clk) and one output (horn), all of type std_logic.

The architecture declares an enumerated type with three possible values: securityOn, securityOff, and securityBreach. It also declares two state variables, named state and nxtState.

The rest of the architecture defines two concurrent processes that interact via the nextState signal. The first process is activated whenever a change occurs in the set or intruder signals, and defines what the new state of the machine will be as of the next clock signal. The second is activated with each rising clock pulse.

1.11 Alphabetical Listing of VHDL Constructs

The following sections provide an encyclopedic reference to each VHDL language element that *Warp* supports.

Each section shows the syntax of each language element, explains the purpose of the language element, and gives an example of its use.

1.11.1 Alias

ALIAS allows the user to define an alternate name by which to reference a VHDL object. Use ALIAS to create a shorter reference to a long object name, or to provide a mnemonic reference to a name that may be difficult to remember otherwise.

```
alias identifier[:subtype_indication] is name;
```

Identifier is the alias for *name* in the alias declaration. An alias of a signal denotes a signal; an alias of a variable denotes a variable; an alias of a constant denotes a constant.

An alias of an object can be updated if and only if the object itself can be updated. Thus, an alias for a constant or for a port of mode **in** cannot be updated.

An alias may be constrained to a sub-type of the object specified in *name*, but *identifier* and *name* must have the same base type.

Example:

```
signal Instrctn:std_logic_vector(15 downto 0);
alias Opcode:std_logic_vector(3 downto 0) is
      Instrctn(15 downto 12);
alias Op1:std_logic_vector(5 downto 0) is Instrctn(11 downto 6);
alias Op2:std_logic_vector(5 downto 0) is Instrctn(5 downto 0);
alias Sign1:std_logic is Op1(5);
alias Sign2:std_logic is Op2(5);
```

The first line of this example declares a signal called Instrctn, containing 16 bits. Succeeding lines define several aliases from sub-elements of this std_logic vector: two six-bit operands (Op1 and Op2) and two sign bits (Sign1 and Sign2). The alias declarations for Sign1 and Sign2 make use of previously declared aliases.

1.11.2 Architecture

An architecture (or, more formally, an "architecture body") describes the internal view of an entity, i.e., it specifies the functionality or the structure of the entity.

```
architecture name of entity is
      architecture_declarations;
```

```
    begin
        concurrent_statements;
    end [name];

architecture_declaration ::=
    subtype_declaration
    | constant_declaration
    | signal_declaration
    | component_declaration
    | attribute_specification
concurrent_statement ::=
    process_statement
    | concurrent_signal_assignment_statement
    | component_instantiation_statement
    | generate_statement
```

Architectures describe the behavior, data flow, or structure of an accompanying entity. See Section 1.6 Entities for more information about entities.

Architectures start with the keyword architecture, followed by a name for the architecture being declared, the keyword of, the name of the entity to which the architecture is being bound, and the keyword is.

A list of architecture declarations follows. This list declares components, signals, types, constants, and attributes to be used in the architecture. If a USE clause appears before the architecture, any elements referenced by the USE clause need not be re-declared.

The architecture body follows, consisting of component instantiation statements, generate statements, processes, and/or concurrent signal assignment statements.

In practice, architectures in *Warp* perform one of the following functions:

- They describe the behavior of an entity.
- They describe the data flow of an entity.
- They describe the structure of an entity.

Examples of each of these uses of an architecture are given in Section 1.5 Operators and Section 1.10 Additional Design Examples.

1.11.3 Attribute

An attribute is a property that can be associated with an entity, architecture, label, or signal in a VHDL description. This property, once associated with the entity, architecture, label, or signal, can be assigned a value, which can then be used in expressions.

1.11.3.1 Attribute Declaration

```
attribute attribute-name:type;
```

1.11.3.2 Attribute Specification

```
attribute attribute-name
    of name-list:name-class is expression;
```

1.11.3.3 Attribute Reference

```
item-name'attribute-name
```

Attributes are constants associated with names. When working with attributes, it is helpful to remember the following order of operations: declare-specify-reference.

- Declare the attribute with an attribute declaration statement.
- Associate the attribute with a name and give the attribute a value with an attribute specification statement.
- Reference the value of the attribute in an expression.

VHDL contains pre-defined and user-defined attributes.

Pre-defined attributes are part of the definition of the language. *Warp* supports a subset of these attributes that relate to synthesis operations. This subset is discussed in Section 1.11.4 Pre-Defined Attributes.

User-defined attributes are additional attributes that annotate VHDL models with information specific to the user's application. Several user-defined attributes are supplied with *Warp* to support synthesis operations.

1.11.3.4 Declaring New Attributes

To declare a new attribute, use an attribute declaration:

```
attribute smart is boolean;
attribute charm is range 1 to 10;
```

This example declares two attributes. The first is called smart, of type boolean. The second is called charm and has as possible values the integers 1 through 10, inclusive.

Associating Attributes With Names

To associate an attribute with a name and assign the attribute a value, use an attribute specification:

```
attribute smart of sig1:signal is true;
attribute charm of ent1:entity is 5;
```

This example associates the attribute smart with signal sig1, and assigns smart a value of TRUE, then associates the attribute charm with entity ent1 and assigns charm a value of 5.

Referencing Attribute Values

To use the value of an attribute in an expression, use an attribute reference:

if (sig1'smart = TRUE) then a <= 1 else a <= 0;

This example tests the value of the attribute smart for signal sig1, then assigns a value to signal a depending on the result of the test.

1.11.4 Pre-Defined Attributes

Warp supports a large set of pre-defined attributes, including value, function, type, and range attributes.

Table 1-6 lists the pre-defined attributes that *Warp* supports:

- Value attributes operate on items of scalar type or subtype.
- Function attributes operate on types, objects, or signals.
- Type attributes operate on types.
- Range attributes operate on constrained (bounded) array types.

