HDL Synthesis Guide

Release 4.2

Exemplar Logic, Inc.
6503 Dumbarton Circle
Fremont, CA 94555
Telephone: 800-632-3742
email: info@exemplar.com

Part No. 31315-R

Contents

Introduction to VHDL Synthesis 1

Overview

VHDL is a high level description language for system and circuit design. The language supports various levels of abstraction. Whereas regular netlist formats support only structural description, and a boolean entry system supports only dataflow behavior, VHDL supports a wide range of description styles. These include structural descriptions, dataflow descriptions and behavioral descriptions.

The structural and dataflow descriptions show a concurrent behavior. That is, all statements are executed concurrently, and the order of the statements is not relevant. On the other hand, behavioral descriptions are executed sequentially in processes, procedures and functions in VHDL. The behavioral descriptions resemble high-level programming languages.

VHDL allows a mixture of various levels of design entry. The Exemplar synthesis tools synthesize all levels of abstraction, and minimizes the amount of logic needed, resulting in a final netlist description in the technology of your choice. The Top-Down Design Flow enabled by the use of the synthesis tools is shown in Figure 1-1.

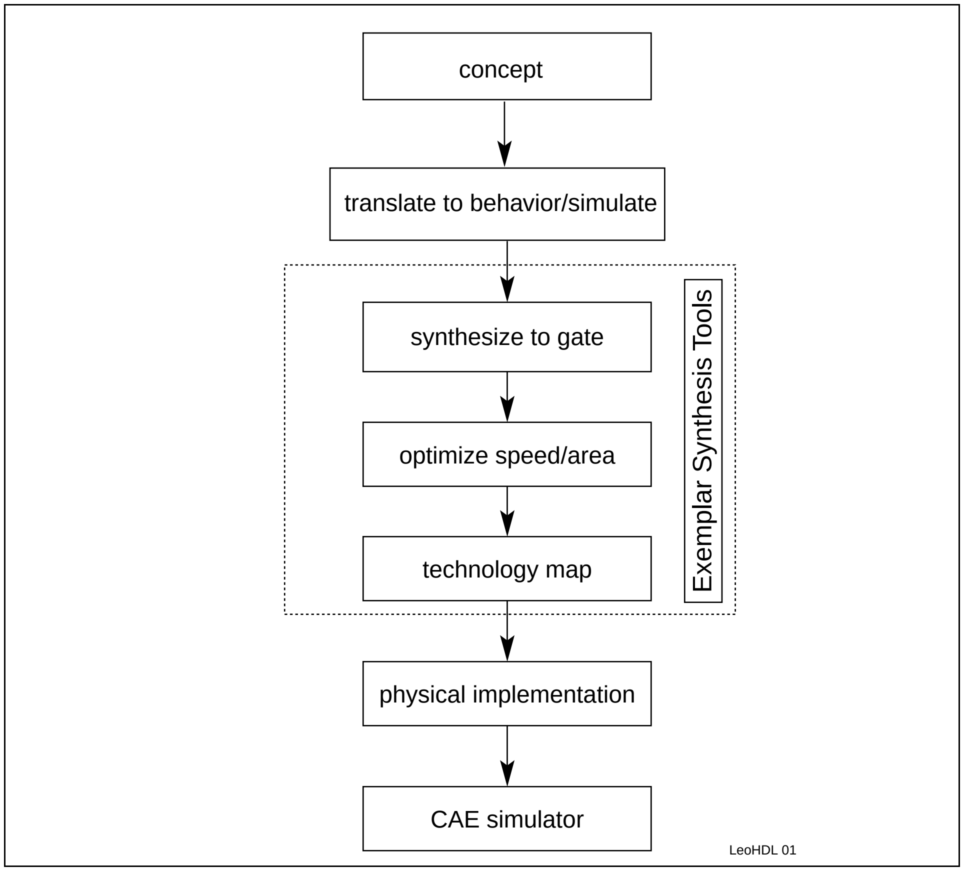

Figure 1-1 Top-Down Design Flow with the Exemplar synthesis tools

VHDL and Synthesis

VHDL is fully simulatable, but not fully synthesizable. There are a number of VHDL constructs that do not have valid representation in a digital circuit. Other constructs do, in theory, have a representation in a digital circuit, but cannot be reproduced with guaranteed accuracy. Delay time modeling in VHDL is an example.

State-of-the-art synthesis algorithms can optimize Register Transfer Level (RTL) circuit descriptions and target a specific technology. Scheduling and allocation algorithms, which perform circuit optimization at a very high and abstract level, are not yet robust enough for general circuit applications. Therefore, the result of synthesizing a VHDL description depends on the style of VHDL that is used. Users of the Exemplar synthesis tools should understand some of the concepts of synthesis specific to VHDL coding style at the RTL level in order to achieve the desired circuit implementation.

Synthesis tools are ideal for solving many of the cumbersome RTL logic optimization problems that occur during a typical top-down design project.

This manual is intended to give the VHDL designer guidelines to achieve a circuit implementation that satisfies the timing and area constraints set for a given target circuit, while still using a high level of abstraction in the VHDL source code. This goal will be discussed both in the general case for synthesis applications, as well as for the Exemplar synthesis tools specifically. Examples are used extensively; VHDL rules are not emphasized.

Knowledge of the basic constructs of VHDL is assumed, although Chapter 2, VHDL Language Features is dedicated to the discussion of all the constructs in VHDL that are useful for synthesis. If you need more details about VHDL, a comprehensive description of VHDL is given in the book "*VHDL*" by Douglas E. Perry (McGraw-Hill, Inc.), and VHDL related to digital circuits is discussed by Randolph E. Harr in "Applications of VHDL to Circuit Design" (Kluwer Academic Publishers). In addition, training on the Exemplar synthesis tools and VHDL for synthesis is available from Exemplar Logic, and training on VHDL and top-down design in general is available from a number of different sources.

In This Manual

The VHDL portion of this manual is organized as follows:

A basic description of the most relevant VHDL constructs is given in Chapter 2, "VHDL Language Features." Chapter 3, "The Art Of VHDL Synthesis," discusses VHDL for synthesis purposes. Within this chapter, a number of common digital circuits are analyzed, with examples of how to properly code these designs in VHDL. Chapter 4, "The VHDL Environment," deals with how the Exemplar synthesis tools are used together with other VHDL and CAE software, and how non-standard issues, such as file handling, are implemented. The Exemplar VHDL package is also presented in this chapter.

Customer Support

If you encounter problems using VHDL or the Exemplar synthesis tools, or if you have any questions or remarks about this VHDL manual, contact the Exemplar Customer Support Hot Line at (510) 337-3742, or send e-mail to `support@exemplar.com`.

VHDL Language Features 2

This chapter provides an introduction to the basic language constructs in VHDL: defining logic blocks, structural, dataflow and behavioral descriptions, concurrent and sequential functionality, design partitioning and more. The Exemplar synthesis tools synthesize all levels of abstraction, and minimizes the amount of logic needed, resulting in a final netlist description in the technology of your choice.

Entities and Architectures

The basic building blocks in VHDL are Entities and Architectures. An *entity* describes the boundaries of the logic block. Its ports and its generics are declared here. An *architecture* describes the contents of the block in structural, dataflow and behavioral constructs.

```
entity small_block is
      port (a, b, c : in bit ;
           o1 : out bit ;
           o2 : out bit
      ) ;
end small_block ;

architecture exemplar of small_block is
      signal s : bit ;
begin
      o1 <= s or c ;
      s <= a and b ;
      o2 <= s xor c ;
end exemplar ;
```

This VHDL description shows the implementation of `small_block`, a block that describes some simple logic functions.

The entity describes the boundary. The port list is given with a direction (in this case `in` or `out`), and a type (`bit`) for each port. The entity's name is `small_block`. The architecture's name is `exemplar` and it is linked to the entity via the name `small_block`. There can be multiple architectures per entity, but always only one architecture is executed. By default, the last defined architecture is linked to the entity.

The architecture describes the contents of the `small_block`. The architecture starts with a declarative region; in this case, the internal signal `s` is declared. It also has a type (`bit`), just like the ports in the entity.

A *signal* is another form of an object in VHDL. All objects and expressions in VHDL are strongly typed. This means that all objects are of a defined type and issues an error message if there is a type mismatch. For example, you cannot assign an integer of type `signal` to a `bit`.

The architecture contents starts after the `begin` statement. This is called the *dataflow environment* (please refer to the previous example). All statements in the dataflow environment are executed concurrently, and thus the order of the statements is irrelevant. This is why it is valid to use `s` before `s` is assigned anything. Assignment of a value to a signal is done with the `<=` sign. In the first statement, `o1` is assigned the result value of `s` or `c`. `or` is a predefined operator.

Additional details about the various dataflow statements and operators are given in the following sections.

Configuration

In summary, a configuration declaration provides the mechanism for delayed component binding specification. The entity name identifies the root entity to be elaborated. The optional architecture name provides the name of the architecture to be elaborated.

A configuration declaration can configure each component instantiation individually with a different entity or architecture. The configuration declaration can also configure some lower level component instantiation of the current component being configured.

With the help of the configuration declaration, you can try out different possible bindings of the component instantiations by keeping the basic hierarchical structure of the top level design intact.

NOTE: If you use “con” for configuration and “ent” for entity then the name of the hierarchy cell created is “`con_ent`”.

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

package global_decl is
    type log_arr is array(std_logic) std_logic;
    constant std_to_bin : log_arr:=('X','X','0','1','X','X', '0','1','X');
    function to_bin (from : std_logic) return std_logic;
end;
package global_decl is

    function to_bin (from : std_logic) return std_logic is
    begin
       return  std_to_bin(from);
    end;

end;
```

continued....

....continued

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

entity en1 is  port
      (a: in std_logic;
      b: out std_logic);
end;

architecture ar1 of en1 is
begin
   b <= to_bin (a);
end;


architecture ar2 of en1 is
begin
   b <= not (to_bin (a));
end;

library ieee;
library work;
use ieee.std_logic_1164.all;
use work.global_decl.all;

entity en2 is  port
      (a: in std_logic;
      b, c: out std_logic);
end;

architecture arc of en2 is
component en1 port
      (a: in std_logic;
      b: out std_logic);
end component;
```

continued....

....continued

```
begin
     c1: en1 port map (a => a, b => b);
     c2: en1 port map (a => a, b => c);

end;

library work;
configuration binding of en2 is
    for arc
        for c1: en1 use entity work.en1 (ar1);
        end for;
        for c2: en1 use entity work.en1 (ar2);
        end for;
    end for;
end binding ;
```

Processes

Processes are sections of sequentially executed statements, as opposed to the dataflow environment, where all statements are executed concurrently. In a process, the order of the statements *does* matter. In fact, processes resemble the sequential coding style of high level programming languages. Also, processes offer a variety of powerful statements and constructs that make them very suitable for high level behavioral descriptions.

A process can be called from the dataflow area. Each process is a sequentially executed program, but all processes run concurrently. In a sense, multiple processes resemble multiple programs that can run simultaneously. Processes communicate with each other via signals that are declared in the architecture. Also the ports defined in the entity can be used in the processes.

```
entity experiment is
        port ( source : in bit_vector(0 to 3) ;
                ce : in bit ;
                wrclk : in bit ;
                selector : in bit_vector(0 to 1) ;
                result : out bit
        );
end experiment;

architecture exemplar of experiment is
        signal intreg : bit_vector(0 to 3) ;

begin   -- dataflow environment
        writer : process         -- process statement
                                 -- declarative region (empty here)
        begin                    -- sequential environment
                -- sequential (clocked) statements
                wait until wrclk'event and wrclk = '1' ;
                if (ce='1') then
                        intreg <= source ;
                end if ;
        end process writer;

        reader : process (intreg, selector)   -- process statement
                                             -- with sensitivity list
                                         -- declarative region (empty here)
        begin
                    -- sequential (not-clocked) statements
                case selector is
                    when "00" => result <= intreg(0) ;
                    when  "01" => result <= intreg(1) ;
                    when  "10" => result <= intreg(2) ;
                    when  "11" => result <= intreg(3) ;
                end case ;
        end process reader;
end exemplar ;
```

This example describes a circuit that can load a source vector of 4 bits, on the edge of a write clock (`wrclk`), store the value internally in a register (`intreg`) if a chip enable (`ce`) is active, while it produces one bit of the register constantly (not synchronized). The bit is selected by a selector signals of two bits.

The description consists of two processes, one to write the value into the internal register, and one to read from it. The two processes communicate via the register value intreg.

The first process (`writer`) includes a wait statement. The wait statement causes the process to execute only if its condition is true (a further explanation is given later in the chapter). In this case, the wait statement waits until a positive edge occurs on the signal `wrclk` (expression `wrclk’event and wrclk=’1’`). Each time the edge occurs, the statements below the wait statements are executed. In this case, the value of the input signal source is loaded into the internal signal `intreg` only if `ce` is`’1’`. If `ce` is`’0’`, `intreg` retains its value. In synthesis terms, this translates into a D-flipflop, clocked on `wrclk`, and enabled by `ce`.

The second process (`reader`) does not have a wait statement. Instead, it has a sensitivity list, with the signals `intreg` and `selector` there. This construct defines that the whole process is executed each time either `intreg` or `selector` changes. If the process is executed, the output signal `result` gets updated with depending on the values of `intreg` and `selector`. Note that this leads to combinational behavior, since `result` depends on only `intreg` and `selector`, and each time either of these signals changes, `result` gets updated.

A process has an optional name (in this case `writer` and `reader`), a sensitivity list OR a wait statement, and a declarative region where signals, variables, functions etc. can be declared which are used only within the process. The next section of the process is the sequential environment where all statements are made. Each statement is executed sequentially, as in a programming language.

Not all constructs, or combinations of constructs, in a process lead to behavior that can be implemented as logic. For more information about synthesizable constructs, refer to “Syntax and Semantic Restrictions” on page 22.

Literals

Constant values in VHDL are given in literals. *Literals* are lexical elements. Below is an overview, with examples given for each type of literal.

Character Literals:	’0’ ’X’ ’a’ ’%’#
String Literals:	“1110100” “XXX” “try me!” “$^&@!”
Bit String Literals:	B“0010_0001” X”5F’ O“63_07”
Decimal Literals:	27 -5 4E3 76_562 4.25
Based Literals:	2#1001# 8#65_07" 14#C5#E+2
Physical Literals:	2 ns 5.0 V 15 pF
Identifiers:	Idle TeSTing a true_story

Literals are used to define types and as constant values in expressions. This list provides a brief description of their function in VHDL which will be more clear after the descriptions of types and expressions.

The `’_’` in bit string literals, decimal literals and based literals helps to order your literal, but does not represent a value.

Character literals contain only a single character, and are single quoted.

String literals contain an array of characters, and are double quoted.

Bit String Literals are a special form of string literals. They contain an array of the characters 0 and 1, and are preceded by one of three representation forms. B is the bit representation (0 or 1 allowed), X the hexadecimal representation (0 to F allowed) and O the octal representation (0 to 7 allowed). X"5F" is exactly the same as B"01011111", which is again the same as the string literal "01011111".

Bit string literals can contain underscores, which are ignored and only inserted for readability.

Decimal literals are `integer` or `real` values.

Based literals are also `integer` or `real` values, but they are written in a based form. 8#75# is the same as decimal 61. However it is not the same as the bit literal value O"75" since the bit literal value is an array (of bits) and the based literal is a integer.

Physical literals are sometimes required for simulation. As they are not used in the synthesized part of the design, we do not go into detail about them.

Identifiers can be enumeration literals. They are case-insensitive, like all identifiers in VHDL. Their use becomes more clear with the discussion of VHDL types.

Types

A *type* is a set of values. VHDL supports a large set of types, but here we concentrate on types that are useful for synthesis.

VHDL is a strongly typed language: every object (see “Objects” on page 30) in a VHDL source needs to be declared and needs to be of a specific type. This allows the VHDL compiler to check that each object stores a value that is in its type. This avoids confusion about the intended behavior of the object, and in general allows the user to catch errors early in the design process. It also allows overloading of operators and subprograms (“User-Defined Attributes” on page 46 and “Resolution Functions” on page 52). It also make coding in VHDL a look more difficult at first sight, but tends to produce cleaner, better maintainable code in the end.

VHDL defines four classes of types:

- Scalar types
- Composite types
- Access types
- File types

Access types and File type cannot be applied for logic synthesis, since they require dynamic resource allocation, which is not possible in a synthesized hardware (see “VHDL Language Restrictions” on page 23). Therefore, we will not discuss these.

Instead, only scalar types and composite types will be discussed. These are all scalar types in VHDL:

- Enumeration types.
- Integer types
- Floating-point types
- Physical types

VHDL has two forms of composite types:

- Array types
- Record types.

This section will discuss the syntax and semantics of scalar and composite types, and comment about the synthesizability of objects of these types.

Finally, this section will discuss some of the built-in standard types of the language (IEEE 1076), and a standardized set of types that are often used for logic synthesis purposes (IEEE 1164).

Enumeration Types

Syntax and Semantics

An *enumeration* type consists of a set of literals (values). It indicates that objects of that type cannot contain any other values than the ones specified in the enumeration type.

An example of an enumeration type is the pre-defined type `bit`. This is how the type `bit` is declared:

```
    type bit is ('0','1') ;
```

Any object of type `bit` can only contain the (literal) values `'0'` and `'1'`. The VHDL compiler will error out (type error) if a different value could be assigned to the object.

Enumeration types are also often used to declare the (possible) states of a state machine. Here is an example of the declaration of the states of an imaginary state machine are declared:

```
    type states is (IDLE, RECEIVE, SEND) ;
```

Once an object of this type is declared, the object can contain only one of these three ‘state’ values.

Synthesis Issues

It is important to understand a logic synthesis tool needs to do state encoding on any enumeration type. For example, the `states` type in the previous section needs at least two bits to represent the three possible values. This section mainly deals with the various forms of controlling the enumeration encoding for each enumeration type in your design.

By default, the synthesis tools perform `onehot` encoding on an enumeration type. With Galileo, any other encoding can be achieved with a global switch (`-encoding`). With Leonardo, other encodings can be achieved by using the `encoding` variable. In addition both tools support alternate encodings by using any of the following attributes:

- `TYPE_ENCODING_STYLE` (define the encoding style for state machine type encoding)
- `TYPE_ENCODING` (define the bit-to-bit encoding for state machine type values manually)
- `LOGIC_TYPE_ENCODING` (define that the type needs to be synthesized into a single binary value)

These three attributes are declared in the `exemplar_1164` package. So you do not need to declare them if you use a `use exemplar.exemplar_1164.all` statement in your design unit. For more information, see “The Exemplar Packages” on page 11.)

The `LOGIC_TYPE_ENCODING` attribute on an enumeration type will give a hint to the compiler that any object of the type should be encoded with a single bit, even though there might be more than two value in the type. An example of a type where `LOGIC_TYPE_ENCODING` is helpful, is the type `std_ulogic` in the IEEE 1164 package (see “IEEE 1076 Predefined Types” on page 28). The type consists of nine values, but the synthesis tools should encode any object of `std_ulogic` as a single bit value. Here is how the synthesis tools encode `std_ulogic` as a single-bit value:

```
-- Declare the LOGIC_TYPE_ENCODING attribute :
attribute LOGIC_TYPE_ENCODING : string ;

-- Declare the std_ulogic type :
type std_ulogic is (’U’,’X’,’0’,’1’,’Z’,’W’,’L’,’H’,’-’) ;

-- Set the LOGIC_TYPE_ENCODING attribute on the std_ulogic type :
attribute LOGIC_TYPE_ENCODING of std_ulogic:type is
           (’X’,’X’,’0’,’1’,’Z’,’X’,’0’,’1’,’X’) ;
```

`LOGIC_TYPE_ENCODING` takes an array of characters, as many as there are values in the type, and each character states how the synthesis tools should treat the related value. There are four values that the synthesis tools accepts as legal single bit values for the `LOGIC_TYPE_ENCODING` attribute: `’0’,’1’,’X’,’Z’`.

`’0’` : Treat the value as a logic zero.

`’1’` : Treat the value as a logic one.

`’X’` : Treat the value as either a logic one or a logic zero. The Exemplar synthesis tools can decide which one, depending on the context it is used in. The synthesis tools will use this freedom to optimize the circuit as much as it can.

`’Z’` : Treat the value as a high-Z values. The synthesis tools will generate a three-state driver if this value is used in an assignment.

The synthesis tools can work with all values of a type with a `LOGIC_TYPE_ENCODING` attribute. Only comparisons of a `NON-STATIC` value with `’X’` or `’Z’` will return `FALSE`.

The `TYPE_ENCODING` and `TYPE_ENCODING_STYLE` attributes on an enumeration type are used to control state-encoding for state-machine descriptions. Normally, state-machines in VHDL are described by giving a enumeration type that identifies each possible state of the state machine. The encoding for this enumeration type is done by the synthesis tools. By default, they use `BINARY` encoding.

