VHDL Reference Manual

096-0400-003

March 1997

SYNARIO®
Design Automation

Synario Design Automation, a division of Data I/O, has made every attempt to ensure that the information in this document is accurate and complete. Synario Design Automation assumes no liability for errors, or for any incidental, consequential, indirect or special damages, including, without limitation, loss of use, loss or alteration of data, delays, or lost profits or savings, arising from the use of this document or the product which it accompanies.

Synario Design Automation
10525 Willows Road N.E., P.O. Box 97046
Redmond, Washington 98073-9746 USA
Corporate Switchboard: (206) 881-6444
Sales: 1-888-SYNARIO or edasales@data-io.com
Technical Support: 1-800-789-6507 or sts@data-io.com
World Wide Web: www.synario.com

Acknowledgments:

Table of Contents

1. Introduction

This manual discusses VHDL and the Synario Programmable IC Solution. This manual is intended to supplement the material presented in the *Programmable IC Entry* manual.

The following topics are discussed in this manual:

- VHDL Language Structure
- How to write Synthesizable VHDL
- How to control the implementation of a VHDL Design
- VHDL Datapath Synthesis
- How to Manage VHDL Design Hierarchies
- VHDL Quick Reference
- Limitations (Constraints and unsupported Constructs)
- VHDL for ABEL-HDL users
- ABEL-HDL Language Reference (Dot extensions)

2. Language Structure

VHDL is a hardware description language (HDL) that contains the features of conventional programming languages such as Pascal or C, logic description languages such as ABEL-HDL, and netlist languages such as EDIF. VHDL also includes design management features, and features that allow precise modeling of events that occur over time.

This chapter introduces a subset of the VHDL language that allows you to begin creating synthesizable designs, and is not intended to describe the full language. For further information on VHDL, consult a standard VHDL reference book. A number of these books are listed at the end of this chapter.

The VHDL Synthesizer supports most of the VHDL language, as described in IEEE Standard 1076-1993. The meaning of some sections of the language, however, is unclear in the context of logic synthesis. Examples of this are found in the standard package **textio**. The file I/O operations supported by **textio** are useful for simulation purposes but are not currently synthesizable.

- For sample syntax and a list of VHDL statements supported by the VHDL Synthesizer, see Appendix A, Quick Reference.
- For a list of exceptions and constraints on the VHDL Synthesizer's support of VHDL, see Appendix B, Limitations.

This chapter shows you the structure of a VHDL design, and then describes the primary building blocks of VHDL used to describe typical circuits for synthesis:

- Library (Design) Units
- Statements
- Objects
- Types
- Operators
- Attributes

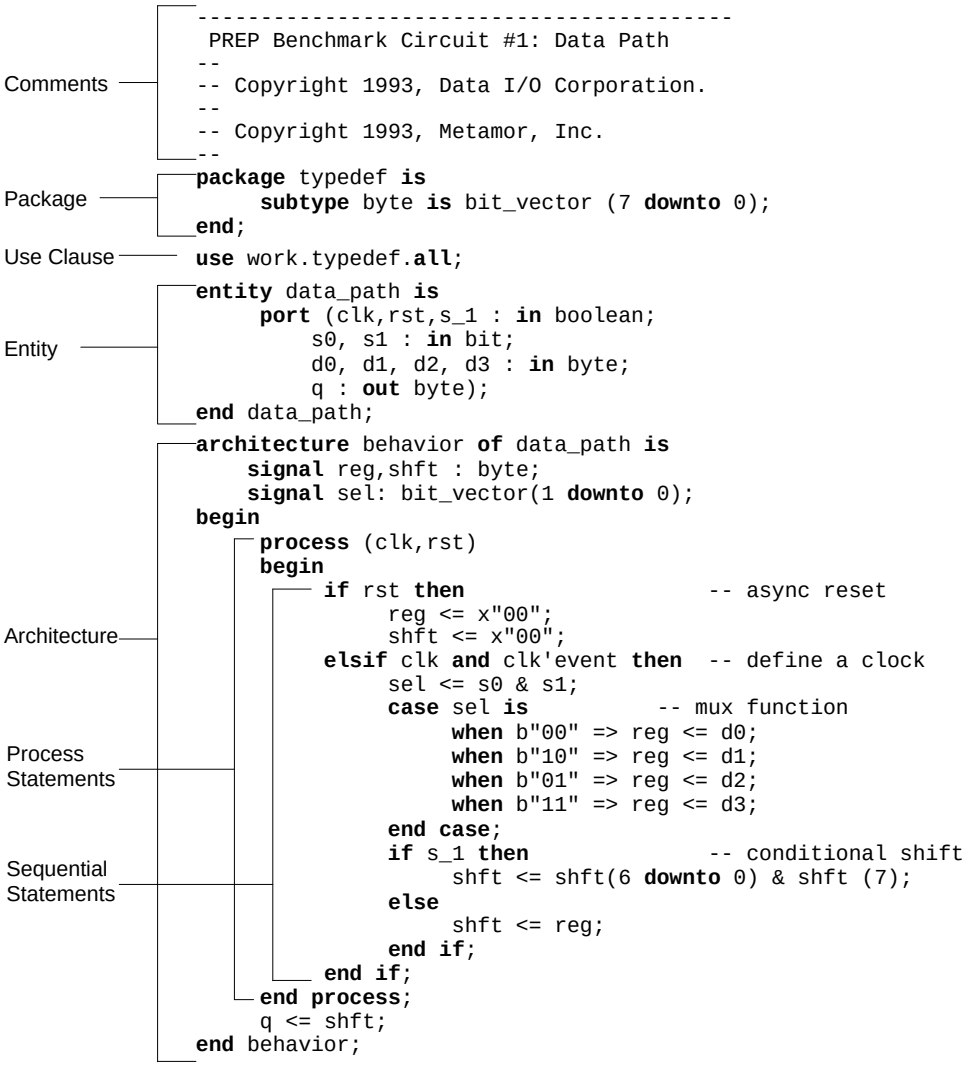
In addition, the three primary methods of VHDL design are discussed:

- Dataflow VHDL
- Behavioral VHDL
- Structural VHDL

Structure of a VHDL Design Description

The basic organization of a VHDL design description is shown in **Figure 2-1**. The sample file shown includes an entity-architecture pair and a package.

Figure 2-1: The Structure of a VHDL Design Description



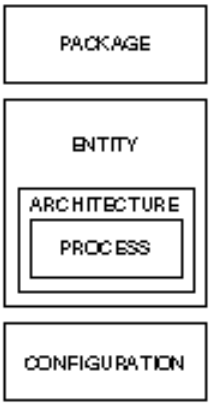
Library Units

Library units (also known as design units) are the main components of a VHDL description. They consist of the following kinds of declarations:

- Package (optional)
- Entity
- Architecture
- Configuration (optional)

A design may include any number of package, entity, architecture, and configuration declarations. The relationship of the four types of design units is illustrated in **Figure 2-2**. Note that only the entity and architecture design units are required; the package and configuration design units are optional.

Figure 2-2: Relationship of VHDL design units



Package

A package is an optional library unit used for making shared definitions. An example of something that might be shared is a type definition, as shown in **Figure 2-1**. When you make definitions in a package, you must use the **library** and **use** statements to make the package available to other parts of the VHDL design.

```
package example_arithmetic is
    type small_int is range 0 to 7;
end example_arithmetic;
```

Entity

Entities contain the input and output definitions of the design. In VHDL designs that contain a hierarchy of lower-level circuits, the entity functions very much like a block symbol on a schematic. An entity usually has one or more ports, which are analogous to the pins on a schematic symbol. All information must flow into and out of the entity through the ports, as shown:

```
library my_lib;
use my_lib.example_arithmetic.all;

entity ent is
    port (a0,a1,b0,b1 : in small_int; c0,c1 : out small_int);
end ent;
```

Note that this example references the package defined in the previous section to gain access to the type small_int. Each **port** has a mode that defines a direction: **in**, **out**, **inout**, or **buffer**.

Modes **in**, **out**, and **inout** all have the obvious meanings. Ports declared to be of type **out** may not be read. Therefore, the assignment:

```
c1 <= c0;
```

would be illegal since c0 is declared to be an **out** port. Mode **buffer** is equivalent to mode **out** except that the value of the port may be read within the entity.

In addition to ports, entities may also contain generics. Generics are similar to ports, except that they pass static information. You can use generics to create two or more instances of an entity where the instances behave in different ways. A common use of generics is in gate-level modeling, where generics pass delay values into the model, as shown:

```
library my_lib;
use my_lib.example_arithmetic.all;

entity ent is
    generic (t_rise, t_fall : time := 5 ns);
    port (a0,a1,b0,b1 : in small_int; c0,c1 : out small_int);
end ent;
```

The preceding example specifies a rise and fall delay using the pre-defined type **time**, and gives the delays a default value of 5 ns. Note that if you use generics when writing code for synthesis, all generic parameters must be given default values.

Architecture

The **architecture** is the actual description of the design. If you think of an entity as a functional block symbol on a schematic, then an **architecture** describes what's inside the block. An **architecture** can contain both concurrent and sequential statements, which are described below. Note that VHDL allows you to have more than one architecture for the same entity. For example, you might have an architecture for synthesis and a gate-level (netlist) architecture. If you have more than one architecture for an entity, use configuration declarations to determine which architecture to use for synthesis or simulation.

An architecture consists of two pieces: the architecture declaration section and the architecture body. Consider the following example:

```
architecture behavioral of ent is
   signal c_internal: small_int;
begin
   c_internal <= a0 + b0;
   c0 <= c_internal;
   c1 <= c_internal + a1 + b1;
end behavioral;
```

The declaration section of the architecture is the area between the keyword **architecture** and the keyword **begin**. Here you may declare objects that are local to the architecture. After the declaration section comes the architecture body, which is where you specify the behavior of the architecture.

Configuration

Configuration declarations may be used to associate particular design entities to *component instances* (unique references to lower-level components) in a hierarchical design, or to associate a particular architecture to an entity. As their name implies, configuration declarations are used to provide configuration management and project organization for a large design.

Statements

There are three basic kinds of statements in VHDL:

- Declaration Statements
- Concurrent Statements
- Sequential Statements

Declaration Statements

Declaration statements are used to define constants (such as literal numbers or strings), types (such as records and arrays), objects (such as signals, variables and components), and subprograms (such as functions and procedures) that will be used in the design. Declarations can be made in many different locations within a VHDL design, depending on the desired scope of the item being declared.

Concurrent and Sequential Statements

Concurrent and sequential statements are the fundamental building blocks of a VHDL design description. These statements, which represent the actual logic of a design, include such things as signal assignments, component instantiations, and behavioral descriptions.

There are important distinctions to be made between concurrent and sequential statements, as discussed below.

Concurrent Statements

Concurrent statements are evaluated independently of the order in which they appear. A concurrent statement is much like a signal assignment used in a PLD programming language such as ABEL-HDL. Signals pass values between concurrent statements, much as wires connect components on a schematic. The components being connected in a VHDL design might be logical elements that have been described using concurrent signal assignments, or they might be instances of lower-level entities.

Concurrent statements define logic (typically in the form of signal assignments that include combinational logic) that is inherently parallel. With concurrent statements, values are carried on signals, which may be the actual input and output ports of the design (defined in an **entity** statement) or local signals declared using a **signal** declaration statement.

Concurrent statements include the following:

- Signal assignments (selected and conditional)
- Component instantiations
- Generate statements
- Process statements
- Procedure and function calls

The following syntax shows an example of an architecture declaration with concurrent statements. Note that this code fragment also demonstrates how to include comments in VHDL source code. The double-hyphen character sequence (--) always begins a comment, and the comment continues until the end of the line.

```
architecture dataflow of my_circuit is
     signal d,e bit;
begin
    -- concurrent statements tied together with signals
    d <= in3 and in4;         -- logic for d
    e <= in5 or in6;          -- logic for e
    out1 <= in1 xor d;        -- output logic
    out2 <= in2 xor e;        -- output logic
end dataflow;
```

Sequential Statements

Sequential statements differ from concurrent VHDL statements in that they are executed in the order they are written. Sequential statements always appear within a **process** statement (which, in its entirety, is a concurrent statement) or within a **function** or **procedure**.

Sequential statements are similar to statements used in software programming languages such as C or Pascal. The term *sequential* in VHDL refers to the fact that the statements execute in order, rather than to the type of logic generated. That is, you can use sequential statements to describe either combinational or sequential (registered) logic. With sequential statements, values may be carried using either signals or variables.

Sequential statements include the following types of statements:

- Variable declarations
- Signal assignments
- Variable assignments
- Procedure and function calls
- If, case, loop, next, exit, return statements
- Wait statements

Following is an example of an architecture declaration that includes sequential statements in a process statement:

```
architecture behavior of some_thing is
begin
    process begin
        wait until clock;
        if (accelerator = '1') then
            case speed is
                when stop => speed <= slow;
                when slow => speed <= medium;
                when medium => speed <= fast;
                when fast => speed <= fast;
            end case;
        end if;
    end process;
end behavior;
```

Note: *Sequential statements do not imply, and are not the same as, sequential logic.*

Data Objects

Data objects hold values. Languages such as C and Pascal generally have only one type of data object: the variable. In addition to the variable, VHDL has two other types of data objects: constants and signals. VHDL variables work in much the same way as variables in conventional programming languages. From a hardware designer s perspective, both signals and variables can be thought of as wires interconnected with various logic gates. The differences among VHDL data objects lay in how they may be used and how much information they contain.

Before they can be used, data objects must be declared with a declaration statement, as explained separately for each of the three data types, below. Note that the VHDL synthesis compiler ignores initial values on both signals and variables, since most types of hardware currently available do not have a guaranteed power-up state. Therefore, when writing VHDL code, it is best not to use initial values unless you know that you are guaranteed a certain power-up state in your target device.

Variables

Like a variable in C or Pascal, a variable in VHDL carries with it only one piece of information: its current value. Variables are assigned a value using the := operator. Consider the following variable assignments:

```
first_var := 45;
SECOND_VAR := first_var;
second_var := 0;
```

In these assignments, the variable named first_var is being assigned an **integer** value of 45 (For more information on data types, including **integer**, see the next section). A variable named SECOND_VAR is then assigned to whatever value first_var currently contains, which is 45. SECOND_VAR is then assigned the **integer** value 0. The variables named second_var and SECOND_VAR are the same, since VHDL is not case-sensitive.

Note: *In VHDL, names (or identifiers, as they are more properly referred to) must begin with a letter, and may consist of any number of letters, digits, or underscores, as long as there is not more than one underscore in a row. As noted earlier, no distinction is made between upper- and lower-case characters.*

Before they can be used, variables must be declared with a variable declaration statement, as in the following example:

```
variable first_var : integer;
variable second_var, third_var : integer := 0;
```

A variable declaration begins with the keyword **variable**, followed by one or more names, the data type, and optionally, an initial value. Variables may be declared only within processes or functions, two constructs that are explained later in this chapter.

Constants

Constants are much like variables, except, as they name implies, their value can never change. Constants are normally employed to make code easier to read and to modify.

Like variables, constants are declared with a declaration statements. An example of a constant declaration is as follows:

```
constant one_grand : integer := 1000;
```

Signals

Signals are declared in much the same manner as variables. Signal declarations may include an initial value, which will be ignored by the synthesis compiler. Examples of signal declarations are as follows:

```
signal first_sig : integer;
signal second_sig, third_sig : integer := 5;
```

Signal assignments are performed using the <= operator, as in the following examples:

```
first_sig <= 9;
second_sig <= first_sig;
third_sig <= first_sig after 5 ns;
```

The first clue as to the fundamental difference between signals and variables is found in the assignment to third_sig. The example specifies that third_sig will take on the value held by first_sig, but with a delay of 5 nanoseconds. This is in essence propagation delay.

Like initial values, delays specified using the optional **after** keyword are ignored by the synthesis compiler, since it has no way of guaranteeing that a particular delay will occur in the target hardware. Therefore, you will not normally use the **after** clause when writing code for synthesis. However, it is important to realize that even without an **after** clause, all signal assignments occur with some infinitesimal delay, known as delta delay. Technically, delta delay is of no measurable unit, but from a hardware design perspective you should think of delta delay as being the smallest time unit you could measure, such as a femtosecond.

The effect of delta delay on the simulation behavior of your code can be profound. Consider the following example. Assume that the signal first_sig is assigned the value 11 at time 100 ns:

```
first_sig <= 11;
```

first_sig actually changes to its new value 1 fs after time 100 ns. Now consider the next two assignments executed at time 200 ns:

```
first_sig <= 25;
first_var := first_sig;
```

If both of these assignments are executed at time 200 ns, first_var immediately takes on the value 11, and 1 fs later first_sig has the value 25.

Data Types

VHDL supports a variety of data types. The type of a variable, signal, or constant determines the operators that are predefined for that object as well as the range of values that it can take on.

The predefined VHDL data types include:

- numeric (integer or real)
- boolean
- character
- time (measured in units from fs to hr)
- string (an array of characters)
- bit (can have a value of 0 or 1)
- bit_vector (an array of bits)

After the language was defined it was acknowledged that the built-in types were not entirely adequate for modeling the behavior of real hardware. The IEEE standard 1164 was developed to address this shortcoming. This standard defines the types:

- std_ulogic and std_logic (the equivalent of bits but with 9 possible data values instead of two)
- std_ulogic_vector and std_logic_vector (an array of std_ulogic and std_logic, respectively)

Definitions for all of the predefined types, with the exception of std_logic and std_logic_vector, are in the file std**.**vhd, which contains the package standard. The types created by the 1164 standard are defined in the file ieee**.**vhd. The primary difference between std_ulogic and std_logic is that std_logic is what is referred to as a resolved type. This means that objects of type std_logic can be used for modeling logic with multiple drivers, such as tristate buses or wired logic. Objects of type std_ulogic may have only one driver.

In addition to types, **subtypes** may be used to define subsets of their base type. For example, a short **integer** type (one with a specified maximum value) can be defined as a **subtype** with the statement:

```
subtype short_int is integer range 0 to 255;
```

VHDL also supports enumerated and user-defined types, which are explained later in this section.

Numeric Types

The numeric types consist of **integer**, floating point (**real**), and physical types. Two encoding schemes are used by the VHDL Synthesizer for numeric types:

- Numeric types and subtypes that contain a negative number in their range definition are encoded as two's complement numbers.
- Numeric types and subtypes that contain only positive numbers are encoded as binary numbers.

The number of wires that are synthesized depends only on the value in the definition that has the largest magnitude. The smallest magnitude is assumed to be zero for numeric types.

Also for synthesis: floating point numbers are constrained to have the same set of possible values as integers, although they can be represented using floating point format with a positive exponent.

Numeric types and subtypes are synthesized as follows:

The declaration:	Is synthesized as:
`type int0 is range 0 to 100`	`-- 7 bit binary encoding`
`type int1 is range 10 to 100`	`-- 7 bit binary encoding`
`type int2 is range -1 to 100`	`-- 8 bit two's complement`
`type int3 is int2 range 0 to 7`	`-- 3 bit binary encoding`

Numeric Operators

If the type of the object to which the result is assigned has more bits than either of the operands, then the result of the numeric operation is automatically sign or zero extended by the VHDL synthesizer. Sequential encoded types are zero extended, and two's complement numbers are sign extended.

If the type of the object to which the result is assigned has fewer bits than either of the operands, then the result of the numeric operation is truncated. If a numeric operation has a result that is larger than either of the operands then the new size is evaluated before the above rules are applied. For example, if an addition operator "+" generates a carry, the result will be truncated, used, or sign (or zero) extended according to the type of the object to which the result is assigned:

```
type short is integer 0 to 255;
subtype shorter is short range 0 to 31;
subtype shortest is short range 0 to 15;

signal op1,op2,res1: shortest;
signal res2: shorter;
signal res3: short
begin
    res1 <= op1 + op2;    -- truncate carry
    res2 <= op1 + op2;    -- use carry
    res3 <= op1 + op2;    -- use carry and zero extend
```

Note: *During simulation, if the result of an arithmetic operation than the size of the specified object, as is the case for signal* ***res1****, then the simulator will produce an error.*

The encoding of integers in a binary format means that all ranges are rounded up to the nearest power of two. This means that if shorter had been declared as:

```
subtype shorter is short range 0 to 16;
```

then the result would have been the same after synthesis. Objects declared type of type integer without a range constraint will be synthesized into 32 wires.

The are two predefined subtypes of **integer**. Subtype **natural** is defined as non-negative integer, while subtype **positive** is defined as a non-negative and non-zero **integer**.

Other Types

The types **bit**, **Boolean**, **character**, std_ulogic and std_logic are enumerated types. Enumerated types are discussed in the following subsection. The type **bit_vector** is an array, as is the type std_logic_vector.

- Bit , std_logic and std_ulogic types are synthesized to one wire.
- Character types are synthesized to seven wires.
- Boolean types are synthesized to one wire.
- Array and record types are composites, and are treated as collections of their elements. Subtypes of composite types are treated as collections of the elements of the subtype only.

Enumerated Types

An enumerated type in VHDL is a special kind of data type that has a symbolic value. A good example of where an enumerated type signal would be used is in a state machine, in which symbolic values are used to represent unique states of the machine:

```
type machine_state is (Init, Ready, Xmit1, Xmit2, Xmit3, Xmit4);
signal present_state, next_state: machine_state;
```

Many of the common data types used in VHDL, such as **bit**, **Boolean**, **character**, and std_ulogic, are actually enumerated types defined in a library such as std or ieee**. Bit**, **Boolean**, and **character** are all enumerated types that are predefined in std**.**vhd.

When synthesized, enumerated types result in a binary encoding, unless the **enum_encoding** attribute has been used to specify alternate values for each element of the type. In the absence of the **enum_encoding** attribute, elements in the enumerated type are assigned numeric values from left to right, with the value of the leftmost element being zero.

By default, the number of wires generated to encode an enumerated type will be the smallest possible *n*, where the number of elements is 2^n. (It will, for example, require three wires to represent an enumerated type with more than four but less than nine different values.)

The Std_ulogic and Std_logic Data Types

Std_ulogic (which is the base type of the more-commonly used resolved type std_logic) is a data type defined by IEEE standard 1164, and defined in the file ieee**.**vhd. Std_ulogic is an enumerated type, and has the following definition (from ieee**.**vhd):

```
type std_ulogic is (
'U',                -- Uninitialized
'X',                -- Forcing Unknown
'0',                -- Forcing 0
'1',                -- Forcing 1
'Z'                 -- High Impedance
'W'                 -- Weak Unknown
'L'                 -- Weak 0
'H'                 -- Weak 1
'-'                 -- Don't care
);
```

The std_ulogic (or std_logic) data type is very important for both simulation and synthesis. Std_logic includes values that allow you to accurately simulate such circuit conditions as unknowns and high-impedance states. For synthesis purposes, the high-impedance and don't-care values provide a convenient and easily recognizable way to represent three-state enables and don't-care logic. For synthesis, only the values `0`, `1`, `Z`, and `-` have meaning and are supported. The version of std_logic_1164 defined in the file ieee.vhd includes an enum_encoding attribute that results in each object of type std_ulogic or std_logic being synthesized into a single wire.

User Defined Types and Subtypes

In addition to the standard types, you can define your own types which may be scalars, arrays, or records. VHDL also allows subtypes, which are simply a mechanism to define a subset of a type.

The use of types other than **bit** and **bit_vector** (or std_logic and std_logic_vector) can make your design much easier to read. It is good VHDL coding practice to put all your type definitions in a package and make the contents of that package visible with a **use** statement. The following example shows how a subtype of **integer** is defined in a package and referenced in the rest of the design:

```
package type_defs is
    subtype very_short is integer range 0 to 3;
end type_defs;

use work.type_defs.all;    -- use clause

entity counter is
    port (clk: in Boolean; p: inout very_short);
end counter;

architecture behavior of counter is
begin
```

```
    process(clk)
    begin
        if clk and clk'event then
            p <= p + 1;
        end if;
    end process;
end behavior;
```

In this example, type **integer** was used because the "+" operator is defined for integers but not for bit_vectors.