1.11.4.1 Value Attributes

All scalar types or subtypes have the following value attributes:

- 'LEFT: returns the leftmost value in the type declaration.
- 'RIGHT: returns the rightmost value in the type declaration.
- 'HIGH: returns the highest value in the type declaration. For enumerated types, this is the rightmost value. For integer sub-range types, this is the value of the highest integer in the range. For other sub-range types, this is the rightmost value if the type declaration uses the keyword "to" or the leftmost value if the type declaration uses the keyword "downto".
- 'LOW: returns the lowest value in the type declaration. For enumerated types, this is the leftmost value. For integer sub-range types, this is the value of the lowest integer in the range. For other sub-range types, this is the leftmost value if the type declaration uses the keyword "to" or is the rightmost value if the type declaration uses the keyword "downto". Constrained array types have the following value attribute:

- ’LENGTH(N): returns the number of elements in the N’th dimension of the array.

Table 1-6 Pre-defined Attributes Supported by *Warp*

Types	Pre-defined Attributes
Value Attributes	’Left, ’Right, ’High, ’Low, ’Length
Function Attributes (types)	’Pos, ’Val, ’Succ, ’Pred, ’Leftof, ’Rightof
Function Attributes (objects)	’Left, ’Right, ’High, ’Low, ’Length
Function Attributes (signals)	’Event
Type Attributes	’Base
Range Attributes	’Range, ’Reverse_range

Constrained array objects also use these same attributes. For objects, the attributes are implemented in VHDL as functions instead of value attributes.

Examples:

For the following type declarations:

```
type countup is range 0 to 10;
type countdown is range 10 downto 0;
type months is (JAN,FEB,MAR,APR,MAY,JUN,
                JUL,AUG,SEP,OCT,NOV,DEC);
type Q1 is months range MAR downto JAN;
```

The value attributes are:

countup’left = 0	countdown’left = 10
countup’right = 10	countdown’right = 0
countup’low = 0	countdown’low = 0
countup’high = 10	countdown’high = 10
countup’length = 11	countdown’length = 11
months’left = JAN	Q1’left = MAR
months’right = DEC	Q1’right = JAN
months’low = JAN	Q1’low = JAN
months’high = DEC	Q1’high = MAR
months’length = 12	Q1’length = 3

1.11.4.2 Function Attributes (Types)

All discrete ("ordered") types and their subtypes have the following function attributes:

- ’POS(V): returns the position number of the value V in the list of values in the declaration of the type.
- ’VAL(P): returns the value that corresponds to position P in the list of values in the declaration of the type.
- ’SUCC(V): returns the value whose position is one larger than that of value V in the list of values in the declaration of the type.
- ’PRED(V): returns the value whose position is one smaller than that of value V in the list of values in the declaration of the type.
- ’LEFTOF(V): returns the value whose position is immediately to the left of that of value V in the list of values in the declaration of the type. For integer and enumerated types, this is the same as ’PRED(V). For sub-range types, this is the same as ’PRED(V) if the type was declared using the keyword "to"; it is the same as ’SUCC(V) if the type was declared using the keyword "downto".
- ’RIGHTOF(V): returns the value whose position is immediately to the right of that of value V in the list of values in the declaration of the type. For integer and enumerated types, this is the same as ’SUCC(V). For sub-range types, this is the same as ’SUCC(V) if the type was declared using the keyword "to"; it is the same as ’PRED(V) if the type was declared using the keyword "downto".

Examples:

For the following type declarations (the same as those used in the previous example set):

```
type countup is range 0 to 10;
type countdown is range 10 downto 0;
type months is (JAN,FEB,MAR,APR,MAY,JUN,
                        JUL,AUG,SEP,OCT,NOV,DEC);
type Q1 is months range MAR downto JAN;
```

the function attributes are:

countup’POS(0) = 0	countdown’POS(10) = 0
countup’POS(10) = 10	countdown’POS(0) = 10
countup’VAL(1) = 1	countdown’VAL(1) = 9
countup’VAL(9) = 9	countdown’VAL(9) = 1
countup’SUCC(4) = 5	countdown’SUCC(4) = 3
countup’PRED(4) = 3	countdown’PRED(4) = 5
countup’LEFTOF(4) = 3	countdown’LEFTOF(4) = 5
countup’RIGHTOF(4) = 5	countdown’RIGHTOF(4) = 3
months’POS(JAN) = 1	Q1’POS(JAN) = 1
months’POS(DEC) = 12	Q1’POS(MAR) = 3
months’VAL(1) = JAN	Q1’VAL(1) = MAR
months’VAL(12) = DEC	Q1’VAL(12) = error
months’SUCC(FEB) = MAR	Q1’SUCC(FEB) = MAR
months’PRED(FEB) = JAN	Q1’PRED(FEB) = JAN
months’LEFTOF(FEB) = JAN	Q1’LEFTOF(FEB) = MAR
months’RIGHTOF(FEB) = MAR	Q1’RIGHTOF(FEB) = JAN

1.11.4.3 Function Attributes (Objects)

All constrained (i.e., bounded) array objects have the following function attributes:

- ’LEFT(N): returns the left bound of the Nth dimension of the array object.
- ’RIGHT(N): returns the right bound of the Nth dimension of the array object.
- ’LOW(N): returns the lower bound of the Nth dimension of the array object. This is the same as ’LEFT(N) for ascending ranges, ’RIGHT(N) for descending ranges.
- ’HIGH(N): returns the upper bound of the Nth dimension of the array object. This is the same as ’RIGHT(N) for ascending ranges, ’LEFT(N) for ascending ranges.

In the discussion above, the value of N defaults to 1, which is also the lower bound for the number of dimensions in an array.

Examples:

For the following type and variable declarations:

```
type two_d_array is array (8 downto 0, 0 to 4);
variable my_array:two_d_array;
```

the function attributes are:

my_array'left(1)= 8	my_array'left(2) = 0
my_array'right(1) = 0	my_array'right(2) = 4
my_array'low(1) = 0	my_array'low(2) = 0
my_array'high(1) = 8	my_array'high(2) = 4

1.11.4.4 Function Attributes (Signals)

Warp supports a single function attribute for signals, namely the 'EVENT attribute. 'EVENT is a boolean function that returns TRUE if an event (i.e., change of value) has just occurred on the signal.