The TYPE_ENCODING_STYLE gives a hint to the compiler as to what kind of encoding style to choose. There are four different styles to choose from: BINARY, GRAY, ONEHOT, RANDOM. Here is an example of how to use the TYPE_ENCODING_STYLE attribute on a (imaginary) state enumeration type:

```
-- Declare the TYPE_ENCODING_STYLE attribute
-- (not needed if the exemplar_1164 package is used) :
type encoding_style is (BINARY, ONEHOT, GRAY, RANDOM) ;
attribute TYPE_ENCODING_STYLE : encoding style ;

-- Declare the (state-machine) enumeration type :
type my_state_type is (SEND, RECEIVE, IGNORE, HOLD, IDLE) ;

-- Set the TYPE_ENCODING_STYLE of the state type :
attribute TYPE_ENCODING_STYLE of my_state_type:type is ONEHOT ;
```

In the above example, the synthesis tools will use one-hot encoding for the values of my_state_type. More specifically, the synthesis tools will use five bits for the type and will encode the states as follows:

```
               bit4   bit3   bit2   bit1  bit0
   SEND          -      -      -      -     1
   RECEIVE       -      -      -      1     -
   IGNORE        -      -      1      -     -
   HOLD          -      1      -      -     -
   IDLE          1      -      -      -     -
```

The '-' value will allow the synthesis tools to only compare a single bit when a state value is tested for. When a state value is assigned, '-' means a 0. This scheme allows the synthesis tools to eliminate almost all logic when testing for the state machine to be in a particular state. On the other hand, since ONEHOT encoding requires more bits than other encoding styles, the number of flip-flops will increase. ONEHOT encoding can therefore be very beneficial for technologies where flip-flops are not expensive, but combinational logic is (like in the Xilinx architectures).

Naming: For `ONEHOT` encoding, the synthesized bits of a state machine will be named after the bit number in the above table. Here is an example:

```
signal state : my_state_type ;
```

The signal `state` will be synthesized with one-hot encoding style, and the synthesis tools will generate five bits for it, where each one gets the state number from the above table:

state(4)	corresponds to bit4 in the state table
state(3)	corresponds to bit3 in the state table
state(2)	corresponds to bit2 in the state table
state(1)	corresponds to bit1 in the state table
state(0)	corresponds to bit0 in the state table

For BINARY encoding (the default) the synthesis tools will use the following state table:

```
                bit2 bit1 bit0
    SEND         0    0    0
    RECEIVE      0    0    1
    IGNORE       0    1    0
    HOLD         0    1    1
    IDLE         1    -    -
```

`BINARY` encoding (as `GRAY` and `RANDOM` encoding) uses the minimum number of bits needed to encode all values. In the above case (five values), `BINARY` encoding needs three bits. The last value (for IDLE) in the above table indicates several `'-'`s. The `'-'` (just as the `'-'`) value is used to reduce the size of comparators needed to test the state.

Naming: For `BINARY` encoding, as well as for `GRAY` and `RANDOM` encoding, the synthesis tools will generate the minimum number of bits needed for an object of the type. The signal `state` will now generate three bits, each with the following name:

`state(2)`	corresponds to `bit2` in the state table
`state(1)`	corresponds to `bit1` in the state table
`state(0)`	corresponds to `bit0` in the state table

`GRAY` encoding lets the synthesis tools build a Gray-code encoding. Gray-code encoding assures that in each successive value, only one single bit changes:

```
          bit2 bit1 bit0
SEND       0    0    0
RECEIVE    0    0    1
IGNORE     0    1    1
HOLD       0    1    0
IDLE       1    1    0
```

Gray encoding does not use the optimization possible with the `'-'` value. Gray encoding reduces glitches in the combinational logic when moving from one value (`state`) to its successor. It can be helpful in designs that require very clean logic switching and state machines that do not perform many jumps to different states.

`RANDOM` encoding will create a random encoding scheme. The state table cannot be predicted, nor is there any way to let the synthesis tools produce it for you. `RANDOM` encoding is interesting if you would like to see whether or not the circuit size of performance depends heavily on the state encoding.

To fully control the state encoding, use the TYPE_ENCODING attribute. With the TYPE_ENCODING attribute you can define the state table used. Here is an example:

```
-- Declare the TYPE_ENCODING attribute :
type exemplar_string_array is array (natural range <>, natural range <>)
    of character ;
attribute array_pin_number : exemplar_string_array ;
attribute TYPE_ENCODING : exemplar_string_array ;

-- Declare the (state-machine) enumeration type :
type my_state_type is (SEND, RECEIVE, IGNORE, HOLD, IDLE) ;

-- Set the type-encoding attribute :
attribute TYPE_ENCODING of my_state_type:type is
                        ("0001","01--","0000","11--","0010") ;
```

The TYPE_ENCODING attribute takes an array of equal-length strings, where each string defines a row in the state table. The TYPE_ENCODING attribute is declared in the exemplar_1164 package, so if you use that, you do not have to enter the declaration for it.

This attribute setting will let the synthesis tools to use the following state table:

	bit3	bit2	bit1	bit0
SEND	0	0	0	1
RECEIVE	0	1	-	-
IGNORE	0	0	0	0
HOLD	1	1	-	-
IDLE	0	0	1	0

Note – The number of bits used in the TYPE_ENCODING attribute value does not have to be the smallest possible number of bits. Just make sure that you specify as many strings as there are values in the enumeration type. Also note that you can use the '-' value to let the Exemplar synthesis tools know to not use these bits when testing is the state machine is in the given state. You can use this to reduce the size of the circuit.

Right now, the synthesis tools do not have an algorithm to find a good state encoding for any enumeration type. Still, the various forms of manual state table control explained in this section should allow you to find a good state encoding for each state machine in your design.

The attributes described in this section allow you to encode each state machine (each state-type) individually. Galileo also provides a command line switch (`-encoding`) that sets the default encoding (`BINARY`) to either `BINARY`, `ONEHOT`, `GRAY` or `RANDOM`. This command-line switch is useful to quickly switch from one state encoding style to another on a design with a single state machine. Any of the above encoding attributes overwrite any default setting. For Leonardo, set the `encoding` variable to `BINARY` (default), `ONEHOT`, `GRAY` or `RANDOM` before reading in a design to use a different encoding style for the state machines in the design.

An interesting effect of this way of handling encoding for enumeration types in synthesis of the predefined type `character` in VHDL. The `character` type is defined in the package `standard`, as an enumeration of all characters in the 8-bit ASCII set. When `BINARY` encoding (default) is chosen, each character will be synthesized into seven bits, with exactly its 8-bit ASCII value. So, the synthesis tools can synthesize characters (and strings) representing them as ASCII values. If a different default encoding is chosen, the encoding of the character type will change accordingly.

Integer Types

Syntax and Semantics

When designing arithmetic behavior, it is very helpful to work with integer types. An integer type defines the set of integer values in its range. This is how an integer type is defined:

```
type my_integer is range 0 to 15 ;
```

Any object of type `my_integer` can only contain integer values in the range specified. VHDL pre-defines an integer type called `integer`, that at least covers a range of integer values that can be represented in two's complement with 32 bits:

```
type integer is range -2147483647 to 2147483647;
```

Actually, VHDL 1076 does not define the maximum bounds of the predefined type `integer` nor of any other integer type, it just states that it should at least include this range.

Synthesis issues

The Exemplar synthesis tools can synthesize with any integer type that contains no values outside the range -2147483648 to 2147483647. The reason is that the synthesis tools store integer values (constant ones) using (32 bit) integers internally. If more than 32 bits are needed for a particular circuit design, you should use arrays to represent them. It is not wise to use integer types that exceed the above range in general, since many other VHDL tools have the same restriction as the Exemplar synthesis tools.

The synthesis tools need to do encoding for integer types, since an integer range requires multiple bits to represent. The synthesis tools will analyze the range of an integer type and calculate the number of bits needed to represent it.

If there are no negative values in the integer range, the synthesis tools will create an unsigned representation. For example, consider the following object of the type `my_integer` from the previous section:

```
   signal count : my_integer ;
```

The signal `count` will be represented as unsigned, consisting of four bits. When synthesized, the four bits will be named as elements of a bus in the resulting netlist:

```
count(0)     the LSB bit
count(1)
count(2)
count(3)     the MSB bit
```

If the range includes negative numbers, the synthesis tools will use two’s-complement representation of the integer values. For example, any object of the predefined type `integer` will be represented with 32 bits where the MSB bit represents the sign bit.

Example:

```
signal big_value : integer ;
```

Now, the synthesis tools will represent the signal `big_value` as 32 bits:

```
big_value(0)   the LSB bit
big_value(1)
    :
    :
big_value(30)  the MSB bit
big_value(31)  the sign bit
```

Floating-point Types

Syntax and Semantics

As any high-level programming language, VHDL defines floating-point types. *Floating-point* types approximate the real numbers.

Here is an example of the declaration of a floating-point type:

```
  type my_real is range 0.0 to 1.0 ;
```

VHDL pre-defines a very general floating-point type called `real.`

```
type real is range -1.0E38 to 1.0E38 ;
```

Just as with the integer types, maximum bounds of any floating-point type is not defined by the language. Still, any floating-point type should but should at least include -1.0E38 to 1.0E38.

Nothing in the language defines anything about the accuracy of the resolution of the floating-point type values.

Synthesis Issues

In general, since the resolution of floating-point types is not defined by the language, it is difficult to come up with a good rule for encoding floating-point types. While in a regular (software) compilers floating-point types are represented in 32, 64 or 128 bits, the floating-point operations just require time. In hardware compilers like a logic synthesis tool, floating-point operations would require massive amounts of actual synthesized hardware, unless the resolution and bounds of the floating-point type are kept under very close control.

For the above reasons, the Exemplar synthesis tools do not currently support synthesis of floating point objects.

Floating-point types and objects can however be used in constant expression.

For example, an attribute could get a (compile time constant) floating-point expression, and the synthesis tools will calculate the expression and set the floating-point value on the attribute.

Physical Types

Syntax and Semantics

VHDL allows the definition of physical types. *Physical* types represent relations between quantities. A good example of a physical type is the predefined type `time`:

```
type time is range -2147483647 to 2147483647
      units
            fs;
            ps = 1000 fs;
            ns = 1000 ps;
            us = 1000 ns;
            ms = 1000 us;
            sec = 1000 ms;
            min = 60 sec;
            hr = 60 min;
      end units;
```

Objects of physical types can contain physical values of the quantities specified in the type, as long as the values do not exceed the type's range. Type `time` is often used in VHDL designs to model delay.

Synthesis Issues

Physical types, objects and values are normally only used for simulation purposes. Objects and values of type `time` are used in `after` clauses to model delay.

The Exemplar synthesis tools attempt to synthesize any physical value that is within the range of the type. The encoding follows the encoding for integer types, and expresses the value with respect to the base quantity (`fs` in the type `time`). It is not common practice however to synthesize logic circuitry to model physical values.

The synthesis tools handles constant expressions of physical values without any problems. For example, attributes of type `time` can receive constant values of type `time`. This is often used to model arrival time and required time properties in the design. (For more information, see “The Exemplar Packages” on page 11.)

Array Types

Syntax and Semantics

An *array* type in VHDL specifies a collection of values of the same type. There are constrained and unconstrained array types.

For an constrained array type, the number of elements and the name of the elements (the index) is defined and fixed.

Example:

```
type byte is array (7 downto 0) of bit ;
```

In this example, type `byte` defines an array of 8 element, each of type `bit`. The elements are named with indexes ranging from 7 (for the left most element in the array) downto 0 (for the right most element in the array). Example of an array object:

```
constant seven : byte := "00000111" ;
```

Individual elements of the array object can now be referred to using indexing:

> `seven(0)` is the name of the right most element in array `seven`. Its value is the bit literal `'1'`.
>
> `seven(7)` is the name of the left most element in array `seven`. Its value is the bit literal `'0'`.

Parts of the array can be retrieved using slicing:

> `seven(3 downto 0)` is the name of the right most four elements in array `seven`. The value is an array of four bits: `"0111"`. The indexes of this array range from `3` down to `0`.

For an unconstrained array type, the number of elements and the name of the elements in not yet defined. An example is the pre-defined type `bit_vector`:

```
type bit_vector is array (natural range <>) of bit ;
```

Here, the array type defines that the element type is `bit`, and that the index type is type `natural`. Type `natural` is a integer subtype that include all non-negative integer. The meaning of this is that the index value for any object of type `bit_vector` can never be negative.

By defining an unconstrained array type, you defer specifying a size for the array. Still, in order to define a valid object of an unconstrained array type, we need to constraint the index range. This is normally done on the object declaration:

```
constant eight : bit_vector (7 downto 0) := "00001000" ;
```

Unconstrained array types are very important, since they allow to declare many different-size objects to be declared and used through each other, without introducing type conflicts.

The type of the element of an (constrained or unconstrained) array type is not restricted to enumerated type `bit` as in the examples above. Actually, an array element type can be any type but an unconstrained array type.

So you could define an array of integers, an array of 6-bit arrays, an array of records etc. But you cannot declare an array of (the unconstrained array type) `bit_vector`.

If you want an unconstrained array type where you need more indexes to remain unconstrained, you need a multi-dimensional array type:

```
type matrix is array (natural range <>, natural range <>) of bit ;
```

Multi-dimensional (constrained and unconstrained) array type are handy when modeling RAMs, ROMs and PLAs in VHDL. The section “Edge-Sensitive Flip-Flops” on page 3 gives some examples. Indexes and slices of multi-dimensional arrays need to specify all index dimensions, separated by a comma. Again, “Edge-Sensitive Flip-Flops” on page 3 gives examples.

Finally, the index type of an array type does not have to be an integer (sub)type. It can also be an enumeration type.

Synthesis Issues

There are no synthesis restrictions in the Exemplar synthesis tools on using arrays. The synthesis tools support arrays of anything (within the language rules), multi-dimensional arrays, array types with enumeration index type. Negative indexes are also allowed.

Naming of array objects is straightforward. The synthesis tools append the index for each element after the array name. If the element type consists of multiple bits, the synthesis tools append the element indexes to the array name with its index.

It is important to understand that there is no Most Significant Bit (MSB) or Least Significant Bit (LSB) defined in an array type or array object. The semantics of what is interpreted as MSB or LSB is defined by the operations on the array. In the example of object `seven` above, the user probably meant the left most bit to be the MSB, and the right most bit the LSB. However, this is not defined by the language, just by the user.

Additions, subtractions, and multiplications have to be defined by the user. Most synthesis tool vendors define (arithmetic) operations on arrays in packages that are shipped with the product. Most of these packages assume that leftmost bit is the MSB and the rightmost bit is the LSB. As an example of this, the packages `exemplar` and `exemplar_1164` (see “The Exemplar Packages” on page 11) define arithmetic operators the `bit_vector` and the IEEE 1164 array equivalent `std_logic_vector` type. In these packages, the leftmost bit is assumed to be the MSB.

Record Types

Syntax and Semantics

A *record* type defines a collection of values, just like the array type.

All elements of an array must be of the same type. Elements of a record can be of different types:

```
type date is
      record
            day : integer range 1 to 31 ;
            month : month_name ;
            year : integer range 0 to 4000 ;
      end record ;
```

The element type `month_name` in this example could be an enumeration type with all names of the months as literals.

The elements of a record type can again be of any type, but cannot be an unconstrained array.

Consider the following object of type `date`:

```
constant my_birthday : date := (29, june, 1963) ;
```

Note – An aggregate is used here to initialize the constant. Aggregates are discussed in the section “IEEE 1076 Predefined Operators” on page 40.

Individual elements of a record object can be accessed with a selected name. A selected name consists of the object name, followed by a dot (.) and the element name:

> `my_birthday.year` selects the `year` field out of `my_birthday` and returns the integer value `1993`.

Synthesis Issues

The Exemplar synthesis tools impose no restrictions (apart from the language rules) on record types and record objects.

Naming of the individual bits that result after synthesizing a record object follow the selected naming rule of the language: Each bit in a record object get the record name followed by a dot, followed by the element name. If the element synthesizes into multiple bits, the index of the bits in each element are appended to that. As an example, the five bits that represent the `day` field in `my_birthday` will be named as follows:

```
my_birthday.day(0)     LSB in my_birthday.day
my_birthday.day(1)
my_birthday.day(2)
my_birthday.day(3)
my_birthday.day(4)     MSB in my_birthday.day
```

Subtypes

A *subtype* is a type with a constraint.

```
subtype <subtype_name> is <base_type> [<constraint>] ;
```

A subtype allows you to restrict the values that can be used for an object without actually declaring a new type. This speeds up the debugging cycle, since the simulator will do a run-time check on values being out of the declared range. Declaring a new type would cause type conflicts. Here is an example:

```
type big_integer is range 0 to 1000 ;
type small_integer is range 0 to 7 ;

signal intermediate : small_integer ;
signal final : big_integer ;

final <= intermediate * 5 ; <- type error occurs because
                         big_integer and small_integer are
                           NOT the same type
```

With a type-conversion (see next section), you can ’cast’ one integer into another one to avoid the above error. Still, it is cleaner to use a subtype declaration for the (more constrained) `small_integer` type:

```
type big_integer is range 0 to 1000 ;
subtype small_integer is big_integer range 0 to 7 ;

signal intermediate : small_integer ;
signal final : big_integer ;

final <= intermediate * 5 ;<- NO type error occurs ! because
                           big_integer and small_integer
                           have the same base-type
                           (big_integer).
```

Subtypes can be used to constraint integer types (as in the above example), floating-point type, and unconstrained arrays.

Declaring a subtype that constraints an unconstrained array type is exactly the same as declaring a constrained array type:

```
type bit_vector is array (natural range <>) of bit ;
subtype eight_bit_vector is bit_vector (0 to 7) ;
```

has the same effect as:

```
type eight_bit_vector is array (0 to 7) of bit ;
```

Just as in the integer type example above, subtypes of one and the same unconstrained base-type are compatible (will not cause type errors), but when two constrained array types are used, they will cause type errors if objects of both types are intermixed in expressions. Type conversion is then the only possibility to let objects of the two types be used together in expressions without type errors (see next section).

There are no synthesis restrictions on the use of subtypes.

Type Conversions

In cases where it is not possible to declare one type and one subtype instead of two separate types, VHDL has the concept of type conversion. *Type conversion* is similar to type 'casting' in high level programming languages. To cast an expression into a type, use the following syntax:

```
<type>(<expression>)
```

Type conversion is allowed between 'related' types. There is a long and detailed discussion in the VHDL LRM about what related types are, but in general, if it is obvious to you that the compiler should be able to figure out how to translate values of one type to values of another type, the types are probably related. For example, all integer types are related, all floating-point types are related and all array types of the same element type are related.

So, the problem of type error between two different types in example of the previous section could be solved with a type conversion:

```
type big_integer is range 0 to 1000 ;
type small_integer is range 0 to 7 ;

signal intermediate : small_integer ;
signal final : big_integer ;

final <= big_integer(intermediate * 5) ;<- NO type error occurs now,
                                         since the compiler knows how to
                                         translate 'small_integer' into
                                         big_integer with the type
                                         conversion.
```

IEEE 1076 Predefined Types

The VHDL IEEE 1076 standard predefines a number of types. Listed below are the ones which are most important for synthesis:

```
type bit is ('0','1') ;
type bit_vector is array (integer range <>) of bit ;
type integer is range MININT to MAXINT ;
subtype positive is integer range 1 to MAXINT ;
subtype natural is integer range 0 to MAXINT ;
type boolean is (TRUE,FALSE) ;
```

The Exemplar synthesis tools also understand the predefined types `CHARACTER`, `STRING`, `SEVERITY_LEVEL`, `TIME`, `REAL` and `FILE`. For more information on synthesis restrictions for these object types, see “Syntax and Semantic Restrictions” on page 22.

IEEE 1164 Predefined Types

A problem with the 1076 standard is that it does not specify any multi-valued logic types for simulation purposes, but rather left this to the user and/or tool vendor. The IEEE 1164 Standard specifies a 9-valued logic. The Exemplar synthesis tools support these types, although some restrictions apply to the values you can use for synthesis. These restrictions are discussed in the section “Syntax and Semantic Restrictions” on page 22.

The meaning of the different type values of the IEEE 1164 standard are given below.

`'U'`	Uninitialized
`'X'`	Forcing Unknown
`'0'`	Forcing Low
`'1'`	Forcing High
`'Z'`	High Impedance
`'W'`	Weak Unknown
`'L'`	Weak Low

`'H'`	Weak High
`'-'`	Dont Care

The weak values on a node can always be overwritten by a forcing value. The high impedance state can be overwritten by all other values.

Most of these values are meaningful for simulation purposes only. Some restrictions apply if you want to use these values for synthesis. Only the values `'0','1','X','-'` and `'Z'` have a well-described meaning for synthesis. For details see “Syntax and Semantic Restrictions” on page 22.

Some examples of IEEE 1164 type statements are:

```
type std_ulogic is ('U','X','0','1','Z','W','L','H','-') ;
type std_ulogic_vector is array (natural range <>) of std_ulogic ;
subtype std_logic is resolution_func std_ulogic ;
type std_logic_vector is (natural range <>) of std_logic ;
subtype X01Z is resolution_func std_ulogic range 'X' to 'Z' ;
                       -- includes X,0,1,Z
```

The identifier `resolution_func` is a function that defines which value should be generated in case multiple values are assigned to an object of the same type. This is called the resolution function of the type. Resolution functions are supported as long as they do not return any metalogical values. For details, refer to “Syntax and Semantic Restrictions” on page 22.

To use the IEEE 1164 types you must load the IEEE package into your VHDL description. This is done with the following statements:

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

Details about how the synthesis tools handle packages are explained in the section “Entity and Package Handling” on page 1.

Objects

Objects in VHDL (signals, variables, constants, ports, loop variables, generics) can contain values. Values can be assigned to objects, and these values can be used elsewhere in the description by using the object in an expression. All objects except loop variables have to be declared before they are used. This section describes the various objects in VHDL and their semantics.

Signals

Signals represent wires in a logic circuit. Here are a few examples of signal declarations:

```
signal foo : bit_vector (0 to 5) := B"000000" ;
signal aux : bit ;
signal max_value : integer ;
```

Signals can be declared in all declarative regions in VHDL except for functions and procedures. The declaration assigns a name to the signal (`foo`); a type, with or without a range restriction (`bit_vector(0 to 5)`); and optionally an initial (constant) value. Initial values on signals are usually ignored by synthesis (For details, see “Restrictions on Initialization Values” on page 24.)