The most common user-defined types are enumerated types, described in the previous section.

Types and Logic Synthesis

In VHDL, types are used for type checking and for operator overload resolution (a situation in which two or more operators or functions have the same name, but specify different argument types). For logic synthesis, each type declaration also defines the encoding and number of wires to be generated. For subtypes, checking and overloading use the base type of the subtype. Each subtype declaration defines a subset of its base type and can specify the number or wires (directly or indirectly) and possibly the encoding scheme.

During compilation, ports with types that synthesize to multiple wires are renamed by appending *_n_*, where *n* is an incremented integer starting from zero.

Type Conversions

Because VHDL is strongly typed, and not all operations are supported for all standard data types, and it is sometimes necessary to convert from one type to another. A good example of this is the previous example, which used an **integer** data type to describe a counter. What if the design required (for external interface reasons) that all I/O ports be of type std_logic? Since there is no pre-defined **+** operator for non-numeric types such as std_logic, it is necessary to either overload the **+** operator (by writing a new **+** function for std_logic_vector data types) or convert the type from std_logic_vector to **integer**, and then from **integer** back to std_logic_vector as shown below:

```
library ieee;
use ieee.std_logic_1164.all;
entity counter is
    port (clk: in std_logic;
          p: inout std_logic_vector(1 downto 0);
end counter;

library dataio;
use dataio.std_logic_ops.all;

architecture behavior of counter is
begin
    process (clk)
    begin
```

```
        if clk = '1' and clk'event then
            p <= To_Vector(2,To_Integer(p) + 1);
        end if;
    end process;
end behavior;
```

The example shown makes use of the package std_logic_ops in the dataio library provided with the VHDL synthesizer. This library includes commonly-used type conversion functions (such as **To_Integer** and **To_Vector**).

Operators

VHDL includes the following kinds of operators:

- Logical
- Relational
- Arithmetic

Logical Operators

Logical operators, when combined with signals and/or variables, are used to create combinational logic. VHDL provides the following logical operators:

and
or
nand
nor
xor
not

These operators are defined for the types **bit** , std_ulogic (which is the base type of std_logic) and **Boolean**, and for one-dimensional arrays of these types (for example, an array of type **bit_vector** or std_logic_vector).

Relational Operators

Relational operators are used to create equality or magnitude comparison functions. VHDL provides the following relational operators:

=	Equal to
/=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The equality operators (= and /=) are defined for all VHDL data types. The magnitude operators (>=, <=, >, <) are defined for numeric types, enumerated types, and some arrays. The resulting type for all these operators is **Boolean**.

Arithmetic Operators

Arithmetic operators are used to create arithmetic functions. VHDL provides the following arithmetic operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division
mod	Modulus
rem	Remainder
abs	Absolute Value
**	Exponentiation

These operators are defined for numeric types such as **integer** and **real**.

Note: *Using arithmetic operators in a design can result in very large amounts of combinational logic being generated.*

Overloading Operators

In addition to the predefined operators, VHDL allows you to create new operators, or to *overload* existing operators to support alternate types of arguments or to give them new meanings. For example, Synario supplies overloaded functions defining the relational operators listed in the previous section for type **bit_vector** as part of the package bit_ops contained in the file \synario\lib5\dataio.vhd. Overloaded relational operators for std_logic_vectors are supplied as part of the package std_logic_ops which is found in the same file. Note that these overloaded operators treat bit_vectors and std_logic_vectors as unsigned quantities.

VHDL Attributes

VHDL has many predefined attributes that allow access to information about types, arrays, and signals. For a complete list of the supported attributes and their definitions, see Appendix A, Quick Reference. Examples of attributes used to modify a type are shown below. In this example, the **'high** and **'low** attributes are used to determine the highest and lowest values of a type:

```
integer'high    -- has a value of 2147483647
integer'low     -- has a value of -2147483647
```

To declare a subtype of type **integer**, use the **'high** and **'low** attributes to determine the resulting new upper and lower bounds of the type :

```
subtype shorter is integer range 0 to 100;
shorter'high   -- has a value of 100
shorter'low    -- has a value of 0
```

Attributes can also be combined, as in:

```
shorter'base'high        -- has a value of 2147483647
```

When used with an array, the **'high** attribute has a value of the highest array index:

```
type my_array is array (0 to 99) of Boolean;
variable info : my_array;
info'high             -- has a value of 99
```

There is a set of attributes that gives access to information about signal waveforms. Most of these signal attributes are for simulation, and have no other meaning. There is one signal attribute, however, that is often used to describe clock logic. You can use **'event** on signals to specify edge sensitivity, usually in combination with a value test to specify a rising or falling edge:

```
signal clock : Boolean;
if not clock and clock'event  -- falling edge.
```

Note: *An alternative to using the "clock and clock'event" method of specifying a clock edge is to use the rising_edge() function provided with the IEEE 1164 library.*

3. How to Write Synthesizable VHDL

The hardware implementation of a design written in VHDL depends on many factors. Coding conventions, fitter technology, and optimization options are all factors. The general nature of a design also has a large impact on its suitability for synthesis to a particular device, independent of the method used to describe the design.

Not all designs can be synthesized. Many VHDL designs (which are often referred to as **models** for simulation) are not suitable for synthesis. These include high level performance models; environmental models (*test benches*) for stimulus/response; or system models that include a mixture of software, hardware, and physical aspects. For the purposes of logic synthesis, the VHDL synthesizer must assume that the entire VHDL design describes digital logic that is to be implemented in hardware.

Hardware design and design for synthesis in particular adds several additional constraints that must be considered above and beyond the requirements for simulation. One example of this is a gated clock . A gated clock may not be an issue for simulation, since values may be written to a computer s memory without concern for electrical glitches. When designing for synthesis, however, care must be taken to ensure that the circuit described will actually control the clocking of memory elements in a manner appropriate for the target hardware.

A simulation model may also describe the **timing** characteristics of a design. Timing specifications (such as inertial or transport delays) are ignored by the VHDL synthesizer, and the actual timing behavior of the design depends on the architecture and mapping of the target device. For this reason, a VHDL model that depends on the timing for correct operation may not synthesize to the expected result when moved from one target device to another.

Simulation models may describe unbounded conditions (such as loops that have no termination, or integers that have no range) that are impossible represent in hardware. In some cases (such as infinite loops) the synthesis tool will produce an error and exit, while in other cases (such as unbounded integers) the VHDL synthesizer will assume a default representation (for example, 32 bits), which may or may not result in the expected circuit being generated.

In addition, a VHDL design written for simulation may use enumerated types to represent the encoding of a group of wires, perhaps as part of a symbolic state machine description. A design may also use enumerated types to represent the electrical characteristics on a signal wire (such as high impedance, resistive, or strong). In this case, the VHDL synthesizer has no way to distinguish the meaning (in terms of how the values should be represented in hardware) of each circuit. Unless you have provided an encoding for these types (using the VHDL synthesis custom attribute **enum_encoding** described in this chapter), the VHDL synthesizer must assume a default encoding for all enumerated types.

Optimization Strategies

Most, but not all, PLDs are constructed with an input logic array (the **and** array) and output register macrocells that are fed by an **or** gate. These devices are optimized for wide sum-of-products logic functions. The macrocells in these devices typically include three-state buffers and one or more possible feedback paths back into the array. Most FPGAs, on the other hand, are constructed with smaller basic logic elements (such as 4- or 5-input lookup tables, or multiplexers). The VHDL synthesizer, and other processes invoked by the Project Navigator, include a number of options (properties) that can be modified to optimize the design for the target device architecture. In most cases, the default property values (which are set depending on the device you have selected) will result in the most efficient implementation. If you want to experiment with different optimization properties, refer to the on-line help for information about each of the available properties.

Note: *An example of modifying VHDL Synthesis and Design Fitting properties can be found in in the Craps Game example, in the tutorials chapter of the* VHDL Entry *manual.*

Describing Combinational Logic

This section describes the relationship between basic VHDL statements and the resulting synthesized combinational logic. Most of the operators and statements that are used to describe combinational logic are the same as found in any programming language. As in a programming language, some operations take more time (path delays) to execute in hardware, and some require more space (in this case, device resources) to implement. Some VHDL operations are more expensive to synthesize into logic than others because they require more gates to implement. This section describes the relative costs associated with various combinational operations, and the kind of circuitry you can expect to get out of synthesis.

Constants and Types

The context in which an operator is used effects the generated circuitry. Using constant values or simple one-bit data types results in the most compact circuitry, while complex data types (such as arrays) in an expression result in correspondingly more circuitry.

If one operand of a combinational expression is a constant, then less logic is generated. If both operands are constants, the logic can be collapsed during compilation and the cost of the operation is zero gates. Using constants wherever possible means that the design description will not contain unwanted functionality, will synthesize faster and produce a more efficient implementation.

Certain operators in VHDL are restricted to specific types, generally following the programming language conventions which are given in the following subsections. These subsections describe the following kinds of operators:

- Logical Operators
- Relational Operators
- Arithmetic Operators

Logical Operators

VHDL provides the following logical operators:

and

or

nand

nor

xor

not

These operators are defined for the types **bit** and **Boolean**, and for one-dimensional arrays of these types (for example, an array of type **bit_vector**). These operators are also defined for the IEEE 1164 **std_logic** (or **std_ulogic**), and **std_logic_vector** data types (if the **ieee** library and **std_logic_1164** package are included in your design). The generation of logic from language constructs is reasonably direct, and results in an implementation in gates as shown in the following two examples.

Example 1:

```
entity logical_ops_1 is
    port (a, b, c, d: in bit;  m: out bit);
end logical_ops_1;

architecture example of logical_ops_1 is
    signal e: bit;
begin
```

```
    m <= (a and b) or e; --concurrent signal assignments
    e <= c xor d;
end example;
```

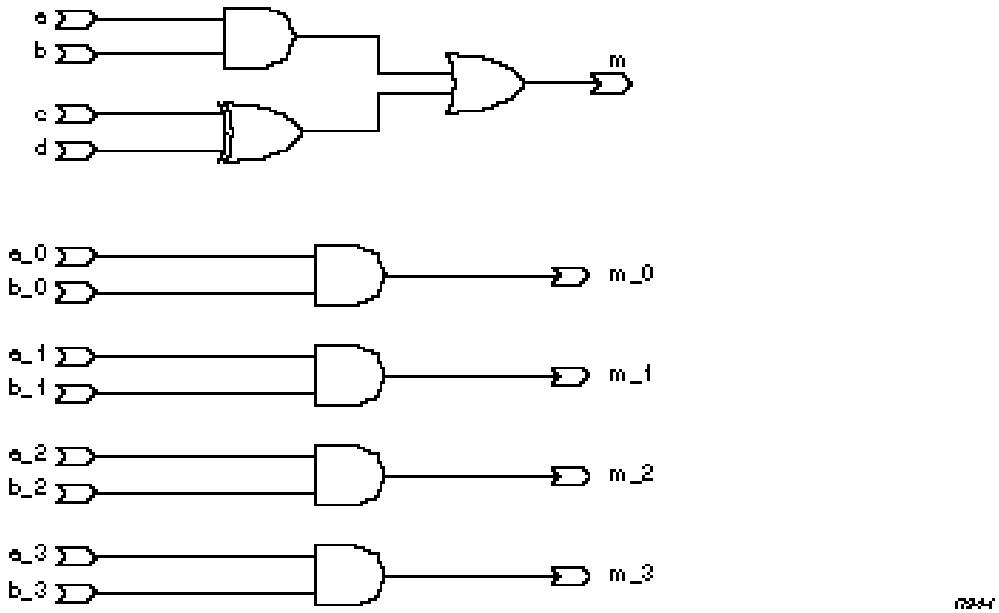
Example 2:

```
entity logical_ops_2 is
    port (a, b: in bit_vector (0 to 3);
          m: out bit_vector (0 to 3));
end logical_ops_2;

architecture example of logical_ops_2 is
begin
    m <= a and b;
end example;
```

Figure 3-1 shows how these examples are implemented in logic. In the first example, notice that the logic is shown in a multilevel implementation. In the logic actually generated, the logic for **m** will be a large sum-of-products function with the exclusive-**or** function (signal **e**) expanded into **and**/**or** logic and preserved (in a multilevel logic structure) or flattened into a larger two-level sum-of-products representation. The actual form of logic generated will depend on the optimization options chosen in the Project Navigator.

The second example shows how bit_vectors are expanded and processed. The **and** operation is distributed through the bit_vector data for **m**, as you would expect.

Figure 3-1: Logical Operators



Relational Operators

VHDL provides relational operators as shown in **Table 3-1**.

Table 3-1*:* Relational Operators

Operator	Description
=	Equal
/=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The equality operators (**=** and **/=**) are defined for all VHDL data types. The magnitude operators (**>=**, **<=**, **>**, **<**) are defined for numeric types, enumerated types, and some arrays. The resulting single-bit type for all these operators is **Boolean**. In addition to the built-in relational operators, overloaded versions of these operators are supplied in the packages bit_ops and std_logic_ops (found in the file \synario\lib5\dataio.vhd) for bit_vector and std_logic_vector types. The overloaded operators found in these packages treat bit_vectors and std_logic_vectors as unsigned quantities.

The simple comparisons (equal and not equal) are more efficient to implement (in terms of gates or product terms) than the magnitude operators. To illustrate, the first example below uses an equal operator while the second uses a greater-than-or-equal-to operator. As you can see from the schematic of **Figure 3-2**, the second example uses more than twice as many gates as the first.

Example 1:

```
entity relational_ops_1 is
    port (a, b: in bit_vector (0 to 3); m: out Boolean);
end relational_ops_1;
architecture example of relational_ops_1 is
begin
    m <= a = b;
end example;
```

Example 2:

```
entity relational_ops_2 is
    port (a, b: in integer range 0 to 3; m: out Boolean);
end relational_ops_2;
architecture example of relational_ops_2 is
begin
    m <= a >= b;
end example;
```

Arithmetic Operators

The arithmetic operators in VHDL are defined for numeric types (**integer** and **real**). The operators are listed in **Table 3-2**. In addition, overloaded versions of the + and - operators are supplied in the packages bit_ops and std_logic_ops for the types bit_vector and std_logic_vector, respectively.

Note: *The VHDL synthesizer does not distinguish between integer and real number values. Floating point values are constrained to the same range of values as integers.*

Figure 3-2: Relational Operators

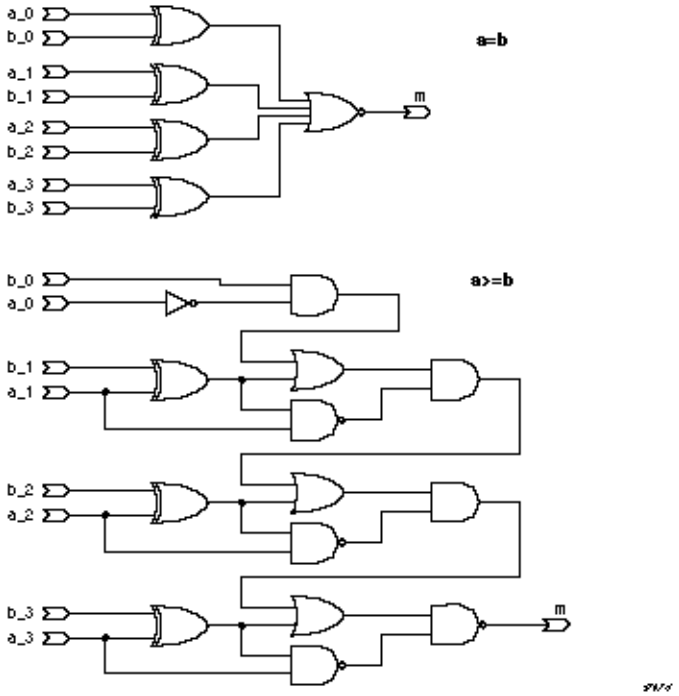


Table 3-2: Arithmetic Operators

Operator	Description
++	Addition
=	Subtraction
*	Multiplication
/	Division
mod	Modulus
rem	Remainder
abs	Absolute Value
**	Exponentiation

While the addition and subtraction operators (**+**, **-**) are somewhat expensive in terms of gates required, the multiplication operators (*****, **/**, **mod**, **rem**) are extremely expensive. The VHDL synthesizer does make special optimizations, however, when the right hand operator is a constant and an even power of 2.

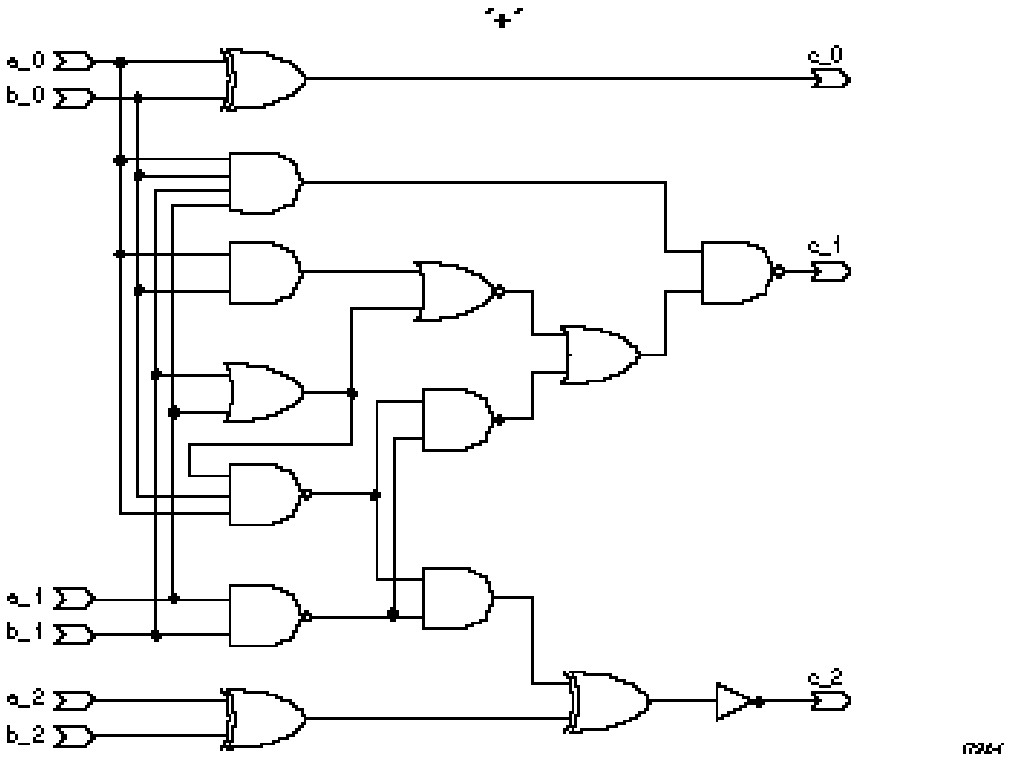
The absolute (**abs**) operator is not expensive to implement. The ** operator is only supported when its arguments are constants.

The following example illustrates the logic generated for an addition operation:

```
package example_arithmetic is
    type small_int is range 0 to 7;
end example_arithmetic;

use work.example_arithmetic.all;

entity arithmetic is
    port (a, b: in small_int; c: out small_int);
end arithmetic;
architecture example of arithmetic is
begin
    c <= a + b;
end example;
```

Figure 3-3 shows the logic generated for this example in schematic form. Again, this logic may be collapsed into a sum-of-products (2-level) form during processing by the VHDL synthesizer and device fitting.

Figure 3-3: Arithmetic Operators



Shift Operators

The shift operators in VHDL are defined for the types bit and boolean. In addition, the package std_logic_ops found in the file \synario\lib5\dataio.vhd supplies overloaded operators for type std_logic_vector. The left-hand argument of these operators must be an array type (such as bit_vector or std_logic_vector) and the right-hand argument must be an integer. The return value is always of the same type as the left-hand argument. The operators are listed in **Table 3-3**.

Table 3-3: Shift Operators

Operator	Description
sll	Shift Left Logical
srl	Shift Right Logical
sla	Shift Left Arithmetic
sra	Shift Right Arithmetic
rol	Rotate Left Logical
ror	Rotate Right Logical

The shift operators are not expensive to implement if the right operand (which must be an integer type) is a constant value. If the right operand is not a constant (and depends on a signal) then the logic can be quite expensive to implement.

Describing Conditional Logic

Conditional logic is combinational logic that implements a multiplexer-like function.

The two forms of concurrent statements used to describe conditional logic are:

- Conditional signal assignment
- Selected signal assignment

There are also two forms of sequential statements for describing conditional logic:

- If statement
- Case statement

These statements are discussed individually below.

Concurrent Statement: Conditional Signal Assignment

The following is an example of a conditional signal assignment:

```
entity control_stmts is
    port (a, b, c: in Boolean;  m: out Boolean);
end control_stmts;

architecture example of control_stmts is
begin
    m <= b when a else  c;
end example;
```

Note: *In IEEE standard 1076-1993, the* ***else*** *clause is optional. If you do not provide an* ***else*** *clause, however, the resulting circuit will probably include a latch, which may not be the desired result.*

Concurrent Statement: Selected Signal Assignment

A selected signal assignment uses the **with** statement, and must include all possible cases. The **others** case ensures that all cases are covered.

The following is an example of a selected signal assignment:

```
entity control_stmts is
    port (sel: bit_vector (0 to 1); a,b,c,d: bit;
          m: out bit);
end control_stmts;

architecture example of control_stmts is
begin
    with sel select
        m <= c    when b"00",
        m <= d    when b"01",
        m <= a    when b"10",
        m <= b    when others;
end example;
```

If Statement

The condition in an **if** statement must evaluate to **true** or **false** (a Boolean type). The following example illustrates the if statement:

```
entity control_stmts is
    port (a, b, c: in Boolean;  m: out Boolean);
end control_stmts;
architecture example of control_stmts is
begin
    process (a, b, c)
        variable n: Boolean;
    begin
        if a then
            n := b;
        else
            n := c;
        end if;
        m <= n;
```

```
    end process;
end example;
```

Case Statement

Like the **with** statement, VHDL requires that all the possible conditions be represented in the condition of a **case** statement. To ensure this, use the **others** clause at the end of a case statement to cover any unspecified conditions.

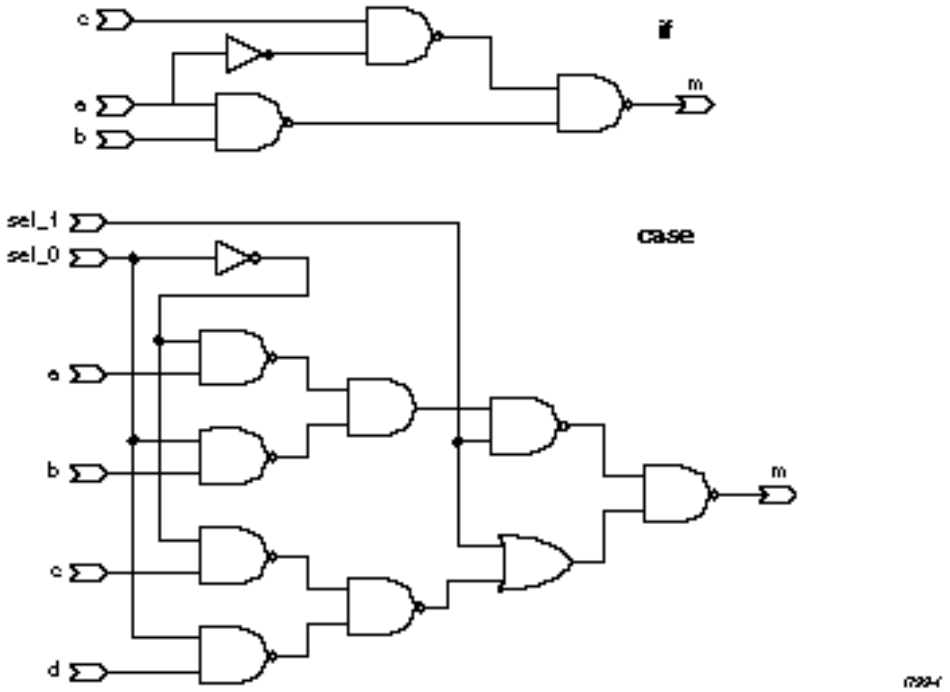
Note: *Since std_ulogic and std_logic types have nine possible values (instead of two possible values for bit types), you should always include an* ***others*** *clause when using these types.*

The following example illustrates the case statement:

```
entity control_stmts is
    port (sel: in bit_vector (0 to 1); a,b,c,d: in bit;
          m: out bit);
end control_stmts;

architecture example of control_stmts is
begin
    process (sel,a,b,c,d)
    begin
        case sel is
            when b"00"  => m <= c;
            when b"01"  => m <= d;
            when b"10"  => m <= a;
            when others => m <= b;
        end case;
    end process;
end example;
```

Schematic representations of the if and case logic generated for these two examples are shown in **Figure 3-4**.