Warp supports the 'EVENT attribute only for clock signals such as in this example.

Example:

```
PROCESS BEGIN
    WAIT UNTIL (clk'EVENT AND clk='1');
    .
    .
    .
END PROCESS;
```

This example shows a process whose statements are executed when an event occurs on signal clk and signal clk goes to '1'.

Type Attributes

All types and subtypes have the following attribute:

- ’BASE: returns the base type of the original type or subtype.

At first glance, this attribute does not appear very useful in expressions, since it returns a type. But it can be used in conjunction with other attributes, as in the following examples.

Examples:

For the following type declarations:

```
type day_of_week is (SUN,MON,TUE,WED,THU,FRI,SAT);
subtype work_day is day_of_week range MON to FRI;
```

the following value attributes are true:

```
work_day’left = MON          work_day’BASE’left = SUN
work_day’right = FRI         work_day’BASE’right = SAT
work_day’low = MON           work_day’BASE’low = SUN
work_day’high = FRI          work_day’BASE’high = SAT
work_day’length = 5          work_day’BASE’length = 7
```

Range Attributes

Constrained array objects have the following attributes:

- ’RANGE(N): returns the range of the Nth index of the array. If N is not specified, it defaults to 1.
- ’REVERSE_RANGE(N): returns the reversed range of the Nth index of the array. If N is not specified, it defaults to 1.

The range attributes give the user a way to parameterize the limits of FOR loops and FOR-GENERATE statements, as in the following example.

Example:

Consider a variable declared as:

```
variable my_bus:std_logic_vector(0 to 7);
```

Then, the value of the ’RANGE and ’REVERSE_RANGE attributes for my_bus are:

```
my_bus’RANGE = 0 to 7
my_bus’REVERSE_RANGE = 7 downto 0
```

You could use this attribute in a FOR loop, like this:

```
for index in my_bus’REVERSE_RANGE loop
.
end loop;
```

1.11.5 CASE

The CASE statement selects one or more statements to be executed within a process, based on the value of an expression.

```
case expression is
    when case1 [| case2...] =>
        sequence_of_statements;
    when case3 [| case4...] =>
        sequence_of_statements;
            .
            .
            .]
    [when others =>
        sequence_of_statements;]
    end case;
```

In *Warp*, the expression that determines the branching path of the CASE statement must evaluate to a vector or to a discrete type (i.e., a type with a finite number of possible values, such as an enumerated type or an integer type).

The vertical bar ('|') operator may be used to indicate multiple cases to be checked in a single WHEN clause. This may only be used if the sequence of statements following the WHEN clause is the same for both cases.

The keyword OTHERS may be used to specify a sequence of statements to be executed if no other case statement alternative applies. Because CASE statements execute sequentially, the test for OTHERS should be the last test in the WHEN list.

When *Warp* synthesizes a CASE statement, it synthesizes a memory element for the condition being tested (in order to maintain any outputs at their previous values) unless one of the following conditions occurs:

- All outputs within the body of the CASE statement are previously assigned a default value within the process.
- The CASE statement completely specifies the design's behavior following any possible result of the conditional test. The best way to ensure complete specification of design behavior is to include an OTHERS clause within the CASE statement.

When a signal/variable is not assigned in all the branches of the CASE statement, Warp may have to create memory elements for such signals/variables.

Therefore, to use the fewest possible resources during synthesis, either assign default values to outputs in a process or make sure all CASE statements include an OTHERS clause.

Example:

In the following example, signal s is declared as

```
s :in std_logic_vector(0 to 2);
```

In addition, i and o are declared as eight-element vectors:

```
i : in std_logic_vector(0 to 7);
o : out std_logic_vector(0 to 7);
```

The architecture follows:

```
architecture demo of Barrel_shifter is
    begin process (s, i)
        begin
        case s is
            WHEN "000"=>
                o <= i;
            WHEN "001"=>
                o <=(i(1),i(2),i(3),i(4),i(5),i(6),i(7),i(0));
            WHEN "010"=>
                o <=(i(2),i(3),i(4),i(5),i(6),i(7),i(0),i(1));
            WHEN "011"=>
                o <=(i(3),i(4),i(5),i(6),i(7),i(0),i(1),i(2));
            WHEN "100"=>
                o <=(i(4),i(5),i(6),i(7),i(0),i(1),i(2),i(3));
            WHEN "101"=>
                o <=(i(5),i(6),i(7),i(0),i(1),i(2),i(3),i(4));
            WHEN "110"=>
                o <=(i(6),i(7),i(0),i(1),i(2),i(3),i(4),i(5));
            WHEN "111"=>
                o <=(i(7),i(0),i(1),i(2),i(3),i(4),i(5),i(6));
            end case;
        end process;
end demo;
```

In this example, signal s evaluates to a 3- element bit-string literal. The appropriate statement is executed, depending on the value of s, and output vector o gets the value given by the specified ordering of elements in input vector i. Note that the CASE statement completely specifies the results of the conditional test; all possible values of s are covered by a WHEN clause. Hence, no OTHERS clause is needed.

1.11.6 Component

A component declaration specifies a component to be synthesized and lists the local signal names of the component. The component declaration serves the same purpose in VHDL as a function declaration or prototype serves in the C programming language. Ports and

Generics in the component declaration must match with those in the entity declaration.

Component Declaration

```
component identifier
    [generic (generic_list);]
    [port (port_list);]
    end component;
```

Example:

```
    component barrel_shifter port(
        clk : IN std_logic;
        s :in std_logic_vector(0 to 2);
        insig :in std_logic_vector(0 to 7);
        outsig :out std_logic_vector(0 to 7));
    end component;
```

This example declares a component called barrel_shifter with a 3-bit input signal, an 8-bit input signal, and an 8-bit output signal.