Signals can be assigned values using an assignment statement (e.g., `aux <= ’0’ ;`). If the signal is of an array type, elements of the signal’s array can be accessed and assigned using indexing or slicing. For more information, see “Statements” on page 33.

Assignments to signals are not immediate, but scheduled to be executed after a delta delay. (This effect is an essential difference between variables and signals.) This is discussed in detail in “Usage Of Attributes” on page 46.

Constants

Constants can not be assigned a value after their declaration. Their only value is the initial constant value. Initialization of a constant is required. An example of declaring a constant is:

```
constant ZEE_8 : std_logic_vector (0 to 7) := "ZZZZZZZZ" ;
```

Variables

Variables can not be declared or used in the dataflow areas or in packages, only in processes, functions and procedures.

An example of declaring a variable is:

```
variable temp : integer range 0 to 10 := 5 ;
```

Assignments to a variable are immediate. This effect is an essential difference between variables and signals. This is discussed in detail in “Usage Of Attributes” on page 46.

The initial assignment to a variable is optional. The initial assignment to a variable in a process is usually ignored by synthesis. (For more information, see “Restrictions on Initialization Values” on page 24.)

Ports

A *port* is an interface terminal of an entity. A port represents an ordinary port in a netlist description. Ports in VHDL are, just like other objects, typed and can have an initial value. In addition, a port has a “direction.” This is a property that indicates the possible information flow through the port. Possible directions are `in`, `out`, `inout` and `buffer`, where `inout` and `buffer` indicate bidirectional functionality.

```
entity adder is
        port (
                input_vector : in bit_vector (0 to 7) ;
                output_vector : out bit_vector (0 to 7)
        ) ;
end adder ;
```

After declaration, a port can be used in the architecture of the entity as if it were a normal signal, with the following restrictions: first, you cannot assign to a port with direction in, and second, you cannot use a port of direction out in an expression.

Generics

A *generic* is a property of an entity. A good example of a generic is the definition of the size of the interface of the entity. Generics are declared in a generic list.

```
entity increment is
       generic ( size : integer := 8 ) ;
       port (  ivec  : in  bit_vector (0 to size-1) ;
              ovec : out bit_vector (0 to size-1)) ;
end increment ;
```

The generic `size` can be used inside the entity (e.g., to define the size of ports) and in the architecture that matches the entity. In this example, the generic `size` is defined as an integer with an initial value `8`. The sizes of the input and output ports of the entity increment are set to be 8 bits unless the value of the generic is overwritten by a generic map statement in the component instantiation of the entity.

```
inst_1 : increment   generic map (size=>16)
                    port map (ivec=>invec, ovec=>outvec) ;
```

Here, a 16-bit incrementer is instantiated, and connected to the signals `invec` and `outvec`. “Component Instantiation” on page 58 explains more about how to use generics when instantiating components.

The Exemplar synthesis tools fully support generics and generic map constructs and imposes no restriction on the type of the generic. Generics are very useful in generalizing your VHDL description for essential properties like sizes of interfaces or for passing timing information for simulation to instantiated components.

Loop Variables

A *loop variable* is a special object in the sense that it does not have to be declared. The loop variable gets its type and value from the specified range in the iteration scheme.

```
for i in 0 to 5 loop
       a(i) <= b(i) and ena ;
end loop ;
```

In this code fragment, `i` becomes an integer with values 0,1,2...5 respectively, when the loop statements are executed 6 times. A loop variable can only be used inside the loop, and there can be no assignments to the loop variable. For synthesis, the range specified for the loop variable must be a compile-time constant, otherwise the construct is not synthesizable.

Statements

This section briefly discusses the basic statements that can be used in VHDL descriptions.

Conditional Statements

```
signal a : integer ;
signal output_signal, x, y, z : bit_vector (0 to 3) ;
....
if a = 1 then
       output_signal <= x ;
elsif a = 2 then
       output_signal <= y ;
elsif a = 3 then
       output_signal <= z ;
else
       output_signal <= "0000" ;
end if ;
```

This code fragment describes a multiplexer function, implemented with an if-then-else statement. This statement can only be used in a sequential environment, such as a process, procedure or a function.

The same functionality in the dataflow environment is accomplished with the use of the conditional signal assignment statement:

```
signal a : integer ;
signal output_signal, x, y, z : bit_vector (0 to 3) ;
....
output_signal <= x when a=1 else
y when a=2 else
z when a=3 else
"0000" ;
```

Selection Statements

If many conditional clauses have to be performed on the same selection signal, a case statement is a better solution than the `if-then-else` construct:

```
signal output_signal, sel, x, y, z : bit_vector (0 to 3) ;
....
case sel is
       when "0010" => output_signal <= x ;
       when "0100" => output_signal <= y ;
       when "1000" => output_signal <= z ;
       when "1010" | ″1100" | "0110" => output_signal <= x and y and z ;
       when others => output_signal <= "0000" ;
end case ;
```

The “|” sign indicates that particular case has to be entered if any of the given choices is true (or functionality). Each case can contain a sequence of statements.

The case statement can only be used in a sequential environment. In the dataflow environment, the selected signal assignment statement has the equivalent behavior:

```
signal output_signal, sel, x, y, z : bit_vector (0 to 3) ;
....
with sel select
      output_signal <= x when "0010",
                       y when "0100",
                       z when "1000",
                       x and y and z when "1010" | "1100"
                       |"0110", "0000" when others ;
```

Loop Statements and Generate Statements

In many cases, especially with operations on arrays, many statements look alike, but differ only on minor points. In that case, you might consider using a loop statement.

```
signal result, input_signal : bit_vector (0 to 5) ;
signal ena : bit ;
....
for i in 0 to 5 loop
      result(i) <= ena and input_signal(i) ;
end loop ;
```

In this code fragment, each bit of a input signal is “anded” with a single bit enable signal, to produce an output array signal. The loop variable `i` does not have to be declared. It holds an integer value since the loop range is an integer range.

The previous example showed a `for` loop. VHDL also has a `while` loop. Here is an example:

```
variable i : integer ;
 ......
i := 0 ;
while (i < 6) loop
   result(i) <= ena AND input_signal(i) ;
   i := i + 1 ;
end loop ;
```

The Exemplar synthesis tools can synthesize any `for` loop. A `while` loop, however, can be synthesized only if the `while` condition evaluates to a constant (as in the example above). If the `while` condition does not evaluate to a run-time constant, then the synthesis tools do not know how many times the loop should be executed, and thus cannot define how must hardware to generate for the statements inside the while loop. A `while` loop with a non-constant condition could be synthesized if there were a `wait` statement inside the loop. However, this implies multiple `wait` statements in a process, which is not supported by the synthesis tools.

Both a for-loop and a while-loop support `EXIT` or `NEXT` statements. An `EXIT` statement tells the synthesis tools to leave the loop, and a `NEXT` statement tells it to go to the next iteration.

For example, we could write the above `while` loop as follows:

```
i := -1 ;
while (TRUE) loop
   i := i + 1 ;

   exit if (i > 5) ;
   if (input_signal(i) = ’0’) then
      result(i) <= ’0’ ;
      next ;
   end if ;
      result(i) <= ena ;
end loop ;
```

This example is just to indicate how the `EXIT` and `NEXT` statements work. We do not want to advise you to use the exit and next statement like this. The synthesis tools however, do synthesize this description into the same logic as the original `for` or `while` loop. The synthesis tools are extremely good in analyzing constant expressions, and that is why this example works.

The loop statement can only be used inside sequential environments. Its equivalent statement in the dataflow environment is the `generate` statement:

```
signal result, input_signal : bit_vector (0 to 5) ;
signal ena : bit ;
....
G1 : for i in 0 to 5 generate
       result(i) <= ena and input_signal(i) ;
end generate ;
```

Note – The `generate` statement is preceded by a label (`G1`). A label is required in the generate statement but is optional in the loop statement.

The `generate` statement does not allow `EXIT` and `NEXT` statements. The reason is that the statements inside the `generate` statement are executed concurrently. So there is no way to know when to exit. The `generate` statement has no `while` equivalent, for the same reason. Instead however, there is a `if` equivalent in the `generate` statement:

```
i := -1 ;
while (TRUE) loop
   i := i + 1 ;

   exit if (i > 5) ;
   if (input_signal(i) = '0') then
      result(i) <= '0' ;
      next ;
   end if ;
      result(i) <= ena ;
end loop ;
```

The condition must evaluate to a run-time constant. That is a language requirement.

Note – There is no `else` part possible in a `generate` statement. We consider this a flaw in the language, but the Exemplar synthesis tools has to comply with it.

The synthesis tools have no synthesis restrictions for the `generate` statement.

Assignment Statements

Assignments can be done to signals, ports and variables in VHDL. Assignments to signals and ports are done with the <= operator.

```
signal o, a, b : std_logic_vector (0 to 5) ;
....
o <= a xor b ;
```

In this code fragment o gets assigned the value of the vector-XOR (bit by bit) of vectors a and b. The type of the object on the left hand side of the assignment should always match the type of the value on the right hand side of the assignment. Signal assignments can be used both in dataflow environment and sequential environments.

Assignments to variables are done with the “:=” sign.

```
variable o : std_logic_vector (0 to 5) ;
signal a, b : std_logic_vector (0 to 5) ;
....
o := a AND NOT b ;
```

Variable assignments can only be used in sequential environments. Types on left and right hand side of the “:=” sign should match.

There is one important difference between assignments to signals and assignments to variables: when the values are updated. The value of a variable in a variable assignment is updated immediately after the assignment. The value of a signal in a signal assignment is not updated immediately, but gets “scheduled” until after a delta (delay) time. This delay time is not related to actual time, but is merely a simulation characteristic. This behavior of the signal assignment does not have any effect for signal assignments in a dataflow environment, since assignments are done concurrently there. However, in a process, the actual value of the signal changes only after the complete execution of the process.

The following example illustrates this effect. It shows the description of a multiplexer that can select one bit out of a four bit vector using two select signals.

```
entity mux is
        port ( s1, s2 : in bit ;
                inputs : in bit_vector (0 to 3) ;
                result : out bit
            ) ;
end mux ;

architecture wrong of mux is
begin
        process (s1,s2,inp)
        signal muxval : integer range 0 to 3 ;
        begin
                muxval <= 0 ;
                if (s1 = '1') then muxval <= muxval+1 ;
                if (s2 = '1') then muxval <= muxval+2 ;
                -- use muxval as index of array 'inputs'
                result <= inputs (muxval) ;
        end process ;
end wrong ;
```

This description does not behave as intended. The problem is because `muxval` is a signal; the value of `muxval` is not immediately set to the value defined by bits `a` and `b`. Instead, `muxval` still has the same value it had when the process started when the `if` statement is executed. All assignments to `muxval` are scheduled until after the process finishes. This means that `muxval` still has the value it got from the last time the process was executed, and that value is used to select the bit from the input vector.

The solution to this problem is to make `muxval` a variable. In that case, all assignments done to `muxval` are immediate, and the process works as intended.

```
entity mux is
      port (  s1, s2 : in bit ;
            inputs : in bit_vector (0 to 3) ;
            result : out bit) ;
end mux ;

architecture right of mux is
begin
      process (s1,s2,inp)
      variable muxval : integer range 0 to 3 ;
      begin
            muxval := 0 ;
            if (s1 = '1') then muxval := muxval+1 ;
            if (s2 = '1') then muxval := muxval+2 ;
            -- Use muxval as index of array 'inputs'
            result <= inputs (muxval) ;
      end process ;
end right ;
```

As a general rule, if you use signal assignments in processes, do not use the value of the signal after the assignment, unless you explicitly need the previous value of the signal. Alternatively, you can use a variable instead.

Operators

IEEE 1076 Predefined Operators

VHDL predefines a large number of operators for operations on objects of various types. The following is an overview:

Relational operators on ALL types (predefined or not):

=	<=
/=	>
<	>=

Logical operators on pre-defined types BIT and BOOLEAN:

AND	NOR
OR	XOR
NAND	NOT

Arithmetic operators on all integer types:

+	mod
-	rem
*	abs
/	
**	

Concatenation of elements into an array of elements:

&	(,,,,)

Relational operators operate on any type. The basis of comparing two values is derived from the order of definition. For example in the `std_logic` type the value `’U’` is smaller than the value `’1’` because `’U’` is defined first in the order of values in the type. The comparison of two arrays is accomplished by comparing each element of the array. The left most element is the most significant one for comparisons.

```
signal a : bit_vector (7 downto 0) ;
signal b : bit_vector (5 to 9) ;
```

In this example, `a(7)` is the most significant bit for comparisons with vector `a`, and `b(5)` is the most significant bit for comparisons with vector `b`.

Logical operators work in a straightforward manner and do the appropriate operations on types `BIT` and `BOOLEAN`, and also for one-dimensional arrays of `BIT` and `BOOLEAN`. In the latter case, the logical operation is executed on each element of the array. The result is a array with the same size and type as the operands.

Arithmetic operators work on integers and on all types derived from integers. The Exemplar synthesis tools support arithmetic operators on vectors, described in the exemplar package. “The Exemplar Packages” on page 11 presents more details about operations on vectors.

Concatenation operators can group elements of the same type into an array of that type. Consider the following examples:

```
signal a, b, c : bit ;
signal x : bit_vector (5 downto 0) ;
signal y : bit_vector (0 to 3) ;
....
-- using concatenation operator
          x <= a & b & c & B"00" & ’0’ ;
-- using an aggregate
          y <= (’1’, ’0’, b, c) ;
```

This description is the same as the following one:

```
signal a, b, c : bit ;
signal x : bit_vector (5 downto 0) ;
signal y : bit_vector (0 to 3) ;
....
          x(5) <= a ;
          x(4) <= b ;
          x(3) <= c ;
          x(2 downto 0) <= "000" ;
          y(0) <= ’1’ ;
          y(1) <= ’0’ ;
          y(2) <= b ;
          y(3) <= c ;
```

The aggregate operator in VHDL is especially useful when assigning to a vector of unknown or large size:

```
signal o : bit_vector (0 to 255) ;
....
       o <= (0=>’1’,others=>’0’) ;
```

In this example, `o(0)` is assigned `’1’` and all other elements of `o` (independent of its size) get value `’0’`.

IEEE 1164 Predefined Operators

The IEEE 1164 standard logic package describes a set of new types for logic values. However, the binary operators that are predefined in VHDL only operate on bit and boolean types, and arrays of bits and booleans. Therefore, the IEEE standard logic type package redefines the logical operators (and, or, not, etc.) for the types `std_logic`, `std_ulogic` and the array types `std_logic_vector` and `std_ulogic_vector`.

Operator Overloading

The operators +, -, *, mod, abs, < ,>, etc. are predefined for integer and floating-point types, and the operators and, or, not etc. are predefined on the type `bit` and `boolean`. If you want to use an operator that is not pre-defined for the types you want to use, use operator overloading in VHDL to define what the operator should do. Suppose you want to add an integer and a bit according to your own semantics, and you want to use the “+” operator:

```
function “+” (a: integer; b: bit) return integer is
begin
   if (b=’1’) then
      return a+1 ;
   else
      return a ;
   end if ;
end “+” ;
signal  o, t: integer range 0 to 255 ;
signal b : bit ;
...
t <= o + 5 + b ;
```

The first “+ ” in the assignment to t is the pre-defined “+” operator on integers. The second “+” is the user defined overloaded operator that adds a bit to an integer. The **“** character around the “+” operator definition is needed to distinguish the operator definition from a regular function definition (see “Resolution Functions” on page 52).

Operator overloading is also necessary if you defined your own logic type and would like to use any operator on it.

If you want to do arithmetic operations (+, -, etc.) on the array types `bit_vector` or `std_logic_vector`, it will be more efficient for synthesis to use the pre-defined operators from the `exemplar` and the `exemplar_1164` packages. For details of these packages operations and their use, see “The Exemplar Packages” on page 11.

The Exemplar synthesis tools fully support operator overloading as described by the language.

Attributes

In VHDL, attributes can be set on a variety of objects, such as signals and variables, and many other identifiers, like types, functions, labels etc.

An *attribute* indicates a specific property of the signal, and is of a defined type. Using attributes at the right places creates a very flexible style of writing VHDL code. An example of this is given at the end of this section.

VHDL Predefined Attributes

VHDL pre-defines a large set of attributes for signals. The following example shows the definition of two vectors and the values of the VHDL predefined attributes for them.

```
signal vector_up : bit_vector (4 to 9) ;
signal vector_dwn : bit_vector (25 downto 0) ;
....
vector_up’LEFT-- returns integer 4
vector_dwn’LEFT-- returns integer 25
vector_up’RIGHT-- returns integer 9
vector_dwn’RIGHT-- returns integer 0
vector_up’HIGH-- returns integer 9
vector_dwn’HIGH-- returns integer 25
vector_up’LOW-- returns integer 4
vector_dwn’LOW-- returns integer 0
vector_up’LENGTH-- returns integer 6
vector_dwn’LENGTH-- returns integer 26
vector_up’RANGE -- returns range 4 to 9
vector_dwn’RANGE-- returns range 25 to 0
vector_up’REVERSE_RANGE-- returns range 9 to 4
vector_dwn’REVERSE_RANGE-- returns range 0 to 25
```

The attributes do not have to be written in capitals; VHDL is case-insensitive for identifiers.

An important predefined attribute for synthesis is the `EVENT` attribute. Its value reveals edges of signals. For more information about the `EVENT` attribute, see “Edge-Sensitive Flip-Flops” on page 3**.**

Exemplar Predefined Attributes

Apart from the VHDL predefined types, Exemplar also supplies a set of predefined attributes that are specifically helpful for guiding the synthesis process or controlling down-stream tools. For details of these attributes, see “Predefined Attributes” on page 12.

User-Defined Attributes

Attributes can also be user defined. In this case, the attribute first has to be declared, with a type, and then its value can be set on a signal or other object. This value can then be used with the “ ’ ” construct. The following is an example:

```
signal my_vector : bit_vector (0 to 4) ;
attribute MIDDLE : integer ;
attribute MIDDLE of my_vector : signal is  my_vector’LENGTH/2 ;
....
      my_vector’MIDDLE         -- returns integer 2
```

Usage Of Attributes

To indicate where attributes in a VHDL description are useful, consider the following example.

```
entity masked_parity is
      port ( source : in bit_vector (0 to 5) ;
            mask : in bit_vector (0 to 5) ;
            result : out bit
      ) ;
end masked_parity ;

architecture soso of masked_parity is
begin
      process (source, mask)
            variable tmp : bit ;
            variable masked_source : bit_vector (0 to 5) ;
      begin
            masked_source := source and mask ;
            tmp := masked_source(0) ;
            for i in 1 to 5 loop
               tmp := tmp XOR masked_source(i) ;
            end loop ;
            result <= tmp ;
      end process ;
end soso ;
```

This example calculates the parity of the bits of a source vector, where each bit can be masked. This VHDL description is correct, but is not very flexible. Suppose the application changes slightly and requires a different size input. Then the VHDL description has to be modified significantly, since the range of the vector affects many places in the description. The information is not concentrated, and there are many dependencies. Attributes can resolve these dependencies.

Here is an improved version of the same example, where attributes `LEFT`, `RIGHT`, and `RANGE` define the dependencies on the size of the vector.

```
entity masked_parity is
      generic ( size : integer := 5) ;
      port ( source : in bit_vector (0 to size) ;
             mask : in bit_vector (source'RANGE) ;
             result : out bit
          ) ;
end masked_parity ;

architecture better of masked_parity is
begin
      process (source, mask)
             variable tmp : bit ;
             variable masked_source : bit_vector (source'RANGE) ;
      begin
             masked_source := source and mask ;
             tmp := masked_source(source'LEFT) ;
             for i in source'LEFT+1 to source'RIGHT loop
                tmp := tmp xor masked_source(i) ;
             end loop ;
             result <= tmp ;
      end process ;
end better ;
```

If the application requires a different size parity checker, this time we only have to modify the source vector range, and the attributes ensure that the rest of the description gets adjusted accordingly. Now the information is concentrated.

Blocks

When using processes and dataflow statements it is possible to use VHDL as a high level hardware description language. However, as the descriptions get more and more complicated, some form of design partitioning, or hierarchy, is required or desirable.

VHDL offers a variety of methods for design partitioning. One form of partitioning is to divide a description into various processes. In the following sections four more forms of partitioning are discussed: blocks, subprograms (functions and procedures), components and packages.

A *block* is a method to cluster a set of related dataflow statements. Signals, subprograms, attributes, etc. that are local to the block can be defined in a block declarative region. All statements in a block are executed concurrently, and thus define a dataflow environment.

```
architecture xxx of yyy is
        signal global_sig ,g1,g2,c bit ;
begin
        B1 : block                          -- block declarative region
                signal local_sig : bit ;
        begin                               -- block concurrent statements
                local_sig <= global_sig ;
                                            -- Block in a block
                B2 : block  (c='1')         -- Block has "GUARD" expression
                    port (o1,o2 : out bit)  -- Block port declarations
                    port map (o1=>g1,o2=>g2) ;
                begin
                    o1 <= guarded local_sig ;
                    o2 <= global_sig ;
                end block ;
        end block ;
end xxx ;
```

Blocks can be nested, as in the example above.

Signals, ports and generics declared outside the block can be used inside the block, either directly (as `global_sig` is used in block `B2`), or via a port map (as `g1` is connected to `o1` in block `B2`) or generic maps (for generics). There is no real difference between the two methods, except that the port (generic) map construct is a cleaner coding style which could reduce errors when using or assigning to global objects.