Figure 3-4: Control Statements



Describing Replicated Logic

VHDL provides the following subprograms and looping constructs for creating replicated logic:

- Function
- Procedure
- Loop Statement
- Generate Statement

Functions and **procedures** are collectively referred to as subprograms. **Generate** is a concurrent **loop** statement. These constructs are synthesized to produce logic that is replicated once for each subprogram call, or once for each iteration of a loop.

Functions and Procedures

Functions are always terminated by a **return** statement, which returns a value. A return statement may also be used in a procedure, where it never returns a value.

The following example illustrates the use of a function:

```
entity func is
    port (a: in bit_vector (0 to 2);
          m: out bit_vector (0 to 2));
end func;

architecture example of func is
    function simple (w, x, y: bit) return bit is
    begin
        return (w and x) or y;
    end;
begin
    process (a)
    begin
        m(0) <= simple(a(0), a(1), a(2));
        m(1) <= simple(a(2), a(0), a(1));
        m(2) <= simple(a(1), a(2), a(0));
    end process;
end example;
```

A procedure differs from a function in that there is no return value, and the arguments of the procedure have modes (**in**, **out**, or **inout**):

```
entity proc is
    port (a: in bit_vector (0 to 2);
          m: out bit_vector (0 to 2));
end proc;
architecture example of subprograms is
    procedure simple (w, x, y: in bit; z: out bit) is
    begin
        z <= (w and x) or y;
    end;
begin
    process (a)
    begin
        simple(a(0), a(1), a(2), m(0));
        simple(a(2), a(0), a(1), m(1));
        simple(a(1), a(2), a(0), m(2));
    end process;
end example;
```

For both functions and procedures, the VHDL synthesizer will generate a block of logic for each instance (unique reference to) the function or procedure.

Loop Statements

If possible, loop ranges should be expressed as constants. Otherwise, the logic inside the loop may be replicated for all the possible values of the loop ranges, which can be very expensive in terms of gates. Loop statements may be terminated with an **exit** statement, and specific iterations of the loop statement may be terminated with the **next** statement.

The following example illustrates the use of a **loop** statement:

```
entity loop_stmt is
    port (a: in bit_vector (0 to 3);
          m: out bit_vector (0 to 3));
end loop_stmt;

architecture example of loop_stmt is
begin
    process (a)
        variable b:bit;
    begin
        b := 1;
        for i in 0 to 3 loop     -- no need to declare i
            b := a(3-i) and b;
            m(i) <= b;
        end loop;
    end process;
end example;
```

While statements are also supported by the VHDL synthesizer, with the constraint that the loop termination depend only on a value that can be determined at the time of synthesis (for example, a *metalogic* value. See Appendix B, Limitations, for more information about metalogic values).

The following example demonstrates the use of a **while** loop:

```
entity while_stmt is
    port (a: in bit_vector (0 to 3);
          m: out bit_vector (0 to 3));
end while_stmt;
architecture example of while_stmt is
begin
    process (a)
        variable b: bit;
        variable i: integer;
    begin
        i := 0;
        while i < 4 loop
            b := a(3-i) and b;
            m(i) <= b;
        end loop;
    end process;
end example;
```

Unconstrained loops (such "while true" loops) are not supported in synthesis.

Example schematics for the loop and subprogram are shown in **Figure 3-5**.

Generate Statements

Generate statements are used to replicate one or more concurrent statement. The generate statement has two forms: **for** and **if**.

For Generation Statement

Following is an example of a for generation statement:

```
Gen1: for i in 0 to 3 generate
SM: mod1 port map(A(i),B(i),Y(i));
end generate Gen1;
```

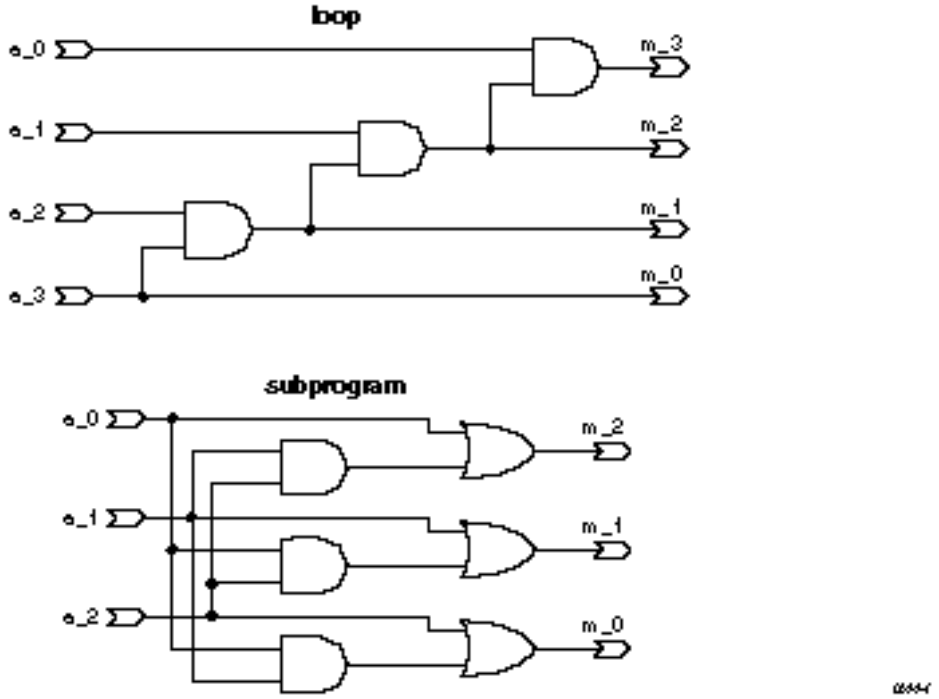
When processed, this statement expands into four statements as follows:

```
SM(0): mod1 port map(A(0),B(0),Y(0));
SM(1): mod1 port map(A(1),B(1),Y(1));
SM(2): mod1 port map(A(2),B(2),Y(2));
SM(3): mod1 port map(A(3),B(3),Y(3));
```

If Generation Statement

The if generation statement is used to describe a conditional selection of concurrent statements:

Figure 3-5: Loop and Subprogram Statements




```
Gen2: if test_flag = 1 generate
    test_pins <= current_state;
end generate Gen2;
```

When an **if** generation statement is used, the conditional expression (in this case "test_flag = 1") must be a metalogic value (one that does not depend on a signal or variable.) For example, the value of **i**, used in the previous **for** generation statement, is a static value and can be used in a nested **if** generation statement, as follows:

```
for i in 0 to 3 generate
   if i /= 2 generate
      SM: modl port map A(i),B(i),Y(i);
   end generate;
end generate;
```

Describing Sequential Logic

This section describes in detail how various kinds of registered sequential circuits can be described using VHDL, and how these descriptions are synthesized into actual circuitry (using latches and flip-flops).

Describing sequential logic in VHDL is very much like programming in a conventional programming language, and less like programming using a traditional PLD programming language. There is no register assignment operator and no special attributes or dot extensions for specifying clocks and resets. In VHDL, you must describe the behavior of a sequential logic element such as a latch or flip-flop, as well as specifying the behavior of more complex sequential machines.

The behavior of a sequential logic element (latch or flip-flop) is to save a value of a signal over time. This section shows how such behavior may be described. This description is extended to add the behavior of set and reset (in both their synchronous and asynchronous forms).

There are often several ways to describe a particular behavior, and the following examples typically show only two of the many possible styles. There is no right style, however. Your choice of style should simply be that which helps you specify the clearest description of your design.

Note: *If you deviate from commonly-used VHDL coding conventions (styles) such as those described in this manual, then you may risk creating designs that are not portable to other synthesis tools.*

There are two commonly-used methods used to describe registered behavior: conditional (**if-then**) specifications and **wait** statements.

Conditional Specification

Describing sequential logic with a conditional specification relies on the inherent behavior of a VHDL if statement. The convention used for conditional statements that describe clocking logic is:

```
process (clk)
    begin
    if clk='1' then
        y <= a;
    else
            -- default: holds previous value
    end if;
end process;
```

This set of statements describes the behavior of a latch; if the clock is high the output (**y**) gets a new value. If the clock is not high then the output retains its previous value. This is unlike a PLD programming language such as ABEL-HDL, where the **else** condition results in the signal going to zero. If both conditions had been written as assignments, then the behavior would be that of a mux:

```
process(clk)
    begin
    if clk='1' then
        y <= a;
    else
        y <= b;
    end if;
end process;
```

Note: *This example would result in an error during synthesis; because the conditional logic is completely specified using an* ***else*** *statement, the process describes a combinational function. The signals* ***a*** *and* ***b*** *are both inputs to the combinational function, and are therefore required to be in the sensitivity list.*

This convention can be summarized: if an **if** statement is not completely specified, then a flip-flop or latch primitive is implied. Incompletely specified assignments within **case** statements can also result in latches being generated, but these latches are constructed using combinational feedback rather than latch primitives. If the **if** statement is completely specified (using an **else** clause), a combinational function is implied. There is no significance to any of the signal names used in these or any other examples. The clock input (in this case **clk**) can have any name. Implied flip-flops and latches can occur on either signals or variables.

Alternatively, you can describe a latch as transparent low by inverting the conditional logic*:*

```
process(clk)
    begin
    if clk='1' then
            -- hold
    else
        y <= a;
    end if;
end process;
```

Or more concisely:

```
process(clk)
    begin
    if clk='0' then
        y <= a;
    end if;
end process;
```

A rising edge flip-flop is created by making the clock input edge sensitive:

```
process(clk)
    begin
    if clk'event and clk='1' then
        y <= a;
    end if;
end process;
```

If you are using the IEEE 1164 std_logic (or std_ulogic) data types, you can simplify the description of clock edges (and improve the accuracy of simulations) by using the rising_edge() function:

```
process(clk)
    begin
    if rising_edge(clk) then
        y <= a;
    end if;
end process;
```

In all these cases, the number of registers or the width of the mux are determined by the type of the signal **y**.

Wait Statement

The second method uses a **wait** statement within the process:

```
process
    wait until expression;
    .
    .
    .
end process;
```

This statement suspends evaluation (over time) until an event occurs, and the expression evaluates to **true**. When a wait statement is used in a process, no process sensitivity list is required (or allowed). A flip-flip may be described as:

```
process
    wait until clk'event and clk='1'
    y <= a;
end process;
```

A constraint of the VHDL synthesizer is that **wait** statements must be located at either the beginning or end of a process, and there may not be more than one **wait** statement in a process.

***Note: Wait** statements are not recommended for use in synthesizable designs. **If-then** conditional statements are a more universally accepted method of describing registered logic.*

Latches

The following three examples each describe a level sensitive latch with an **and** function connected to its input. In all three of these examples the signal **y** retains its current value unless the clock input (**clk**) is **high**.

Example 1:

This example uses a **process** statement and conditional (**if**) statement. The sensitivity list contains the clock input and two data inputs because when **clk** is high the output **y** changes asynchronously with any change in **a** or **b**:

```
process (clk,a,b)
begin
    if clk='1' then
        y <= a and b;
    end if;
end process;
```

Example 2:

This example uses a **procedure** statement in combination with a concurrent procedure call. In this example the procedure is called twice to generate two latches from the declared procedure:

```
architecture dataflow of latch is
    procedure my_latch(signal clk,a,b: in Boolean;
        signal y : out Boolean)
    begin
        if clk='1' then
            y <= a and b;
        end if;
    end;
begin
    latch_1: my_latch (clock,input1,input2,outputA);
    label_2: my_latch (clock,input1,input2,outputB);
end dataflow;
```

Example 3:

This example uses a concurrent conditional assignment to describe a latch function for **y**. Note that **y** is used as an input to the conditional statement as well as being used as the output:

```
architecture dataflow of latch is
begin
    y <= a and b when clk else y;
end dataflow;
```

Flip-flops

The following four examples describe an edge sensitive flip-flop with an AND function connected to its input. In all these cases the signal **y** retains its current value unless the clock is changing.

Example 1:

A **process** statement for a flip-flop is identical to the first latch example, above, with addition of the **'event** attribute to specify an edge. The sensitivity list for the process contains only the clock input, since the output of a flip-flop only changes when the clock transitions from low to high:

```
process (clk)
begin
    if clk='1' and clk'event then
        y <= a and b;
    end if;
end process;   -- A Process statement :
```

Example 2:

This example shows how to use a **wait** statement to describe a flip-flop. When a wait statement is used, there is no sensitivity list associated with the process statement. To accurately describe an edge triggered flip-flop on the output, the wait statement must be the first statement in the process:

```
process
begin
    wait until clk'event and clk='1';  -- rising edge
    y <= a and b;
end process;
```

Example 3:

This example uses a **procedure** declaration, and is identical to the second latch example, above. The only difference is the addition of the **'event** attribute to define the clock as an edge triggered signal:

```
architecture dataflow of flipflop is
    procedure my_ff(signal clk,a,b: Boolean;
        signal y : out Boolean)
    begin
        if clk='1' and clk'event then
            y <= a and b;
        end if;
    end;
begin
    ff_1: my_ff (clock,input1,input2,outputA);
    ff_2: my_ff (clock,input1,input2,outputB);
end dataflow;
```

Example 4:

This example shows how to use a concurrent conditional assignment to describe the flip-flop:

```
    architecture concurrent of my_register is
    begin
        y <= a and b when clk='1' and clk'event;
    end concurrent;
```

Note: *In examples 1, 3 and 4, above, the* ***clk and clk'event*** *condition expression can be replaced with the IEEE 1164* ***rising_edge()*** *function, if* ***std_logic*** *(or* ***std_ulogic****) is used for the* ***Clk*** *signal. Using* ***rising_edge()*** *can improve the accuracy of simulations, particularly in cases where you are simulating transitions from uninitialized states.*

Note: *The concurrent conditional assignment shown in example 4 is allowed in the 1076-1993 VHDL specification, and is not supported in earlier versions of the language.*

Gated Clocks and Clock Enable

The examples in this chapter have assumed that the clock is a simple signal. In principle, any complex Boolean expression could be used to specify clocking. The use of a complex clock expression implies a gated clock.

As with any kind of hardware design, there is a risk that gated clocks may cause glitches in the register clock, and hence produce unreliable hardware. You need to be aware of the constraints of the target hardware and, as a general rule, use only simple logic in the **wait** or **if-then** expression.

It is possible to specify a gated clock with a statement such as:

```
if clk='1' and clk1'event and ena then
    q <= d;
end if;
```

which implies a logical **and** in the clock line. If you want to use a clock enable feature in the target device, however, you should use nested **if** statements as follows:

```
if clk='1' and clk1'event then
    if ena then
        q <= d;
    end if;
end if;
```

This style causes a clock enable feature to be specified in the target architecture if the Generate Clock Enable property of the Synthesize VHDL process is specified in the Project Navigator. If the Generate Clock Enable property has not been specified, multiplexer logic will be generated to hold the value of **q** when **ena** is low.

Synchronous Set/Reset

To add the behavior of synchronous set or reset you can simply add a conditional assignment to a constant value immediately inside the clock specification. This is a VHDL coding convention that is recognized as a hardware reset by the VHDL synthesizer. Other methods of reset control may have the desired behavior, but do not result in a hardware reset feature being utilized.

The following example shows a simple preset of a 1-bit (Boolean) output:

```
process(clk)
begin
    if clk='1' and clk'event then
        if set='1' then
            y <= '1';
        else
            y <= a and b;
        end if;
    end if;
end process;
```

The example sets **y** to **true** when the **set** input is high during a rising clock edge.

The next example shows how a constant value can be specified to reset the output to an arbitrary encoding. When this example is processed by VHDL, a mixture of reset and preset features is utilized to create the desired reset output encoding:

```
process (clk)
begin
    if clk='1' and clk'event then
        if init='1' then
            y <= 7;      -- y is type integer
        else
            y <= a + b;
        end if;
    end if;
end process;
```

Note: *If the target device supports only a reset feature and does not have a preset feature, then you can use the Generate Reset Logic property of the VHDL synthesizer to ensure that only resets are generated.*

Asynchronous Set/Reset

To describe the behavior of asynchronous set or reset, the initialization is no longer within the control of the clock. Instead, simply add a conditional assignment to a constant immediately outside the clock specification. Because the outputs must change asynchronously when the reset input is high, the reset input must be included in the sensitivity list of the process:

```
process (clk,reset)
begin
    if reset='1' then
        y <= false;       -- y is type Boolean
```

```
    elsif clk='1' and clk'event then
        y <= a and b;
    end if;
end process;
```

Devices with both an asynchronous reset and preset are also supported, as long as the reset overrides the preset condition. For example:

```
process (clk, reset, preset)
begin
   if reset='1' then
        y <='0'; -- must be a constant value
   elsif preset='1' then
        y <='1'; -- must be a constant value
   elsif rising_edge(clk) then
        y <= a and b;
   end if;
end process;
```

Asynchronous Reset/Preset

You can combine the asynchronous reset and preset conditions in a single process:

```
process (clk, reset, preset)
begin
   if (reset = '1') then
        q <= '0';
   elsif (preset = '1') then
        q <= '1';
   elsif (rising_edge(clk)) then
        q <= d;
   end if;
end process;
```

Note that the asynchronous reset condition overrides the preset, as is generally the case in most flip-flops that actually have both reset and preset.

You can combine the asynchronous reset and preset to create an asynchronous load:

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

entity asyn_load_cnt is
   port(
   clk : in std_logic;
   ce : in std_logic;
   reset : in std_logic;
   preset : in std_logic;
   load : in std_logic;
   d : in std_logic_vector(7 downto 0);
   q : buffer std_logic_vector(7 downto 0));
end;

architecture behavioral of asyn_load_cnt is
   signal next_q : std_logic_vector(7 downto 0);
begin

   next_q <= q + '1';

   process(clk, reset, preset, load, d)
   begin
      for i in q'range loop
         if (reset = '1') or (load = '1' and d(i) = '0') then
            q(i) <= '0';
         elsif (preset = '1') or (load = '1' and d(i) = '1') then
            q(i) <= '1';
         elsif (rising_edge(clk)) then
            if (ce = '1') then
               q(i) <= next_q(i);
            end if;
         end if;
      end loop;
   end process;
end behavioral;
```

This logic defines an 8 bit up counter, with asynchronous reset, preset, load, and clock enable. Since the VHDL synthesis compiler requires that the value assigned by the reset or preset condition be a constant expression, a loop construct inside the process generates the logic for each flip-flop individually.

Describing Finite State Machines

This section describes the relationship between various types of finite state machines (FSMs), the VHDL description methods that are most commonly used to specify them, and the logic that is generated as a result of synthesis. Each example illustrates a single state machine. (This is not a constraint of VHDL or the VHDL synthesizer, just a simplification. Multiple state machines are supported in VHDL, and the different machines can operate independently using multiple clocks.)

Template State Machine

The recommended method for describing synthesizable state machines in VHDL is to use an enumerated type to define the states of the machine, and use two processes to clearly distinguish between the registered portion of the machine (the state registers and registered state machine outputs) and the combinational portion (the state transition logic and combinational state machine outputs). The following sample state machine provides a template that you can use to create your own state machine designs:

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

entity machine is
   port (clk,reset: in std_logic;
         state_inputs: in std_logic_vector (0 to 1);
         comb_outputs: out std_logic_vector (0 to 1));
end machine;

architecture behavior of machine is
    type states is (st0, st1, st2, st3);
    signal present_state, next_state: states;
begin
    register: process (reset,clk)
    begin
        if reset = '1' then
            present_state <= st0;        -- async reset to st0
        elsif rising_edge(clk) then
            present_state <= next_state; -- transition on clock
        end if;
    end process;
    transitions: process(present_state, state_inputs)
    begin
        case current_state is            -- describe transitions
            when st0 =>                  -- and comb. outputs
                comb_outputs <= "00";
                if state_inputs = "11" then
                    next_state <= st0;   -- hold
                else
                    next_state <= st1;   -- next state
                end if;
            when st1 =>
                comb_outputs <= "01";
                if state_inputs = "11" then
                    next_state <= st1;   -- hold
                else
                    next_state <= st2;   -- next state
                end if;
            when st2 =>
                comb_outputs <= "10";
                if state_inputs = "11" then
                    next_state <= st2;   -- hold
                else
                    next_state <= st3;   -- next state
                end if;
            when st3 =>
                comb_outputs <= "11";
                if state_inputs = "11" then
                    next_state <= st3;   -- hold
                else
                    next_state <= st0;   -- next state
                end if;
         end case;
    end process;
end behavior;
```

This state machine description includes two combinational outputs (port **comb_outputs**) that decode the current state of the machine. If these outputs were registered, rather than combinational, then a third process would have been written for the outputs, using the **clk** and **reset** inputs in the process sensitivity list.

Note: *Methods for specifying the encoding of enumerated types, and the impact that these types have on optimization results, are described later in this chapter in the section, "Enumerated Types".*

Feedback Mechanisms

All state machines require some form of feedback to implement the current state memory and next state decoding. There are two ways to create feedback in VHDL: using **signals** and using **variables**. The recommended method for state machines is to use signals, as shown in the previous template example.

Note: *As a general rule, use variables to pass data within a process, and use signals to pass data outside a process (between concurrent statements).*

Feedback on Signals

The following design demonstrates how signals are used to provide register feedback. This example uses a **process** and **if-then** statement to provide the clocking function for the flip-flop. The flip-flop output (**c**) is fed back and used in the assignment of combinational signal **b** in the second process:

```
architecture example of some_entity is
    signal b: bit;
begin
    process(clk)               -- a sequential process
    begin
      if clk = '1' and clk'event then   -- clock function
        if reset = '1' then
            c <= '0';          -- synchronous reset
        else
            c <= b;            -- load flip-flop from b
        end if;
      end if;
    end process;
    process (a, c)             -- a combinational process
    begin
        b <= a and c;
    end process;
end example;
```

When relating this circuit to a classic state machine, you can consider the signal **c** to the *current state* register, and signal **b** to the *next state* decode function.

For simple state machine circuits such as this one (or for somewhat more complex circuits such as counters), a more concise method of specifying the feedback can be used:

```
architecture example of some_entity is
begin
    process (clk)
    begin
      if clk = '1' and clk'event then
        if reset = '1' then
            c <= '0';
        else
            c <= a and c;  -- c is fed back directly
        end if;
      end if;
    end process;
end example;
```

This method eliminates the intermediate signal **b**.

Feedback on Variables

Variables exist within a **process**, and preserve their data as processes suspend and reactivate. Feedback is created whenever variables are used (placed on the right hand side of an assignment or used in a conditional statement) before they are assigned (placed on the left hand side of an assignment.)

Feedback paths typically contain registers, so to use feedback in a process you will usually want to insert a conditional statement that implies a clock into the process. The following example shows how registered feedback is created (to form a counter function) from the use of a variable in a process:

```
    process (clk)
        variable Count: integer range 0 to 255;
    begin
      if clk = '1' and clk'event then
        if reset = '1' then
            Count := 0;
        elsif Count = 255 then
            Count := 0;
        else
            Count := Count + 1; -- Counter feeds back
        end if;
      end if;
      C <= Count;
   end process;
```

In this example, the counter output (**C**) is assigned the value of the variable Count, creating an 8-bit counter on the design output. In the above example, eight flip-flops are generated for the output **C**, and the feedback logic for variable **Count** is folded into the logic for **C**.