Component Instantiation

```
instantiation_label:component_name
    [generic generic_mapping]
    [port port_mapping];
```

A component instantiation creates an instance of a component that was previously declared with a component declaration statement. Think of component instantiation as "placing" a previously declared component into an architecture, then "wiring" the newly placed component into the design by means of the generic map or port map.

Example:

```
    a1:barrel_shifter
        port map(
            clk=>clock,
            s(0)=>s0,
            s(1)=>s1,
            s(2)=>s2,
            insig=>myinput,
            outsig=>myoutput);
```

The line a1:barrel_shifter in the example above instantiates a component named a1 of type barrel-shifter. The port map statement that follows maps each signal from this instance of barrel_shifter to signals at a higher level.

Note that, in this case, the barrel_shifter was defined as accepting a single bit signal clk, and three vectors s, insig and outsig. For vectored signals, you can either split the vector up into individual signals, as in the case of s or perform direct vector to vector connections.

Note also the direction of the "arrow" in each mapping: from the "formal" (the signal name on the component) to the "actual" (the name of the pin to which the signal is being mapped).

1.11.7 Constant

A constant is an object whose value may not be changed.

constant identifier_list:type[:=expression];

Example:

```
TYPE stvar is std_logic_vector(0 to 1);
constant s0:stvar := "00";
constant s1:stvar := "01";
constant s2:stvar := "10";
constant s3:stvar := "11";
```

This example declares a vector subtype with length 2, called stvar. It then defines four constants of type stvar, and gives them values of "00", "01", "10", and "11", respectively.

```
subtype vec8 is std_logic_vector(0 to 7);
type v8_table is array(0 to 7) of vec8;
constant xtbL1:v8_table := (
"00000001",
"00000010",
"00000100",
"00001000",
"00010000",
"00100000",
"01000000",
"10000000");
```

This example declares a vector subtype with length 8 called vec8, a 1-dimensional array type of vec8 called v8_table with eight elements, and a constant of type v8_table called xtbL1.

Values are assigned to the constant by concatenating a sequence of string literals (e.g., "00000001") into vector form. Only characters ’0’, ’1’, ’Z’ and ’-’ are allowed in these string literals, but the values could have been written in hex format (e.g., x"01" is the same as "00000001" for this purpose). The result is a table of constants such that xtbL1(0) is "00000001" and xtbL1(7) is "10000000".

1.11.8 Entity

An entity declaration names a design entity and lists its ports (i.e., external signals). The mode and data type of each port are also declared.

```
entity identifier is port(
    port_name: mode type [;
    port_name: mode type...])
    end [identifier];
```

Choices for mode are IN, OUT, BUFFER, and INOUT.

Example:

```
    entity Barrel_Shifter is port(
        clk : IN std_logic;
        s :in std_logic_vector(0 to 2);
        insig :in std_logic_vector(0 to 7);
        outsig :out std_logic_vector(0 to 7));
    end Barrel_Shifter;
```

This example declares an entity called barrel_shifter with a 3-bit and an 8-bit input signal as well as an 8-bit output signal.

1.11.9 Exit

The EXIT statement causes a loop to be exited. Execution resumes with the first statement after the loop. The conditional form of the statement causes the loop to be exited when a specified condition is met.

```
exit [loop_label] [when condition];
```

Example:

```
    i <= 0;
    loop
        outsig(i)<=barrel_mux8(i,s,insig);
        i <= i+1;
        exit when i>7;
        end loop;
```

The EXIT statement in the example above causes the loop to exit when variable i becomes greater than 7. The example thus calls the function barrel_mux8 eight times.

1.11.10 Generate

Generate statements specify a repetitive or conditional execution of the set of concurrent statements they contain. Generate statements are especially useful for instantiating an array of components.

```
label:generation_scheme generate
      {concurrent_statement}
            end generate [label];

      generation_scheme ::=
            for generate_parameter_specification
      | if condition
```

A "generation scheme" in the syntax above refers to either a FOR-loop specification or an IF-condition specification, as shown in the example below.

Example:

```
architecture test of serreg is
      signal zero : std_logic := ’0’ ;
begin
      m1: for i in 0 to size-1 generate
            m2: if i=0 generate
                  x1:dsrff port map (si, zero, mreset, clk, q(0));
            end generate;
            m3: If i>0 generate
                  x2:dsrff port map(q(I-1), zero, mreset, clk, q(I));
            end generate;
      end generate;
end test;
```

The example above instantiates a single component labeled x1, and size-1 components labeled x2. For size=3, for instance, the code shown above is the equivalent of

```
architecture test of serreg is
      signal zero : std_logic := ’0’ ;
begin
      x1:dsrff port map(si, zero, mreset, clk, q(0));
      x2:dsrff port map(q(0), zero, mreset, clk, q(1));
      x3:dsrff port map(q(1), zero, mreset, clk, q(2));
end test;
```

1.11.11 Generic

Generics are the means by which instantiating (parent) components pass information to instantiated (child) components in VHDL. Typical uses are to specify the size of array objects or the number of subcomponents to be instantiated.

```
      generic(identifier:type[:=value]);
```

Example:

```
      component serreg
            generic (size:integer:=8);
```

```
    port (si,
          clk,
          mreset : in std_logic;
          q : inout std_logic_vector(0 to size-1));
    end component;
```

This example declares a component with a bidirectional array of 8 bits, among other signals. The number of bits is given by the value of the size parameter in the generic statement.

1.11.12 If-Then-Else

The IF statement selects one or more statements to be executed within a process, based on the value of a condition.

```
IF condition THEN sequence_of_statements
   [ELSIF condition THEN
      sequence_of_statements...]
   [ELSE sequence_of_statements]
      END IF;
```

A condition is a boolean expression, i.e., an expression that resolves to a boolean value. If the condition evaluates to true, the sequence of statements following the THEN keyword is executed. If the condition evaluates to false, the sequence of statements following the ELSIF or ELSE keyword(s), if present, are executed.