A block can also have a GUARD expression (c='1' in block B2). In that case, an assignment inside the block that contains the keyword GUARDED will only be executed when the GUARD expression is TRUE. In the example above, o1 only gets the value of local_sig when c='1'. GUARDED blocks and assignments provide a interesting alternative to construct latches or flip-flops in the synthesized circuit. For examples, refer to “Registers, Latches and Resets” on page 1.

The Exemplar synthesis tools fully support blocks, with port/generic lists and port/generic maps and the GUARD options of blocks.

Functions And Procedures

Subprograms (function and procedures) are powerful tools to implement functionality that is repeatedly used. *Functions* take a number of arguments that are all inputs to the function, and return a single value. *Procedures* take a number of arguments that can be inputs, outputs or inouts, depending on the direction of the flow of information through the argument. All statements in functions and procedures are executed sequentially, as in a process. Also, variables that are local to the subprogram can be declared in the subprogram. Local signals are not allowed.

As an example, suppose you would like to add two vectors. In this case, you could define a function that performs the addition. The following code fragment shows how an addition of two 6-bit vectors is done.

```
function vector_adder (x : bit_vector(0 to 5);  y : bit_vector(0 to 5))
return bit_vector(0 to 5) is
                        -- declarative region
       variable carry : bit ;
       variable result : bit_vector(0 to 5) ;
begin
                        -- sequential statements
       carry := '0' ;
       for i in 0 to 5 loop
              result (i) := x(i) xor y(i) xor carry ;
              carry := carry AND (x(i) OR y(i))  OR  x(i) AND y(i) ;
       end loop ;
       return result ;
end vector_adder ;
```

Note – That vector addition, implemented this way, is not very efficient for synthesis. The packages `exemplar` and `exemplar_1164` provide vector additions that can implement efficient/fast adders more easily. For more information, see "The Exemplar Packages" on page 11.

An example of a procedure is shown below. The procedure increments a vector only if an enable signal is high.

```
procedure increment (vect : inout bit_vector(0 to 5); ena : in bit :='1')
is
begin
      if (ena='1') then
            vect := vector_adder (vect, "000001") ;
      end if ;
end increment ;
```

This incrementer procedure shows the behavior of an inout port. The parameter `vect` is both set and used in this procedure. Also, the procedure statements use a call to the previously defined `vector_adder` function. If an input of a function or a procedure is not connected when it is used, that input will get the initial value as declared on the interface list.

For example, input `ena` will get (initial) value `'1'` if it is not connected in a procedure call to the procedure `increment`. It is an error if an input is not connected and also does not have an initial value specified.

One important feature of subprograms in VHDL is that the arguments can be unbound. The given examples operate on vectors of 6 bits. If you want to use the subprograms for arbitrary length vectors, you could specify the length-dependencies with attributes and not specify a range on the parameters (leave them unbound). Here is a redefinition of both the vector addition function and the incrementer procedure for arbitrary length vectors.

```
function vector_adder (x : bit_vector; y : bit_vector) return bit_vector
is
        variable carry : bit := '0' ;
        variable result : bit_vector(x'RANGE) ;
begin
        for i in x'RANGE loop
                result (i) := x(i) XOR y(i) XOR carry ;
                carry := carry AND (x(i) OR y(i))  OR  x(i) AND y(i) ;
        end loop ;
        return result ;
end vector_adder ;

procedure increment (vect : inout bit_vector; ena : in bit :='1') is
begin
        if (ena='1') then
                vect := vector_adder (x=>vect, "000001" ) ;
        end if ;
end increment ;
```

In the procedure increment example, name association was added in the parameter list of the `vector_adder` call. The name association (e.g., `x=>vect`) is an alternative way to connect a formal parameter (`x`) to its actual parameter (`vect`). Name associations (as well as positional associations) are helpful if the number of parameters is large.

Subprograms can be called from the dataflow environment and from any sequential environment (processes and other sub-programs). If a procedure output or inout is a signal, the corresponding parameter of the procedure should also be declared as a signal.

Subprograms can be overloaded. That is, there could be multiple subprograms with the same name, but with different parameter list types or return types. The synthesis tools perform the overlaod resolution.

In the last example, the variable carry was initialized in when it was declared. This is a more compact way of setting the starting value of a variable in a function or procedure. The initial value does not have to be a constant. It could be a nonconstant value also (like the value of one of the parameters).

The Exemplar synthesis tools fully support all VHDL language features of functions and procedures.

Resolution Functions

Syntax and Semantics

In a concurrent area in VHDL (see the section “Entities and Architectures” on page 1), all statements happen concurrently. That means that if there are two assignments to one and the same signal, that the final value of the signal needs to be resolved. In VHDL, you can only have multiple concurrent assignments to a signal if the type of the signal is `resolved`. A *resolved* type is a type with a resolution function. A good example of a resolved type is the type `std_logic` from the IEEE 1164 package:

```
subtype std_logic is resolved std_ulogic ;
```

The word `resolved` in this declaration refers to a resolution function called `resolved`. Here is how it is specified in the `std_logic_1164` package:

```
function resolved ( s : std_ulogic_vector ) return std_ulogic is
  variable result : std_ulogic := ’Z’;  -- weakest state default
  attribute synthesis_return of result:variable is “WIRED_THREE_STATE” ;
  begin
    -- the test for a single driver is essential otherwise the
    -- loop would return ’X’ for a single driver of ’-’ and that
    -- would conflict with the value of a single driver unresolved
    -- signal.
    if    (s’LENGTH = 1) then    return s(s’LOW);
    else
      for i in s’range loop  result := resolution_table(result, s(i));
      end loop;
    end if   return result;
  end resolved;
```

The resolution function of type `std_logic` takes a vector of the (unresolved) base-type of `std_logic` : `std_ulogic`. It returns a single `std_ulogic`.

Now if you have two concurrent assignments to any signal of type `std_logic`, the resolution function will be called to determine the final value of the signal. The resolution function will be called with a vector with two elements, where each element contains the value of a concurrent assignment. Inside the resolution function, the final value of the signal is defined, based on the two assignment values.

Synthesis Issues

Resolution functions are especially useful when you want to model nets with multiple drivers (like busses with three-state drivers). However, VHDL lets you define a resolution function freely, without any special restrictions. The resolution function is thus just another function, only it gets called wherever there are multiple assignments to a signal of the (sub)type it is attached to.

The Exemplar synthesis tools synthesize resolution functions without restriction.

You can define a resolution function and attach it to a subtype, and the synthesis tools will synthesize the circuitry it implies for each multiple assignment.

In many cases, the resolution function mimics a certain electrical behavior for the simulator. In the case of the IEEE type `std_logic`, and its resolution function `resolved` (described above), the resolution function resembles tri-states being wired together. Therefore, the synthesis directive attribute (`synthesis_result`) is set to `WIRED_THREE_STATE`. This synthesis directive is a hint to the synthesis tools to interpret the elements of the incoming vector as parallel three-state assignments, where the three-state condition is derived from the assignment. That way, any three-state drivers can be created with multiple assignments (For more information, see “Three-state Buffers” on page 14).

Let’s go through one example step by step, to show what the resolution function is doing:

```
entity test_resolver is
   port (a, b : bit ;
      o : out bit ) ;
end test_resolver ;
architecture exemplar of test_resolver is
   signal tmp : bit ;
begin
   tmp <= a ;
   tmp <= b ;
   o <= tmp ;
end exemplar ;
```

When the above example is executed, the synthesis tools will give the following error:

```
file, line 9: Error, multiple sources on unresolved signal TMP; also line 10.
```

This message is obvious, since you did not explain what should happen when `a` and `b` force (different) values concurrently onto signal `TMP`. For that, write a resolution function. Suppose you want the concurrent assignments to be `AND`ed. Then you should write a resolution function that performs an `AND` operation of the elements of its input vector.

Also attach the resolution function to `TMP`. You could do that in two ways:

1. Create a subtype of `bit`, say, `rbit`, and attach the resolution function to that subtype, just as we did for the type `std_logic`.
2. Directly attach the resolution function to the signal `TMP`. This is the easiest way, and it is useful if there are not many signals that need the resolution function.

The second method is used below:

```
entity test_resolver is
   port (a, b : bit ;
      o : out bit ) ;
end test_resolver ;

architecture exemplar of test_resolver is
   -- Write the resolution function that ANDs the elements:
   function my_and_resolved (a : bit_vector) return bit is
      variable result : bit := ’1’ ;
   begin
      for i in a’range loop
         result := result AND a(i) ;
      end loop ;
      return result ;
   end my_and_resolved ;

   -- Declare the signal and attach the resolution function to it:
   signal tmp : my_and_resolved bit ;
begin
   tmp <= a ;
   tmp <= b ;
   o <= tmp ;
end exemplar ;
```

The synthesis tools will synthesize this description and `tmp` becomes the `AND` of `a` and `b`.

BUS and REGISTER

In the previous section, multiple concurrent assignments were discussed. Each concurrent assignment to a signal in VHDL creates what is called a ‘driver’ to the signal, and the resolution function resolves the values of the (multiple) drivers on the signal.

Now it is possible to (temporarily) switch-off drivers to a signal. Lets investigate an example:

```
process (c,d)
begin
   if (c = '1') then
      o <= d ;
   else
      o <= NULL ;
   end if ;
end process ;
```

In this example, `o` gets a driver from (concurrent) process statement. However, if `c` is not `'1'`, the NULL value is assigned to `o`. The NULL value is called a 'disconnection statement'. In VHDL this means that the driver of `c` is switched off if `c` is not `'1'.` A VHDL simulator will NOT include the driver value as an element in the input vector of the resolution function as long as the driver is switched off.

Since drivers can be switched off, we have to consider the case that ALL drivers are switched off. For that particular reason, VHDL defines what is called an entity class for a signal. There are two entity classes: BUS and REGISTER.

If the entity class is BUS, and all drivers on the signal are switched off, then VHDL defines that the resolution function should still be called, but with a vector of zero elements (a NULL vector).

If the entity class is REGISTER, and all drivers on the signal are switched off, then VHDL defines that the signal should hold its previous value.

Signals of BUS or REGISTER entity class are called resolved signals. A resolved signal always needs a resolution function.

Here is the full example where o gets a BUS entity class:

```
-- include the IEEE 1164 package to use type std_logic.
library ieee ;
use ieee.std_logic_1164.all ;
-- An entity with a BUS entity-class signal
entity test_bus is
   port (c,d : std_logic ;
      o : out std_logic BUS) ;
end test_bus ;
architecture exemplar of test_bus is
begin
   process (c,d)
   begin
      if (c = '1') then
         o <= d ;
      else
         o <= NULL ;
      end if ;
   end process ;
end exemplar ;
```

In this example, o is of entity class BUS, and thus the resolution function of std_logic will be executed if all drivers on o are switched off. That means that o will get the 'Z' value. That means that the synthesis tools will synthesize a three-state driver for o.

If o would be declared with the REGISTER entity class, the synthesis tools would synthesize a LATCH for it, since o should retain its value if all drivers are off.

Switching off drivers can also be done with a GUARDED block, or with a disconnection statement in a concurrent signal assignment. The Exemplar synthesis tools support all these statements.

The synthesis tools synthesizes BUS and REGISTER entity classes according to the semantics described above with the following restrictions. The Exemplar synthesis tools guarantee ONLY behavior compliant with VHDL language for BUS and REGISTER signals if the resolution function contains the WIRED_THREE_STATE synthesis directive. Also, multiple concurrent assignments to REGISTER entity class signals is not supported right now.

Component Instantiation

Components are a method of introducing structure in a VHDL description. A component represents a structural module in the design. Using components, it is possible to describe a netlist in VHDL. Components are instantiated in the dataflow environment. Here is an example of a structural VHDL description where four one-bit rams and a counter module are instantiated.

```
entity scanner is
        port ( reset  : in bit ;
                stop   : in bit ;
                load   : in bit ;
                clk : in bit ;
                load_value : in bit_vector (0 to 3) ;
                data   : out bit_vector (0 to 3)
        ) ;
end scanner ;

architecture exemplar of scanner is

        component RAM_32x1
                port ( a0, a1, a2, a3, a4 : in bit ;
                       we, d : in bit ;
                       o : out bit
                ) ;
        end component ;

        component counter
                generic (size : integer := 4 ) ;
                port ( clk : in bit ;
                       enable : in bit ;
                       reset : in bit ;
                      result : out bit_vector (0 to 4)
                ) ;
        end component ;
        signal ena : bit ;
        signal addr : bit_vector (0 to 4) ;
```

```
begin
      for i in 0 to 3 generate
            ram : RAM_32x1 port map (a0=>addr(0), a1=>addr(1),
            a2=>addr(2), a3=>addr(3), a4=>addr(4), d=>data(i),
            we=>load, o=>data(i) ) ;
      end generate ;

      ena <= not stop ;
      count : counter    generic map (size=>addr'length)
                       port map(clk=>clk, enable=>ena,
                               reset=>reset, result=>addr) ;
end exemplar ;
```

The generate statement is used here to instantiate the four RAMs.

Components have to be declared before they can be used. This is done in the declaration area of the architecture, or in a package (see next section). The declaration defines the interface of the component ports with their type and their direction. Actually this example is just a netlist of components. We added one dataflow statement (the assignment to `ena`) to show that structure and behavior can be mixed in VHDL.

The ports of the component are connected to actual signals (or ports) with the port map construct. The generics of the component are connected to actual values with the generic map construct. In this example the generic `size` is set to 4 with the attribute length on the array `addr`. If no generic value was set to `size` (or if the generic map construct was completely absent), `size` gets value 4, as indicated by the initial value on `size` in the generic list of the component. It is an error if a generic (or input port) is not connected in a generic map (or port map) construct and there is no initial value given in the component generic (or port) list.

In the example above, the input ports of the component `RAM_32x1` are individual bits (`a0`, `a1`, `a2`, `a3`, `a4`). If the input would have been declared as a `bit_vector` (0 to 4), then the individual bits could be connected with indexed formal names:

```
.. port map (a(0) => addr(0), a(1) => addr(1), a(2) => addr(2),
             a(3) => addr(3), a(4) => addr(4), ...
```

or with a sliced formal name:

```
.. port map (a(0 to 4) => addr(0 to 4), .....
```

or simply with a full identifier association:

```
.. port map ( a => addr, .....
```

The Exemplar synthesis tools support any form of slicing or indexing of formal parameter names, as long as the VHDL language rules are obeyed (formal name should be static).

The synthesis tools also support type-transformation functions in port and generic associations as long as they are synthesizable. Type transformation functions are not very often used and so are not explained here.

The definition of the components counter and `RAM_32x1` are not yet given in the example. The process of giving a contents definition for a component is called *binding* in VHDL. With the Exemplar synthesis tools, there are four ways to do component binding:

1. Specify an entity with the same name as the component and an architecture for it. This way, the component gets bound to the entity with the same name. This is called ’default binding’ in VHDL.
2. Specify a configuration specification. Here you can bind a component to an entity with a different name, and you can even connect component ports to entity ports with a different name.
3. Use a source technology in the synthesis tools that contains a cell with the same name as the component. The synthesis tools will bind the component to the technology cell (and include functional, timing and area information for it).
4. Do not specify any entity for the component. This way, the synthesis tools will issue a warning and create a black-box for the component.

The component counter is a good example of the first option:

```
entity counter is
      generic (size : integer ) ;
      port ( clk : in bit ;
             enable : in bit ;
             reset  : in bit ;
             result : out bit_vector (0 to size-1)
          ) ;
end counter ;

architecture exemplar of counter is
begin
      process (clk,reset)
      begin
             if (reset='1') then
                result <= (others=>'0') ;
             elsif (clk'event and clk='1') then
                if (enable='1') then
                   result <= result + "1" ;
                end if ;
             end if ;
      end process ;
end exemplar ;
```

This description only includes behavior. There is no component instantiated, although it is possible, and it makes hierarchical design possible.

Note that in this case the overloaded '+' operator is used on vectors, as defined in the `exemplar` package. (See "The Exemplar Packages" on page 11 for details.) Also note that an asynchronous reset construction is used to reset the counter value. For details about various synthesizable forms of reset, see "Registers, Latches and Resets" on page 1.

The second option gives more freedom to bind an entity to a component. Suppose you have a counter entity that does exactly what you need, but it is named differently, and (or) has differently named ports and generics:

```
entity alternative is
   generic (N : integer) ;
   port (clock : in bit ;
         ena : bit :
         reset : bit ;
         output : out bit_vector (0 to N-1) ) ;
end alternative ;
architecture ex of alternative is
begin
   .....
end ex ;
```

In our example, the following configuration specification could be used to bind the component `counter` to the entity `alternative`, for a particular or all instances of the `counter` component. The configuration specification is added after the `counter` component declaration:

```
component counter
    generic (size : integer) ;
    port (clk : in bit ;
           enable : in bit ;
           reset : in bit ;
           result : out bit_vector(0 to 4)) ;
end counter ;
for all:counter use entity work.alternative(ex) generic map (N=>size)
                     port map (clock=>clk, ena=>enable,
                               reset=>reset,output=>result) ;
```

This configuration specification binds `all` instances of component `counter` to an entity called `alternative` (architecture `ex`) in the `work` library, and it connects the generics and ports of the entity to differently named generics and ports in the component. If the ports and generics have the same name in the entity and the architecture, the generic map and port map don't have to be given. If there is only one architecture of the entity `alternative` then the architecture (`ex`) does not have to be given either. If not all, but just one or two instances of the component `counter` should be bound to the entity `alternative`, then replace `all` by a list of instance (label) names.

Configuration specifications are a very powerful method to quickly switch definitions of components to a different alternative.

Core fully supports all forms of configuration specifications that are allowed in the language.

If no configuration specification is given, the synthesis tools use the default binding as explained in the first option.

For the third option, use a source technology in the synthesis tools that includes the component `RAM_32x1`. If the source technology is *lib_name*, the synthesis tools recognize the component in the *lib_name* library, and instantiates it in the design. In this case, `RAM_32x1` is a RAM cell, and the synthesis tools cannot really optimize that behavior (and `RAM32x1` shows up in the netlist as a hard macro). But if the macro contained combinational logic, the synthesis tools would include that logic in the optimization process, and map it to other target technology cells.

Note – Galileo and Leonardo use different techniques to indicate which source technology to use. Galileo uses the `-source=`*lib_name* switch. Leonardo requires that you load the source technology by using the `load_library` *lib_name* command before reading the design in the database.

The fourth option, omitting any entity for the component, is helpful when hierarchy has to be preserved. This technique can be effectively used in Galileo to maintain hierarchy. The synthesis tools generate an empty module for each component it cannot find in the present file as an entity or as a library cell in the source technology. Empty modules show up as blocks in the final netlist. They are not touched by the synthesis and optimization process. Components without a definition can also help to isolate a particular difficult or user-defined part of the design from the synthesis operations. Clock generators or other asynchronous circuits or time-critical user-defined modules are an example of this.

Packages

A *package* is a cluster of declarations and definitions of objects, functions, procedures, components, attributes etc. that can be used in a VHDL description. You cannot define an entity or architecture in a package, so a package by itself does not represent a circuit.

A package consists of two parts. The package header, with declarations, and the package body, with definitions. An example of a package is `std_logic_1164`, the IEEE 1164 logic types package. It defines types and operations on types for 9-valued logic.

To include functionality from a package into a VHDL description, the `use` clause is used.

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

entity xxx is
      port ( x : std_logic ; -- type std_logic is known since it is
                              -- defined in package
                              -- std_logic_1164
...
```

This example shows how the IEEE 1164 standard logic types and functions become accessible to the description in entity `xxx`.

This is the general form to include a package in a VHDL description:

```
library lib ;
use lib.package.selection ;
```

The `use` clause is preceded by a `library` clause. The predefined libraries `work` and `std` do not have to be declared in a `library` clause before they are used in a `use` clause. All other libraries do need to be declared.

The *selection* can consist of only one name of a object, component, type or subprogram that is present in the package, or the word all, in which case all functionality defined in the package is loaded into the synthesis tools, and can be used in the VHDL description.

Aliases

An *alias* is an alternate name for an existing object. By using an alias of an object, you actually use the object to which it refers. By assigning to an alias, you actually assign to the object to which the alias refers.