Note: *Variables only imply registers when used within* ***process*** *statements. If a variable is declared inside a* ***function*** *or* ***procedure****, the variable is local to the subprogram; it exists only within the scope of the subprogram.*

Types of State Machines

Classical state machines can be classified as Moore or Mealy machines. In a Moore machine, the output is a function of the current state only, and can change only on a clock edge. Mealy machines, on the other hand, have outputs that are a function of the current state and the current inputs. The outputs of a Mealy machine may change when any input changes.

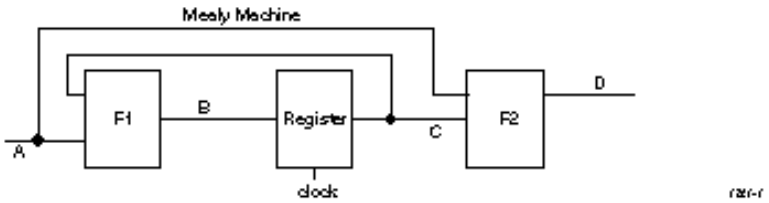
Moore Machine

In the following architecture, F1 and F2 are combinational logic functions of an arbitrary complexity. A simple state machine implementation maps each block to a VHDL process:

```
library ieee;
use ieee.std_logic_1164.all;
entity system is
    port (clock: in std_logic;
          A: in std_logic;
          D: out std_logic);
end system;
architecture moore1 of system is
    signal B, C: std_logic;
begin
    F1: process (A, C)         -- Next state logic
    begin
        B <= F1(A, C);
    end process;
    F2: process (C)            -- Output logic
    begin
        D <= F2(C);
    end process;
    Register: process (clock)  -- State registers
    begin
        if rising_edge(clock) then
            C <= B;
        end if;
    end process;
end moore1;
```

A block diagram that shows how the three processes of this architecture are related is shown in **Figure 3-6**.

Figure 3-6: Moore State Machine



A more compact description of this architecture could be written as follows:

```
architecture moore2 of system is
    signal C: std_logic;
begin
    Combinational: process (C)   -- combinational -logic
    begin
        D <= F2(C);
    end process;
    Registers: process (clock)   -- sequential logic
    begin
        if rising_edge(clock) then
            C <= F1(A, C);
        end if;
    end process;
end moore2;
```

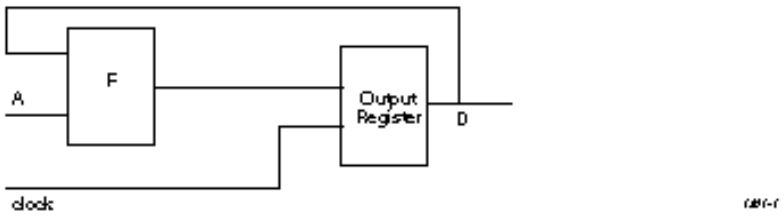
Output Registers

If the system timing requires that there be no logic between the registers and the output (the shortest output propagation delay is desired) then the following architecture can be used:

```
architecture moore3 of system is
begin
    process (clock)
    begin
        if rising_edge(clock) then
            D <= F(A,D)
        end if;
    end process;
end moore3;
```

This is the simplest form of a Moore state machine, and is diagrammed in **Figure 3-7**.

Figure 3-7: Moore3 State Machine



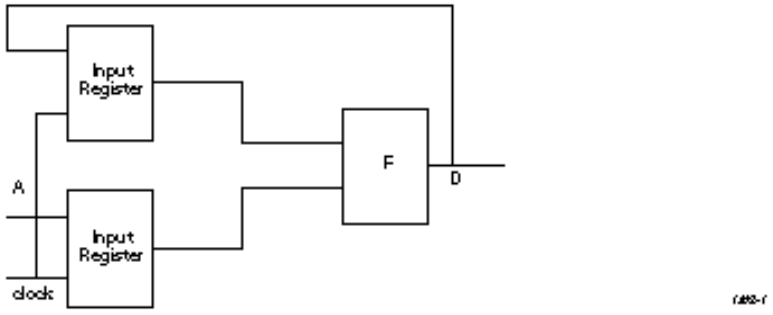
Input Registers

If the system timing requires no logic between the registers and the input (a short setup time is required) the following architecture can be used:

```
architecture moore4 of system is
    signal A1, D1 : std_logic;
begin
    Registers: process (clock)
    begin
        if rising_edge(clock) then
            A1 <= A;
            D1 <= D;
        end if;
    end process;
    F1: process (A1, D1)
    begin
        D <= F(A1,D1);
    end process;
end moore4;
```

The resulting circuitry is diagrammed in **Figure 3-8**. Note that this form of a state machine does not map well into most PLD devices.

Figure 3-8: Moore4 State Machine



Mealy Machine

A Mealy machine always requires two **processes** (or one process for the machine and separate concurrent statements for the outputs,) as its timing is a function of both the clock and the data inputs:

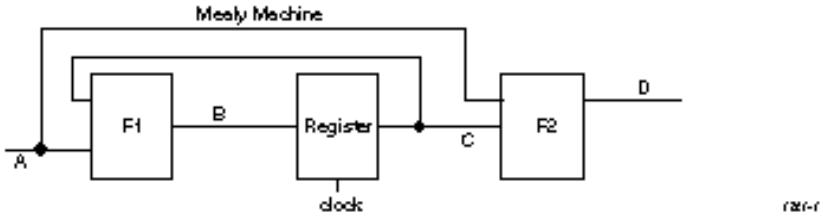
```
architecture mealy of system is
    signal C: std_logic;
begin
    Combinational: process (A,C) -- Mealy outputs
    begin
        D <= F2(A, C);
    end process;

    Registers: process (clock)   -- State machine logic
    begin
        if rising_edge(clock) then
            C <= F1(A, C);
        end if;
    end process;
end mealy;
```

Mealy state machines can be written more clearly, however, if three processes are used in the following style:

```
architecture mealy of system is
    signal C: std_logic;
    signal B: std_logic;
begin
    Registers: process (clock)  -- State register
    begin
        if rising_edge(clock) then
            C <= B;
        end if;
    end process;
    Transitions: process (A, C) -- Transition logic
    begin
        B <= F1(A, C);
    end process;
    Outputs: process (A,C) -- Mealy outputs
    begin
        D <= F2(A, C);
    end process;
end mealy;
```

Figure 3-9 shows a block diagram of this type of state machine. The block labeled F1 represents the combination logic function for the transition logic, while the block labeled F2 represents a combinational logic function of the state machine s current state and the design inputs.

Figure 3-9: Mealy State Machine



Avoiding Unwanted Latches

When describing state machines in VHDL, you must be careful to avoid the creation of unwanted asynchronous feedback paths that form latches. The rules of VHDL state that a signal within a process whose value is not completely specified (provided with an explicit assignment for all possible input conditions) will hold its previous value for the unspecified conditions. Latches can therefore be inadvertently created by incompletely specifying the transitions from one or more states in a state machine, or by failing to specify the value of all outputs in the states of the machine.

Perhaps the most common mistake made by new VHDL users (particularly those who have had experience with PLD-oriented languages) is in assuming that unspecified conditions will have no effect on the logic of the generated circuit. This is not the case in VHDL, and you need to be aware of the logic that will be generated for incompletely specified conditions.

To help you detect and correct situations such as this, the VHDL synthesizer will display a warning message whenever asynchronous feedback paths are generated. (You can view these warning messages by viewing the Process Log in the Project Navigator.)

4. How to Control the Implementation of VHDL

Using Enumerated Types

Enumerated types are very important in VHDL, and are the only way in which signals that require more than two values (such as those with output enables) can be described in the language.

For many circuits, most notably state machines, enumerated types can help to make complex designs more readable, and can also help to increase the efficiency of the resulting synthesized circuitry.

The VHDL synthesizer also gives you control over the encoding of enumerated types, by providing you with a special attribute called **enum_encoding**.

Enumerated types can reduce combinational logic requirements because they make it easy to add don t-care logic to a design. A don t-care (for the purpose of logic synthesis) is a circuit condition (a combination of inputs, for example) under which the resulting output of the circuit is not important. The VHDL synthesizer (and other software, such as devices fitters) can take advantage of don t-care conditions to reduce the amount of logic required to implement a combinational logic function.

A Review of Enumerated Types

Enumerated types in VHDL are user-defined (or predefined) data types that are specified using symbolic representations. The state machine template example presented earlier in this chapter was one example of an enumerated type, and was declared with the following statement:

```
type states is (st0, st1, st2, st3);
```

Many of the standard types used in VHDL are actually enumerated types defined in the standard library (std.vhd) or in the IEEE 1164 library (ieee.vhd). The types **bit** and **Boolean**, for example, are enumerated types with the following definitions (from std.vhd):

```
type boolean is (false,true);
type bit is ('0','1');
```

The type **std_ulogic** (from which **std_logic** is derived) is also an enumerated type, and has the definition (from ieee.vhd):

```
type std_ulogic is ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

The **std_ulogic** (or **std_logic**) data type is very important for both simulation and synthesis. Std_logic includes values that allow you to accurately simulate such circuit conditions as unknowns and high-impedance states. For synthesis purposes, the high-impedance and don t-care values provide a convenient and easily recognizable way to represent three-state enables and don t-care logic.

Synthesis of Enumerated Types

When synthesized into logic, enumerated types result in a binary encoding. Elements in the enumerated type are assigned numeric values from left to right, with the value of the leftmost element being zero. For example, the state values defined earlier would be assigned the binary values 00, 01, 10, and 11 for the states **st0**, **st1**, **st2** and **st3**, respectively.

If the number of elements in the enumerated type is not a power of 2 (for example, there are only three states, instead of four), the remaining binary encodings are processed as don t-cares by the VHDL synthesizer.

The default (binary) encoding for enumerated types is rarely the optimal encoding for a complex state machine, so it is important to have a way to override the default with an encoding more appropriate for the machine being described. The default binary encoding can be changed if needed, using the **enum_encoding** attribute that will be described in this section.

By default, the number of wires generated to encode an enumerated type is the smallest possible n, where the number of elements is 2^n. (It will, for example, require three wires to represent an enumerated type with more than four but less than nine different values.) The enum_encoding attribute allows you to specify no only the encoding of each member of the type, but the width of the encoding (number of wires) as well.

Enum_encoding attribute

The VHDL synthesizer supports a custom synthesis attribute for objects of enumerated types that allows alternate encodings to be specified. An alternate encoding may be required for a state machine that is described with enumerated types, or may be required to resolve multi-valued logic into a single-bit representation.

An example of a state machine design that uses enum_encoding to specify a particular (non-sequential) encoding of state values is presented in the tutorials chapter of the *VHDL Entry* manual. The prep4 (complex state machine) example presented in that chapter uses **enum_encoding** to specify an encoding that is optimized for a complex PLD implementation, as well as using explicit assignments to the - value to include don t-care information about the circuit s outputs.

An example of resolving multi-valued logic can be found in the definition for the **std_ulogic** data type (upon which the type **std_logic** is based). As described above, the **std_ulogic** data type is defined in the package **std_logic_1164** (which is located in the file **ieee.vhd**) to have nine possible values: U , X , 0 , 1 , Z , W , L , H and - . If these nine values were given the default binary enumerations, then any signal of type **std_ulogic** (or the derived type **std_logic**) would require four wires when synthesized, rather than the intended single wire. To solve this dilemma, the following **enum_encoding** attribute specification (which is provided in **ieee.vhd**) results in a single-bit (and non-unique) encoding for each value of the **std_ulogic** data type:

```
attribute enum_encoding: string;
attribute enum_encoding of std_ulogic: type is
    "- - 0 1 Z - - - -"
```

Enum_encoding is a custom attribute recognized by the VHDL synthesizer (and ignored by other tools, such as simulators) that maps the elements of an enumerated type into an actual encoding, using 1 of 4 possible enumeration values. The enumerations for each type are listed in the attribute string (with entries separated by a space character) and may consist of one or more of the characters shown in **Table 4-1**.

Table 4-1: Logical values for enumeration characters

Enumeration character	Logic Value
0	0
1	1
-	Don t care (for logic optimization)
Z	High impedance (implies an output enable)

Of these four possible values, only the **1** and **0** result in logic being generated for a signal, so an element of an enumerated type that is defined using any of the four characters results in only a single wire being generated. For enumerated type elements that are defined using the **-** character, the VHDL synthesizer generates a don t-care for the signal. This often results in better logic optimization. Enumerated type elements that are defined using the **Z** character result in output enable logic being generated for the associated signal. (This matches the convention for output enables commonly used in simulation environments.)

The **enum_encoding** attribute must be entered as a single string value. Strings in VHDL cannot span multiple lines, so to enter a long **enum_encoding** value, you must concatenate multiple strings using the & (concatenate) operator as shown in the following example:

```
attribute enum_encoding of states: type is
    "00101 00000 10000 00100 10100 " &
    "01100 01000 10101 10001 11000 " &
    "10011 00011 00001 01101 01001 11001";
```

One hot Enumeration

In addition to the enumeration values shown above, the VHDL synthesizer supports a special type of enumeration attribute value for state machines. The following statements define an enumerated type for a state machine and specify that the state machine is to be encoded using a one hot (one bit active per state) representation:

```
type machine is (init, s1, s2, s3, s4);
attribute enum_encoding of machine : type is "one hot";
```

In this example, the five states of the machine (init, s1, s2, s3, and s4) are encoded automatically with the values **00001**, **00010**, **00100**, **01000**, and **10000**, respectively by the VHDL synthesizer. When used as next and previous state values in a subsequent **case** statement, the value of the type is decoded using only that bit that is hot (has a value of 1) for each enumerated value. This results in a dramatic decrease in the amount of decoding logic required for each condition in the **case** statement logic, at the expense of a few additional registers.

Note: *One hot encoding is particularly effective when you are generating circuits intended for implementation in register-rich architectures such as FPGAs.*

Don t-cares and Enumerated Types

Unused encodings result in don t-care conditions, which allow the VHDL synthesizer to perform additional logic optimizations. Subtypes use the element encodings of their base, and may result in additional don t-care conditions being generated. For example:

The declaration:	Is synthesized as:
type direction **is** (left, right, up, down);	Two wires with the values 00, 01, 10, and 11.

type cpu_op **is** (execute, load, store);	Two wires with the values 00, 01, and 10; the binary encoding 11 is a don t-care.
subtype mem_op **is** cpu_op **range** load **to** store;	Two wires with the values 01 and 10; the encodings 00 and 11 are don t-cares.

Describing Output Enables

Output enables are commonly used in PLD-based designs, but the VHDL language does not define an explicit output enable function such as that found in ABEL-HDL or other PLD and FPGA oriented languages. To describe an output enable, you must write your VHDL design using the special enumerated value **Z** and the IEEE 1164 **std_logic** data type, or some other enumerated type that includes **Z** as a possible value.

Using Std_logic to Describe Output Enables

VHDL does not provide an explicit method for describing a signal that can be disabled, so various conventions have been adopted for this purpose. The most common way to describe a three-state signal is to use the IEEE 1164 **std_logic** data type, which includes a value of **Z** as one of its possible values. The following design uses the **std_logic** data type, and describes the disabled state of the output using the **Z** constant:

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

package example_bus is
    subtype bundle is std_logic_vector (0 to 4);
end example_bus;

use work.example_bus.all;
library ieee;
use ieee.std_logic_1164.all;
entity tbuf is
    port (enable: in std_logic;  a: in bundle;
          m: out bundle);
end tbuf;
architecture example of tbuf is
begin
    process (enable, a)
    begin
        if enable = '1' then
            m <= a;
        else
            m <= "ZZZZZ";
        end if;
    end process;
end example;
```

Note: *Output enables generated by the VHDL synthesizer correspond to the three-state output pins found in most PLD and FPGA devices. They do not correspond to internal tri-states such as those found in Xilinx FPGAs.*

Controlling Output Inversion

Many common PLDs feature inverted registered outputs, or outputs that have programmable inverters between the outputs of the flip-flops and the actual output pins. When designing for these devices, precise control over the polarity of the outputs is often required. Using properties of the VHDL synthesizer, you have three options available for controlling output polarities.

If the target device has outputs with fixed output inversion on all registered outputs, then the Invert Yes property of the Synthesize VHDL process should be specified when compiling your design. The Invert Yes property results in all registered ports in the design being generated with inverted outputs. This option does not affect the logic of the outputs as observed on the pins; instead it simply instructs the VHDL synthesizer to change the inversion of the output pins, then invert the internal logic polarity (and swap presets and resets as needed) to preserve the correct pin-to-pin behavior of the circuit.

If the target device has outputs with fixed non-inverted registered outputs, then you should specify Invert No when processing your design. This property results in all design outputs being generated with non-inverting outputs. Once again, the logic of the circuit as observed in the device output pins remains unaffected, regardless of how the outputs are described in the VHDL program.

If the target device features programmable output inversion for registered outputs (as is the case in the P22V10 and the smaller MACH devices) then you can either select Invert Yes or Invert No, or do not specify the inversion and let the VHDL synthesizer choose the output inversion based on how the outputs are used in the design description.

If the Invert property is not specified, the VHDL synthesizer attempts to infer the inversion of outputs based on the use of inverting signals in the design.

Note: *Most PLD Device Kits include a flip-flop transformation process that can adjust output polarities to match the architecture of the target device. For this reason, you will probably not have to use the Invert Yes or Invert No property to achieve a successful fit.*

The following example shows two design outputs. The **o2** output is described in such a way that it will result in inverted outputs (using a variable to provide the inversion) while the **o1** output is described in a way that will result in a non-inverted implementation. (Both **o1** and **o2** have the same pin-to-pin behavior.)

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

entity polarity is
    port (clock : in std_logic;
          a,b: in std_logic;
```

```
         o1,o2: out std_logic);
end polarity;

architecture inversion of polarity is
    signal n2: std_logic;
begin
    process(clock)
    begin
        if rising_edge(clock) then
            o1 <= (a and b);
            n2 <= (not a or not b);
        end if;
    end process;
    o2 <= not n2;
end inversion;
```

Controlling Feedback Paths

If the design description specifies feedback, then the VHDL synthesizer will generate the feedback logic according to how the fed back signal was used in the design. There are two basic types of feedback generated from the VHDL synthesizer: macrocell (register) feedback and pin feedback. The source of the feedback (register or pin) may be controlled by using appropriate VHDL coding conventions.

If a port of type **inout** is both read from and written to, (as is the case for bi-directional I/O) then the pin feedback path is generated. If the specified port is written to only and the feedback is assigned to a signal, the register feedback is generated. A port that is both read and written must be of mode **inout**, and a port that is written to only may be either **out** or **inout**.

For example, consider two implementations of a 3-bit counter. The first counter uses the register feedback path, and the second uses the pin feedback path.

In the first case, the counter is specified by a variable named **count1**, and the output of the counter drives the pin **count**. This describes register feedback, since the value is fed back with the variable **count1**.

```
entity counter0 is
    port (clock: in Boolean;
          count: out integer range 0 to 7);
end counter0;

architecture register_feedback of counter0 is
begin
    process (clock)
        variable count1: integer range 0 to 7;
    begin
        if clock and clock'event then
            if count1 = 7 then
                count1 := 0;
            else
                count1 := count1 + 1;
            end if;
        end if;
        count <= count1;
    end process;
end register_feedback;
```

In the second case, the counter is described directly with the port **count**. Note that count is now an **inout** so that it may be read from and written to. The counter feedback uses the VHDL port directly, and will therefore result in the pin feedback path being generated.

```
entity counter1 is
    port (clock: in Boolean;
          count: inout integer range 0 to 7;
end counter1;
architecture pin_feedback of counter1 is
begin
    process (clock)
    begin
        if clock and clock'event then
            if count = 7 then
                count <= 0;
            else
                count <= count + 1;
            end if;
        end if;
    end process;
end pin_feedback;
```

Note: *Mode* ***inout*** *should only be used to describe true directional ports those that have an output enable function associated with them. Using* ***inout*** *to describe signals that are simply fed back to create circuits such as counters is not recommended.*

Selecting a Base Data Type

An important consideration when starting a VHDL design project is the data type upon which your design is to be based. Typically, you will use one of the following types:

- Integer
- Bit and bit_vector
- Std_ulogic and std_ulogic_vector
- Std_logic and std_logic_vector
- Std_logic and the unsigned/signed data types defined by IEEE 1076.3

The default type for wires and pins in Synario's schematic editor is std_logic (or std_logic_vector for busses). If you wish to use a different type on schematics, you must either change the net_type and bus_type settings in the \synario\config\vhdl.ini file, or change the type of individual wires and buses by setting the VHDL_NetType VHDL_BusType attribute on the schematic. Refer to the *Netlist Application Note* available through online help for details.

The advantages and disadvantages of designing using different base types is discussed below.

Using the Integer Type

Using an **integer** type has the advantage that all the normal arithmetic operators are built into the VHDL language for this type. Another advantage of using integers is that they use memory efficiently during simulation. To understand why this is so, consider the following declarations:

```
signal a_int : integer range 0 to 255;
signal a_vec : bit_vector(7 downto 0);
```

Both a_int and a_vec are capable of representing exactly the same range of values. Note that the declaration of a_int defines one signal, while the declaration of a_vec really defines eight signals (one per bit of the vector). To support attributes, there is a variety of information that a VHDL simulator must store for each signal (such as the last value, whether the signal is active or not, and so on), so using an integer is more efficient.

There are several disadvantages to using integers, however. The first is that there is no way for an integer to represent several common logic states such as unknown, tristate, or don t-care. The second problem is that when writing VHDL code that performs integer operations, you must include extra code to check for boundary conditions. Consider the following code fragment:

```
signal a, b, z : integer range 0 to 7;
z <= a + b;
```

The problem with this code is that whenever the sum of a and b exceeds 7, a fatal error will occur during simulation. To avoid this, the code must be re-written to handle this boundary condition:

```
process(a, b)
    variable a_var, b_var : integer range 0 to 15;
begin
    a_var := a;
    b_var := b;
    if (a_var  +  b_var > 7) then
        z <= a_var + b_var - 8;
    else
        z <= a_var + b_var;
    end if;
end process;
```

This model correctly handles overflow conditions, but at great expense in complexity of the code. More importantly, the synthesizer has no way of knowing that this code is only present to handle simulation issues, and will synthesize more logic than is really needed to implement a 3-bit, unsigned adder.

Using Bit and Bit_vector Types

Types **bit** and **bit_vector** have the advantage of being built into the VHDL language, and are therefore highly portable. These types also have many disadvantages, however. The first is that built-in arithmetic operators are not defined for **bit_vector**. You can overcome this limitation by using the package bit_ops contained in the file \synario\lib5\dataio.vhd supplied with your Synario installation. A second and more serious limitation is the fact that **bit** may represent only two states, 0 and 1 . Therefore, it is impossible to model tristate or wired logic properly using the **bit** type.

Using Std_ulogic and Std_ulogic_vector Types

Types std_ulogic and std_ulogic_vector are attractive since they allow modeling of unknown, tristate, and other such logic states. As unresolved types, though, they may not be used to model wired logic or tristate buses. Also, there are no overloaded arithmetic operators defined for std_ulogic_vector.

Using Std_logic and Std_logic_vector Types

Types std_logic and std_logic_vector offer numerous advantages. As resolved types, they not only have the necessary logic states to model tristate and wired logic, but they also allow for multiple drivers on the same signal. Overloaded arithmetic operators are provided for std_logic_vector in the package std_logic_ops found in the file dataio.vhd. Overloaded textio functions are available as part of the package std_logic_textio found in the file \synario\lib5\stdtxtio.vhd. In addition, all new IEEE packages and libraries (such as VITAL or 1076.3) are based on std_logic. A final advantage of std_logic and std_logic_vector is that it is the assumed type for wires and buses drawn on a Synario schematic.

Using IEEE 1076.3 Unsigned/Signed Types

In late 1995 the IEEE 1076.3 commitee approved a new VHDL package called numeric_std. This package defines two vector types, unsigned and signed, and the appropriate overloaded operators for these types. Both unsigned and signed are based upon the std_logic type. This has several implications:

- You can not assign a signal of type std_logic_vector to one of type unsigned (or signed) or visa-versa. For example:

  ```
  signal slv : std_logic_vector(3 downto 0);
  signal uns : unsigned(3 downto 0);
  .
  .
  slv <= uns;
  ```

  will not compile.

- Since the post-route netlists produced for VHDL simulation are usually defined in terms of std_logic and std_logic_vector, is is impossible to "plug in" a post-route netlist into a testbench in place of a functional design that had ports of type unsigned or signed.

For these reasons, it is recommended that types unsigned/signed only be used in places where the overloaded functions defined in the numeric_std package are needed. Explicit type conversions can be performed at the point of use, for example:

```
signal a,b,z : std_logic_vector(3 downto 0);
.
.
z <= std_logic_vector(unsigned(a) + unsigned(b));
```

In this code fragment, the vectors a and b are converted to unsigned, summed using the overloaded '+' operator defined by numeric_std, and the result is then converted back to a std_logic_vector type before being assigned to z. If you prefer to use unsigned/signed throughout your design, it is suggested that you still make your top-level inputs and outputs type std_logic_vector, and convert them in your top level module before using them throughout the rest of your design.

Synthesis of Don't Cares

If you are designing using the type std_logic as your base type (as recommended in the previous section) you are probably aware that one of the values in the enumeration of std_logic is '-', which is *commented* in the std_logic_1164 package as meaning "don't care". Unfortunatly, none of the functions defined by 1164 actually give '-' such a meaning. For example, the following expression:

```
if ('1' = '-') then
...
end if;
```

will always evaluate to false during simulation.

Prior versions of Synario's VHDL synthesis compiler evaluated the '-' the same as '0' in relational expressions.The IEEE 1076.3 standard has defined new semantics for the '-' value that Synario now conforms to. In particular, any use of the '-' value in an expression involving one of the relational operators (=, /=, >, etc.) is defined to always return false. To get the simulation and synthesis semantics of "don't cares" it is necesary to use the std_match functions defined in the new numeric_std package. For example:

```
signal a,b,z : unsigned(3 downto 0);
.
.
z <= a when (std_match(b, "000-")) else "0000";
```

which makes the assignment to z conditional on only the left-most 3 bits of b.

Using Device Fitting Attributes

The VHDL synthesizer supports a small number of custom attributes that can be used to control features of synthesis that are not normally accessible from the VHDL language. These synthesis features include control over pin placement, encoding of enumerated types and other important optimization and device-related capabilities. All of the custom attributes are defined in the file **metamor.vhd**, which can be included in your VHDL source file with the following **library** and **use** statements:

```
library metamor;
use metamor.attributes.all;
```

Alternatively, the required attribute declarations can be entered directly into your VHDL source file. The declaration syntax for each attribute is shown in the following subsections.

Pinnum Attribute

You can specify pin and node numbers in the VHDL source file using the custom attribute **pinnum**. special attribute is recognized by the VHDL synthesizer and is written to the output for use by device fitting software. This attribute has no other meaning to the VHDL synthesizer, or to any other VHDL software (such as a simulator). The information is simply passed on to the device fitting software.

To use the pinnum attribute:

1. Declare the **pinnum** attribute in the top-level entity as follows:
   ```
   attribute pinnum : string;
   ```
2. Specify the value of **pinnum** attributes as strings, as in the following example:
   ```
   entity my_design is
        port(a, b : in integer range 0 to 7;
             c: bit_vector (3 to 5);
             d: bit_vector (27 downto 25);
             e: out Boolean);

       attribute pinnum: string;
       attribute pinnum of c: signal is "1,2,3";
       attribute pinnum of d: signal is "6,5,4";
       attribute pinnum of e: signal is "2";

   end my_design;
   ```

To specify pin numbers for an array of signals:

You must consider the direction of the array. In the following example, the pin numbers being assigned to each element of the arrays **A** and **B** correspond to the index numbers for each array element:

```
entity toplevel is
     port (Clock, reset: in std_logic;
           A out std_logic_vector(1 to 3);
           B out std_logic_vector(6 downto 4);
    attribute pinnum: string;
    attribute pinnum of Clock: signal is "1";
    attribute pinnum of reset: signal is "19";
    attribute pinnum of A: signal is "1,2,3";
    attribute pinnum of B: signal is "6,5,4";
end toplevel;
```

If the **pinnum** attribute is applied to signals that are ports of the top-level entity in the design (as in the above example), the values specified in the attribute will be passed to the fitting software as pin numbers. If the **pinnum** attribute is applied to signals that are internal to an architecture, or are ports of a lower-level entity in the design, the values will be ignored.

Note: *Pin numbers that are entered using the **pinnum** attribute may be overridden by device fitting software, if the software is unable to map the design as specified. Most device fitting software includes Project Navigator properties that can be used to control how pin number mapping is performed. Refer to the device kit documentation for more information.*

To specify node numbers:

1. Declare the signal in the entity (not the architecture).
2. If the signal is combinational, associate a critical attribute with it, as in the following example:

```
library metamor;
use metamor.attributes.all;
entity nnum1a is
   port ( A, B, Clk:   in     boolean ;
          o1:          out    boolean);
   attribute pinnum of A  :  signal is "1";
   attribute pinnum of B  :  signal is "2";
   attribute pinnum of Clk  :  signal is "3";
   attribute pinnum of o1  :  signal is "4";
   -- maps to NODE #20 for combinational signal
   signal t1: boolean;
   attribute pinnum of t1  :  signal is "20";
   attribute critical of t1  :  signal is true;
   -- maps to NODE #21 for registered signal
   signal t2: boolean;
   attribute pinnum of t2  :  signal is "21";
end;
```

Property Attribute

The **property** attribute is used to pass device-specific fitting data to device fitting or place-and-route software. The attribute strings are passed to the fitter software (through the use of Open-ABEL property fields in the intermediate data files) exactly as specified in the **property** attribute statements. The case is maintained, and the syntax of the strings is not checked or parsed in any way by the VHDL synthesizer.

To use the **property** attribute, you must first declare the attribute as a string using the following statement:

```
attribute property: string;
```

and then write one attribute statement containing all properties needed for the entire design:

```
entity example is
    port (Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0: in bit;
          D3,D2,D1,D0: out bit);
    attribute property: string;
    attribute property of example: entity is
        "AMDMACH GROUP A Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0" & CR &
        "AMDMACH GROUP B D3 D2 D1 D0";
end example;
```

In VHDL, it is not legal to have a newline imbedded within a string, so to create multiple properties it is necessary to concatenate strings with new lines using the & CR & syntax as shown.

If you will be assigning properties to signals that are part of non-binary data types (such as **bit_vector**, **integer** or **std_logic_vector**), then you will need to be aware of how the VHDL synthesizer generates signal names from these data types. When the VHDL synthesizer expands an array data type into signals for each bit, it appends a numeric suffix to the array name in the format *_n_*, where *n* is the array index for each element of the data type. The following example shows how to write a property for **std_logic_vector** data types.

```
library ieee;
use ieee.std_logic_1164.all;
entity example is
  port (Q: in std_logic_vector (7 downto 0);
        D: out std_logic_vector (3 downto 0));
    attribute property: string;
    attribute property of example: entity is
    "AMDMACH GROUP A Q_7_ Q_6_ Q_5_ Q_4_ Q_3_ Q_2_ Q_1_ Q_0_"
    & CR &
    "AMDMACH GROUP B D_3_ D_2_ D_1_ D_0_";
end example;
```

The actual property string that you must enter in the attribute statements are defined in the appropriate device-specific documentation provided with the device kit being used.

Macrocell Attribute

The **macrocell** attribute allows components in a design to be flagged as external, device-specific macrocells in a hierarchical design. Components that are specified with the **macrocell** attribute do not have VHDL source files associated with them (unless VHDL source files have been written or provided for simulation purposes), and are resolved by the device fitting software during device mapping.

When the **macrocell** attribute is specified for a component, the resulting output will include a hierarchical reference to the specified external module, and the Project Navigator will not attempt to find a source module in the design to resolve the hierarchy.

The following example demonstrates how the **macrocell** attribute can be used:

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

entity top is
    port (a: in std_logic;
          b: in std_logic;
          out1: out std_logic;
          out2: out std_logic );
end top;
architecture schematic of top is
    component submod_def
      port (in1: in std_logic;
            in2: in std_logic;
            and1: out std_logic;
```

```
            or2: out std_logic);
    end component;
    attribute macrocell: boolean;
    attribute macrocell of submod_def: component is true;
begin
    myblock: submod_def
        port map(in1=>a, in2=>b, and1=>out1, or2=>out2);
end schematic;
```

In this example, the hierarchical reference to the **submod_def** design unit will result in a reference being generated for an external module named **submod_def**. The VHDL synthesizer will not attempt to find or synthesize the **submod_def** design unit.

To use the **macrocell** attribute when referencing hard and soft macros, you must specify the actual name of the macro, as defined in the appropriate device kit documentation, and provide component port mapping that exactly matches the argument list of the hard or soft macro. Refer to your device kit documentation for information about available hard or soft macros.

Critical Attribute

The **critical** attribute allows you to flag signals used in your design as nodes to be preserved in the final implementation. This is particularly useful for debugging. To flag a signal as a node (and prevent its being collapsed out of the design), you must first define the **critical** attribute as a Boolean type:

```
attribute critical: boolean;
```

and then write an attribute statement for the desired signal (or signals):

```
attribute critical of a,b,c: signal is true;
```

If you have some knowledge of the structure of your design, you may be able to improve the timing characteristics of a large, combinational function by using the **critical** attribute to flag certain signals in the design. One example of this is a carry chain, in which a **critical** flag applied to each carry signal can result in the individual logic blocks operating faster than if the entire combinational circuit was optimized for a minimum gate count.

Note: *When you specify the **critical** attribute for a signal, the resulting preserved node (or nodes) may have an unexpected name. For example, if the signal is used at a lower level in a hierarchical design, it is possible that the name will be prefixed with instance or block labels, or will have a numeric suffix. If the signal is redundant, it may be reduced out of the design entirely, even with the **critical** attribute. Device fitting software may also modify or remove signals that have been flagged as **critical**, depending on the device-specific optimizations performed by the software.*

Enum_encoding Attribute

The **enum_encoding** attribute allows you to override the default encoding of VHDL enumerated types. This is most useful when you want to specify a non-sequential encoding for an enumerated type. For example, you might want to use **enum_encoding** to specify the exact encoding of each state in a state machine. The **enum_encoding** attribute can also be used to encode don t-care information into the logic of an enumerated type output, or can be used to define synthesizable 1-bit values for a multi-valued data type (such as the IEEE 1164 type **std_ulogic**).

The syntax for the **enum_encoding** attribute is demonstrated in the following example, which specifies an alternate encoding for a state machine. To use the **enum_encoding** attribute, you must first define it as a string type:

```
attribute enum_encoding: string;
```

and then write an attribute statement for the desired encoding values:

```
architecture behavior of state_machine is
    type state_values is (Init, S1, S2, S3, S4, S5);
    attribute enum_encoding: string;
    attribute enum_encoding of state_values: type is
        "000 001 011 111 110 100";
```

A special **enum_encoding** string, "one hot", can be used to specify that an enumerated type represents a list of symbolic state machine states, and is to be encoded using a one register per state (one hot) encoding method. This method is particularly useful for state machines that will be implemented in FPGA devices, and results in less combinational logic being required for state decoding. The syntax for the one hot encoding attribute is:

```
architecture behavior of state_machine is
    type state_values is (Init, S1, S2, S3, S4, S5);
    attribute enum_encoding: string;
    attribute enum_encoding of state_values: type is
        "one hot";
```

An example of one hot state encoding can be found in the tutorials chapter in the *VHDL Entry* manual, in the complex state machine example.

5. VHDL Datapath Synthesis

A common problem with using VHDL synthesis for FPGA and PLD design is that synthesis tools often to a relatively poor job of implementing datapath logic (wide adders, counters, multipliers, and the like). There are a number of potential reasons for this:

- Datapath functions such as adders have many different potential implementations, all representing different speed/density trade-offs.
- Many FPGA and PLD vendors offer optimized libraries of datapath functions, designed for efficient implementation in their specific architecture.
- Synthesis tools to break all logic down to the level of Boolean equations so that datapath functions are no longer recognizable as such.

In order to avoid the efficiency issues that crop up when datapath functions are decomposed into Boolean equations, Synario s VHDL synthesis compiler has the ability to infer the use of certain common datapath functions from your VHDL code and extract them from your design. This inferencing capability is based on the LPM (Library of Parameterized Modules) Specification, which is simply a industry-standard set of variable-width macrofunctions such as adders, counters, etc. After these macrofunctions are extracted from your design they are mapped to a specific implementation based upon the chosen target device.

LPM inferencing from VHDL is currently supported for three Synario Device Kits; Actel, Altera, and Actel. For Xilinx designers, a similiar capability is available through the X-BLOX standard, consult the LCA Device Kit manual for details.

There are two ways to take advantage of Synario s datapath synthesis and mapping capability in your VHDL designs:

- Allow Synario s VHDL compiler to infer the use of LPM macrofunctions from generic VHDL code that you write.
- Instantiate macrofunctions from Synario s Generic Datapath (gen_dp) library.

How Inferencing Works

Inferencing is currently supported for the following macros:

- ADD_SUB
- MULT
- COUNTER
- COMPARE

Inferencing is performed based upon a combination of the operators (+, -, >, etc.) that appear in your VHDL code and the context in which those operators appear. For example,

```
signal a, b, c: std_logic_vector(7 downto 0);
c <= a + b;
```

would infer an ADD_SUB configured to perform addition.

Similarly, the code

```
signal clk : std_logic;
signal a : std_logic_vector(7 downto 0);
process(clk)
begin
   if (rising_edge(clk)) then
      a <= a + 1;
   end if;
end process;
```

would infer an 8 bit COUNTER.

In both of these examples, the overloaded operator '+' is used, but the context is different.

Controlling Datapath Inferencing

The inferencing feature of Synario s VHDL compiler is automatically enabled if you are targeting a device where this feature is supported. You may disable it by changing the properties for Synthesize Logic.

To disable inferencing :

1. In the Sources list in Project Navigator, click once to highlight a VHDL source file.
2. In the Processes list, highlight the Synthesize Logic process.
3. Click the Properties button.
4. Set the LPM Inferencing property to False.

To enable inferencing for certain source files and not for others, use Synario s Strategy feature. This feature allows you to associate one or more source files in your design with a particular synthesis and optimization style . See Strategies in the Synario online help for more information on using the Strategy feature.

To View the Results of Inferencing:

1. Select the Synthesize Logic process, and click on the Properties button.
2. Set the property Show Inferred Structure to True.
3. Run the Synthesize Logic process for the module.
4. Click the Log button. The Synario Report Viewer opens on the log file.

See Figure 5-1 for an example of the inferencing portion of a log file.

Figure 5-1: Example of an Inferencing report in the Synario Log File

```
process :   accumulate                    4.00 sec
     Inferred structure :
       flip flop: ce       fdata_t 0        filter.vhd line 87
       flip flop: ce       fdata_t 1        filter.vhd line 87
       flip flop: ce       fdata_t 2        filter.vhd line 87
       flip flop: ce       fdata_t 3        filter.vhd line 87
       flip flop: ce       fdata_t 4        filter.vhd line 87
       flip flop: ce       fdata_t 5        filter.vhd line 87
       flip flop: ce       fdata_t 6        filter.vhd line 87
       flip flop: ce       fdata_t 7        filter.vhd line 87
       macrocell:          LPM_ADD_SUB
```

Limitations of Inferencing Support

There are some limitations to Synario's support for inferred datapath macrofunctions:

1. ADD_SUB and COMPARE macros will not be inferred when one or more of the operands of a function are constants (e.g. a+1, b < 2, etc.).
2. Only UNSIGNED arithmetic is supported.
3. Not all datapath macrofunctions are equally well supported in all target architectures. Some targets may not have the ability to implement certain features. For example, the Lucent Orca architecture does not support both asynchronous presets and sets on flip-flops. Therefore, a COUNTER macrofunction that uses these ports can not be mapped to an Orca implementation. This will result in Synario generating an error such as the one shown below when it attempts to map the COUNTER for an ORCA device:

```
Unable to Map Parameterized Module of Type: LPM_COUNTER, Instance
Name: p63_0 to Vendor macro.
Reason for failure was:
   LPM module type LPM_COUNTER can not be mapped if the port ACLR
is connected.
Fatal Error 19254: Mapping problem forces termination. Try
resynthesizing the source without LPM inferencing
Done: failed with exit code: 0002.
```

If this happens because of an inferred macrofunction, turn LPM inferencing off. If it happens because of an instantiated macrofunction, either replace that macrofunction, or consider using it in a different manner. For example, in this case using a synchronous reset will map to an Orca device.

In all cases be sure to consult the Device Kit manual for your target device, it will contain details on any limitations that apply to that device.

Examples of How to Infer Datapath Macrofunctions

This section will discuss the particulars of how to write code to infer the use of different datapath macrofunctions.

ADD_SUB

The following are examples of code that will infer an ADD_SUB:

```
c <= a + b;

p0: process (d, e)
begin
   f <= d+e;
end process;

p1: process(clk)
begin
   if (rising_edge(clk)) then
      i <= g - h;
   end if;
end process;
```

The first two cases, a simple signal assignment statement and a combinational process, are equivalent. Process p1 will infer an ADD_SUB, configured to do subtraction and with the output registered.

There is currently no way to infer use of the Add_Sub or Cin ports of an ADD_SUB. Therefore, the following examples will each infer two ADD_SUBs:

```
l <= j + k + my_carry_bit;

p2: process(my_control,  m, n)
begin
   if (my_control = '1') then
      o <= m + n;
   else
      o <= m - n;
   end if;
end process;
```

For these cases, direct instantiation of the G_ADD_SUB macrofunction from the Synario Generic Datapath library is the solution, see the section on instantiation.

COUNTER

Inference of the COUNTER macro works very similarly to that for the ADD_SUB. Examples of code that will infer counters are:

```
signal a : std_logic_vector(3 downto 0) := (others => '0');

p0: process(clk)
begin
   if (rising_edge(clk)) then
      a <= a + '1';
   end if;
end process;

signal c : std_logic_vector(7 downto 0) := (others => '1');

p1: process(clk, my_async_ctrl)
begin
   if (my_async_ctrl = '0') then
      c <= (others => '1');
   elsif (rising_edge(clk)) then
      if (my_state =  load_state) then
            c <= b;
      else
            c <= c + 1;
      end if;
   end if;
end process;
```

Process p0 infers a simple COUNTER with only the Clock and Q ports connected. The initial state of signal a during simulation would be "UUUU", and this code would fail to count properly except that we have used an initial assignment on the declaration of signal a. If you use a default assignment like this to get proper simulation behavior, be sure that your choice of a default value matches the register power-up state in your target device, otherwise mismatches will occur between the functional and timing simulations and your device will not work as intended.

Process p1 infers a COUNTER configured as an up counter with a low-active asynchronous preset and synchronous load. Note that here we are assuming that our registers will power-up to the logic 1 state.

COMPARE

The COMPARE macrofunction will be inferred from the use of any of the following relational operators: <, <, >=, <=, =, /=. A COMPARE will only be inferred when both of the operands are not constant values. The following code will infer three COMPAREs:

```
p0: process (a, b, my_state)
begin
   c <= '0';
   case (my_state) is
      when "00" =>
            if (a = b) then
                  c <= '1';
            end if;
      when "01" =>
            if (a /= b) then
                  c <= '1';
            end if;
      when others =>
            if (a > b) then
                  c <= '1';
            end if;
   end case;
end process;
```

Note that the VHDL synthesis compiler will only infer COMPARE macrofunctions with a single output (this is the least common denominator). Some targets (such as Altera) support COMPARE functions with multiple outputs. In these cases it may be more efficient to instantiate a single G_COMPARE directly, rather than letting the synthesis compiler infer multiple COMPAREs. See the section on instantiation for more details.

MULT

The MULT macrofunction will be inferred from any use of the * operator where both operands are not constant. For example:

```
z <= a * b;

process(clk)
begin
   if (rising_edge(clk)) then
      iq <= i * q;
   end if;
end process;
```

The only difference in these two examples is that in the second case the outputs of the MULT will be registered.

Inferencing Details

This section provides additional details on inferencing.

Supported Types

Inferencing is supported for operands of type bit_vector, std_logic_vector, IEEE unsigned, and integer. If you are synthesizing designs originally developed in the Synopsys environment, the type unsigned in the package stdarith.std_logic_arith may also be used.

In general, integer is a poor choice. The reason for this can be seen by examining the following code fragment, which will infer an ADD_SUB:

```
signal a, b, c: integer range 0 to 255;
c <= a + b;
```

For situations where the sum of a+b is greater than 255, a fatal assertion will occur during simulation. This is different than the behavior of an actual ADD_SUB macro, which will rollover when the sum is greater than 255.

The behavior of code written using the overloaded arithmetic operators supplied in either the dataio.std_logic_ops package (for std_logic_vector) or in the IEEE package numeric_std (for types unsigned and signed) will mimic the behavior of the LPM macrofunctions that are inferred, and therefore we recommend the use of these types.

Matching Semantics

In order for datapath macrofunctions to be inferred, the simulation semantics of VHDL code must match the semantics of the corresponding macrofunction. For example:

```
process(clk)
begin
   if (rising_edge(clk)) then
       if (load = '1') then
              a <= b;
       elsif (clk_en = '1') then
          a <= a + 1;
       end if;
   end if;
end process;
```

would not infer COUNTER. The reason for this is that the clock enable input to an LPM COUNTER must override the load input, and this is not the case for the code fragment shown.

Resource Sharing

Consider the following simple ALU:

```
p0: process(operand0, operand1, operand2, opcode)
begin
   case(opcode)
      when  ADD_OPERAND1 =>
            result <= operand0 + operand1;
      when SUB_OPERAND1 =>
            result <= operand0 - operand1;
      when ADD_OPERAND2 =>
            result <= operand0 + operand2;
      when others =>
            result <= operand0 - operand2;
   end case;
end process;
```

Because resource sharing is not currently supported, this code will infer 4 different ADD_SUB macros. A better approach would be to instantiate a single ADD_SUB, and generate the inputs to it from a process:

```
p0: process(opcode)
begin
   case(opcode)
      when ADD_OPERAND1 =>
            add_sub <=  '1';
            b <= operand1;
      when SUB_OPERAND1 =>
            add_sub <=  '0';
            b <= operand1;
      when ADD_OPERAND2 =>
            add_sub <=  '1';
            b <= operand2;
      when others =>
            add_sub <=  '0';
            b <= operand2;
   end case;
end process;

p1: add_sub
generic map (width => 8, representation => "UNSIGNED")
port map(dataa => operand0, datab => b, add_sub => add_sub, sum =>
result);
```

Essentially, you should separate datapath from control logic, instantiating the datapath and leaving the control logic for synthesis.

Instantiation Details

If you are planning to do direct instantiation of the Synario Generic Datapth macrofunctions, you will need to add the following library/use statements to your source code:

```
library gen_dp;
use gen_dp.components.all;
```

The source code for this package can be found in <SYNARIO>\lib5\gen_dp.vhd, where <SYNARIO> refers to your Synario installation directory. This package contains the component declarations for all supported macrofunctions, and should be consulted when doing direct instantiation.

Functional Description

The following tables describe the functional behavior of the datapath macrofunctions.

G_ADD_SUB

The following table describes the ports of the G_ADD_SUB:

Port	Usage	Default Value	Description	Comments
Dataa	Required	None	First data input	Size equals WIDTH generic.
Datab	Required	None	Second data input	Size equals WIDTH generic.
Add_sub	Optional	Logic 1	Controls whether add or sub function is performed.	Defaults to add.
Cin	Optional	Logic 0	Carry in, active high.	
Sum	Optional	None	Sum output.	Sum = Dataa +/- Datab +/- Cin
Cout	Optional	None	Carry out output.	Indicates overflow for unsigned arithmetic.
Overflow	Optional	None	Overflow output	Has no meaning for UNSIGNED arithmetic.