When *Warp* synthesizes an IF-THEN-ELSE statement, it synthesizes a memory element for the condition being tested (in order to maintain any outputs at their "previous" values) unless one of the following conditions is true:

- All outputs within the body of the IF-THEN-ELSE statement are previously assigned a "default" value within the process.
- The IF-THEN-ELSE statement specifies the design's behavior following any possible result of the conditional test. The best way to ensure complete specification of design behavior is to include an ELSE clause within the IF statement.

When a signal/variable is not assigned in all the branches of the IF-THEN-ELSE statement, Warp may have to create memory elements for such signals/variables. This could use up more PLD resources than would otherwise be required.

In short, to use the fewest possible PLD resources during synthesis, either assign default values to outputs in a process, or make sure all IF-THEN-ELSE statements include ELSE clauses.

Example:

```
   if (not tmshort=one)then stvar <= ewgo;
```

```
elsif (tmlong=one)then stvar <= ewwait;
elsif (not ew=one AND ns=one)then stvar<=ewgo;
elsif (ew=one AND ns=one)then stvar <= ewwait;
elsif (not ew=one)then stvar <= ewgo;
else stvar <= ewwait;
end if;
```

Note – In the case above, the ELSE statement is to be left out, as opposed to suggestions made on the previous page.

The example above sets a state variable called stvar. The value that stvar receives depends on the value of variables tmshort, tmlong, ew, and ns.

Asynchronous Sets, Resets, Loads

Use an IF-THEN-ELSE statement to synthesize the operation of a synchronous circuit containing asynchronous sets or resets.

To do so, use a sensitivity list in the PROCESS statement, naming the sets, resets, and clock signals that will trigger the execution of the process. Then, use a sequence of IF-ELSIF clauses to specify the behavior of the circuit.

The structure of the process should be something like this:

```
process (set, reset, clk) begin
    if (reset = ’0’)then
        --assign signals to their "reset" values;
    elsif (set = ’0’)then
        --assign signals to their "set" values;
    elsif (load = ’1’)then
        assign signals an aysnchronous load value;
    elsif (clk’EVENT AND clk=’1’)then
        --perform synchronous operations;
    end if;
end process;
```

The example above shows the VHDL description for active-low set and reset signals. It could just as easily have been coded for active-high sets and resets by using the conditions set=’1’ and reset=’1’.

The assignments made in the statements that follow the set or reset signal conditions must be "simple" assignments (i.e., of the form *name=constant)*, to a signal of type `std_logic`, `std_logic_vector`, or enumerated type (for state variables). If the value being assigned is a non-constant, it is treated as an asynchronous load.

1.11.13 Library

In *Warp*, a library is a storage facility for previously analyzed design units.

library library-name [, library-name];

A library clause declares logical names to make libraries visible within a design unit. A design unit is an entity declaration, a package declaration, an architecture body, or a package body.

Example:

```
library mylib;
```

The above example makes a library named *mylib* visible within the design unit containing the library clause.

1.11.14 Loops

Loops execute a sequence of statements repeatedly.

```
[loop_label:] [iteration_scheme] loop
    sequence_of_statements
    end loop [loop_label];
iteration_scheme ::=
    while condition
    | for loop_parameter in
        {lower_limit to upper_limit
        |upper_limit downto lower_limit}
```

There are three kinds of loops in VHDL:

- Simple loops are bounded by a loop/end loop statement pair. These kinds of loops require an exit statement, otherwise they execute forever;
- FOR loops execute a specified number of times; and
- WHILE loops execute while a specified condition remains true.

Examples:

Simple loop:

```
    i := 0;
    loop
        outsig(i)<=barrel_mux8(i,s,insig);
        i := i+1;
        if (i>7) then
            exit;
            END IF;
        end loop;
```

FOR loop:

```
    for i in 0 to 7 loop
        outsig(i)<=barrel_mux8(i,s,insig);
        end loop;
```

WHILE loop:

```
    i := 0;
    while (i<=7) loop
        outsig(i)<=barrel_mux8(i,s,insig);
        i := i+1;
        end loop;
```

The examples above show three ways of implementing the same loop. All of these loops call the function barrel_mux8 eight times. (In all three, i must be defined as a variable.)

1.11.15 Next

NEXT advances control to the next iteration of a loop.

next [loop_label] [when condition];

Example:

```
for i in 0 to 7 loop
    if ((i =0) or (i=2) or (i=4) or (i=6)) then
        outsig(i)<=barrel_mux8(i,s,insig)
    else
        next i;
    end if;
end loop;
```

The example above performs an operation on the even-numbered bits of an 8-element std_logic_vector. It uses a NEXT statement to advance to the next iteration of the loop for the odd-numbered bits.

1.11.16 Package

A VHDL package is a collection of declarations that can be used by other VHDL descriptions. A VHDL package consists of two parts: the package declaration and the package body.

Package Declaration

```
package identifier is
    function_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    [;{function_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause}...]
    end [identifier];
```

Package Body

```
package body identifier is
        {function_declaration
        | function_body
        | type_declaration
        | subtype_declaration
        | constant_declaration
        | use_clause}
        [;{function_declaration
        | function_body
        | type_declaration
        | subtype_declaration
        | constant_declaration
        | use_clause}...]
        end [identifier];
```

The package declaration declares parts of the package that can be used by other VHDL descriptions, i.e., by other designs that use the package.

The package body provides definitions and additional declarations, as necessary, for functions whose interfaces are declared in the package declaration.

Example: (package declaration)

```
        package bv_tbl is
        subtype vec8 is std_logic_vector(0 to 7);
        type v8_table is array(0 to 7) of vec8;

        -- defining vectors for i2bv8 function
        constant xtbl1:v8_table := (
        "00000001",
        "00000010",
        "00000100",
        "00001000",
        "00010000",
        "00100000",
        "01000000",
        "10000000");
        -- function declaration
        function i2bv8(ia:integer) return vec8;
        subtype vec3 is std_logic_vector(0 to 2);
        type v3_table is array(0 to 7) of vec3;

        -- defining vectors for i2bv3 function
        constant xtbl2:v3_table := (
        "000",
        "001",
"010",
"011",
"100",
"101",
"110",
"111"
);

--function declaration
function i2bv3(ia:integer) return vec3;

end bv_tbl;
```

The example above declares several types, subtypes, constants, and functions. These items become available for use by any VHDL description that uses package bv_tbl.