```
signal vec : std_logic_vector (4 downto 0) ;
alias mid_bit : std_logic is vec(2) ;
-- Assignment :
mid_bit <= ’0’ ;
-- is the same as
vec(2) <= ’0’ ;
```

Aliases are often useful in unbound function calls. For instance, if you want to make a function that takes the AND operation of the two left most bits of an arbitrary array parameter. If you want to make the function general enough to handle arbitrary sized arrays, this function could look like this:

```
function left_and (arr: std_logic_vector) return std_logic is
begin
     return arr(arr’left) and arr(arr’left-1) ;
end left_and ;
 -- Function does not work for ascending index ranges of arr.
```

This function will only work correctly if the index range of `arr` is descending (`downto`). Otherwise, `arr'left-1` is not a valid index number. VHDL does not have a simple attribute that will give the one-but-leftmost bit out of an arbitrary vector, so it will be difficult to make a function that works correctly both for ascending and descending index ranges. Instead, you could make an alias of `arr`, with a known index range, and operate on the alias:

```
function left_and (arr : std_logic_vector) return std_logic is
      alias aliased_arr : std_logic_vector (0 to arr'length-1) is arr ;

begin
            return aliased_arr(0) and aliased_arr(1) ;
      end left_and ;
      -- Function works for both ascending and descending index
      -- ranges of arr.
```

The Exemplar synthesis tools fully support aliases.

The Art Of VHDL Synthesis 3

This chapter explains the relationship between constructs in VHDL and the logic which is synthesized. It focuses on coding styles with the best performance for synthesis. Actual synthesis restrictions on VHDL are discussed in the section, Syntax and Semantic Restrictions.

Registers, Latches and Resets

VHDL synthesis produces registered and combinational logic at the RTL level. All combinational behavior around the registers is, unless prohibited by the user, optimized automatically. The style of coding combinational behavior, such as `if-then-else` and `case` statements, has some effect on the final circuit result, but the style of coding sequential behavior has significant impact on your design.

The purpose of this section is to show how sequential behavior is produced with VHDL, so that you understand why registers are generated at certain places and not in others.

Most examples explain the generation of these modules with short VHDL descriptions in a process. If you are not working in a process, but just in the dataflow area of an architecture in VHDL, it is possible to generate these modules using predefined procedures in the `exemplar.vhd` package. For details about this package, refer to the section The Exemplar Packages.

Level-Sensitive Latch

This first example describes a level-sensitive latch:

```
signal input_foo, output_foo, ena : bit ;
...
process (ena, input_foo)
begin
      if (ena = '1') then
            output_foo <= input_foo ;
      end if ;
end process ;
```

In this example, the sensitivity list is required, and indicates that the process is executed whenever the signals `ena` or `input_foo` change. Also, since the assignment to the global signal `output_foo` is hidden in a conditional clause, `output_foo` cannot change (will preserve its old value) if `ena` is `'0'`. If `ena` is `'1'`, `output_foo` is immediately updated with the value of `input_foo`, whenever it changes. This is the behavior of a level-sensitive latch.

In technologies where level-sensitive latches are not available, the Exemplar synthesis tools translate the initially generated latches to the gate-equivalent of the latch, using a combinational loop.

Latches can also be generated in dataflow statements, using a guarded block:

```
b1 : block (ena='1')
begin
      output_foo <= GUARDED input_foo ;
end block ;
```

Edge-Sensitive Flip-Flops

The Event Attribute

An edge triggered flip-flop is generated from a VHDL description only if a signal assignment is executed on the leading (or on the falling) edge of another signal. For that reason, the condition under which the assignment is done should include an edge-detecting mechanism. The EVENT attribute on a signal is the most commonly used edge-detecting mechanism.

The EVENT attribute operates on a signal and returns a boolean. The result is always FALSE, unless the signal showed a change (edge) in value.   If the signal started the process by a change in value, the EVENT attribute is TRUE all the way through the process.

Here is one example of the event attribute, used in the condition clause in a process. The synthesis tools recognize an edge triggered flip-flop from this behavior, with `output_foo` updated only on the leading edge of `clk`.

```
signal input_foo, output_foo, clk : bit ;
....
process (clk)
begin
      if (clk’event and clk=’1’) then
            output_foo <= input_foo ;
      end if ;
end process ;
```

The attribute STABLE is the boolean inversion of the EVENT attribute. Hence, CLK’EVENT is the same as NOT CLK’STABLE. The Exemplar synthesis tools support both attributes.

Flip-flops and registers can also be generated with dataflow statements (as opposed to from a process) using a GUARDED block.

```
b2 : block (clk’event and clk=’1’)
begin
      output_foo <= GUARDED input_foo ;
end block ;
```

By adding the GUARDED statement option, a flip-flop will be inserted in between input_foo and output_foo, since the output_foo expression of the block specifies a clock edge.

Synchronous Sets And Resets

All conditional assignments to signal output_foo inside the if clause translate into combinational logic in front of the D-input of the flip-flop. For instance, we could make a synchronous reset on the flip-flop by doing a conditional assignment to output_foo:

```
signal input_foo, output_foo, clk, reset : bit ;
...
process (clk)
begin
        if (clk'event and clk = '1') then
                if reset = '1' then
                        output_foo <= '0' ;
                else
                        output_foo <= input_foo ;
                end if ;
        end if ;
end process ;
```

Note – Signals reset and input_foo do not have to be on the sensitivity list (although it is allowed) since a change in their values does not result in any action inside the process.

Alternatively, dataflow statements could be used to specify a synchronous reset, using a GUARDED block and a conditional signal assignment.

```
b3 : block (clk'event and clk='1')
begin
    output_foo <= GUARDED '0' when reset='1' else input_foo ;
end block ;
```

Asynchronous Sets And Resets

If the reset signal should have immediate effect on the output, but the assignment to `output_foo` from `input_foo` should happen only on the leading clock edge, an asynchronous reset is required. Here is the process:

```
signal input_foo, output_foo, clk, reset : bit ;
...
process (clk,reset)
begin
      if (reset = '1') then
            output_foo <= '0' ;
      elsif (clk'event and clk = '1') then
            output_foo <= input_foo ;
      end if ;
end process ;
```

Now reset HAS TO BE on the sensitivity list! If it were not there, VHDL semantics require that the process should not start if reset changes. It would only start if `clk` changes. That means that if reset becomes `'1'`, `output_foo` would be set to `'0'` if `clk` either goes up, or goes down, but not before any change of `clk`. This behavior cannot be synthesized into logic. The synthesis tools issue an error message that reminds you to put reset on the sensitivity list.

Asynchronous set and reset can both be used. It is also possible to have expressions instead of the fixed `'0'` or `'1'` in the assignments to `output_foo` in the reset and set conditions. This results in combinational logic driving the set and reset input of the flip-flop of the target signal. The following code fragment shows the structure of such a process:

```
process (clock, asynchronously_used_signals )
begin
        if (boolean_expression) then
                asynchronous signal_assignments
        elsif (boolean_expression) then
                asynchronous signal_assignments
        elsif (clock'event and clock = constant ) then
                synchronous signal_assignments
        end if ;
end process ;
```

There can be several asynchronous `elsif` clauses, but the synchronous elsif clause (if present) has to be the last one in the if clause. A flip-flop is generated for each signal that is assigned in the synchronous signal assignment. The asynchronous clauses result in combinational logic that drives the set and reset inputs of the flip-flops. If there is no synchronous clause, all logic becomes combinational.

Clock Enable

It is also possible to specify an enable signal in a process. Some technologies have a special enable pin on their basic building blocks. The synthesis tools recognize the function of the enable from the VHDL description and generates a flip-flop with an enable signal from the following code fragment:

```
signal input_foo, output_foo, enable, clk : bit ;
...
process  (clk)
begin
      if (clk'event and clk='1') then
            if (enable='1') then
                  output_foo <= input_foo ;
            end if ;
      end if ;
end process ;
```

In dataflow statements, a clock enable can be constructed with a `GUARDED` block and a conditional signals assignment.

```
b4: block  (clk'event and clk='1')
begin
      output_foo <= GUARDED input_foo when enable='1'
            else output_foo ;
end block ;
```

Wait Statements

Another way to generate registers is by using the `wait until` statement. The `wait until` clause can be used in a process, and is synthesizable if it is the first statement in the process. “Syntax and Semantic Restrictions” on page 22 gives more details about the synthesis restrictions of the wait statement. The following code fragment generates an edge triggered flip-flop between signal `input_foo` and `output_foo`:

```
signal input_foo, output_foo, clk : bit ;
...
process
begin
      wait until clk’event and clk=’1’ ;
      output_foo <= input_foo ;
end process ;
```

Note – There is no sensitivity list on this process. In VHDL, a process can have a sensitivity list *or* a wait statement, but not both. In this example, the process is executed if clk changes since clk is present in the wait condition. Also, the wait condition can be simplified to `wait until clk=’1’ ;`, since the process only starts if `clk` changes, and thus `clk’event` is always true.

The Exemplar synthesis tools do not support asynchronous reset behavior with wait statements. A synchronous reset remains possible however, by describing the reset behavior after the wait statement.

Variables

Variables (like signals) can also generate flip-flops. Since the variable is defined in the process itself, and its value never leaves the process, the only time a variable generates a flip-flop is when the variable is used before it is assigned in a clocked process. For instance, the following code segment generates a three-bit shift register.

```
signal input_foo, output_foo, clk : bit ;
...
process (clk)
        variable a, b : bit ;
begin
        if (clk'event and clk='1') then
                output_foo <= b ;
                b := a ;
                a := input_foo ;
        end if ;
end process ;
```

In this case, the variables a and b are used before they are assigned. Therefore, they pass their values from the last run through the process, which is the assigned value delayed by one clock cycle. If the variables are assigned before they are used, you will get a different circuit:

```
signal input_foo, output_foo, clk : bit ;
...
process (clk)
        variable a, b : bit ;
begin
        if (clk'event and clk='1') then
                a := input_foo ;
                b := a ;
                output_foo <= b ;
        end if ;
end process ;
```

Here, a and b are assigned before used, and therefore do not generate flip-flops. Instead, they generate a single wire. Only one flip-flop remains in between input_foo and output_foo because of the signal assignment in the clocked process.

Predefined Flip-flops and Latches

Flip-flops and latches can also be generated by using predefined procedures from the exemplar package. These procedure calls cause the synthesis tools to instantiate the required flip-flop or D-latch. There are various forms of these procedures available, including versions with asynchronous preset and clear. For details of the procedures, see “Predefined Procedures” on page 20.

Assigning I/O Buffers From VHDL

There are three ways to assign I/O buffers to your design from VHDL:

- Run the synthesis tools in “chip” mode.
- Use the buffer_sig attribute on a port in the VHDL source
- Use the buffer_sig command.
- Use direct component instantiation in VHDL of the buffer you require.

The buffer_sig attribute or the direct component instantiation will overwrite any default buffer assignment that the synthesis tools would do in “chip” mode.

The buffer_sig command is implemented differently for Galileo and Leonardo. For Galileo, you put the command in the control file. For Leonardo, you use the buffer_sig procedure.

It is important to realize that if you specify buffer names in the VHDL source, the synthesis tools will check the source technology library to find the buffer you requested. If you specify buffers in the control file, the synthesis tools will check the target technology library for a matching buffer.

Automatic Assignment Using Chip Mode

The easiest way of assigning buffers is to use the `-chip` option in the synthesis tools. (For Galileo, run the tool with the `-chip` option, or choose “Chip” mode from the Graphical User Interface. For Leonardo, use the `-chip` option with the `optimize` command.) This automatically assigns appropriate input, output, three-state, or bidirectional buffers to the ports in your entity definition. For instance,

```
entity example is
   port (  inp, clk   : in std_logic;
      outp   : out std_logic;
      inoutp : inout std_logic
   );
end example;
```

targeted to the Actel technology translates into an `INBUF` for `inp` and `clk`, an `OUTBUF` for `outp`, and a `BIBUF` for `inoutp` (if it is both used and assigned). `outp` would become a `TRIBUFF` if it was assigned to a three-state value under a condition:

```
outp <= inp when ena = ’1’ else ’Z’ ;
```

The above example also holds for buses, of course. The sections “Three-state Buffers” on page 14 and “Bidirectional Buffers” on page 17 give more details on how to generate three-state buffers and bidirectional buffers from VHDL.

Manual Assignment Using The BUFFER_SIG Property

For Galileo only, special buffers, e.g. clock buffers, can be assigned using the `buffer_sig` property. This can be done in the control file, with the `BUFFER_SIG` command. Here is an example:

```
BUFFER_SIG CLOCK_BUFFER clk
```

For Leonardo, special buffers can be assigned by using the `BUFFER_SIG` procedure. After reading in a design, use the command `BUFFER_SIG  CLOCK_BUFFER` *net_names*.

The `buffer_sig` property can also be set on a port using the `buffer_sig` attribute in the VHDL source.

```
entity example is
port (  inp, clk   : in std_logic;
   outp   : out std_logic;
   inoutp : inout std_logic
   );
   attribute buffer_sig : string ;
   attribute buffer_sig of clk:signal is “CLOCK_BUFFER” ;
end example;
```

Port `clk` will be connected to the input of the external clock buffer `CLOCK_BUFFER`. An intermediate node called `manual_clk` appears on `CLOCK_BUFFER`’s output. Gates specified in the control file are searched for in the target technology library. Gates specified in the VHDL source are searched for in the source technology library.

Buffer Assignment Using Component Instantiation

It is also possible to instantiate buffers in the VHDL source file with component instantiation. In particular, if you want a specific complex input or output buffer to be present on a specific input or output, component instantiation is a very powerful method:

```
entity special is
   port ( inp : in std_logic ;
      clk : in std_logic ;
      ...
      outp : out std_logic;
      inoutp : inout std_logic
   ) ;
end special ;

architecture exemplar of special is
   component OUTPUT_FLIPFLOP
      port (  c,d,t : in std_logic ;
         o : out std_logic
      ) ;
   end component ;
   component INPUT_BUFFER
      port (  i : in std_logic ;
         o : out std_logic
      ) ;
   end component ;
   signal intern_in, intern_out, io_control : std_logic ;
begin
   b1 : OUTPUT_FLIPFLOP port map (c=>clk, d=>intern_out,
                                  t=>io_control, o=>inoutp) ;
   b2 : INPUT_BUFFER port map (i=>inoutp, o=>intern_in) ;
   ...
end exemplar ;
```

In this example, component instantiation forces an `OUTPUT_FLIPFLOP` buffer on the bidirectional pin `inoutp`. Also an input buffer `INPUT_BUFFER` is specified to pick up the value from this pin to be used internally.

The synthesis tools will look for definitions of VHDL instantiated components in the source library. Make sure that you specify a source library (`-source=lib_name`) or set the attribute `NOBUFF` on the I/O pin of the instantiated buffer, otherwise The synthesis tools will consider the buffer to be a user-defined block and will add a buffer from the target technology.

Three-state Buffers

Three-state buffers and bidirectional buffers (covered in the next section) are very easy to generate from a VHDL description.

A disabled three-state buffer will be in a high-impedance state. VHDL itself does not predefine a high-impedance state, but the IEEE 1164 standard logic package defines the `'Z'` character literal to have a behavior that exactly resembles the behavior of the high-impedance state of a three-state buffer. A signal (a port or an internal signal) of the standard logic type can be assigned a `'Z'` value. The synthesis tools recognize the `'Z'` value and creates a three-state buffer from a conditional assignment with `'Z'`:

```
entity three-state is
   port (   input_signal : in std_logic ;
      ena : in std_logic ;
      output_signal : out std_logic
            ) ;
end three-state ;

architecture exemplar of three-state is
begin
   output_signal <= input_signal when ena = '1' else 'Z' ;
end exemplar ;
```

Note – In the when clause, both `input_signal` and the condition `ena='1'` can be full expressions. The synthesis tools generate combinational logic driving the input or the enable of the three-state buffer for these expressions.

Normally, simultaneous assignment to one signal in VHDL is not allowed for synthesis, since it would cause data conflicts. However, if a conditional `'Z'` is assigned in each assignment, simultaneous assignment resembles multiple three-state buffers driving the same bus.

```
entity three-state is
   port (   input_signal_1, input_signal_2 : in std_logic ;
      ena_1, ena_2 : in std_logic ;
      output_signal : out std_logic
            ) ;
end three-state ;

architecture exemplar of three-state is
begin
   output_signal <= input_signal_1 when ena_1 = '1' else 'Z' ;
   output_signal <= input_signal_2 when ena_2 = '1' else 'Z' ;
end exemplar ;
```

Note – The synthesis tools do not check for bus-conflicts on three-state assignments. Therefore, make sure that the enable signals of the three-state drivers are never simultaneously active. In this example, `ena_1` and `ena_2` should never be `'1'` simultaneously.

These examples show assignments to output ports (device ports). It is also possible to do the assignments to an internal signal. This will create internal busses in such a case.

Three-state buffers can also be generated from process statements:

```
driver1 : process (ena_1, input_signal_1) begin
   if (ena_1='1') then
      output_signal <= input_signal_1 ;
   else
      output_signal <= 'Z' ;
   end if ;
end process ;
driver2 : process (ena_2, input_signal_2) begin
   if (ena_2='1') then
      output_signal <= input_signal_2 ;
   else
      output_signal <= 'Z' ;
   end if ;
end process ;
```

If the target technology does not have any internal three-state drivers, Galileo can transform the three-state buffers into regular logic with the `-tristate` option. Leonardo performs this transformation when the `tristate_map` variable is set to TRUE.

Bidirectional Buffers

Bidirectional I/O buffers will be created by the synthesis tools if an external port is both used and assigned inside the architecture. Here is an example:

```
entity bidir_function is
    port (   bidir_port : inout std_logic ;
        ena : in std_logic ;
                 ...
              ) ;
end bidir_function ;

architecture exemplar of bidir_function is
    signal internal_signal, internal_input : std_logic ;
begin
    bidir_port <= internal_signal when ena = '1' else 'Z' ;
                internal_input <= bidir_port ;
                ...
                -- use internal_input
                ...
                -- generate internal_signal
end exemplar ;
```

The difference with the previous example is that in this case, the output itself is used again internally. Note that for that reason, the port `bidir_port` is declared to be `inout`.

The enable signal `ena` could also be generated from inside the architecture, instead of being a primary input as in this example.

The synthesis tools select a suitable bidirectional buffer from the target technology library. If there is no bidirectional buffer available, it selects a combination of a three-state buffer and an input buffer.

Busses

The examples in the previous sections all use single bits as signals. In reality, busses are often used: arrays of bits with (multiple) three-state drivers. In that case, the type of the bus signal should be `std_logic_vector`. All examples given still apply for busses, although the `'Z'` character literal now has to be a string literal. Here is one example:

```
entity three-state is
   port (   input_signal_1, input_signal_2 : in
                std_logic_vector (0 to 7) ;
      ena_1, ena_2 : in std_logic ;
      output_signal : out std_logic_vector(0 to 7)
            ) ;
end three-state ;

architecture exemplar of three-state is
begin
   output_signal <= input_signal_1 when ena_1 = '1'
                else "ZZZZZZZZ" ;
   output_signal <= input_signal_2 when ena_2='1'
                else "ZZZZZZZZ" ;
end exemplar ;
```

This generates two set of eight three-state buffers, two on each line of the bus `output_signal`.

As with single three-state drivers, busses can be internal signal, or ports. Similarly, busses can be created using processes.

State Machines

This section describes a basic form of a general state machine description. VHDL coding style, power-up and reset, state encoding and other issues will be discussed.

General State Machine Description

There are various ways to describe a state machine in VHDL. This section will only show the most commonly used description.

The possible states of the state machine are listed in an enumerated type. A signal of this type (`present_state`) defines in which state the state machine appears. In a `case` statement of one process, a second signal (next_state) is updated depending on present_state and the inputs. In the same `case` statement, the outputs are also updated. Another process updates present_state with next_state on a clock edge, and takes care of the state machine reset.

Here is the VHDL code for such a typical state machine description. This design implements a RAS-CAS controller for DRAM refresh circuitry.

```
entity ras_cas is
   port ( clk, cs, refresh, reset : in bit ;
      ras, cas, ready : out bit ) ;
end ras_cas ;

architecture exemplar of ras_cas is
   -- Define the possible states of the state machine
   type state_type is (s0, s1, s2, s3, s4) ;
   signal present_state, next_state : state_type ;
begin

   registers : process  (clk, reset)
   begin
      -- process to update the present state
      if (reset='1') then
               present_state <= s0 ;
      elsif clk'event and clk = '1' then
               present_state <= next_state;
      end if ;
   end process ;
```

```
transitions : process (present_state, refresh, cs)
begin
    -- process to calculate the next state and the outputs
    case present_state is
            when s0 =>
                    ras <= '1' ; cas <= '1' ; ready <= '1' ;
                    if (refresh = '1') then
                        next_state <= s3 ;
                    elsif (cs = '1') then
                        next_state <= s1 ;
                    else
                        next_state <= s0 ;
                    end if ;
            when s1 =>
                    ras <= '0' ; cas <= '1' ; ready <= '0' ;
                    next_state <= s2 ;
            when s2 =>
                    ras <= '0' ; cas <= '0' ; ready <= '0' ;
                    if (cs = '0') then
                        next_state <= s0 ;
                    else
                        next_state <= s2 ;
                    end if ;
            when s3 =>
                    ras <= '1' ; cas <= '0' ; ready <= '0' ;
                    next_state <= s4 ;
            when s4 =>
                    ras <= '0' ; cas <= '0' ; ready <= '0' ;
                    next_state <= s0 ;
            end case ;
    end process ;
end exemplar ;
```

VHDL Coding Style For State Machines

There are various issues of coding style for state-machines that might affect performance of the synthesized result.

A first issue is the form of state machine that will be created. There are basically two forms of state machines, Mealy machines and Moore machines. In a Moore machine, the outputs do not directly depend on the inputs, only on the present state. In a Mealy machine, the outputs depend directly on the present state and the inputs.

In the RAS-CAS state machine described in the previous section, the outputs ras, cas and ready only depend on the value of `present_state`. This means that the description implements a Moore machine. If the outputs would be set to different values under the input conditions in the `if` statements inside the `case` statement, a Mealy machine would have been created. In a Moore machine, there is always a register in between the inputs and the outputs. This does not have to be the case in Mealy machines.

A second issue in coding style is the `case` statement that has been used to test the `present_state`. A `case` statement is more efficient than a `if-then-elsif-else` statement, since that would build a priority encoder to test the state (which could mean more logic in the implementation). It is also important to note that there is no `OTHERS` entry in the `case` statement. An `OTHERS` entry could create extra logic if not all the states are mentioned in the `case` statement. This extra logic will have to determine if the machine is in any of the already mentioned states or not. Unless there are a number of states where the state machine behaves exactly the same (which is not likely since then you could reduce the state machine easily) an `OTHERS` entry is not beneficial and will, in general, create more logic than is required.

A third issue is the assignments to outputs and `next_state` in the state transition process. VHDL defines that any signal that is not assigned anything should retain its value. This means that if you forget to assign something to an output (or `next_state`) under a certain condition in the `case` statement, the synthesis tools will have to preserve the value. Since the state transition process is not clocked, latches will have to be generated. You could easily forget to assign to an output if the value does not matter. The synthesis tools will warn you about this, since it is a common user error in VHDL:

```
"file.vhd", line xx : Warning, latches might be needed for XXX.
```

Make sure to always assign something to `next_state` and the state machine outputs under every condition in the process to avoid this problem. To be absolutely sure, you could also assign a value to the signal at the very beginning of the process (before the start of the `case` statement).

Note – Graphical state-machine entry tools often generate state machine descriptions that do not always assign values to the outputs under all conditions. The Exemplar synthesis tools will warn about this, and you could either manually fix it in the VHDL description, or make sure you fully specify the state machine in the graphical entry tool. The synthesis tools cannot fill in the missing specifications, since it is bounded by the semantics of VHDL on this issue.

Power-up And Reset

For simulation, the state machine will initialize into the leftmost value of the enumeration type, but for synthesis it is unknown in which state the machine powers up. Since the Exemplar synthesis tools do state encoding on the enumeration type of the state machine (see “State Encoding” below), the state machine could even power up in a state that is not even defined in VHDL. Therefore, to get simulation and synthesis consistency, it is very important to supply a reset to the state machine.

In the example state machine shown in “General State Machine Description” on page 18, an asynchronous reset is used, but a synchronous reset would be possible. “Registers, Latches and Resets” on page 1 explains more about how to specify resets on registers in VHDL.

State Encoding

The Exemplar synthesis tools have a variety of methods to control state encoding for state machines that use an enumeration type for the declaration of the states. “Enumeration Types” on page 10 discusses all forms of state encoding in detail**.**

Arithmetic And Relational Logic

This section gives an overview of how arithmetic logic is generated from VHDL, what the synthesis tools do with this logic and how to avoid getting into combinational explosion with large amounts of arithmetic behavior.

In general, logic synthesis is very powerful in optimizing “random” combinational behavior, but has problems with logic which is arithmetic in nature. Often special precautions have to be taken into consideration to avoid ending up with inefficient logic or excessive run times. Alternatively, macros may be used to implement these functions. For more information see “Technology-Specific Macros” on page 29.

The Exemplar synthesis tools support the overloaded operators “+”, “-”, “*”, and “abs.” These operators work on integers (and on arrays; with the exemplar package).

If you use overloaded operators to calculate compile time constants, the synthesis tools will not generate any logic for them. For example, the following code segments do not result in logic, but assign a constant integer 13 to signal `foo`.