G_ADD_SUB also has the following generics:

Generic	Usage	Description	Comments
Width	Required	The width, in bits, of the Dataa, Datab, and Sum ports.	Must be an integer value >= 2.
Representation	Required	Indicates whether UNSIGNED or SIGNED math is to be performed.	Only the value UNSIGNED is currently supported.

An example of an instantiated G_ADD_SUB is:

```
i_addsub0: g_add_sub
  generic map (width => 8, representation => "UNSIGNED")
  port map(dataa=>din0, datab=>din1, sum=>add_sub_out);
```

Note that the REPRESENTATION generic must be assigned a valid value, even though it is defined to have default value in the component declaration. This is true for all datapath macrofunctions.

G_COMPARE

The following table describes the ports of the G_COMPARE:

Port	Usage	Default Value	Description	Comments
Dataa	Required	None	First data input	Size equals WIDTH generic.
Datab	Required	None	Second data input	Size equals WIDTH generic.
Alb	Optional	None	High if Dataa < Datab	Not(Alb) = Dataa >= Datab
Aeb	Optional	None	High if Dataa = Datab	Not(Aeb) = Dataa /= Datab
Agb	Optional	None	High if Dataa > Datab	Not(Agb) = Dataa <= Datab

G_COMPARE also has the following generics:

Generic	Usage	Description	Comments
Width	Required	The width, in bits, of the Dataa, Datab, and Sum ports.	Must be an integer value >= 2.
Representation	Required	Indicates whether UNSIGNED or SIGNED math is to be performed.	Only the value UNSIGNED is currently supported.

An example of an instantiated G_COMPARE is:

```
i_compare0: g_compare
    generic map (width => 8, representation => "UNSIGNED")
    port map(dataa=>din0, datab=>din1, alb=>a_lt_b, aeb=>a_eq_b,
    agb=>a_gt_b);
```

G_COUNTER

The following table describes the ports of the G_COUNTER:

Port	Usage	Default Value	Description	Comments
Data	Optional	None	Data input	Size equals WIDTH generic.
Clock	Required	None	Clock input	Rising edge sensitive clock input.
Enable	Optional	Logic 1	Active high clock enable	Defaults to enabled.
Updown	Optional	Logic 1	Up/down counter control	Counter counts up when high, counts down when low. Defaults

				to up counter.
Aset	Optional	Logic 0	Asynchronous set input	Sets all Q outputs high
Aclr	Optional	Logic 0	Asynchronous clear input	Sets all Q outputs low
Aconst	Optional	Logic 0	Asynchronous set/clear input	Sets the Q output to the value of the AVALUE generic. If used, neither the Aset or Aclr inputs can be used.
Aload	Optional	Logic 0	Asynchronous load input	Sets the Q output to the value of the Data input.
Sset	Optional	Logic 0	Synchronous set input	Sets all Q outputs high on the next rising edge of Clock with Enable high.
Sclr	Optional	Logic 0	Synchronous clear input	Sets all Q outputs high on the next rising edge of Clock with Enable high.
Sconst	Optional	Logic 0	Synchronous set/clear input	Sets the Q output to the value of the SVALUE generic on the next rising edge of Clock with Enable high. If used, neither the Sset or Sclr inputs can be used.
Sload	Optional	Logic 0	Synchronous load input	Sets the Q output to the value of the Data input on the next rising edge of Clock with Enable high. If used, the Data input must be used.
Testenab	Optional	Logic 0	Active high test enable.	When this input is high, COUNTER enters a test mode and will serially shift data through from the Testin input to the Testout output.
Testin	Optional	Logic 0	Serial test data input	The value on this input is shifted into Q(0) on the rising edge of Clock with Testenab active.
Testout	Optional	None	Serial test data output	This output is always equal to Q(WIDTH-1).
Q	Optional	None	Registered Counter output	Size equals WIDTH generic.
Eq	Optional	None	Asynchronous decode of the Q output.	Size equals 2**WIDTH generic.

Notes:

1. Most targets do not support the Testenab, Testin, Testout, Aconst, Aload, and Sconst ports.
2. The Eq port, if used, will be 2**WIDTH bits wide. The compile time will grow exponentially with the width of this port; therefore we recommend that if you use this port that you do not set WIDTH > 6. If necessary, you can cascade individual G_COUNTERs together to create wider counter functions.

G_COUNTER also has the following generics:

Generic	Usage	Description	Comments
Width	Required	The width, in bits, of the Dataa, Datab, and Sum ports.	Must be an integer value >= 2.
Representation	Required	Indicates whether UNSIGNED or SIGNED math is to be performed.	Only the value UNSIGNED is currently supported.
Modulus	Optional	The terminal count of the counter.	If not set, terminal count will default to 2**WIDTH - 1.
Avalue	Optional	Value loaded by the Aconst input.	See Note 2., above.
Svalue	Optional	Value loaded by the Sconst input.	See Note 2., above.
Pvalue	Optional	Power-up value of the Q output.	For simulation, defaults to 0.

Notes:

1. Most targets do not support Modulus, Avalue, and Svalue.
2. At the current time no targets support Pvalue. With the default value for this generic, the Q output of the functional simulation model will initialize to 0. This may or may not match the behavior of the target implementation.

An example of an instantiated G_COUNTER is:

```
i_count0: g_counter
    generic map (width=>16, REPRESENTATION => "UNSIGNED")
   port map (data=>din0, clock=>clock, aclr=>areset,  sload=>sload,
q=>count_out);
```

G_MULT

The following table describes the ports of the G_MULT:

Port	Usage	Default Value	Description	Comments
Dataa	Required	None	First data input	Size equals WIDTHA generic.
Datab	Required	None	Second data input	Size equals WIDTHB generic.
Sum	Optional	Logic 0	Partial Sum.	Size equals WIDTHS generic.
Product	Optional	None	Product output.	Product = Dataa * Datab + Sum. Size equals WIDTHP generic.

Notes:

1. Most targets do not support the Sum port.

G_MULT also has the following generics:

Generic	Usage	Description	Comments
Widtha	Required	The width, in bits, of the Dataa port.	Must be an integer value >= 2.
Widthb	Required	The width, in bits, of the Datab port.	Must be an integer value >= 2.
Widths	Required	The width, in bits, of the Sum port.	Must be an integer value >= 2.
Widthp	Required	The width, in bits, of the Product port.	Must be an integer value >= 2.
Representation	Required	Indicates whether UNSIGNED or SIGNED math is to be performed.	Only the value UNSIGNED is currently supported.

An example of an instantiated G_MULT is:

```
i_mult0: g_mult
    generic map(widtha=>4, widthb=>4, widthp=>8,
   representation => "unsigned")
    port map(dataa=>din0, datab=>din1, product=>mult_out);
```

G_RAM_DQ

At the current time G_RAM_DQ is never inferred. It may be instantiated, but is only supported for Altera targets, where it maps directly to the Altera LPM_RAM_DQ macro. See the Altera MaxPlus documentation for details.

G_ROM

At the current time G_ROM is never inferred. It may be instantiated, but is only supported for Altera targets, where it maps directly to the Altera LPM_ROM macro. See the Altera MaxPlus documentation for details.

Functional Simulation

Functional simulation models for the Synario generic datapath macrofunctions are supplied in the file <SYNARIO>\generic\vhdl\gen_dpe.vhd. This file will automatically be compile into the appropriate library in the Vsystem simulator if you following the instructions outlined in the SYN-VHDL ReadMe.

Note that at the current time, the following limitations apply with respect to these models:

1. Models for the G_RAM_DQ and G_ROM do not support loading their initial contents from a data file. To load these memories with initial values during simulation, use the FORCE command in Vsystem.
2. All models only support a value of UNSIGNED for the REPRESENTATION generic.

6. How to Manage VHDL Design Hierarchies

Managing Large Designs

This section shows you how to use partitioning and design management to manage larger designs. The VHDL constructs for partitioning and sharing code modules are the **component**, **configuration**, **block** and **library** statements. You should refer to a standard VHDL reference text for detailed descriptions of these partitioning statements. In this section, methods for design partitioning that are most relevant to synthesis will be discussed. Additional methods of design partitioning, using packages, will be discussed in the section that follows.

Hierarchy

Hierarchy is a way of managing a design by creating references to external, lower-level design modules (entities) from within a higher-level design module. The concept of hierarchy in VHDL is similar to the concept of hierarchy as implemented in many schematic entry systems. The basic unit of hierarchy in VHDL is the **component**. A component is a VHDL entity that is referenced as a lower-level module from another, higher-level entity.

A VHDL **entity** can have multiple **architectures**, and a particular entity/architecture pair (referred to as a **design entity**) can be referenced from another architecture as a VHDL **component**. Instantiating components within another design provides a mechanism for partitioned designs, or for using existing designs as reusable components of larger designs.

You can manage the relationship between a component declaration and various design entities by using **configuration** specifications. Configuration statements are supported in the VHDL synthesizer, and are described later in this chapter. There are default configurations, so it usually not necessary to include configuration specifications in your designs.

Components

VHDL design entities can be referenced from other architectures as components. VHDL allows you to manage the mapping of design entities to components with a configuration specification (described in the next section) that associates particular component instances with a specified design entity. In most cases, however, you will have only one design entity for each component instance, and you will simply let the VHDL synthesizer (or simulator) select the default entity (default binding) for each component.

The following example contains three entity/architecture pairs: two lower level entities (**add** and **mult**) and a top level entity (**addmult**) that includes two component instantiations of the lower-level entities:

```
-----------------------------
-- lower-level entity: adder
--
entity add is
    port(op1,op2: in integer range 0 to 7;
         result: out integer range 0 to 63);
end add;
architecture dataflow of add is
begin
    result <= op1 + op2;   -- add the operands
end dataflow;
---------------------------------
-- lower-level entity: multiplier
--
entity mult is
    port(op1,op2: in integer range 0 to 7;
         result: out integer range 0 to 63);
end mult;
architecture dataflow of mult is
begin
    result <= op1 * op2;   -- multiply the operands
end dataflow;
-----------------------------------------------------
-- top-level entity: mux of add and multiply results
--
entity addmult is
    port(op1,op2: in integer range 0 to 7; sel: in boolean;
          result: out integer range 0 to 63);
end addmult;
architecture structure of addmult is
    signal s_add, s_mult: integer range 0 to 63;
    component add   -- component declaration
        port(op1,op2: in integer range 0 to 7;
             result: out integer range 0 to 63);
    end component;
    component mult  -- component declaration
        port(op1,op2: in integer range 0 to 7;
             result: out integer range 0 to 63);
    end component;
begin
    add1: add port map (op1,op2,s_add);
    mult1: mult port map (op1,op2,s_mult);
    with sel select    -- Mux the add and multiply results
        result <= s_add when FALSE,
                  s_mult when TRUE;
end structure;
```

In this example, the architecture of **addmult** contains a declaration of two components, **add** and **mult**, and one instance of each component. The three design entities are arranged in the hierarchy shown in **Figure 6-1**.

Figure 6-1: Hierarchy of the Addmult Multiplexer

Hierarchical designs contain multiple design entities, and may be written using more than one source file, or by entering multiple design entities (entity/architecture pairs) in the same source file.

Components And Synthesis

Using components to partition a large design can have advantages for synthesis. Many of the optimizations performed by the VHDL synthesizer operator at the component level, meaning that a design written using components can process more efficiently. Large designs can create very long run times during synthesis, so breaking up the design into smaller pieces using components is recommended. In addition to performing logic optimizations at the level of components, the VHDL synthesizer will detect multiple references to (instances of) a component, and will not attempt to perform the same optimizations twice for the same circuitry. The synthesizer will instead make a copy of the already-optimized circuitry to create the additional instances.

Using Multiple Hierarchical VHDL Files

If you use multiple VHDL source files to describe the hierarchy of your design, and have entered only a single entity/architecture pair in each VHDL file, then the Project Navigator will determine the hierarchy of your design for you, and will display the VHDL sources as a hierarchy (by indenting them in the Sources Window). When the design is synthesized, the Project Navigator runs the VHDL synthesizer once for each source file. During synthesis, hierarchy references are be generated for each VHDL file. After synthesis, the synthesized logic for each module is combined (by linking or merging) during the device fitting process.

Note: *The point at which a hierarchical design is linked or merged depends on the type of device selected. Designs being implemented in PLD-type devices are linked prior to the device fitting process, while designs being implemented in FPGA devices are merged later, during the fitting process. For FPGA devices, partitioning a large design into multiple source files can greatly reduce the processing time required for the design.*

When you simulate a design consisting of multiple VHDL files, you must compile the design from the bottom up, beginning with the VHDL files containing the lowest-level design entities (those that contain no references to other VHDL design units). The VHDL simulator does not allow VHDL source files to be compiled if they contain references to lower-level design entities that have not yet been compiled.

Note: *The Project Navigator expects the entity and architecture declarations for each design entity to be located in the same source file.*

Using A Single Hierarchical VHDL File

If you choose to enter your design as a single VHDL source file, however, the VHDL synthesizer will attempt to flatten the entire design into a non-hierarchical form. When doing this, the VHDL synthesizer will assume that the last entity (or configuration declaration) encountered in the VHDL source file is the top-level design entity. If the top-level design entity is not the last entity or architecture in the file, then you must specify the name of the actual top-level design entity by setting the Top-Level Entity property in the Project Manager.

Configurations

Configurations are one of five primary design units in VHDL (the others being entities, architectures, package declarations and package bodies). During synthesis or simulation, you may choose to have a configuration statement represent the highest level design unit in your design instead of the top-level entity. When you process a design with a configuration statements as the highest level design unit, the configuration statement provides the necessary information to define the design, in terms of the relationships between all the other design units and their references to lower-level design units (entities, architectures, and component references to lower-level entities and architectures).

For structural VHDL (designs with many components arranged in a hierarchy) it is useful to think of the configuration statement as a parts list. The configuration statement may contain statements associating each component instance in the design with a specific architecture (perhaps allowing a different architecture to be used for simulation than is used for synthesis) or may specify *generics* (compile-time values) that configure components prior to their use. (One common use of generics in this way is to pass delay values into an architecture prior to simulation.)

Note: *Generics are fully supported in the VHDL synthesizer, but are not described in this manual. Refer to a standard VHDL text for information about this language feature.*

Configurations can also be used to re-map the ports of a component and its lower-level entity, making it possible to substitute lower-level entities that do not directly correspond to the component instance being described at the higher-level. In this context, you can think of configurations as being analogous to a configurable socket connecting an IC package (the component) to a possibly mismatched set of through-holes on a printed circuit board (which in turn corresponds to a component reference in the higher-level entity).

The VHDL synthesizer supports the use of configurations, and allows a configuration to be specified as the top-level design unit for synthesis purposes. For more complete information about configurations, refer to a standard VHDL text.

Blocks

Designs can be partitioned using either **block** statements or **component** statements. While **component** statements are used to create a hierarchy of connected components, **block** statements are used to provide partitioning within a single architecture. Using **block** statements to partition an architecture is analogous to using multiple drawing sheets to enter a schematic-based design.

Block statements can be used to partition concurrent statements, as in the following example:

```
architecture partitioned of some_design is
begin
 a_block: block
 begin
 -- concurrent statements here
 end block;
 another: block
 begin
 -- concurrent statements here
 end block;
end partitioned;
```

Blocks such as this have no special meaning for synthesis, and are not frequently used in designs intended for synthesis.

Using Packages

Package declarations can be used to declare common types and subprograms. For example:

```
package example_package is
    type shared_enum is (first, second, third, last);
end example_package;
```

In order for the contents of a package to be visible from inside an entity or an architecture, you need to place a **use** clause before the entity declaration, as in the following example:

```
use work.example_package.all;
entity design_io is
```

```
    ...
end design_io;
```

Placing a **use** clause before an entity causes the contents of the specified package contents to be visible to that entity and its architecture(s), but nowhere else. This means that you must include a **use** statement before each entity/architecture pair in the design that requires access to package contents.

Package Visibility Rules

Standard VHDL package visibility rules ignore file boundaries, meaning that a package could be in one file, the **use** clause and entity declaration in another, and the architecture in a third file. The VHDL synthesizer and Project Navigator, however, require that each entity appear in the same source file as its corresponding architecture, and that **library** and **use** statements appear in the same file as the entity and architecture in which they are used.

For example, if the following **library** statement is placed in the VHDL source file:

```
library my_lib;
```

then the VHDL synthesizer will analyze the file named **my_lib.vhd** in the current directory, then the VHDL synthesizer will search for and analyze the contents of a file called my_lib.vhd, looking first in the current working directory and then in the \synario\lib5 directory. (Note that library names cannot exceed 8 characters, to comply with DOS filename restrictions.)

To make the library units within the library visible outside that library, it is necessary to add the following statement to the VHDL source file:

```
use my_lib.my_package.all;
```

This **use** clause must be place just prior to (and in the same source file as) any entity statement that requires access to the contents of **my_package**. (You do not have to repeat the **use** statement prior to the architectures associated with the entities; architecture design units inherit the visibility rules of their parent entities.)

Design Libraries

Each package declaration that you write, and all other design units, including entities and architectures, are compiled (during simulation or synthesis) into a special area called a *design library*. In simulation environments, if you do not specify a named library during compilation, then the default library, **work**, is where the package will be compiled. For packages that you place in the same source file in which they are referenced, you will use the name **work** for all **use** clause references to those packages.

Note: *There are differences between the way in which the **work** library is implemented in the VHDL synthesizer and many VHDL simulation products. Refer to the information in the next section, "Using Design Libraries," for a complete explanation.*

Using Packages For Common Declarations

To define common declarations (such as types, subtypes or subprograms), you may want to use a package that is shared between different VHDL sources files. Although this is quite easy to do in simulation systems (you simply all of the source files, including the file containing the package, into **work** and place appropriate **use** statements before each design unit requiring the package), there are some restrictions on how you can do this using the VHDL synthesizer. If you need to reference a package from within two or more different source files (for example, from your actual design description and from a test bench, or from two or more source files referenced in the hierarchy of the design), you must place the package declaration in a separate VHDL source file and reference it as a named library. You do this by referencing the external source file containing the package in each of the source files using **library** statements. During synthesis, the VHDL synthesizer will include the external source file containing the package declaration into the files that contain the **library** statements. During simulation, you must compile the external source file containing the package into an appropriately named library before compiling the higher-level source files. For example, if you have the following package:

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

package typedef is
    subtype byte is std_logic_vector (7 downto 0);
end;
```

and you wish to reference the type **byte** in more than one source file in your design, then you must place the package in a unique source file, and place **library** statements prior to the entity declarations in all source files that require the **byte** data type:

```
library mytypes;
use mytypes.typedef.all;
```

In this **use** statement, **mytypes** corresponds to the name of the external VHDL source file containing the package (in this case, mytypes.vhd). During synthesis, the VHDL synthesizer will include the mytypes.vhd file each time it encounters the **library** statement. When you simulate the design, you will compile the mytypes.vhd file into a named library (**mytypes**) prior to compiling the other source files into the **work** library.

For more information on these and other uses of packages and libraries, refer to a standard VHDL text.

Using Design Libraries

In most VHDL simulation environments, *design libraries* are areas in which pre-compiled (analyzed) design units are stored. An good example of such a library is the IEEE library, which is pre-compiled into a library accessible to the VHDL simulator. The form that the pre-compiled library takes is dependent upon the simulator being used, and compiled libraries are generally not compatible between different simulator programs.

The VHDL synthesizer, on the other hand, implements libraries as source files that are read in and synthesized as they are encountered in a higher-level VHDL source file (as **library** statements). This means that a library such as the IEEE library must exist as a VHDL source file, and must be accessible to the VHDL synthesizer during the compilation process. The VHDL source code for the IEEE library, for example, is provided in the file **ieee.vhd**.

Note: *The VHDL implicit library* ***std*** *does not have to be referenced in a* ***library*** *or* ***use*** *statement. The library* ***std*** *is contained in the file* ***std.vhd****.*

Library Search Paths

When the VHDL synthesizer encounters a library statement, it attempts to find a corresponding VHDL source file (one with a name matching the referenced library name) in the current directory (the project directory). If a VHDL source file with the appropriate name is not found in the current directory on your system, the VHDL synthesizer will attempt to find the referenced VHDL source file in the library area (normally **./lib5/**) within the Project Navigator installation directory.

If you create library files that will be shared between multiple projects, you may want to place those files in the library area, rather than make local copies in each project directory.

The Work Library

The differences in library implementations between most simulation environments and the VHDL synthesizer does not normally make any difference; library files are read in by the synthesizer as needed, and the **library** and **use** statements function normally. The only exception is in the treatment of the default library, **work**.

Work is the default name of the current library. The **work** library is where, in a simulation system, all of your design units (entities, architectures, packages, and configurations) will be placed after they are analyzed, unless you have specified an alternate (named) library. Unlike simulation environments, the VHDL synthesizer only considers design units that are currently being compiled (those in the current source file) to be in the **work** library. This means that the VHDL synthesizer will not be able to access design units (such as packages) that are located in another source file using the **work** library.

The Dataio and Generics Libraries

In addition to the **ieee** and **std** libraries, two custom libraries, **dataio** and **generics**, are provided with the VHDL synthesizer. These libraries (which are provided in source file form in the **lib5** installation subdirectory, and in simulation compiled form in the **generic/vhdl** subdirectory) are used for type conversions and for simulation of generic symbols, respectively. Examples of using these libraries can be found in the tutorials chapter of the *VHDL Entry* manual.

Using Schematics With VHDL

The Project Navigator allows schematic and VHDL sources to be intermixed in an arbitrary hierarchy. When such a design is entered, there are certain rules that you must follow to ensure that the design is able to be simulated (using the VHDL simulator) and properly synthesized.

Note: *Although VHDL is case-insensitive, the VHDL synthesizer preserves the case (upper- and lower-case characters) used in all VHDL signal names. Because some device-specific fitting software is sensitive to case (and will not, for example, recognize that two signals named Reset and RESET are the same), you should be careful to use consistent signal names in different parts of your design. This is particularly true when combining schematics with VHDL; always check to make sure that the signal names used on the schematic match properly with the names used in VHDL portions of the design.*

Using A Top-Level Schematic With VHDL

You can create designs that are a mixture of top-level schematic files and lower-level VHDL files, but there are a few rules that you must observe:

- Use **std_logic** or **std_logic_vector** data types as interfaces between blocks on the schematic and lower-level VHDL design entities. When installed, the schematic VHDL netlist writer is set up (via .INI files settings) for this data type, and will make reference to lower-level design units using **std_logic** and **std_logic_vector** as appropriate. If necessary, use the type conversion functions provided in the dataio library (**/lib5/dataio.vhd**) to convert non-**std_logic** data types to **std_logic** within the VHDL design units.

- The names that you use in your schematic for net or instance names must be valid VHDL identifiers. Refer to the rules for VHDL identifier names, and check the list of VHDL keywords provided in Appendix A if you are unsure if a particular name is a valid VHDL identifier. Avoid using underscore (_) characters at the beginning or end of names, as this is not allowed in VHDL names.
- To allow simulation to work properly on post-route models, you should avoid using arrays (busses) on the top-level schematic for your design. Instead, you should provide single-bit I/O signals, and refer to these signals in your test bench.

The Using Schematics with VHDL tutorial in the *VHDL Entry* manual provides an example of using a top-level schematic with lower-level VHDL source files.

Using Lower-Level Schematics With VHDL

In most cases, you can mix higher-level VHDL sources with lower-level schematics representing portions of your design. (Check your device kit documentation for restrictions.) When you are referencing a lower-level schematic module from VHDL, you do not need to specify any special attributes or flag the external component in any special way within the VHDL source file. When the VHDL synthesizer encounters a component instantiation within a VHDL file that has no corresponding lower-level VHDL entity declaration, it will simply create a reference to the missing module in the generated Open-ABEL 2 format intermediate file. The Project Navigator will then attempt to resolve the hierarchy by looking for an appropriately named source file (either schematic or VHDL) in the current project. If the Project Navigator is unable to resolve the hierarchy reference, it will display a warning icon with the name of the missing module.

Note: *If you are referencing a lower-level component that is not represented by a lower-level schematic or VHDL file, and is instead a hard or soft macro defined by the device kit chosen, then you must use the* ***macrocell*** *attribute (described in the next section of this chapter) to specify that the synthesizer should make an external reference to the module.*

Using Generic Symbols With VHDL

When you use the VHDL simulator to simulate a design schematic that includes generic symbols (such as the MUX symbol used in the **prep2** tutorial design), you must provide the simulator with information about the function of the generic symbols. To do this, you must reference the **generics** library provided with the Project Navigator, using the Library Mapping menu item in the simulator. The generics library (which can be found in compiled form in the **generic\vhdl\generics** installation subdirectory) contains functional models for all of the generic symbols provided with the schematic editor.

The **generics** library is also provided in source file form in the **generic\vhdl** subdirectory, but this file (**generics.vhd**) is not used during processing for simulation or for synthesis. It is only provided for your information.

An example of how to map the **generics** library during simulation can be found in the tutorials chapter in the *VHDL Entry* manual, in the final tutorial example (**prep2**).

How Schematics Are Processed For VHDL Simulation

When creating a VHDL functional simulation model from a schematic, the Project Navigator reads the schematic file and generates a VHDL source file corresponding to a netlist representing all wires, components and block symbols on the schematic. To create a VHDL source file, the Project Navigator s netlist generator must assign valid VHDL data types to all wires and busses used on the schematic, and assign matching data types to the ports of all components and blocks used on the schematic.

By default, the data types used are **std_logic** and **std_logic_vector**. These data types are specified in a configuration file (**vhdl.ini**), and can be modified if needed. It is strongly recommended, however, that you standardize on the **std_logic** and **std_logic_vector** data types for all schematic/VHDL interfaces.

Note: *The Project Navigator allows schematics containing references to generic symbols (such as G_DEC or G_MUX21) to be functionally simulated using the VHDL simulator. The* ***generics*** *library included with the VHDL option contains VHDL descriptions (models) for each generic symbol. If you are using device-specific symbols (such as the Xilinx TBUF internal tri-state or OSC oscillator), refer to your device kit documentation for information about which symbols are supported in VHDL simulation, and for information on how to access these model libraries during simulation.*

A. VHDL Quick Reference

This appendix contains basic reference information for VHDL syntax. For complete information, refer to the *IEEE Standard 1076-1993 VHDL Language Reference Manual* or to a standard VHDL text.

Reserved Words

The following words are reserved in VHDL (regardless of case) and cannot serve as user-defined identifiers:

abs
access
after
alias
all
and
architecture
array
assert
attribute

begin
block
body
buffer
bus

case
component
configuration
constant

disconnect
downto

else
elsif
end
entity
exit

file
for
function

generate
generic
guarded

if
impure
in
inertial
inout
is

label
library
linkage
literal
loop

map
mod

nand
new
next
nor
not
null

of
on
open
or
others
out

package
port
postponed
procedure
process
pure

range
record
register
reject
rem
report
return
rol
ror

select
severity
shared
signal
sla
sll
sra
srl
subtype

then
to
transport
type

unaffected
units
until
use

variable

wait
when
while
with

xnor
xor

VHDL Syntax Basics

The following code fragments illustrate the syntax of VHDL statements:

Declarations