Example: (package body)

```
package body bv_tbl is

function i2bv8(ia:integer) return vec8 is
-- translates an integer between 1 and 8
-- to an 8-bit vector
begin
return xtbl1(ia);
end i2bv8;

function i2bv3(ia:integer) return vec3 is
-- translates an integer between 1 and 8
-- to a three-bit vector
begin
    return xtbl2(ia);
end i2bv3;

end bv_tbl;
```

The example above defines two functions whose declarations appeared in the package declaration example, shown previously.

1.11.17 Port Map

A port map statement associates the ports of a component with the pins of a physical part.

```
port map ([formal_name =>] actual_name
            [,[formal_name =>] actual_name]);
```

The port map statement associates ports declared in a component declaration, known as *formals*, with the signals (known as *actuals*) being passed to those ports.

If the signals are presented in the same order as the formals are declared, then the formals need not be included in the port map statement.

Port map statements are used within component instantiation statements. See Section 1.11.6 Component, for more information about component instantiation statements.

Example:

```
and_1: AND2
    port map(A => empty_1,
                B => empty_2,
                Q => refill_bin);
```

The example above instantiates a two-input AND gate. The port map statement associates three signals (empty_1, empty_2, and refill_bin, respectively) with ports A, B, and Q of the AND gate.

If the three ports appear in the order A, B, and Q in the AND2 component declaration, the following (shorter) component instantiation would have the same effect:

```
and_1: AND2
    port map(empty_1,empty_2,refill_bin);
```

Cypress recommends, however, using the name association method instead of positional association for both port maps and generic maps. In general, name association is a lot more readable, less error prone, and insulates the code from changes to lower level components to a certain extent.

1.11.18 Generic Map

A generic map statement associates the generics of a component with the values defined at an upper level architecture.

```
generic map ([formal_name =>] actual_name
            [,[formal_name =>] actual_name]);
```

The generic map statement associates generics declared in a component declaration, known as *formals*, with the values (known as *actuals*) being passed to those generics.

If the values are being passed in the same order as the formals declared, then the formals need not be specified in the generic map.

Generic map statements are used within component instantiation statements. See Section 1.11.6 Component for more information about component instantiation statements.

When using association pairs (formals being specified in the generic map), the user can omit the specification of certain generics if those generics have defaults defined for them in the component declaration.

Example:

```
u0: MADD_SUB
    generic map(lpm_width => myOutput'length,
                lpm_representation => lpm_unsigned,
                lpm_direction => lpm_add,
                lpm_hint => speed) ;
    port map(dataA => myDataA, dataB => myDataB,
                cin => zero, add_sub => one,
                result => myOutput, cout => open,
                overflow => open) ;
```

This example instantiates an add/sub component defined in the LPM library supplied with Warp. In the instantiation, the Madd_sub LPM modules is being configured as an adder only without a carry-in. The carry-out or the overflow outputs are not being used.

Since the Madd_sub moduled defined in the LPM library provided a default for lpm_hint (speed), this association could have been omitted in the generic map. The very first generic lpm_width has no default value, however, and must be specified. The choice of what can be omitted and what has to be defined depends on the component declaration.

Using positional association, the same example could have been written as

```
    u0: MADD_SUB
        generic map(myOutput'length, lpm_unsigned,
                    lpm_add, speed) ;
        port map(myDataA,myDataB, zero, one,
                    myOutput, open, open) ;
```

Cypress recommends using the name association method instead of positional association for both port maps and generic maps. In general, name association is a lot more readable, less error prone, and insulates the code from changes to lower level components to a certain extent.

1.11.19 Process

A process statement is a concurrent statement that defines a behavior to be executed when that process becomes active. The behavior is specified by a series of sequential statements executed within the process.

```
[label:] process [(sensitivity_list)]
    [process_declarative_part]
    begin
        sequential_statement
        [;sequential_statement...];
    end process [label];

process_declarative_part ::=
    function_declaration
    | function_body
    | type declaration
    | subtype declaration
    | constant declaration
    | variable declaration
    | attribute specification

sequential_statement ::=
    wait_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement
```

A process which is executing is said to be active; otherwise, the process is said to be suspended. Every process in the VHDL description may be active at any time. All active processes are executed concurrently with respect to simulation time.

Processes can be activated in one of two ways: by means of a WAIT statement or by means of a sensitivity list (a list of signals enclosed in parentheses appearing after the process keyword).

When a process includes a WAIT statement, the process becomes active when the value of the clock signal goes to the appropriate value ('0' or '1').

When a process includes a sensitivity list, the process becomes active when one or more of the signals in the list changes value.

Example:

```
process (s, insig) begin
    for i in 0 to 7 loop
        outsig(i)<=barrel_mux8(i,s,insig);
        end loop;
    end process;
```

The example above shows a process that executes whenever activity is sensed on either of two signals, s or insig.

1.11.20 Signal

A signal is a pathway along which information passes from one component in a VHDL description to another. Signals are also used within components to pass values to other signals, or to hold values.

Signal Declaration

```
signal name [, name ...]:type;
```

Signal Assignment)

```
signal_name <= expression
    [when condition [ else expression]];
```

Signals must be declared before they can be used. Declaring a signal gives it a name and a type. Signal declarations often appear as part of port or component declarations.

To assign a value to a signal, replace its current value with the value of some expression of the same type as the signal, using the signal name and the "<=" operator.

The user may also specify a condition under which the replacement is to be made, as well as an alternative value for the signal if the condition is not met. This form of the signal assignment statement uses the "when" and "else" keywords.