```
function add_sub (a: integer, b: integer, add : boolean)
                           return integer is
begin
             if (add = TRUE) then
                return a + b ;
             else
                return a - b ;
             end if ;
end my_adder ;
signal foo : integer ;
constant left : integer := 12 ;
....
foo <= add_sub (left,6,TRUE) - 5 ;-- Expression evaluates to 13
```

If you are not working with compile time constant operands, arithmetic logic is generated for arithmetic operators.

The pre-defined “+” on integers generates an adder. The number of bits of the adder depends on the size of the operands. If you use integers, a 32 bit adder is generated. If you use ranged integers, the size of the adder is defined so that the entire range can be represented in bits. For example, if variables `a` and `b` do not evaluate to constants, the following code segment:

```
variable a, b, c : integer ;
c := a + b ;
```

generates a 32-bit (signed) adder, but

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

generates an 8-bit (unsigned) adder.

If one of the operands is a constant, initially a full-sized adder is still generated but logic minimization eliminates much of the logic inside the adder, since half of the inputs of the adder are constant.

The pre-defined “-” on integers generates a subtracter. Same remarks apply as with the “+” operator.

The pre-defined “*” multiplication on integers generates a multiplier. Full multiplication is supported when a module generator is used. See the *Synthesis and Technology Reference Guide* for information on module generators supported for specific technologies. You can also define your own technology specific multiplier.

The pre-defined “/” division on integers generates a divider. Only division by a power of two is supported. In this case, there is no logic generated, only shifting of the non-constant operand. With module generation you could define your own technology-specific divider.

The predefined “**” exponentiation on integers is only supported if both operands are constant.

“=,” “/=,” “<,” “>,” “<=,” and “>=” generate comparators with the appropriate functionality.

Operations on integers are done in two-complement implementation if the integer range extends below 0. If the integer range is only positive, an unsigned implementation is used.

There are a number of other ways to generate arithmetic logic. The predefined exemplar functions `add`, `add2`, `sub`, `sub2`, `+`, and `-` on `bit_vector` and `std_logic_vector` types are examples of functions which do this. For descriptions of these functions, see “Predefined Functions” on page 14.

By default, the synthesis tools will generate “random” logic for all pre-defined operators. Alternatively, if a module generator for a particular target technology is supplied, the synthesis tools will generate technology specific solutions (e.g., hard macros) instead of random logic.

Module Generation

When arithmetic and relational logic are used for a specific VHDL design, the synthesis tools provide a method to synthesize technology specific implementations for these operations. Generic modules (for bit-sizes > 2) have been developed for many of the FPGAs supported by the Exemplar synthesis tools to make the most efficient technology specific implementation for arithmetic and relational operations.

For Galileo, use the `-modgen=`*modgen_library* option to include a module generation library of the specified technology and infer the required arithmetic and relational operations of the required size from a user VHDL design. For Leonardo, use the `modgen_read` *modgen_library* command to load the module generation library into the HDL database. Since these modules have been designed optimally for a target technology, the synthesis result is, in general, smaller and/or faster and takes less time to compile.

If you want to define your own module generator for a specific technology, you can do so by describing a module generator in VHDL. For more information on module generation, see Chapter 9-Chapter 11.

Resource Sharing

The synthesis tools perform automatic common subexpression elimination for arithmetic and boolean expressions. The following example has two adders in the code, but they are adding the same numbers, `a` and `b`.

```
signal a,b,c,d : integer range 0 to 255 ;
...
process (a,b,c,d) begin
   if ( a+b = c ) then    <statements>
   elsif ( a+b = d) then <more_statements>
   end if ;
end process ;
```

After automatic common subexpression elimination, only one adder will be used in the final circuit. Thus, it would create the same logic as the following example.

```
process  (a,b.c.d)
          variable tmp : integer range 0 to 255 ;
begin
          tmp := a+b ;
          if ( tmp = c ) then  <statements>
          elsif ( tmp = d) then <more_statements>
          end if ;
end process ;
```

Proper use of parentheses guide the synthesis tools in eliminating common subexpressions. The following code segment, for example, can be properly modified to share an adder.

```
o1 <= a + b + c;
o2 <= b + c + d;
```

Using parentheses, the logic can share an adder for inputs `b` and `c`, as shown below.

```
o1 <= a + (b + c);
o2 <= (b + c) + d;
```

The synthesis tools automatically perform a limited amount of resource sharing of arithmetic expressions that are mutually exclusive. Consider the following example:

```
process (a,b,c,test) begin
      if (test=TRUE) then
            o <= a + b ;
      else
            o <= a + c ;
      end if ;
end process ;
```

Initially, two adders and a multiplexer are created, but after the automatic resource sharing one adder is reduced, and the final circuit is same as would be created from the following code:

```
process (a,b,c,test) begin
        variable tmp : integer range 0 to 255 ;
begin
        if (test=TRUE) then
                tmp := b ;
        else
                tmp := c ;
        end if ;
        o <= a + tmp ;
end process ;
```

The limitations of automatic resource sharing are as follows:

- Complex operators must drive the same signal.
- Complex operators must be of the same type (for example, two adders) and have the same width (for example, 8-bit adders).

Ranged Integers

It is best to use ranged integers instead of “unbound” integers. In VHDL, an unbound integer (integer with no range specified) is guaranteed to include the range -2147483647 to +2147483647. This means that at least 32 bits are needed to implement an object of this type. The synthesis tools have to generate large amounts of logic in order to perform operations on these objects. Some of this logic may become redundant and get eliminated in the optimization process, but the run time is slowed down considerably. If you use integers as ports, all logic has to remain in place and synthesis algorithms are faced with a complex problem. Therefore, if you do not need the full range of an integer, specify the range that you need in the object declaration:

```
signal small_int : integer range 255 downto 0 ;
```

`small_int` only uses eight bits in this example, instead of the 32 bits if the range was not specified.

Advanced Design Optimization

Module generation, resource sharing and the use of ranged integers are all examples of how a particular design can be improved for synthesis without changing the functionality. Sometimes it is possible to change the functionality of the design slightly, without violating the design specification constraints, and improve the implementation for synthesis. This requires understanding of VHDL and what kind of circuitry is generated, as well as understanding of the specifications of the design. One example of this is given, in the form of a loadable loop counter.

Often, applications involve a counter that counts up to a input signal value, and if it reaches that value, some actions are needed and the counter is reset to 0.

```
process begin
   wait until clk’event and clk=’1’ ;
      if (count = input_signal) then
         count <= 0 ;
      else
         count <= count + 1 ;
      end if ;
end process ;
```

In this example, the synthesis tools build an incrementer and a full-size comparator that compares the incoming signal with the counter value.

In this example, a full comparator has to be created since the VHDL description indicates that the comparison has to be done each clock cycle. If the specification allows that the comparison is only done during the reset, we could re-code the VHDL and reduce the overall circuit size by loading the counter with the `input_signal`, and then counting down to zero:

```
process begin
   wait until clk’event and clk=’1’ ;
      if (count = 0) then
         count <= input_signal ;
      else
         count <= count - 1 ;
      end if ;
end process ;
```

Here, one decrementer is needed plus a comparison to a constant (0). Since comparisons to constants are a lot cheaper to implement, this new behavior is much easier to synthesize, and results in a smaller circuit.

This is a single example of how to improve synthesis results by changing the functionality of the design, while staying within the freedom of the design specification. However, the possibilities are endless, and a designer should try to use the freedom in the design specification to get truly optimal synthesis performance.

Technology-Specific Macros

In many cases, the target technology library includes a number of hard macros and soft macros that perform specific arithmetic logic functions. These macros are optimized for the target technology and have high performance.

This section will explain how to instantiate technology specific macros in the VHDL source to assure full control over the synthesized logic. The VHDL description will become technology dependent.

Note that the Exemplar synthesis tools do automatic inference of technology specific macros from standard (technology independent) arithmetic and relational operators when Module Generation is used. The section “Resource Sharing” on page 25 explains more about this and details can be found in Chapter 9–Chapter 11. However, if a particular hard-macro is required, or there is no Module Generator available for the your technology, manual instantiation will be needed.

With the Exemplar synthesis tools, it is possible to use component instantiation of soft macros or hard macros in the target technology, and use these high performance macros. An added benefit is that the time needed for optimization of the whole circuit can be significantly reduced since the synthesis tools do not have to optimize the implementation of the dedicated functions anymore.

As an example, suppose you would like to build an 8-bit counter in the device family FPGAX. There is a hard-macro available in the `FPGAX` library that will do this. Call it the `COUNT8`. In order to directly instantiate this macro in VHDL, declare a component `COUNT8` and instantiate it with a component instantiation statement.

```
component COUNT8
      port (pe, c, ce, rd : in std_logic ;
            d : in std_logic_vector (7 downto 0) ;
            q : out std_logic_vector (7 downto 0)
      ) ;
end component ;
...
-- clock, count_enable, reset, load, load_data and output are signals
-- in the VHDL source
...
counter_1 : COUNT8 port map (c=>clock, ce=>count_enable,
                        rd=>reset, pe=>load, d=>load_data, q=>output) ;
```

The synthesis tools will synthesize this component as a black-box, since there is no entity/architecture description for it. It will appear in the output file as a symbol.

If you use hard-macros in a VHDL description, specify a source technology so the synthesis tools can include area and timing information. For this example, you would use the option `-source=fpgax` with Galileo. With Leonardo, you would use the `load_library fpgax` command to load the source library into the design database.

If simulation is required on the source VHDL design, you have to supply an entity and architecture for `COUNT8`. In that case, make sure to set the attribute `NOOPT` to TRUE on the component `COUNT8`, so that the synthesis tools treat the component as a black-box, otherwise they will synthesize `COUNT8` into general logic. For more information about setting the `NOOPT` attribute on a component, see the section "Finding Definitions of Components" on page 3.

Using technology specific macro instantiation can speed-up the synthesis and optimization process considerably. It also often leads to more predictable area and delay costs of the design. The VHDL description however becomes technology dependent.

Multiplexers and Selectors

From a `case` statement, the synthesis tools create either muxes or selector circuits. In the following example, a selector circuit is created.

```
case test_vector is
   when  “000"  =>     o <= bus(0)  ;
   when  “001" | ”010" | “100" => o <= bus(1)  ;
   when  “011" | ”101" | “110" => o <= bus(2) ;
   when  “111" =>     o <= bus(3) ;
end case ;
```

If the selector value is the index to be selected from an array, the selector will resemble a multiplexer. It is still possible to express this in a `case` statement, but it is also possible to use a variable indexed array. For example, if an integer value defines the index of an array, a variable indexed array will create the multiplexer function:

```
signal vec : std_logic_vector (0 to 15) ;
signal o : std_logic ;
signal i : integer range 0 to 15 ;
...
o <= vec(i) ;
```

selects bit `i` out of the vector `vec`. This is equivalent to the more complex writing style with a `case` statement:

```
case i is
   when 0 => o <= vec(0) ;
   when 1 => o <= vec(1) ;
   when 2 => o <= vec(2) ;
   when 3 => o <= vec(3) ;
   ...
end case ;
```

For the prior description, the synthesis tools create the same multiplexers as they do for the variable-indexed array.

The Exemplar synthesis tools fully support variable-indexed arrays, including index values that are enumerated types rather then integers, and index values that are expressions rather then singe identifiers.

ROMs, PLAs And Decoders

There are many ways to express decoder behavior from a ROM or PLA table. The clearest description of a ROM would be a `case` statement with the ROM addresses in the case conditions, and the ROM data in the `case` statements. In this section, two other forms are discussed:

1. Decoder as a constant array of arrays.

2. Decoder as a constant two-dimensional array.

Here is an example of a ROM implemented with an array of array type. The ROM defines a hexadecimal to 7-segment decoder:

```
type seven_segment is array (6 downto 0) ;
type rom_type is array (natural range <>) of seven_segment ;
constant hex_to_7 : rom_type (0 to 15)   :=
    (“0111111", -- 0
     “0011000", -- 1
     “1101101", -- 2         Display segment index numbers :
     “1111100", -- 3                               2
     “1011010", -- 4                           1       3
     “1110110", -- 5                               6
     “1110111", -- 6                           0       4
     “0011100", -- 7                               5
     “1111111", -- 8
     “1111110", -- 9
     “1011111", -- A
     “1110011", -- B
     “0100111", -- C
     “1111001", -- D
     “1100111", -- E
     “1000111") ; -- F
-- Now, the ROM field can be accessed via a integer index
display_bus <= hex_to_7 (i) ;
```

The ROM with array of array implementation has the advantage that it can be accessed via a simple integer value as its address. A disadvantage is that each time another ROM is defined, a new element type (`seven_segment`) and a new ROM type (`rom_type`) have to be defined.

PLA descriptions should allow a `’X’` or `’-’` dont-care value in the input field, to indicate a product lines’ insensitivity for a particular input. You cannot use a `case` statement for a PLA with dont cares in the input field since a comparison with a value that is not `’0’` or `’1’` will return FALSE in a case condition (as opposed to just ignoring the input). Instead, a small procedure or function is needed that explicitly defines comparisons to `’X’` or `’-’`. The following example describes such a procedure. First, a general 2-dimensional PLA array type is declared.

```
type std_logic_pla is array (natural range <>, natural range <>)
of std_logic;
...
procedure pla_table (constant invec: std_logic_vector;
                     signal outvec: out std_logic_vector;
                     constant table: std_logic_pla) is
     variable x : std_logic_vector (table’range(1)) ; -- product lines
     variable y : std_logic_vector (outvec’range) ;    -- outputs
     variable b : std_logic ;
begin
     assert (invec’length + outvec’length = table’length(2))
     report “Size of Inputs and Outputs do not match table size”
     severity ERROR ;
```

```
-- Calculate the AND plane
       x := (others=>'1') ;
       for i in table'range(1) loop
          for j in invec'range loop
            b := table (i,table'left(2)-invec'left+j) ;
            if (b='1') then
                 x(i) := x(i) AND invec (j) ;
            elsif (b='0') then
                 x(i) := x(i) AND NOT invec(j) ;
            end if ;
-- If b is not '0' or '1' (e.g. '-') product line is insensitive to
invec(j)
            end loop ;
       end loop ;
-- Calculate the OR plane
       y := (others=>'0') ;
       for i in table'range(1) loop
          for j in outvec'range loop
            b := table(i,table'right(2)-outvec'right+j) ;
            if (b='1') then
                 y(j) := y(j) OR x(i);
            end if ;
          end loop ;
       end loop ;
       outvec <= y ;
end pla_table ;
```

Once the two-dimensional array type and the PLA procedure are defined, it is easy to generate and use PLAs (or ROMs). As a simple example, here is a PLA description of a decoder that returns the position of the first `’1’` in an array. The PLA has five product lines (first dimension) and seven IOs (four inputs and three outputs) (second dimension).

```
constant pos_of_fist_one : std_logic_pla (4 downto 0, 6 downto 0) :=
   (“1---000",--  first ’1’ is at position 0
    “01--001",--  first ’1’ is at position 1
    “001-010",--  first ’1’ is at position 2
    “0001011",--  first ’1’ is at position 3
    “0000111") ;--  There is no ’1’ in the input
signal test_vector : std_logic_vector (3 downto 0) ;
signal result_vector : std_logic_vector (2 downto 0) ;
...
-- Now use the pla table procedure with PLA pos_of_first_one
-- test_vector is the input of the PLA, result_vector the output.
...
pla_table ( test_vector, result_vector, pos_of_first_one) ;
```

The PLA could have been defined in a array-of-array type also, just as the ROM described above. A procedure or function for the PLA description will always be necessary to resolve the dont-care information in the PLA input field.

Note – The synthesis tools will do a considerable amount of compile-time constant propagation on each call to the procedure `pla_table`. This does not affect the final circuit result at all. It just adds the possibility to specify dont-care information in the PLA input table. In fact, a ROM described with an array-of-array type and a variable integer index as its address will produce the same circuit as the ROM specified in a two-dimensional array and using the `pla_table` procedure. If the modeled ROM or PLA becomes large, consider a technology-specific solution by directly instantiating a ROM or PLA component in the VHDL description. Many FPGA and ASIC vendors supply ROM and/or PLA modules in their library for this purpose.

The VHDL Environment 4

This chapter discusses the Exemplar synthesis tools and the VHDL tool environment, including search paths, interfacing with other VHDL tools, and the Exemplar package.

Entity and Package Handling

Loading Entities and Packages (Galileo)

Packages and entities in VHDL are stored in libraries. VHDL tools often have the possibility to load VHDL files (with packages and entities) separately into a directory that is assigned to a library. Galileo does not have the ability to pre-load VHDL files into libraries. Instead, all VHDL sources need to be specified for each run of Galileo.

Galileo can get VHDL source from three different areas:

1. Predefined VHDL package files
2. Optionally included VHDL files
3. The source (input) VHDL file for the run of the tool

An example of a predefined package is the package STANDARD (which is pre-defined for VHDL), that Galileo loads from file `standard.vhd` in `$EXEMPLAR/data/packages.syn`. Other packages are available both in that directory, and in `$EXEMPLAR/data`.

```
-vhdl_file=filename
```

With the `-vhdl_file=<filename>` option, it is possible to load a VHDL file into Galileo before the source VHDL file is read. In the Graphical User Interface, use the “VHDL Files” option in the Input Options menu. Multiple `-vhdl_file` options allow you to load multiple files. The order in which the files are included is important. If you use a package A in file B, make sure that the file in which A is defined is loaded before file B.

After all the `-vhdl_file` options are executed, and their corresponding VHDL files are loaded into Galileo, the source VHDL file is read.

Galileo can handle either VHDL IEEE 1076-1987 or IEEE 1076-1993 dialects of VHDL. The default is 87. To run 93-style VHDL, use the switch -vhdl_93 on the command line, or use the “VHDL Style” option the (VHDL) input options menu on the GUI.

Galileo does not handle all 93 style features. They support the most commonly used features of the ’93 extension: shifter and rotator operators, xnor operator and extended identifiers.

Loading Entities and Packages (Leonardo)

If there is only design file, you can read the file directly into Leonardo. If the design is split into multiple source files, however, you need to analyze them in the proper order so that all terms are defined before they are used in the design. For example, if there is a package declaration in one file that must be used by the whole design, that file must be analyzed first. In Leonardo, all the design units are stored in the HDL database, and you can analyze as many of them as you want.

Entity Compiled as the Design Root

When the VHDL source is loaded, Galileo will start compiling the top level entity and start the synthesis process. By default, Galileo uses the last entity found in the source file as the top-level entity. This behavior can be changed, however.

```
-entity=entity_name
```

The option `-entity=`*entity_name* on the command line will let Galileo find the entity specified and consider that the root of the design. In the Graphical User Interface, use the “Top Entity” option in the VHDL Input Options window. An entity from an included VHDL file can be specified as the root of the design.