```
-- OBJECTS

constant alpha : character := 'a';
variable total : integer ;
variable sum : integer := 0;
signal data_bus : bit_vector (0 to 7);

-- TYPES

type opcodes is (load,store,execute,crash);
type small_int is range 0 to 100;
type big_bus is array ( 0 to 31 ) of bit;
type glob is record
     first : integer;
     second : big_bus;
     other_one : character;
end record;

-- SUBTYPES

subtype  shorter is integer range 0 to 7;
subtype  smaller_int is small_int range 0 to 7;
```

Names

```
-- Array element
big_bus(0)

-- Record element
record_name.element
```

Sequential Statements

```
--IF STATEMENT
if increment and not decrement then
    count := count +1;
elsif not increment and decrement then
    count := count -1;
elsif increment and decrement then
    count := 0;
else
    count := count;
end if;

--CASE STATEMENT
case day is
when Saturday to Sunday =>
    work := false;
    work_out := false;
when Monday | Wednesday | Friday =>
    work := true;
    work_out := true;
when others =>
    work := true;
    work_out := false;
end case;

-- LOOP,NEXT,EXIT STATEMENTS
L1 : for i in 0 to 9 loop
    L2 : for j in opcodes loop
        for k in 4 downto 2 loop    -- loop label is optional
            if k = i next L2;       -- go to next L2 loop
        end loop;
        exit L1 when j = crash;     -- exit loop L1
    end loop;
end loop;

-- WAIT STATEMENT
wait until clk;

-- VARIABLE ASSIGNMENT STATEMENT
var1 := a or b or c;
-- SIGNAL ASSIGNMENT STATEMENT
sig1 <= a or b or c;
```

Subprograms

```
-- FUNCTION DECLARATION
-- parameters are mode in
-- return statements must return a value
function is_zero (n : integer) return Boolean is
    -- type, variable,constant,subprogram declarations
begin
    -- sequential statements
    if n = 0 then
        return true;
    else
        return false;
    end if;
end;

-- PROCEDURE DECLARATION
-- parameters may have mode in , out or inout
procedure count (incr : Boolean; big : out bit;
                                        num : inout integer) is
    -- type, variable,constant,subprogram declarations
begin
    -- sequential statements
    if incr then
        num := num +1;
    end if;
    if num > 101 then
        big := '1';
    else
        big := '0';
    end if;
end;
```

Concurrent Statements

```
-- BLOCK STATEMENT
label5 :    -- label is required
block
    -- type, signal,constant,subprogram declarations
begin
    -- concurrent statements
end block;

-- PROCESS STATEMENT , sequential first form
label3 :    -- label is optional
process
    -- type, variable,constant,subprogram declarations
begin
    wait until clock1;
    -- sequential statements
end process;

-- PROCESS STATEMENT , sequential second form
process ( en1, en2, clk) -- ALL signals used in
                         -- process
    -- type, variable,constant,subprogram declarations
begin
    if clk then
        -- sequential statements
        local <= en1 and en2;
        -- sequential statements
    end if;
end process;

-- PROCESS STATEMENT , combinational
process ( en1, en2, reset ) -- ALL signals used in
                            -- process
    -- type, variable,constant,subprogram declarations
begin
        -- sequential statements
        local <= en1 and en2 and not reset;
        -- sequential statements
end process;
```

```
-- GENERATE STATEMENT
label4 : -- label required
for i in 0 to 9 generate
    -- concurrent statements
    if i /= 0 generate
       -- concurrent statements
            sig(i) <= sig(i-1);
    end generate;
end generate;

-- COMPONENT INSTANTIATION
-- label is required
-- positional association
U1 : decode port map (instr, rd, wr);
-- named association
U2 : decode port map (r=> rd, op => instr, w=> wr);

-- CONDITIONAL SIGNAL ASSIGNMENT
total <= x + y;
sum <= total + 1 when increment else total -1;

-- SELECTED SIGNAL ASSIGNMENT;
with reg_select select
    enable <= "0001" when "00",
              "0010" when "01",
              "0100" when "10",
              "1000" when "11";
```

Library Units

```
-- PACKAGE DECLARATION
package globals is
    -- type,constant, signal ,subprogram declarations
end globals;

-- PACKAGE BODY DECLARATION
package body globals is
    -- subprogram definitions
end globals;

-- ENTITY DECLARATION
entity decoder is
    port (op : opcodes; r,w : out bit);
end decoder;

-- ARCHITECTURE DECLARATION
architecture first_cut of decoder is
    -- type, signal,constant,subprogram declarations
begin
    -- concurrent statements
end first_cut;

-- CONFIGURATION DECLARATION
configuration example of decoder is
    -- configuration
end example;

-- LIBRARY CLAUSE
-- makes library , but not its contents visible
library utils;

-- USE CLAUSE
use utils.all;
use utils.utils_pkg.all;
```

Attributes

- ATTRIBUTES DEFINED FOR TYPES

```
T'base      the base type of T
T'left      left bound of T
T'right     right bound of T
T'high      high bound of T
T'low       low bound of T
T'pos(N)    position number of N in T
T'val(N)    value in T of position N
T'succ(N)   T'val(T'pos(N) +1)
T'pred(N)   T'val(T'pos(n) -1)
T'leftof(N) T'pred(N) if T is ascending
    T'succ(N) if T is descending
T'rightof(N)        T'succ(N) if T is ascending
    T'pred(N) if T id descending
T'image(N)          string representing value of N
T'value(N) value of string N

-- ATTRIBUTES DEFINED FOR ARRAYS
A'left(N)   left bound of Nth index of A
A'right(N)  right bound of Nth index of A
A'high(N)   high bound of Nth index of A
A'low(N)    low bound of Nth index of A
A'range(N)          range of Nth index of A
A'reverse_range(N)        reverse range of Nth index of A
A'length(N)         number of values in Nth index of A
A'ascending         true if array range ascending

-- ATTRIBUTES DEFINED FOR SIGNALS
S'event     true if an event has just occurred on S
S'stable    true if an event has not just occurred on S
S'last_value        the previous value of S, before last change

-- STRING ATTRIBUTES
E'simple_name     string "E"
E'path_name       hierarchy path string
E'instance_name   hierarchy and binding string
```

B. Limitations

VHDL is a technology-independent language and has a very large feature set. Because the VHDL software is specifically targeted toward logic design, some VHDL constructs are not applicable to synthesis.

Constraints and Unsupported Constructs

This section lists the VHDL constructs that are not supported by the VHDL synthesizer, or whose use is constrained.

Unsupported Constructs

The following constructs are not supported; using them results in a constraint error.

- Access types
- File types
- Signal attributes (except **'event** and **'stable**)
- Textio package
- Impure Functions
- Shared Variables

Constrained Constructs

The following constructs are constrained in their usage. Constrained constructs fall into two classes:

- Statements constrained in where they may be used.
- Constrained expressions. The use of a constrained construct will result in a constraint message.

Constrained Statements

- A **wait** statement may be first statement in a process only.
- Signal attributes **'event** and **'stable** are valid only when they specify a clock edge.
- Subprogram calls cannot be recursive.
- The formal part of a named association may not be a function call.

- A process sensitivity list must contain all signals that the process is sensitive to.

Constrained Expressions

Certain expressions metalogic expressions which simply means they evaluate to a constant value, and are not dependent on a signal (they do not change over time).

- Operands of ** must be metalogic expressions.
- Assignments to elements of an array must have an index that is a metalogic expression.
- An assertion statement condition, severity, and message must consist of metalogic expressions, if the message is to be reported.
- Type and subtype declarations must be metalogic expressions.
- Floating point and physical types are constrained to the same set of values as the equivalent integer type.
- While loop and unconstrained loop execution completion must depend only on metalogic expressions.

Ignored Constructs

The following constructs are ignored. They may be included in the VHDL file for simulation purposes, but the VHDL compiler will not generate any logic for them.

- Disconnect specifications
- Resolution functions
- Signal kind **register**
- Waveforms, except the first element value

C. VHDL for the ABEL-HDL Designer

This Appendix compares ABEL-HDL and VHDL design strategies and is intended for the experienced ABEL-HDL designer who has little or no experience with VHDL. Included in this chapter are the following topics:

- Design Input/Output
- Pin Numbers
- Combinational Logic
- Sequential Logic
- Registers
- State Machines

Design I/O

 Describing Design I/O in ABEL-HDL

In ABEL-HDL, you use the **pin** keyword to declare input and output signals that correspond to device I/O pins:

```
Clock, !Reset, S1 pin istype 'REG';
```

This statement specifies that the three signals (**Clock**, **Reset** and **S1**) are all registered. Information about whether individual signals are inputs or outputs not included in an ABEL pin declaration statement, but is instead implied in the way that the signals are used in subsequent design descriptions (such as equations).

 Describing Design I/O in VHDL

In VHDL, you describe design I/O using port statements within an entity declaration. A port statement is similar to a pinout description for a circuit element: each pin has a type of data (value) associated with it, and a flow direction (mode) associated with that data. Correspondingly, each entry in a VHDL **port** statement has a mode and value associated with it. The **port** statement has the following syntax:

```
port (pin_list: [mode] type [; pin_list: [mode] type ...]);
```

The mode of a port in VHDL describes its dataflow direction. A port's mode can be **in**, **out**, **buffer**, or **inout**. The default mode is **in**. (It is good VHDL coding practice to include **in** for all input ports.) There is no equivalent in ABEL-HDL to **mode**. Pin declarations in ABEL-HDL do not indicate whether signals are inputs, outputs, or bidirectional. In ABEL-HDL, direction information is determined by how the declared signals are actually used in the design.

The following program segment illustrates simple design I/O in VHDL. This example describes a circuit element with inputs **a**, **b**, and **sel**, and output **c**. **a**, **b** and **c** are 6-bit data types that can transmit data corresponding to the integer values 0 through 63.

```
entity ent1 is
port (a,b: std_logic_vector (0 to 5); sel: std_logic;
    c: out std_logic_vector(0 to 5);
end ent1;
```

Note: *Unlike ABEL-HDL, VHDL is not case-sensitive. In VHDL, ABC=AbC=abc. You should be aware, however, that some device - specific ("back-end") programs are case sensitive, so you should choose signal (and other) names that do not rely on case sensitivity or insensitivity.*

Pin and Node Numbers

Describing Pin Numbers in ABEL-HDL

In ABEL-HDL pin declarations, actual pin numbers can be specified in the pin declarations, as in

```
clk, clr, Dir, OE   pin 1,2,3,11;
```

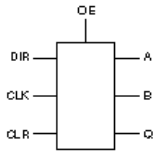
Describing Pin and Node Numbers in VHDL

VHDL does not have a language equivalent that allows pin number declarations, so special signal attributes are used to pass pin and node number information through the VHDL Synthesizer.

Note: You specify pin and node numbers in the VHDL source file using the custom attribute pinnum*. This special attribute is recognized by the VHDL synthesizer and is written to the output for use by device fitting software. For more information on the pinum* **attribute***, refer to the online help for VHDL.*

The following example (the **entity** portion of a VHDL version of the standard ABEL-HDL example **cntbuf.abl**) shows how to use port and attribute statements in VHDL to make pin assignments. **Figure C-1** is the block diagram.

Figure C-1: Block Diagram for cntbuf Design



In this example, the port statement defines the data types. (This example uses the IEEE 1164 std_logic data types. These data types are described in detail in Chapter **Error! Reference source not found.**, **Error! Reference source not found.**.) The VHDL **pinnum** attribute statement is then used to assign actual pin numbers to the design's I/O ports:

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

entity cntbuf is
    port(Dir: in std_logic;
        Clk,Clr,OE: in std_logic;
        A,B: inout std_logic_vector (0 to 1);
        Q: inout std_logic_vector (3 downto 0));

    attribute pinnum : string;  -- Define the attribute
    attribute pinnum of Clk: signal is  "1";
    attribute pinnum of Clr: signal is  "2";
    attribute pinnum of Dir: signal is  "3";
    attribute pinnum of OE: signal is  "11";
    attribute pinnum of A: signal is  "13,12";
                     -- Assigns A_0_ to 13, A_1_ to 12
    attribute pinnum of B: signal is  "19,18";
                     -- Assigns B_0_ to 19, B_1_ to 18
    attribute pinnum of Q: signal is  "17,16,15,14";

end cntbuf;
```

In this design, the bit vectors **A**, **B**, and **Q** must be given a list of pin numbers according to the width of their data types. The order of the list of pin numbers is significant. If the VHDL bit vectors are ordered from least- to most-significant bit (LSB to MSB) using the **to** range specifier, then the mapping signals to pins is also LSB to MSB (A(0) would be mapped to pin 13 and A(1) would be mapped to pin 12).

Combinational Logic

Describing Combinational Logic in ABEL-HDL

In ABEL-HDL, you use the combinational assignment operator ('=') to specify combinational logic in the Equations section of your program. The following ABEL-HDL design uses equations to describe the function of a simple 2-bit adder circuit:

```
module add
    a0,a1,b0,b1   pin;            "operands A and B
```

```
    s1,s0         pin istype 'com'; "sum
    c0,c1         pin istype 'com'; "carry bits
equations
    c0 = b0 & a0;         " carry from bit 0
    s0 = b0 $ a0;         " sum of bit 0
    c1 = b1 $ a1;         " carry out
    s1 = b1 $ a1 $ c0;    " sum for bit 1
end
```

Describing Combinational Logic in VHDL

In VHDL, you can describe combinational logic using concurrent statements in the architecture section of your program. The following is a VHDL architecture describing the same adder:

```
architecture adder of add is
begin
    c0 <= b0 and a0;         -- carry from bit 0
    s0 <= b0 xor a0;         -- sum for bit 0
    c1 <= b1 xor a1;         -- carry out
    s1 <= b1 xor a1 xor c0;  -- sum for bit 1
end adder;
```

Registers

Describing Registers in ABEL-HDL

In ABEL-HDL, you can describe registered circuit elements by specifying the various flip-flop inputs, such as clocks, resets and data through the use of dot extensions like .CLK, .AR, and .D. ABEL-HDL also allows you to write registered output functions using pin-to-pin syntax through the use of a registered assignment (:=). Using pin-to-pin syntax, a D flip-flop is described in ABEL-HDL as:

```
foo.CLK = clock;
foo := Data # Preset
```

Describing Registers in VHDL

In ABEL-HDL, you specify registers when you supply inputs to register macrocells that are inherently predefined in the language. In VHDL, however, there is no inherent register behavior or macrocell, unless one has been provided (written in VHDL as a procedure or component, or implied by the defined behavior of a VHDL process). There is also no direct equivalent in VHDL to ABEL-HDL's register assignment statement. In VHDL, your description of registered operation will differ depending on whether you are using structural, dataflow or behavioral design methods. In structural or dataflow VHDL, your program must define how the flip-flop operates. In behavioral VHDL, the actual flip-flops can be implied, rather than specified.

To describe a registered function in structural or dataflow VHDL, you can add a procedure to define the memory elements. In the following dataflow design example, a **procedure** is added to the previous design description to implement the 2-bit adder with registered outputs:

```
architecture adder_ff of add is
    signal f,g: bit;

    procedure dff(signal clk,d: bit;
                  signal q: out bit) is
    begin
        if clk and clk'event then
            q <= d;
        end if;
    end;
    procedure add(signal a0,a1,b0,b1: bit;
                  signal c0,c1 out bit) is
    variable x,y : bit
    begin
        c0 <= b0 and a0;        -- carry from bit 0
        s0 <= b0 xor a0;        -- sum for bit 0
        c1 <= b1 xor a1;        -- carry out
        s1 <= b1 xor a1 xor c0; -- sum for bit 1
    end;

begin
    add(a0,a1,b0,b1,f,g);
    dff(clk,f,c0);
    dff(clk,g,c1);
end adder_ff;
```

Since descriptions of registered logic using procedures can become rather unwieldy, it is often easier to use behavioral descriptions for designs. This is done by placing the combinational logic within a process as shown below:

```
architecture behavior of my_and is
begin
    process(clk)
    begin
        if (clk and clk'event) then
            y <= a and b;
        end if;
    end process;
    q <= y;
end adder_ff;
```

See Behavioral VHDL in Chapter **Error! Reference source not found.**, **Error! Reference source not found.** for more information.

An alternative method of describing a concurrent registered assignment (using features of the VHDL 1076-1993 standard) is to use a selected signal assignment such as the following:

```
architecture dataflow of my_and is
  signal y: Boolean;
begin
    y <= a and b;
    q <= y when clk and clk'event;
end dataflow;
```

or, using the IEEE 1164 std_logic data types (which are described in more detail in Chapter **Error! Reference source not found.**, **Error! Reference source not found.** and in Chapter **Error! Reference source not found.**, **Error! Reference source not found.**):

```
architecture dataflow of my_and is
  signal y: std_logic;
begin
    y <= a and b;
    q <= y when rising_edge(clk);
end dataflow;
```

Note: *If you intend to process your VHDL designs using synthesis or simulation tools that do not support the IEEE 1076-1993 standard, then you should avoid using the preceding language style to describe registered logic.*

Avoiding Unwanted Latches

When you are describing combinational or registered logic using ABEL-HDL, you are describing the conditions under which one or more design outputs are to be asserted with a high value. For example, you might describe a simple multiplexer using the following ABEL-HDL statements:

```
module mux

    s0,s1         pin;       "select inputs
    a0,a1         pin;       "A inputs
    b0,b1         pin;       "B inputs
    c0,c1         pin;       "C inputs
    y1,y0         pin istype 'com'; "output Y

equations
    [y1,y0] = [a1,a0] & [s1,s0]
            # [b1,b0] & [s1,!s0]
            # [c1,c0] & [!s1,s0];
end
```

This simple design assigns the value of outputs **y1** and **y0** to the values of **a1** and **a0** when **s1** is high and **s0** is high, to **b1** and **b0** when **s1** is high and **s0** is low, and to **c1** and **c0** when **s1** is low and **s0** is high.

As written, the function of the multiplexer is incompletely specified, since there is no value specified for the condition in which **s1** and **s0** are both low. In ABEL-HDL, the default logical condition is false (low) so the values of **y1** and **y0** will be a logic low when **s1** and **s0** are both low.

In VHDL, however, it is not always the case that an unspecified logic condition will result in a low value on the output. *Instead, the rules of VHDL state that an unspecified condition will result in the output holding its state.* For synthesis purposes, this rule implies that a latch must be inserted into the circuit.

Perhaps the most common mistake that is made by new VHDL users (who have had experience with PLD-oriented languages) is the assumption that unspecified conditions will have no effect on the logic of the generated circuit. Perhaps the most common example of this is in the use if conditional assignments within process statements. Consider, for example, the following VHDL description of the same multiplexer just presented:

```
entity mux is
port (s0,s1,a0,a1,b0,b1,c0,c1: in bit;      --inputs
      y1,y0: out bit);                      --output Y
end mux;

architecture behavior of mux is
begin
    process(s0,s1,a0,a1,b0,b1,c0,c1)
    begin
        if    (s1,s0) = "11" then
              (y1,y0) <= (a1,a0);
        elsif (s1,s0) = "10" then
              (y1,y0) <= (b1,b0);
        elsif (s1,s0) = "01" then
              (y1,y0) <= (c1,c0);
    end process;
end
```

This design description is also incompletely specified; no value is specified for the condition in which **y1** and **y0** are both low. Unlike the ABEL-HDL version of this circuit, however, the rules of VHDL dictate that the value of the outputs must be held over time, rather than transition to a low state. For this design, the circuit that results must include a latch, and the VHDL synthesizer will construct this latch by feeding back the outputs to create an asynchronous feedback loop.

To prevent the creation of unwanted latches, you must make sure to include all of the possible input conditions in your design descriptions. In the case of conditional assignments such as the multiplexer, you should include a terminating else statement that defines the default value:

```
        if    (s1,s0) = "11" then
              (y1,y0) <= (a1,a0);
        elsif (s1,s0) = "10" then
              (y1,y0) <= (b1,b0);
        elsif (s1,s0) = "01" then
              (y1,y0) <= (c1,c0);
        else
              (y1,y0) <= "00";
```

State Machines

Describing State Machines in ABEL-HDL

In ABEL-HDL, you describe a state machine similar to the way you create a behavioral model in VHDL. You create a state description for each possible state of the machine, beginning each with a description of the state value to be stored in the state registers. For example:

```
module st_mach
    q1,q0       pin      istype 'reg';
    A,B         pin      istype 'com';

equations
    [q1,q0].clk = clock;

state_diagram [q1,q0]
    state [1,1]:
        A = 1;    "A is high in this state
        B = 1;    "B is high in this state
        if (start)
            then [1,1]   "Hold state
        else
            goto [0,0];  "Transition to 0,0
    .
    .
    .
end st_mach
```

In ABEL-HDL, you must specify the actual registers that are used (in this case, **q0** and **q1**) and the values to be stored in those registers when in each state.