Example:

(Signal Declaration examples)

```
signal c0,c1,cin1,cin2:std_logic;
```

This example declares four signals (c0, c1, cin1, and cin2), each of type std_logic.

```
type State is (s1, s2, s3, s4, s5);
signal StVar : State;
```

This example declares an enumerated type named State, with five possible values. It then declares a signal named StVar of type State. Thus, StVar can have values s1, s2, s3, s4, or s5.

(Unconditional Signal Assignment examples)

```
c_in <= '1';
```

This example sets a variable of type std_logic named c_in to ’1’.

```
StVar <= s1;
```

This example assigns the value s1 to a signal named StVar. Presumably, StVar is a signal of some enumerated type, having s1 as one of its possible values.

(Conditional Signal Assignment example)

```
c_in2 <= '1' when (stvar=s1) OR (stvar=s2) else '0';
```

The above example illustrates the use of conditional signal assignment. Signal c_in2 is assigned one value if the specified conditions are met; otherwise, it is assigned a different value.

1.11.21 Subprograms

Subprograms are sequences of declarations and statements that can be invoked repeatedly from different parts of a VHDL description. VHDL includes two kinds of subprograms: procedures and functions.

Procedure Declaration

procedure designator [(formal-parameter-list)];

Procedure Body

```
procedure designator [(formal-parameter-list)] is
        [declarations]
    begin
        {sequential-statement;}
    end [designator];
```

Function Declaration

```
function designator [(formal-parameter-list)]
    return type_mark;
type_mark ::= type_name | subtype_name
```

Function Body

```
function designator [(formal-parameter-list)]
    return type_mark is
        [declarations]
    begin
        {sequential-statement;}
    end [designator];
```

A subprogram is a set of declarations and statements that can be used repeatedly from many points within a VHDL description.

There are two kinds of subprograms in VHDL: procedures and functions. Procedures may return zero or more values. Functions always return a single value. In practice, procedures are most often used to sub-divide a large behavioral description into smaller, more modular sections. Functions are most often used to convert objects from one data type to another or to perform frequently needed computations.

VHDL allows the user to declare a subprogram in one part of a VHDL description while defining it in another. Subprogram declarations contain only interface information: name of the subprogram, interface signals, and return type (for functions). The subprogram body contains local declarations and statements, in addition to the interface information.

Function calls are expressions; the result of a function call is always assigned to a signal or variable, or otherwise used in a larger statement (e.g., as a parameter to be passed to a procedure call). Procedure invocations, by contrast, are entire statements in themselves.

In both procedures and functions, actual and formal parameters may use positional association or named association.

To use positional association, list the parameters to be passed to the subprogram in the same order that the parameters were declared, without naming the parameters.

To use named association, give the formal parameter (the name shown in the subprogram declaration) and the actual parameter (the name passing to the subprogram) within the procedure invocation or function call, linking the formal and actual parameters with the "=>" operator. In named association, parameters can be listed in any order.

Example:

Consider a procedure, whose declaration is shown below, that takes three signals of type `std_logic` as input parameters:

```
    procedure crunch(signal sig1,sig2,sig3:in std_logic);
```

Then, the invocation

```
    crunch(trex,raptor,spitter);
```

uses positional association to map signal `trex` to `sig1`, signal `raptor` to `sig2`, and signal `spitter` to `sig3`. You could use named association in the following procedure invocation, however, and get the same result:

```
    crunch(sig2=>raptor,sig3=>spitter,sig1=>trex);
```

Procedures and functions are described in the next sections.

1.11.21.1 Procedures

Procedures describe sequential algorithms that may return zero or more values. Procedures are most frequently used to decompose large behavioral descriptions into smaller, more modular sections, which can be used by many processes.

Procedure parameters may be constants, variables, or signals, and their modes may be in, out, or inout. Unless otherwise specified, a parameter is by default a constant if it is of mode in, and a variable if it is of mode out or inout.

In general, procedures can be used both concurrently (outside of any process) and sequentially (inside a process). If any of the procedure parameters are variables, the procedure can only be used sequentially, as variables are only defined inside a process. Variables declared inside a procedure cease to exist when the procedure terminates.

A procedure body can contain a wait statement, while a function body cannot; however, a process that calls a procedure with a wait statement in it cannot have a sensitivity list. Processes can not be sensitive to signals and made to wait simultaneously.

1.11.21.2 Functions

Functions describe sequential algorithms that return a single value. Their most frequent uses are: (1) to convert objects from one data type to another; and (2) as shortcuts to perform frequently used computations.

Function parameters must be of mode in and must be signals or constants. If no mode is specified for a function parameter, the parameter is interpreted as having mode in.

A function body cannot contain a wait statement. (Functions are only used to compute values that are available instantaneously.)

Any variables declared inside a function cease to exist when the function terminates (i.e., returns its value).

One common use of functions in VHDL is to convert objects from one data type to another. *Warp* provides several conversion functions in the various packages for type conversions (integer to vector, vector to integer etc.). Functions can also be used to overload operators or perform simple combinatorial functions.

Example:

```
function count_ones(vec1:std_logic_vector)
    return integer is
-- returns the number of ’1’ bits in a std_logic vector
    variable temp:integer:=0;
    begin
        for i in vec1’low to vec1’high loop
            if vec1(i) = ’1’ then
                temp := temp+1;
                end if;
            end loop;
        return temp;
    end count_ones;
```

This function counts the number of ’1’s in a vector.

1.11.22 Type

In VHDL, objects are anything that can hold a value. Signals, constants, or variables are common objects. All VHDL objects have a type, which specifies the kind of values that the object can hold.