```
-architecture=architecture_name
```

After the root entity is found, Galileo will try to find a matching architecture for it. By default, the tools will choose the LAST architecture described in the source VHDL file that matches the top-level entity. Use the `-architecture=`*architecture_name* to overwrite this default. In the Graphical User Interface, use the option “Top Architecture” in the VHDL Input Options window.

By default, Leonardo assumes that the last entity or configuration analyzed is the root entity. By default, the LAST architecture analyzed for the root entity is compiled. You can use the `elaborate` command with `-entity` *entity_name* and `-architecture` *arch_name* arguments to selectively compile a particular entity-architecture pair.

Finding Definitions of Components

In order to instantiate an entity into a VHDL description, you must first declare a component for it. If you use a component instantiation in your VHDL design, the synthesis tools try to find the definition of that component. There are three possibilities.

1. The component is a cell in a source technology library.
2. The component has a matching (named) entity in the VHDL source
3. The component has no definition.

If a source technology is specified, the synthesis tools try to find the component in the source technology library. This is especially helpful if the component represents a particular macro in the source technology. For an example, see "Technology-Specific Macros" on page 29.

If the component is not present in the source technology, the synthesis tools try to find an entity and architecture for it. The entity (and architecture) could be present in the same file, or in an included VHDL file.

If the synthesis tools cannot find a matching entity for the component, they issue the following warning and leave the contents component undefined:

```
Warning, component component_name has no definition
```

Working with components without a definition can be useful if a particular module of the design is not synthesizable. A clock generator or a delay-module is an example of this. The contents of that module should be provided separately to the physical implementation tools.   Leaving components undefined is also useful in two other cases:

- With Galileo, to preserve hierarchy through the synthesis process.
- With all the Exemplar synthesis tools, for using hard and soft macros in the target technology (see "Technology-Specific Macros" on page 29).

It is possible to explicitly leave the contents of a component empty, even though there is a entity/architecture for it or a cell in the source technology library. In that case, specify the boolean attribute `NOOPT` on the component, or on its corresponding entity, or use the `-noopt=`*entity_name* option (for Galileo only) as described below. This can be useful when only a part of the hierarchy of a design has to be synthesized or if a user-defined simulatable but not synthesizable block is run through the synthesis tools. Here is an example of how to set the `noopt` attribute:

```
component clock_gen
   .....
end component ;
attribute noopt : boolean ;
attribute noopt of clock_gen:component is TRUE ;
```

Components with a `noopt` attribute or undefined components will be handled as black boxes by the synthesis tools, and will show up as cells in the target netlist. Supplying the technology-specific contents of these cells is left to the user. It is also possible to only noopt a particular instance of a component by setting the noopt attribute on the label of the component instantiation statement. This will have the same effect as if the attribute was added to the underlying entity.

How to Use Packages

A functionality described in a VHDL package is included into the VHDL design using the use clause. This is the general form of the use clause:

```
library lib ;
use lib.package.selection ;
```

The use clause is preceded by a library clause. There are predefined libraries `work` and `std` that do not have to be declared in a library clause before they are used in a use clause. All other libraries do need top be declared. Library `std` is normally only used to include packages predefined in VHDL1076, but library `work` is free to be used for any user-defined packages. User-defined library names are also allowed.

If a particular package is not found in the specified library, the synthesis tools perform the following steps to find the package:

1. The current `work` library is searched for the package.
2. If it is not there, it searches for a file with the name *package*.`vhd` in the present working directory. The present working directory is the directory where a synthesis tool is running.
3. If the file is not there, the synthesis tools try to find it in the `$EXEMPLAR/data` or the `$EXEMPLAR/data/packages.syn` directory to check if it is a pre-defined package.
4. If the file is not there, the synthesis tools issue an error message that the package can not be found.

The *selection* can consist of only one name of an object, component, type or subprogram that is present in the package, or the word `all`, in which case all functionality defined in the package is loaded into the synthesis tools and can be used in the VHDL description.

As an example, the IEEE 1164 `std_logic_1164` package (that defines the multi-valued logic types that are often used for circuit design), is included with the following statements:

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

This package is loaded from the `$EXEMPLAR/data/packages.syn` file. This file contains only the declarations of the functions of the `std_logic_1164` package. The bodies of the functions are built into the Exemplar synthesis tools for synthesis efficiency.

Note – The contents of the package you include with a `use` clause becomes visible and usable only within the scope where you use the `use` clause. It would be beyond the scope of this manual to explain the VHDL scoping rules, but if you start a new entity (and architecture), always make sure that you include the packages you need with `use` clauses just before the entity.

Interfacing With Other VHDL Tools

The VHDL parsers in the Exemplar synthesis tools are compliant with the IEEE VHDL 1076-1987 standard. Hence, apart from the VHDL restrictions for synthesis, interfacing with tools that generate VHDL or operate on VHDL should not introduce compatibility problems.

However, since VHDL 1076 does not define file handling, there might be mismatches in the way the tools handle files. Many VHDL simulators incorporate a directory structure to store separately compiled VHDL files. The synthesis tools do not use separate compilation of VHDL files. Therefore, all packages and components that are used for a VHDL design description should be identified before running the synthesis tools, as explained in the previous section.

VHDL Simulators

Always make sure to load the packages and entities in your design into the simulator prior to simulating your root entity. For simulation, the `exemplar` and `exemplar_1164` packages can be found in the `$EXEMPLAR/data` directory. For details on using these packages, see “The Exemplar Packages” on page 11.

Post-Synthesis Functional Simulation

If desired, post-synthesis functional simulation can be performed using the structural VHDL output from the synthesis tools. In your design flow, choose the appropriate netlist output for the target technology. Then use the `-effort=reformat` switch (with Galileo) to produce structural VHDL for simulation. The flow with Galileo is, assuming an ASIC as the target technology for this example,

1. VHDL synthesis with Galileo:

```
galileo my_design.vhd my_design.edf -target=asic
-effort=exhaustive -report=2
```

2. Produce VHDL netlist:

```
galileo my_design.edf my_test.vhd -source=asic
-target=asic -effort=reformat -report=2
```

This produces the structural VHDL file `my_test.vhd`, which may now be simulated.

The synthesis tools synthesize all port types into single-bit values. These get written out in VHDL as ports of type `std_logic`. The original port types are not preserved.

In Leonardo, the same design can be written into multiple files in multiple formats. After optimization, choose the appropriate netlist output format for the target technology; then, you can write a VHDL description of the same synthesized design. By using a simulatable library of the target technology, this VHDL output can be simulated. The sequence of synthesis statements should be similar to the following:

```
load_lib asic
read original.vhd
optimize -tar asic <other options>
write synthesized.edf -- required for target technology
write synthesized.vhd -- can be used for simulation.
```

When doing synthesis from a VHDL description, one goal of post-synthesis VHDL simulation is to simulate the design with the original set of ports (same type, io mode etc.). With Galileo, the `-vhdl_wrapper=`*filename* option is used for that. On the GUI, you can find the wrapper option in the ’VHDL Input Options’ menu. With Leonardo, use the `create_wrapper` command to create the wrapper file.

The wrapper consists of an architecture (that connects to the original entity) that instantiates a component that refers to the synthesized description. Type-conversion functions connect ports of the synthesized description to the ports of the original description. Since both the synthesized description and the original description have the same name, we need to store the synthesized description into a different library (in the simulator) than the original one.

Load the synthesized VHDL description in a library called `synthesis` in your simulator. Then load the wrapper architecture into the work library. It will link with the originally compiled entity of the original VHDL description. The wrapper file uses type transformation functions from a package called `typetran` to translate the port types. This packages in the file `$EXEMPLAR/vhdl/typetran.vhd`. You have to load this package into the simulator before you load the wrapper description.

Now, the original entity can be simulated with the wrapper architecture. Since the wrapper instantiates the synthesized description, simulation will be done of the synthesized design by using the original entity (ports), and thus the original test vectors can be used to simulate.

Viewlogic

Users with VHDL files originally written for the Viewlogic synthesis or simulation systems will be using the `pack1076` and `stdsynth` packages. Galileo supports all behavior from these packages, as long as they are included according to VHDL rules with a `use` clause. Viewlogic accepts descriptions without the `use` clause.

```
-viewlogic
```

To avoid having to re-code the VHDL files from Viewlogic to Exemplar, Galileo accepts an option (`-viewlogic`) that triggers the VHDL parser to adjust to the Viewlogic semantics of VHDL. In the Graphical User Interface, use the manual options line in the Global Options window to set the `-viewlogic` option. This option also makes sure that the search path for packages is changed according to the

Viewlogic rules. This search path includes the current working directory, and the `./`*lib*`/behv`, `./vhdllibs/`*lib*, `$`*WDIR*`/`*lib*`/behv` and `$`*WDIR*`/vhdllibs/`*lib* directories.

Note – The Exemplar synthesis tools do not support the old Viewlogic package `synth`. Only the packages `pack1076` and `stdsynth` are supported and recognized as the packages that define Viewlogic's synthesis functions and types.

To use Viewlogic VHDL files with Leonardo, you must set the variable `viewlogic_vhdl` to `TRUE`.

Synopsys

Users that have existing VHDL files for Synopsys VHDL Compiler will rely on one or more of the Synopsys pre-defined VHDL packages. The Exemplar synthesis tools support all these packages; a `use` clause includes them into your design. The Exemplar versions of these packages cause an implementation that is efficient for the Exemplar synthesis tools to be used.

The Synopsys packages define a set of types and functions that contain Synopsys progamas that VHDL Compiler uses as synthesis directives. These pragmas are correctly interpreted by the following Exemplar tools:

pragma translate_on
pragma translate_off
synopsys translate _on
synopsys translate_off
synopsys synthesis_on
synopsys synthesis_off

Apart from a `use` clause for each Synopsys package that you need in your VHDL file, you should NOT have to load any Synopsys package into the Exemplar synthesis tools. They will search for the packages that you want to use in the directory `$EXEMPLAR/data`. Here is the list of files with the packages they contain:

File Name	Package Name
`syn_ari.vhd`	`ARITHMETIC`
`syn_attr.vhd`	`ATTRIBUTES`
`syn_type.vhd`	`TYPES`
`syn_arit.vhd`	`STD_LOGIC_ARITH`
`syn_misc.vhd`	`STD_LOGIC_MISC`
`syn_unsi.vhd`	`STD_LOGIC_UNSIGNED`
`syn_sign.vhd`	`STD_LOGIC_SIGNED`

It is very important that you let the synthesis tools find these packages themselves (from the use clause in your VHDL description). The synthesis tools should load any of the files above from the `$EXEMPLAR/data` directory, or it will probably read a file without the synthesis directives. Without the synthesis directives, the synthesis tools can NOT efficiently synthesize any of the Synopsys packages.

The synthesis tools assume that the Synopsys libraries are called from either the VHDL library `SYNOPSYS` or the VHDL library `IEEE` (this is where Synopsys advises its packages to be stored). If you store your Synopsys library (on your VHDL simulator) somewhere else than in these libraries, they you have to manually include the (package) files needed from the `$EXEMPLAR/data` directory, since the synthesis tools will not recognize them as Synopsys packages. For Galileo, the technique to manually include such packages is to use the option `-vhdl_file=`*libname*`::`*filename* to include the files (packages) you need into the library you want. For Leonardo, use the `analyze` *libname filename* command and argument. Make sure again that you use the files from the `$EXEMPLAR/data` directory (with synthesis directive attributes in there).

Mentor Graphics

The Exemplar synthesis tools are source-code compatible with the latest version of Autologic II. Therefore, you should not encounter any problems when running VHDL designs from Mentor Graphics. The Exemplar synthesis tools support two VHDL packages from Autologic II, both of which are stored in the `$EXEMPLAR/data` directory:

File Name	Package Name
`std_arit.vhd`	`STD_LOGIC_ARITH`
`qsim_logic.vhd`	`QSIM_LOGIC`

These files will be automatically read when you specify the package names in a use clause in your VHDL description.

The Exemplar Packages

There are a number of operations in VHDL that occur regularly. An example is translation of vectors to integers and back. For this reason, Exemplar provides packages that define attributes, types, functions and procedures that are often used. Using the functions and procedures reduces the amount of initial circuitry that is generated, compared to writing the behavior explicitly in a user-defined function or procedure. This reduces the cpu-time for compilation and also could result in a smaller circuit implementation due to improved optimization.

This section discusses all the defined functionality in the Exemplar packages `exemplar` and `exemplar_1164`. The package bodies are not read by the synthesis tools; the functions are built-in. The packages are used for simulation only, and editing them will NOT change the synthesized logic. The VHDL source for these packages is given in the files `exemplar.vhd` and `ex_1164.vhd`, respectively in the `$EXEMPLAR/data` directory.

The `exemplar_1164` package defines the same functionality of the exemplar package, but operates on the IEEE 1164 multi-valued logic types.

If you are using the IEEE 1164 types in your VHDL description, include the IEEE standard logic type definition into your VHDL description with a use clause. The VHDL source of the IEEE 1164 types package is in the file `std_1164.vhd` in the

`$EXEMPLAR/data` directory. For details about the IEEE 1164 types, see “IEEE 1164 Predefined Types” on page 28. If you also want to use the Exemplar functions that operate on these types, include the package `ex_1164` with a `use` clause.

If you do not use the IEEE 1164 types, but still want to use the Exemplar functions, just include the package exemplar in your VHDL description with a use clause. All functions are then defined on the predefined types `bit` and `bit_vector`, and on the four-valued types `elbit` and `elbit_vector`.

Predefined Types

The `exemplar` package defines a four-valued type called `elbit` and its array equivalent `elbit_vector`. The `elbit` type includes the bit values `’0’`, `’1’`, `’X’` and `’Z’`.

Exemplar recommends that you use the IEEE 1164 standard logic types, and the `exemplar_1164` package. The Exemplar data types are included only for backward compatibility with Galileo releases prior to 1.2.

Predefined Attributes

The Exemplar synthesis tools use attributes to control synthesis of the described circuit. With Galileo, these attributes can be set in the control file. With Leonardo, you can use the `set_attribute` command to set object attributes within the hierarchical database.

In many cases, though, it is more convenient to define attributes in the VHDL source. The following attributes are recognized by the VHDL parser, and declared in both the `exemplar` and the `exemplar_1164` package:

Attribute	**Type**	**Description**
`required_time`	time	Set required time on output
`arrival_time`	time	Set `arrival_time` on input
`output_load`	real	Specify load set on output
`max_load`	real	Specify max load allowed on input
`clock_cycle`	time	Specify clock length on clock pin
`pulse_width`	time	Specify pulse width on clock pin
`input_drive`	time	Specify delay/unit load for input

Attribute	Type	Description
nobuf	boolean	Reject buffer insertion for a input
pin_number	string	Specify location of input or output pin
array_pin_number[1]	array of strings	Specify location for each bit of a bus
preserve_signal	boolean	Signal's function will survive synthesis
buffer_sig	string	Specify explicit buffer on a pin
modgen_sel	modgen_select	Specify time requirement for module generators driving this signal

1. This attribute can be set *only* in the VHDL source.

In order to set a particular attribute on a signal (or port) in VHDL, use the normal attribute specification statement in VHDL. Here are some examples:

```
library exemplar ;
use exemplar.exemplar.all ;  -- Include the 'exemplar' package
entity test is
      port ( my_input : in bit ;
             my_output : out bit_vector (5 downto 0) ;
   ) ;
attribute pin_number of my_input:signal is "P15" ;
attribute array_pin_number of my_output:signal is
          ("P14","P13","P12","P11","P10","P9") ;
attribute required_time of my_output:signal is 5 ns ;

end test ;

architecture exemplar of test is
      signal internal_signal : bit ;
      attribute preserve_signal of internal_signal:signal is TRUE ;
      attribute modgen_sel of internal_signal:signal is FAST ;
begin
...
```

Since variables do not represent one unique node in the circuit implementation (they represent a different circuit node after each assignment) the attributes will be effective on all circuit nodes the variable represents. This could lead to unexpected behavior. So be careful using the exemplar attributes on variables.

All attributes work both on single-bit signals and on arrays of bits. In the case an attribute is set on a signal that is an array of bits (`bit_vector`, `elbit_vector` or `std_logic_vector`) the value of the attribute is set to all circuit nodes in the vector. An exception is the `pin_number` attribute which only operates on single bit ports. Use the `array_pin_number` attribute to set pin numbers on all bits of a bus.

Predefined Functions

The package exemplar defines a set of functions that are often used in VHDL for synthesis. First of all, the package defines the overloaded operators `and`, `NAND`, `or`, `nor`, `xor`, and `not` for the types `elbit` and `elbit_vector`, as well a for `elbit_matrix`, a two-dimensional array type of `elbit` values.

The Exemplar package defines a large set of functions for both the standard `bit` and `bit_vector` types. For backwards compatibility, these functions are also defined for `elbit` and `elbit_vector` types. These functions are discussed below.

All functions are also defined with the IEEE 1164 types `std_logic`, `std_ulogic`, `std_logic_vector`, and `std_ulogic_vector` in the package `ex_1164` in file `ex_1164.vhd`.

bool2elb (l: boolean) return std_logic;

Takes a boolean, and returns a `std_logic` bit. Boolean value TRUE will become `std_logic` value `'1'`, FALSE will become `'0'`.

elb2bool (l: std_logic) return boolean;

Takes a `std_logic` value and returns a boolean. The `std_logic` value `'1'` will become TRUE, all other values become FALSE.

int2boo (l: integer) return boolean;

Takes an integer and returns a boolean. Integer value `'0'` will return FALSE, all other integer values return TRUE.

boo2int (l: boolean) return integer;

Takes a boolean and returns an integer. Boolean value TRUE will return `1`, FALSE will return `0`.

evec2int (l: std_logic_vector) returninteger;

Takes a vector of bits and returns the (positive) integer representation. The left most bit in the vector is assumed the MSB for the value of the integer. The vector is interpreted as an unsigned representation.

int2evec (l: integer, size : integer := 32) return std_logic_vector;

Takes a integer and returns the vector representation. The size of the vector becomes equal to the value of an optional second argument (size). If this argument is not specified, the size of the return vector defaults to 32. The left most bit in the resulting vector is the MSB of the returned value. If the integer value of the first parameter is negative, the MSB is the sign bit.

Note – The second parameter in the `int2evec` function is new. Prior to Galileo 2.1, `int2evec` took only a single parameter. This created simulator-synthesis inconsistencies that have been eliminated with the introduction of the second parameter. In some cases this means that Galileo 2.1 will give an array-size error on a design that used to run fine under older versions of Galileo. Make sure you add the second parameter to return the right-sized array.

elb2int (l: std_logic) return integer;

Takes a `std_logic` value and returns an integer. The `std_logic` value `'1'` will return integer value `1`, all other values will return integer value `0`.

For all shifter functions that follow, the shift amount (r) could either be a compile time constant or not. If it is, the synthesized circuit will only consist of a re-ordering of the wires in the array. Otherwise, the synthesis tools will synthesize a shifter circuit.

sl (l: std_logic_vector; r: integer) return std_logic_vector;

Takes a vector *l* and an integer *r* and returns a vector. The resulting vector is the same size as *l*, but all bits of *l* are shifted left *r* places. The bits on the right side of the result vector are zero-filled. The integer *r* must be non-negative.

sl2 (l: std_logic_vector; r: integer) return std_logic_vector;

Same as `sl`, but the vector *l* is treated as a 2-complement (signed) representation. Sign bit is the left most bit in vector. Bits on the right are zero-filled.

sr (l: std_logic_vector; r: integer) return std_logic_vector;

Same as `sl`, but bits are shifted to the right side of the vector. Bits on left side are zero-filled.

sr2 (l: std_logic_vector; r: integer) return std_logic_vector;

Same as `sr`, but the vector *l* is treated as a 2-complement representation. Sign bit is the left most bit in vector. Bits on the left side are sign-bit filled.

add (op_l, op_r: std_logic_vector) return std_logic_vector;

Takes two vectors and returns a vector. The resulting vector is one bit larger than the largest of the input vectors, and represents the addition of the input vectors, including the carry bit. The left most bit is assumed to be the MSB. The add function is a vector addition of two unsigned vectors. The smallest input vector is `'0'`, extended on the MSB side to the size of the largest input vector before addition is performed.

```
add ("1011","0100") result : "01111"    (add (11,4) == 15)
add ("0011","100")  result : "00111"     (add (3,4) == 7)
```

add2 (op_l, op_r: std_logic_vector) returnstd_logic_vector;

Same as add, but now the vectors are assumed to be in 2-complement representation. Sign bit is the left most bit in the vectors. The smallest input vector is sign-bit extended on the MSB side to the size of the largest vector before addition is performed.

```
add2 ("1011","0100") result : "00001"   (add2 (-5,4) == 1)
add2 ("0011","100")  result : "11111"   (add2 (3,-4) == -1)
```

sub (op_l, op_r: std_logic_vector) returnstd_logic_vector;

Same as add, but the subtraction function is implemented on unsigned vectors. *op_r* is subtracted from *op_l*.

```
sub  ("1011","0100")result : "00111"    (sub (11,4) == 7)
sub  ("0011","100") result : "11111"    (sub(3,4) == 31)
```

Actually this is an under-flow of unsigned !

sub2 (op_l, op_r: std_logic_vector) return std_logic_vector;

Same as add2, but the subtraction function is implemented on 2-complement representation vectors. *op_r* is subtracted from *op_1*.

```
sub2 ("1011","0100") result : "10111"    (sub2(-5,4) == -9)
sub2 ("1011", "100") result : "11111"   (sub2(-5,-4) == -1)
```

extend (op_l: std_logic_vector; op_r: integer) return std_logic_vector;

Takes a vector *op_l* and an integer *op_r* and returns a vector. The vector *op_l* is extended in size up to *op_r* elements. The input vector *op_l* is zero-extended on the MSB side. The left most bit in the vector is assumed the MSB. There is also a version of extend that takes a single (`std_logic`) value and extends it to a vector of size *op_r*.