ABEL-HDL state diagrams are normally accompanied by one or more equations describing the clock, reset or other additional functions for the state registers.

Describing State Machines in VHDL

In VHDL, you write a **behavioral** model for the state machine, typically by adding an **if-then** conditional statement and a **case** statement inside of a **process** statement in your program. To more clearly distinguish the clock logic of state machines (the registered portion) from the transition logic (the combinational logic portion), it is good practice to use two process statements as shown in the following example. In this example, the first process describes the registered behavior of the state machine, while the second process describes the transition logic, and the combinational output logic for the machine:

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

entity machine is
   port (clk,reset: in std_logic;
         state_inputs: in std_logic_vector (0 to 1);
         state_outputs: out std_logic_vector (0 to 1));
end machine;

architecture behavior of machine is
    type states is (st0, st1, st2, st3);
    signal present_state, next_state: states;
begin
    register: process (clk)
    begin
        if reset = '1' then
            present_state <= st0;        -- async reset to st0
        elsif rising_edge(clk) then
            present_state <= next_state; -- transition on clock
        end if;
    end process;

    transitions: process(present_state, state_inputs)
    begin
        case current_state is            -- describe transitions
            when st0 =>                  -- and comb. outputs
                state_outputs <= "00";
                if state_inputs = "11" then
                    next_state <= st0;   -- hold
                else
                    next_state <= st1;   -- next state
                end if;

            when st1 =>
                state_outputs <= "01";
                if state_inputs = "11" then
                    next_state <= st1;   -- hold
                else
                    next_state <= st2;   -- next state
                end if;

            when st2 =>
                state_outputs <= "10";
                if state_inputs = "11" then
                    next_state <= st2;   -- hold
                else
                    next_state <= st3;   -- next state
                end if;

            when st3 =>
                state_outputs <= "11";
                if state_inputs = "11" then
                    next_state <= st3;   -- hold
                else
                    next_state <= st0;   -- next state
                end if;

         end case;
    end process;
end behavior;
```

A variety of state machines designs are provided with your VHDL Synthesizer software. The **craps**, **prep3** and **prep4** designs included in the tutorials chapter of the *VHDL Entry* manaul are examples of alternative ways to describe state machine designs in VHDL.

A Standard ABEL-HDL Design in VHDL

The VHDL source file, **cntbuf.vhd**, shows how one of the standard ABEL-HDL examples can be written in VHDL. This example demonstrates:

- Pin assignments
- Bidirectional I/O
- Output enable conventions
- Synchronous reset logic

The complete VHDL description is listed below.

```
------------------------------------------------------
-- VHDL Version of standard ABEL example CNTBUF.ABL
-- Michael Holley, Data I/O Corp.
--
-- Copyright 1994, Data I/O Corporation
--
library ieee;
use ieee.std_logic_1164.all;

entity cntbuf is
    port( Dir: in std_logic;
          Clk,Clr,OE: in std_logic;
          A,B: inout std_logic_vector(0 to 1) bus;
          Q: inout std_logic_vector (3 downto 0) bus);

    attribute pinnum : string;      -- Must define the attribute

    attribute pinnum of Clk : signal is  "1";
    attribute pinnum of Clr : signal is  "2";
    attribute pinnum of Dir : signal is  "3";
    attribute pinnum of OE : signal is  "11";
    attribute pinnum of A : signal is  "13,12";--A_0_=3,A_1_=12
    attribute pinnum of B : signal is  "19,18";--B_0_=19,B_1_=18
    attribute pinnum of Q : signal is  "17,16,15,14";

end cntbuf;
library dataio;
use dataio.std_logic_ops.To_Vector;
architecture example of cntbuf is
    signal Count: integer range 0 to 15;
begin
    process (Dir,A,B)             -- Bi-directional buffer
    begin
        if Dir = '1' then
            B  <= "ZZ";           -- Make B high Z
            A  <= B;
        else
            B  <= A;
            A  <= "ZZ";           -- Make A high Z
```

```
        end if;
    end process;
    process (Clk,OE,Count)        -- Counter
    begin
        if rising_edge(Clk) then -- Edge triggered
            if Clr = '1' then
                Count <= 0;
            elsif Count = 15 then
                Count <= 0;
            else
                Count  <= Count + 1;
            end if;
        end if;
        if OE = '1' then
            Q <= "ZZZZ";          -- Make Q high Z
        else
            Q <= To_Vector(4,Count);
        end if;
    end process;
end example;
```

Design I/O

In this design, the I/O ports are assigned to pins as described earlier in this chapter. (This design is intended for implementation in a 20-pin PLD such as an Altera E0320.) Std_logic_vectors **A** and **B** represent two bidirectional buffers controlled by the **Dir** input. The convention for defining an output enable function in VHDL is to specify an assignment to **'Z'** for the disabled state. In this design, **A**, **B** and **Q** all have output enable functions defined using '**Z**'.

Combinational Logic

The combinational logic for **A** and **B** is defined in a process statement, but could just as easily have been defined using concurrent statements. (Whether to use concurrent statements or sequential statements for combinational logic is largely a matter of personal taste.) When a process statement is used to define a combinational logic function, all of the inputs to that logic function must be entered in the sensitivity list. In this case, the three inputs are **Dir**, **A** and **B**.

Registered Logic

The counter portion of this design is described in the second process statement. This process statement includes both **Clk** and **OE** in its sensitivity list because the output enable control for **Q** is not dependent on the clock. The sensitivity lists for a process must contain all inputs that are to be processed asynchronously. To provide a clocking function for the counter, the counter logic is contained in an **if** statement that describes the clock input using the previously described convention for edge triggered flip-flop behavior.

Because IEEE 1164 std_logic_vector data types do not have a '+' operator defined for them, the counter portion of the design has been described using an integer data type (the signal **Count**). A type conversion function (**To_Vector**) has been used to convert the integer data type into a std_logic_vector data type suitable for the design's output. This type conversion function (and others) is provided in the **dataio library supplied with the VHDL option.**

D. ABEL-HDL Language Reference

The information in this appendix is provided to help you read and interpret the logic equations that the Project Navigator produces in reports and error messages. These equations use a subset of the ABEL-HDL equation language to represent the logic of your design. These equations are produced by most device fitter software, as well as the equation report generator, which displays the Synthesized, Reduced and Linked Equations.

The following equation is an example of the ABEL-HDL equation syntax:

```
Q0_.D = A & Dir & Sel
      # B & !Dir & Sel
      # C  & Sel;
```

The equations displayed are in a sum-of-products (2-level) form, and include the operators shown in **Table C-1**.

Table C-1: ABEL-HDL Operators

Operator	Description
=	Assignment
:=	Registered Assignment
!	Not (invert)
&	AND
#	OR
$	Exclusive-OR
!$	Exclusive-NOR

Dot extensions

Identifier names used in ABEL-HDL equations may include dot extensions. Dot extensions provide a means to refer specifically to internal signals and nodes that are associated with a primary signal in a design.

Dot extensions are used in complex language constructs, such as nested sets or complex expressions.

Pin-to-Pin Vs. Detailed Dot Extensions

Dot extensions refer to various circuit elements (such as register clocks, presets, feedback and output enables) that are related to a primary signal.

Some dot extensions are general purpose and are used with a wide variety of device architectures. These dot extensions are therefore referred to as pin-to-pin (or "architecture-independent"). Other dot extensions are intended for specific classes of device architectures, or require specific device configurations. These dot extensions are referred to as detailed (or "architecture-dependent" or "device-specific") dot extensions.

Table C-2 lists the ABEL-HDL dot extensions. Pin-to-pin dot extensions are indicated with a check in the **Pin-to-Pin** column.

Table C-2: Dot Extensions

Dot Ext.	Pin-to-pin	Description
.ACLR	✓	A device-independent asynchronous register reset, equivalent to .AR with ISTYPE 'buffer' (or .AP with ISTYPE 'invert').
.AP		Asynchronous register preset
.AR		Asynchronous register reset
.ASET	✓	A device-independent asynchronous register preset, equivalent to .AP with ISTYPE 'buffer' (or .AR with ISTYPE 'invert').
.CE		Clock-enable input to a gated-clock flip-flop
.CLK1	✓	Clock input to an edge-triggered flip-flop
.CLR	✓	A device-independent synchronous register reset, equivalent to .SR with ISTYPE 'buffer'.

Dot Ext.	Pin-to-pin	Description
.COM	✓	A combinational feedback from the flip-flop data input, normalized to the pin value and used to distinguish between pin (.PIN) and internal logic array (.COM) feedback.
.D		When on the left side of an equation, .D is the data input to a D-type flip-flop; on the right side, .D is combinational feedback.
.FB	✓	Register feedback
.FC		Flip-flop mode control
.J		J input to a JK-type flip-flop
.K		K input to a JK-type flip-flop
.LD		Register load input
.LE		Latch-enable input to a latch
.LH		Latch-enable (high) to a latch
.OE	✓	Output enable
.PIN	✓	Pin feedback
.PR		Register preset (synchronous or asynchronous)
.Q		Register feedback
.R		R input to an SR-type flip-flop
.RE		Register reset (synchronous or asynchronous)
.S		S input to an SR-type flip-flop
.SET	✓	A device-independent synchronous register preset, equivalent to .SP with ISTYPE 'buffer'.
.SP		Synchronous register preset
.SR		Synchronous register reset
.T		T input to a T-type (toggle) flip flop

Detailed Design Dot Extensions

Table C-3 shows the dot extensions that are used to describe different register types. The required dot extensions are indicated with a check in the **Extension Required** column.

Table C-3: Dot Extensions for Device-specific (detailed) Designs

Register Type	Extension Required	Supported Extensions	Definition
combinational (no register)		.oe	output enable
		.pin	pin feedback
		.com	combinational feedback
D-type flip-flop	✓	.clk	clock
	✓	.d	data input
		.fc	flip-flop mode control
		.oe	output enable
		.q	flip-flop feedback
		.sp	synchronous preset
		.sr	synchronous reset
		.ap	asynchronous preset
		.ar	asynchronous reset
		.pin	pin
JK-type flip-flop	✓	.clk	clock
	✓	.j	j input
	✓	.k	k input
		.fc	flip-flop mode control
		.oe	output enable
		.q	flip-flop feedback
		.sp	synchronous preset
		.sr	synchronous reset
		.ap	asynchronous preset
		.ar	asynchronous reset
		.pin	pin feedback
SR-type flip-flop	✓	.clk	clock
	✓	.s	set input
	✓	.r	reset input
		.oe	output enable
		.q	flip-flop feedback
		.sp	synchronous preset
		.sr	synchronous reset
		.ap	asynchronous preset
		.ar	asynchronous preset
		.pin	pin feedback

Register Type	Extension Required	Supported Extensions	Definition
T-type flip-flop	✓	.clk	clock
	✓	.t	toggle input
		.oe	output enable
		.q	flip-flop feedback
		.sp	synchronous preset
		.sr	synchronous reset
		.ap	asynchronous preset
		.ar	asynchronous reset
		.pin	pin feedback
L-type latch	✓	.d	data input
	✓	.le	latch enable input to a
		.lh	latch
		.oe	latch enable (high) input
		.q	to a latch
		.pin	output enable
			flip-flop feedback
			pin feedback
Gated clock D	✓	.clk or	clock or clock enable
flip-flop	✓	.ce	data input
		.d	output enable
		.oe	flip-flop feedback
		.q	pin feedback
		.pin	

Pin-to-Pin Design Dot Extensions

Table C-4 shows the dot extensions that are used (and which of those are required) for pin-to-pin design descriptions. The required dot extensions are indicated with a check in the **Required** column.

Table C-4: Dot Extensions for Architecture-independent (pin-to-pin) Designs

Register Type	Required	Allowable Extensions	Definition
combinational		none	output
(no register)		.oe	output enable
		.pin	pin feedback
registered	✓	.clr	synchronous preset
logic		.aclr	asynchronous preset
		.set	synchronous set
		.aset	asynchronous set
		.clk	clock
		.com	combinational feedback
		.fb	registered feedback
		.pin	pin feedback

Figure C-1 through **Figure C-9** show the effect of each dot extension. The actual source of the feedback may vary from that shown.

Figure C-1: Pin-to-pin Dot Extensions in an Inverted Output Architecture

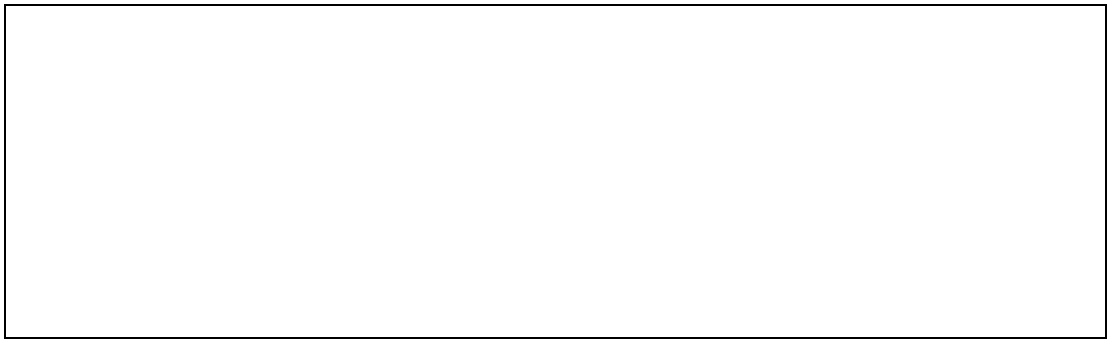


Figure C-2: Pin-to-pin Dot Extensions in a Non-inverted Output Architecture

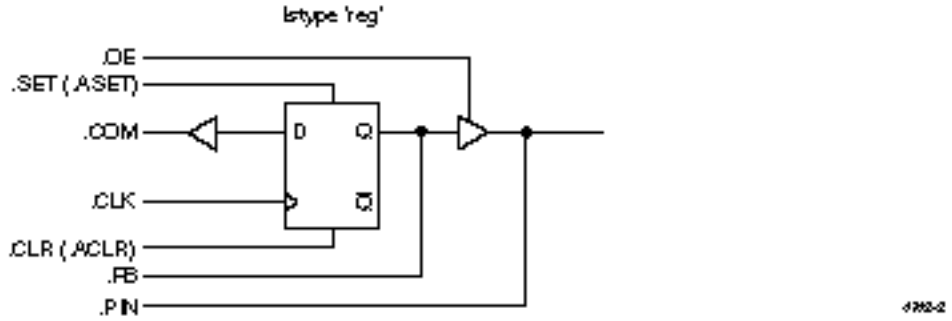


Figure C-3: Detailed Dot Extensions for an Inverted D-type Flip-flop Architecture

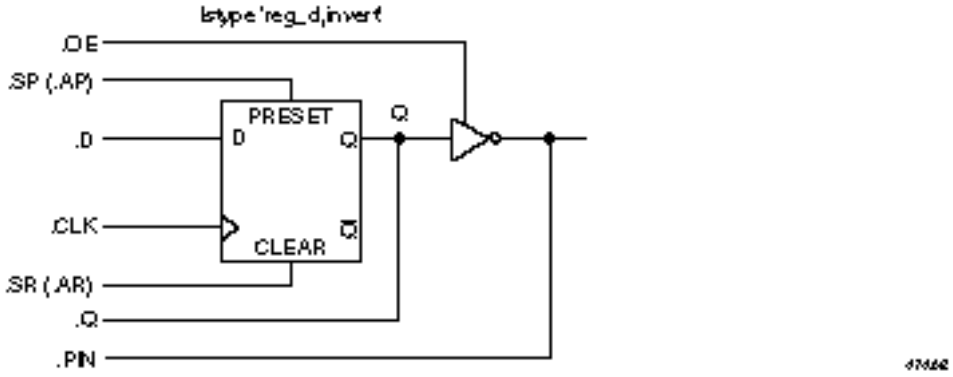


Figure C-4: Detailed Dot Extensions for an Inverted T-type Flip-flop Architecture

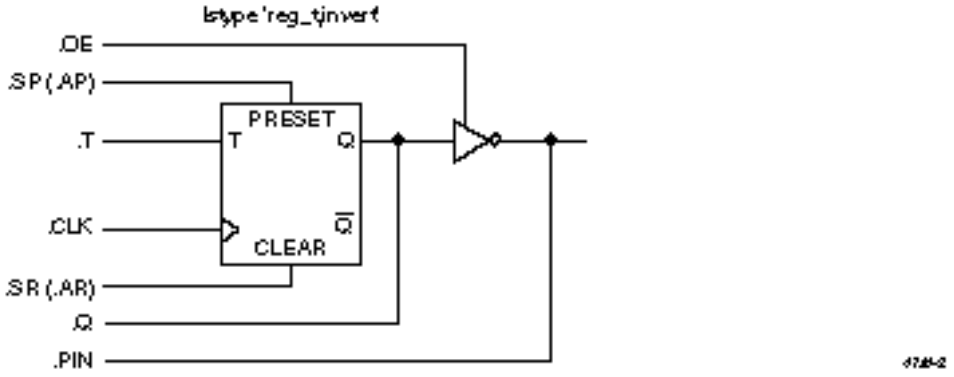


Figure C-5: Detailed Dot Extension for an Inverted RS-type Flip-flop Architecture

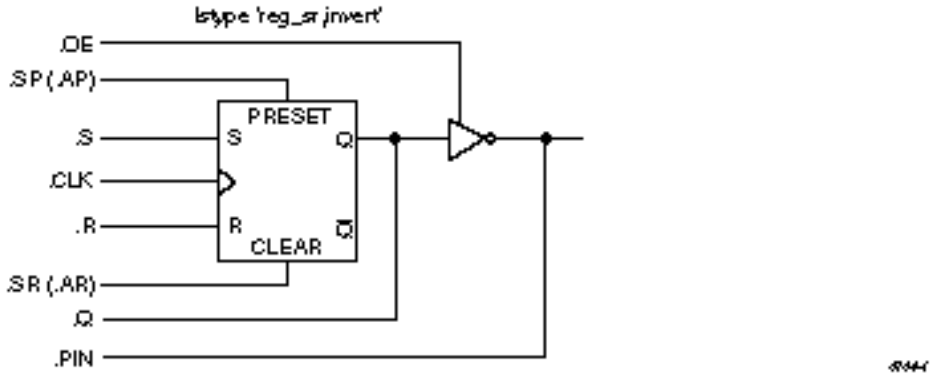


Figure C-6: Detailed Dot Extensions for an Inverted JK-type Flip-flop Architecture

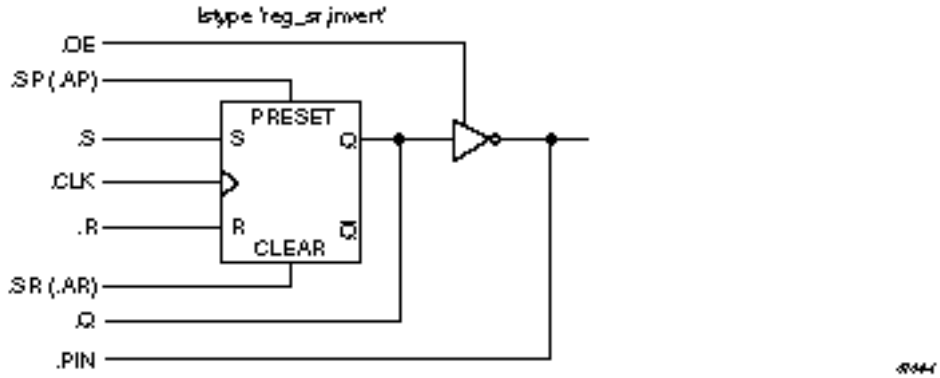


Figure C-7: Detailed Dot Extensions for an Inverted Latch with Active High Latch Enable

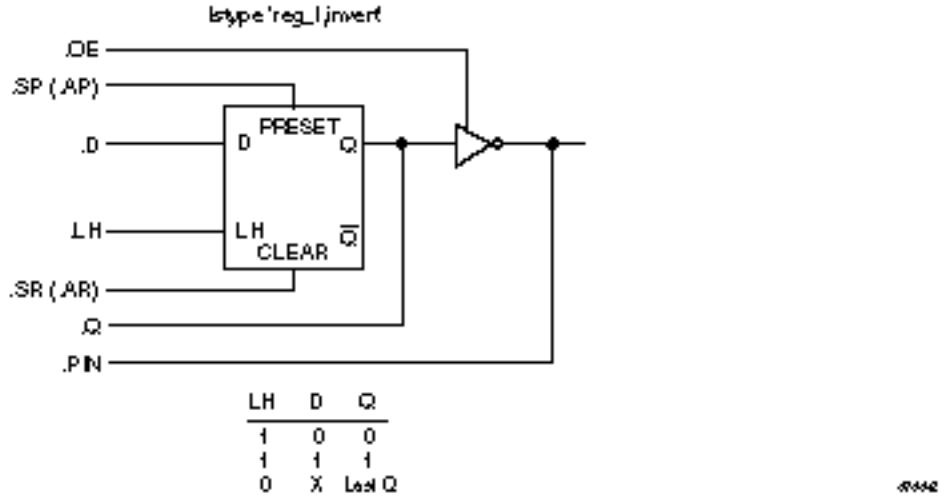


Figure C-8: Detailed Dot Extensions for an Inverted Latch with Active Low Latch Enable

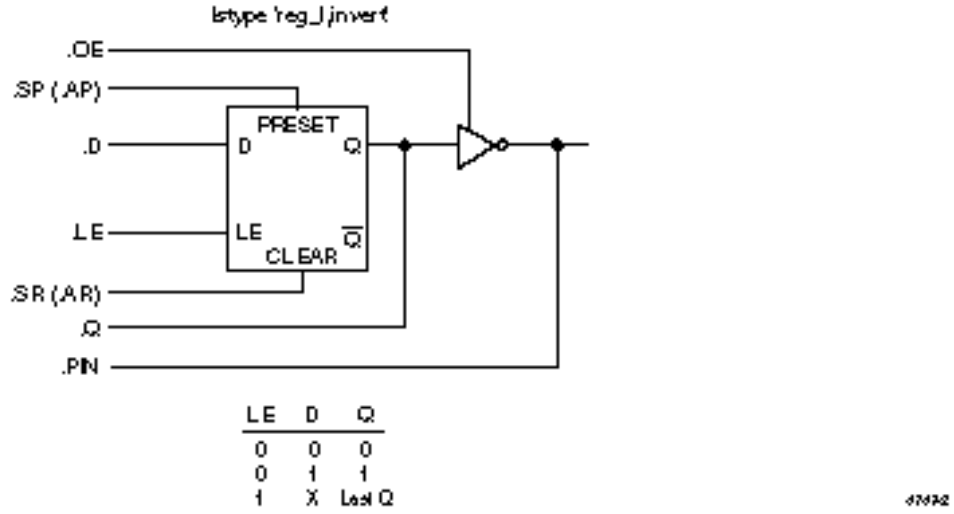
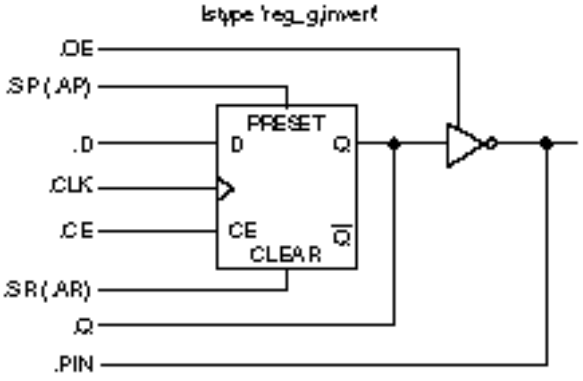


Figure C-9: Detailed Dot Extensions for an Inverted Gated-clock D Flip-flop



CE	CLK	D	Q
X	0	0	0
1	0	1	1
1	1	X	Last Q

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X