Enumerated Type Declaration

```
type name is (value [,value...]);
```

Subtype Declaration

```
subtype name is base_type
    [range {lower_limit to upper_limit
            | upper_limit downto lower_limit}];
```

Vector Declaration

```
subtype name is std_logic_vector
      (lower_limit to upper_limit
      | upper_limit downto lower_limit);
```

Record Declaration

type name is record
name:type;
[name : type;...]
end record;

Warp has the following pre-defined types:

- integer: VHDL allows each implementation to specify the range of the integer type differently, but the range must extend from at least -(2**31-1) to +(2**31-1), or -2147483647 to +2147483647. Only variables (not signals) can have type integer.
- boolean: an enumerated type, consisting of the values "true" and "false";
- std_logic: an enumerated type, consisting of the values '0','1', ’Z’, ’-’. std_logic in its strictest definition also has values of ’H’, ’L’, ’U’, ’X’ and ’W’ which should not be used in VHDL that is intended for synthesis.
- character: an enumerated type, consisting of one literal for each of the 128 ASCII characters. The non-printable characters are represented by two or three-character identifiers, as follows: NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL, BS, HT, LF, VT, FF, CR, SO, SI, DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB, CAN,EM, SUB, ESC, FSP, GSP, RSP, and USP.
- VHDL objects can take other, user-defined, types as well. Possibilities include:
- enumerated types: This type has values specified by the user. A common example is a state variable type, where the state variable can have values labeled state1, state2, state3, etc.
- sub-range types: This type limits the range of a larger base type (such as integers) to a smaller set of values. Examples would be positive integers, or non-negative integers from 0 to 100, or printable ASCII characters.
- arrays (especially vectors): This type specifies a collection of elements of a single base type. A commonly used example is the std_logic_vector type, declared in the std_logic_1164 package from IEEE, which denotes an array of std_logic bits.
- records: This type specifies a collection of elements of possibly differing types.

Enumerated Type Declaration Example

```
type sigstates is (nsgo, nswait, nswait2,
      nsdelay, ewgo, ewwait, ewwait2, ewdelay);
```

This example declares an enumerated type and lists eight possible values for it.

Sub-range Type Declaration Example

```
type column is range (1 to 80);type row is
      range (1 to 24);
```

These examples declare two new sub-range integer types, column and row. Legal values for objects of type column are integers from 1 to 80, inclusive. Legal values for objects of type row are integers from 1 to 24, inclusive.

Vector Type Declaration Example

```
subtype vec8 is std_logic_vector(0 to 7);
signal insig, outsig : vec8;
.
.
insig <= "00000010";
.
.
outsig<=insig(2 to 7) & insig(0 to 1);
```

This example declares a vector type called vec8. It then declares two signals of type vec8, insig and outsig. The signal assignment statement that concludes the example left-shifts insig by two places. (Outsig then contains the value "00001000".)

Record Type Declaration Example

```
subtype vec5 is std_logic_vector(1 to 5);
-- define a record type containing a 5-bit vector
-- and a single bit
type arec is record
      abc:vec5;
      def:std_logic;
      end record;
-- define a type containing
-- an array of five records
type twodim is array (1 to 5) of arec;
-- now define a couple of signals
signal v:twodim;
signal vrec:arec;
.
.
```

```
-- in record 1 of v, set array field to all 0's
v(1).abc <= "00000";

-- in record 2 of v, set first three bits to 0's,
-- others to 1's
v(2).abc <= ('0', '0', '0', others => '1');

-- set fifth bit in array within record #5 to 0.
v(5).abc(5) <= '0';

-- set bit field within record to 1
vrec.def <= '1';

-- set 3rd bit within vrec's array to 1
vrec.abc(3) <= '1';
```

The example above defines a record type and a type consisting of an array of records. The example then demonstrates how to assign values to various elements of these arrays and records.

1.11.23 USE

The USE statement makes items declared in a package visible inside the current design unit. A design unit is an entity declaration, a package declaration, an architecture body, or a package body.

use name {, name};

Example:

```
use work.cypress.all;
use work.rtlpkg.all;
use work.brl4pkg.all;
```

The example above makes the items declared in the specified packages available for use in the design unit in which the use statements are contained.

Note – The scope of a USE statement extends only to the design unit (entity, architecture, or package) that it immediately precedes.

1.11.24 Variable

A variable is a VHDL object (similar to a signal) that can hold a value.

Variable declaration

```
variable name [, name ...]:type [:=expression];
```

Variable assignment

```
signal_name <= expression;
```

Variables differ from signals in that variables have no direct analogue in hardware. Instead, variables are simply used as indices or value-holders to perform the computations incidental to higher-level modeling of components.

Example:

```
function admod(i,j,max:integer) return integer is
    variable c:integer;
    begin
    c:=(i+j) mod (max+1);
    return c;
    end admod;
```

The function in the example above declares a variable c as a value-holder for a computation, then uses it and returns its value in the body of the function.

1.11.25 Wait

The WAIT statement suspends execution of a process until the specified condition becomes true.

```
wait until [condition];
```

A WAIT statement, if used, must appear as the first sequential statement inside a process.

Example:

```
wait until clk='1';
```

The example above suspends execution of the process in which it is contained until the value of signal clk goes to ’1’.

1.12 List of VHDL Reserved Words

The following list of reserved words should not be used for object (signals, variables, entities, or libraries) names.

Table 1-7 VHDL Reserved Words

abs	downto	library	postponed	srl
access	else	linkage	procedure	subtype
after	elsif	literal	process	then
alias	end	loop	pure	to
all	entity	map	range	transport
and	exit	mod	record	type
architecture	file	nand	register	unaffected
array	for	new	reject	units
assert	function	next	rem	until
attribute	generate	nor	report	use
begin	generic	not	return	variable
block	group	null	rol	wait
body	guarded	of	ror	when
buffer	if	on	select	while
bus	impure	open	severity	with
case	in	or	signal	xnor
component	inertial	others	shared	xor
configuration	inout	out	sla	
constant	is	package	sll	
disconnect	label	port	sra	

In addition to the reserved words listed by the language, also avoid declaring objects (signals, variables, entities, or libraries) whose names can conflict with objects declared in Cypress supplied packages, such as those in the files *$CYPRESS_DIR\lib\common\cypress.vhd* or *rtlpkg.vhd*.