```
extend ("1001",7)      result : "001001"
extend ('1',3)         result : "001"
extend ("011001001", 4)result : "1001"    -- Truncation
```

extend2 (op_l: std_logic_vector; op_r: integer) return std_logic_vector;

Same as `extend`, but the vector is in 2's-complement representation. The input vector is sign-bit extended. There is also a version of extend2 that takes a single (std_logic) value and sign-extends it to a vector of size *op_r*.

```
extend2 ("1001",7)         result : "1111001"
extend2 ('1',3)            result : "111"
extend2 ("011001001",4)    result : "1001" -- Truncation
```

comp2 (op: std_logic_vector) return std_logic_vector;

Takes a vector and returns a vector of the same size. This function assumes the input vector to be in 2-complement representation and will return the complement (negative) value of the input value. The right most bit is assumed to be the LSB.

```
comp2 ("1001") result : "0111"       ( comp2 (-7) == 7)
```

"+" (op_l, op_r: std_logic_vector) return std_logic_vector;

Takes two vectors and returns a vector. As add, but now the carry bit is not saved. The resulting vector is the same size as the largest input vector. Overflow wraps around. This function implements addition of unsigned vectors.

```
"10110" + "101"
          result :  "11011"       (22 + 5 == 27)
```

"-" (op_l, op_r: std_logic_vector) returnstd_logic_vector;

Same as "+", only the subtraction function is performed. *op_r* is subtracted from *op_l*. This function implements subtraction of unsigned vectors.

```
"10110" - "101"
          result : "10001"       (22 - 5 == 17)
```

"mult" (op_l, op_r: std_ulogic_vector) return std_ulogic_vector;

Takes two vectors and returns a vector. The size of the resulting vector is the size of both input vectors added. In each vector, the left most bit is the MSB. The mult function performs UNSIGNED multiplication of the two input vectors. In case of unequal-length input vectors, the smallest vector is zero-extended on the MSB side to the size of the largest input vector before the multiplication is performed.

```
mult ("1011", "0100") result: "00101100" (mult(11,4)==44)
mult ("1", "1111") result: "00001111" (mult(1,15)==15)
```

"mult2" (op_l, op_r: std_ulogic_vector) return std_ulogic_vector;

Like `mult`, but now the vectors are assumed to be in 2-complement representation. The sign bit is the left most bit in each vector. In case of unequal-length input vectors, the smallest vector is sign-bit extended on the MSB side to the size of the largest input vector before the multiplication is performed.

Predefined Procedures

There are various ways to generate flip-flops and d-latches with VHDL, such as using processes and specifying behavior that represents the behavior of flip-flops and dlatches. However, in some cases it is useful to instantiate technology independent flip-flops or dlatches in the VHDL dataflow environment immediately. A more structural oriented VHDL style will be possible that way. The exemplar package includes the definition of procedures that represent flip-flops or dlatches with various set or reset facilities that operate on single bits or vectors (to create registers).

The `exemplar` package defines these procedures on signals of type `bit`, `bit_vector`, `elbit` and `elbit_vector`, while the package `exemplar_1164` defines the same procedures for the IEEE 1164 types `std_logic`, `std_ulogic`, `std_logic_vector` and `std_ulogic_vector`. In the description below only examples for `bit` and `bit_vector` are given, but the full definition of the procedures, for the types listed above, is available for simulation purposes in the files `exemplar.vhd` and `exemplar_1164.vhd`.

Flip-flops

```
dff[_v](data, clock, q)
dffc[_v](data, clear, clock, q)
dffp[_v](data, preset, clock, q)
dffpc[_v](data, preset, clear, clock, q)
```

Here `dff` is the single bit D flip-flop and `dff_v` is the vectored D flip-flop. `dff` has no preset or clear inputs, `dffc` has an active-high asynchronous clear (set `q` to ’0’) input, `dffp` has an active-high asynchronous preset (set `q` to ’1’) input, and `dffpc` has both a preset and a clear input. If both preset and clear are asserted, `q` is not defined. All inputs are active high, the clock input is positive edge triggered. For the vectored dffs, the number of flip-flops that will be instantiated is defined by the size of the input (`d`) and output (`q`) vectors of the `dff#_v` instantiation. (The size of `d` and `q` vectors must be the same.)

If `q` is a port of the VHDL entity, it must be declared as an INOUT port, since `q` is used bidirectionally in each of these functions.

Latches

```
dlatch[_v](data, enable, q)
dlatchc[_v](data, clear, enable, q)
dlatchp[_v](data, preset, enable, q)
dlatchpc[_v](data, preset, clear, enable, q)
```

These define a level sensitive D-type latch with an enable. The latch is enabled (transparent) when the enable input is 1, disabled when the input is 0. `dlatch` has no preset or clear capability, `dlatchc` has an asynchronous active-high clear (set `q` to `'0'`) input, dlatchp has an asynchronous active-high preset (set `q` to `'1'`), and `dlatchpc` has both preset and clear. If both preset and clear are asserted, `q` is not defined. `dlatch_v` creates the vector equivalent procedures to generate registers of dlatches.

Tristate Busses

When a signal is assigned in multiple concurrent statements, the synthesis implementation requires that in each statement the signal is assigned a `'Z'` value under at least one condition. A tristate gate is created in this case, with the enable of the gate corresponding to the inverse of the condition where the `'Z'` is assigned in the model. This is the only case where multiple assignments to a signal in different concurrent statements is allowed.

It is also possible for the user to specify what to do in the case where none of the drivers of the bus are enabled. To address this situation, three pre-defined procedures have been declared to handle the three standard tristate bus conditions: `PULLUP`, `PULLDN` and `TRSTMEM`. These drive an otherwise undriven bus to the values `1`, `0`, or `retain the current value`, respectively. Only one of these functions may be specified for a given bus. The synthesis tools will build the appropriate logic to implement the specified function in the technology. If the technology includes pull-up or pull-down resistors or repeater cells on internal busses these will be used. If they are not available, an additional tristate gate, whose enable is the NOR of all the other enables and whose input is either VCC, GND or the value on the bus will be created to implement the specified function. The synthesis tools also know what the default state for a bus is in the technology, and if that matches the specified function, no extra logic is created. If no termination is specified, then its undriven value depends on the technology used.

The tristate bus procedures defined below may be used with signals of type `bit`, `elbit`, (package `exemplar`) `std_logic` and `std_ulogic` (package `ex_1164`).

pullup(busname)

When a bus is not driven, this procedure will pull the bus up to 1.

pulldn(busname)

When a bus is not driven, this procedure will pull the bus down to 0.

trstmem(busname)

When a bus is not driven, this procedure will drive the bus to its last driven state.

Syntax and Semantic Restrictions

VHDL as the IEEE Standard 1076 is a extended language with many constructs that are useful for simulation. However, during the initial development of the language, logic synthesis was not taken into account. Therefore, a number of constructs or combination of constructs cannot be implemented in actual circuits.   VHDL 1076 is fully simulatable, but not fully synthesizable.

Synthesis Tool Restrictions

This section discusses the syntax and semantic restrictions of the VHDL parsers of the Exemplar synthesis tools.

- Operations on files not supported. Files in VHDL could behave like ROMs or RAMs, but the synthesis tools do not support using file (types). The synthesis tools will ignore, but accept, file (type) declarations.
- Operations on objects of `real` types are not supported. Objects of `real` types have no defined bit-resolution. The synthesis tools will ignore, but accept, declarations of (objects of) `real` types.
- Operations on objects of `access` types are not supported, since they lead to unsynthesizable behavior. The synthesis tools will ignore, but accept, declarations of (objects of) `access` types.

- Attributes `BEHAVIOR`, `STRUCTURE`, `LAST_EVENT`, `LAST_ACTIVE`, and `TRANSACTION` are not supported.
- Configurations are ignored; default component binding (by name) is assumed.
- Global, non-constant signals are not supported, that is, signals declared in a package.
- Allocators are not supported, because they perform dynamic allocation of resources, which is not synthesizable.
- Configuration declarations are ignored. The synthesis tools allow only entities or components as the main building blocks of the design. Configuration specifications (binding a component (instance) to an entity) ARE supported.
- `REGISTER` and `BUS` signal declarations are not supported. Only resolution functions with a synthesis directive are allowed (see the section “BUS and REGISTER” on page 55).

VHDL Language Restrictions

Apart from these restrictions, which are mostly tool-related, there are some basic restrictions that apply to VHDL descriptions for synthesis. Since they occur quite often, additional descriptions are presented here to clarify the problems involved for synthesis. Here is the list:

- `after` clause ignored.
- Restrictions on Initialization values.
- Ranges of loops have to evaluate to constants during compile time.
- Restrictions on edge-detecting attributes (EVENT and STABLE).
- Restrictions on wait statements.
- Restrictions on multiple drivers on one signal.

A more detailed description of these restrictions follows below:

After Clause Ignored

The `after` clause refers to delay in a signal. Since delay values cannot be guaranteed in synthesis, they are ignored by the synthesis tools after they issue a warning.

Restrictions on Initialization Values

Initialization values are allowed in a number of constructs in VHDL:

1. Initial value of a signal in a signal declaration.
2. Initial value of a variable in a variable declaration in a process.
3. Initial value of a variable in a variable declaration in a subprogram (procedure or function).
4. Initial value of a generic or port in a component declaration.
5. Initial value of a parameter in a subprogram interface list.

The problem with initialization values for synthesis is that some initial values define the initial value of an object before actual simulation is done. This behavior corresponds to controlling the power-up state of a device that would be synthesized from the VHDL description. Since synthesis cannot control the power-up state of a device, this kind of initial value cannot be synthesized. However, if after initialization there is never an change of value, the behavior can be synthesized, and resembles a simple constant value.

The synthesis tools fully support initialization values, except for initializing objects that can change their value after initialization. That is, the following form of initialization values are NOT supported because they imply power-up behavior of the synthesized device:

1. Initial values of a signal in a signal declaration.
2. Initial value of a variable in a variable declaration in a process.
3. Initial value of an OUTPUT or INOUT port in an interface list.

All other forms of initialization values are supported by the synthesis tools.

Ranges Of Loops Have To Evaluate To Constants During Compile Time

Loops with no compile time bounds (especially infinite loops) have no RTL logic representation. Therefore, make sure that the loop bounds depend on “constant” values like the bounds of a vector or ordinary decimal literals. The attributes `'LEFT`, `'RIGHT`, `'RANGE`, etc. are normally sufficient to indicate bounds of a loop.

Restrictions On Edge-Detecting Attributes ('event)

Most restrictions on VHDL to assure correct compilation into a logic circuit are on the constructs that define edges or changes on signals. The `'EVENT` attribute is the best example of this. *signal*`'EVENT` is TRUE only if *signal* changes. Then it is TRUE for one simulation delta of time. In all other cases it is FALSE. The `STABLE` attribute is the boolean inversion of `EVENT`.

There are two restrictions for synthesis on usage of the `EVENT` and the `STABLE` attribute:

1. An `EVENT` or `STABLE` attribute can be used only to specify a leading or falling clock edge. For example:

```
clk'event and clk='1'      -- Leading edge of clk
clk'event and clk='0'      -- Falling edge of clk
NOT clk'stable and clk='0' -- Falling edge of clk
clk'event and clk          -- Leading edge of (boolean) clk
```

2. Clock edge expressions can only be used as conditions. For example:

```
if (clk'event and clk='1') then ...
wait until NOT clk'stable and clk='0' ;
wait until clk='1' ;          --Implicit clock edge due to
                              --VHDL semantics of 'wait'
block (clk'event and clk='1'... --Block GUARD condition
```

These restrictions originate from the fact that binary logic circuits have a restricted number of elements that are active ONLY during signal edges. Basically, only (set/resettable) edge triggered flip-flops show that behavior. Within these restrictions, the synthesis tools allow free usage of the clock edge conditions, either in guarded blocks, processes or subprograms.

Restrictions on Wait Statements

All state-of-the-art VHDL synthesis tools on the market right now have strong restrictions with respect to wait statements and use of edge-detecting attributes (`'event` and `'stable`). Here are the (informal) restrictions for the wait statement:

- Only one wait (until) statement is allowed in a process.
- That wait (until) statement (if present) must be the first or last statement in the process.
- The expression in the "until" condition must specify a leading or falling single clock edge. (Examples are shown above in the `EVENT` attribute section.)

All assignments inside the process result in the creation of registers. Each register (flip-flop) is clocked with the single clock signal.

There are a number of cases where multiple waits are synthesizable and resemble state-machine behavior. In the Exemplar synthesis tools, however, multiple waits are not supported. State machine behavior, however, can always be re-written to a `case` statement and register process, as explained in "State Machines" on page 18.

Restrictions on Multiple Drivers on One Signal

VHDL does not allow multiple drivers on a signal of an unresolved type. For signals of resolved types, VHDL defines that a (user-defined) resolution function defines what the signal value is going to be in case there are multiple driver (simultaneous assignments) to the signal.

A resolution function with meta-logical values (`'Z'`, `'X'`, etc.) in general leads to behavior that is not synthesizable (since logic circuits cannot produce meta-logical values). Therefore, in general, VHDL synthesis tools do not allow multiple drivers on a signal. However, if the resolution function defines the behavior of multiple three-state drivers on a bus, multiple drivers of a signal could represent synthesizable behavior.

The `'Z'` value is in general used to identify three-state behavior. The resolution function of the IEEE `std_logic` (resolved) type is written so that multiple drivers on a signal of `std_logic` do resemble multiple three-state drivers on a bus. Therefore, the synthesis tools accept multiple assignments to the same signal as long as each assignment is conditionally set to the `'Z'` value. The synthesis tools allow free usage of `'Z'` assignments (either from dataflow statements, process statements or from within procedures). The synthesis tools will implement three-state drivers to mimic the three-state behavior.

It is important to note that the synthesis tools do not check if there could be a bus-conflict on the driven bus. In this case, the simulation would just call the resolution function again to resolve the value (normally producing a meta-logical value), but the behavior for synthesis is not defined. Avoiding bus conflicts is the responsibility of the user.

Introduction to Verilog HDL Synthesis 5

Verilog HDL is a high level description language for system and circuit design. The language supports various levels of abstraction. Where a regular netlist format supports only structural description, Verilog supports a wide range of description styles. This includes structural descriptions, data flow descriptions and behavioral descriptions.

The structural and data flow descriptions show a concurrent behavior. All statements are executed concurrently, and the order of the statements does not matter. On the other hand, behavioral descriptions are executed sequentially in always blocks, tasks and functions in Verilog. The behavioral descriptions resemble high-level programming languages.

Verilog allows a mixture of various levels of design entry. The Exemplar synthesis tools synthesize all levels of abstraction, and minimizes the amount of logic needed, resulting in a final netlist description in the technology of your choice.

The high level design flow enabled by the use of the Exemplar synthesis tools is shown in Figure 5-1.

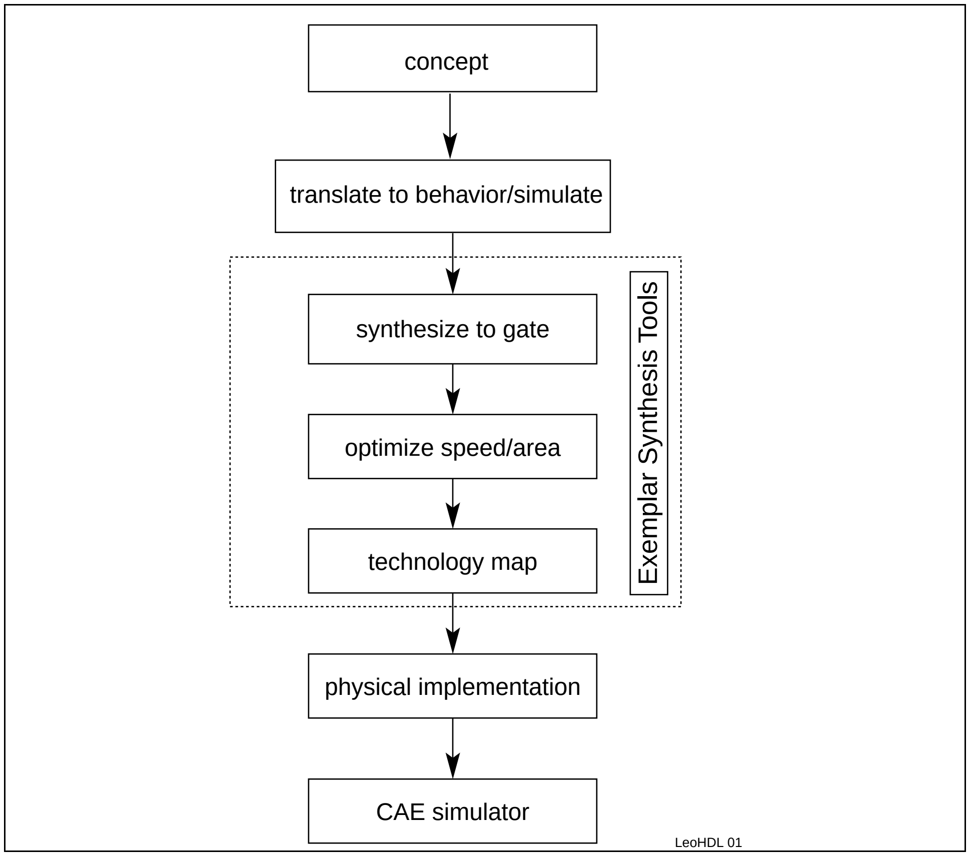

Figure 5-1 Top-Down Design Flow with Exemplar Synthesis Tools

Verilog and Synthesis

Verilog is completely simulatable, but not completely synthesizable. There are a number of Verilog constructs that have no valid representation in a digital circuit. Other constructs do, in theory, have a representation in a digital circuits, but cannot be reproduced with guaranteed accuracy. Delay time modeling in Verilog is an example of that.

State-of-the-art synthesis algorithms can optimize Register Transfer Level (RTL) circuit descriptions and target a specific technology. Scheduling and allocation algorithms, that perform circuit optimization at a very high and abstract level, are not yet available for general circuit applications. Therefore, the result of synthesis of a Verilog description depends on the style of Verilog that is used. Users of the Exemplar synthesis tools should understand some of the concepts of synthesis specific to Verilog coding style at the RTL level, in order to achieve the desired circuit implementation.

What synthesis tools do best then is to automatically solve many of the cumbersome RTL logic optimization problems that occur during a typical top-down design project.

This manual is intended to give the Verilog designer guidelines to achieve a circuit implementation that satisfies the timing and area constraints that are set for the target circuit, while still using a high level of abstraction in the Verilog source code. This goal will be discussed both in the general case for synthesis applications, as well as for the Exemplar synthesis tools specifically. Examples are used extensively; Verilog rules are not emphasized.

Knowledge of the basic constructs of Verilog is assumed, although one chapter is dedicated to the discussion of all the constructs in Verilog that are useful for synthesis. For more information on the Verilog language, refer to the *Verilog Hardware Description Language Reference Manual*, published by Open Verilog International.

Synthesizing the Verilog Design

Using the Exemplar synthesis tools to synthesize your Verilog design is easy. If you run Galileo from the command line, use the following option:

```
-input_format=verilog
```

If you run Leonardo from the command line, use the following command and argument:

```
read -format verilog file_name
```

If using the graphical user interface, use the interface to choose “Verilog as the Input Format.” Target technology and other options are chosen as usual with the synthesis tools.

Verilog Language Features 6

This chapter provides an introduction to the basic language constructs in Verilog: defining logic blocks:

- Data flow and behavioral descriptions
- Concurrent and sequential functionality
- Numbers and data types.

The Exemplar synthesis tools synthesize all levels of abstraction and minimizes the amount of logic needed resulting in a final netlist description in the technology of your choice.

Modules

A basic building block in Verilog is a module. The module describes both the boundaries of the logic block and the contents of the block, in structural, data flow and behavioral constructs.

```
module small_block (a, b, c, o1, o2);
input a, b, c;
output o1, o2;
wire s;
       assign o1 = s || c ;
       assign s = a && b ;
       assign o2 = s ^ c ;
endmodule
```

This Verilog description shows the implementation of `small_block`, a block that describes some simple logic functions.

The port list is declared, the port directions are specified, then an internal `wire` is declared. A `wire` in Verilog represents physical connection in hardware. It can connect between `modules` or gates, and does not store a value. A `wire` can be used anywhere inside the `module`, but can only be assigned by:

- Connecting it to an output of a gate or a `module`.
- Assigning to it using a continuous assignment.

This `module` contains only dataflow behavior. Dataflow behavior is described using continuous assignments. All continuous assignments are executed concurrently, thus the order of these assignments does not matter. This is why it is valid to use `s` before `s` is assigned. In the first statement `o1` is assigned the result of the logical OR of `s` and `c`. "| |" denotes the logical OR operation.

More details about the various dataflow statements and operators are given in the following sections.

The Exemplar synthesis tools support empty top level modules.

'macromodule'

The Exemplar synthesis tools support 'macromodule', which is treated as 'module'.

Numbers

Numbers in Verilog can be either constants or parameters. Constants can be either sized or unsized. Either one can be specified in binary, octal, hexadecimal, or decimal format.

Name	Prefix	Legal Characters
binary	’b	01xXzZ_?
octal	’o	0-7xXzZ_?
decimal	’d	0-9_
hexcadecimal	’h	0-9a-fA-FxXzZ_?

If a prefix is preceded by a number, this number defines the bit width of the number, for instance, `8’b 01010101`. If no such number exists, the number is assumed to be 32 bits wide. If no prefix is specified, the number is assumed to be 32 bits decimal.

The synthesis tools produce a warning when encountering non-synthesizable constants such as float. The value 0 is assumed.

For example, in

```
x = 2.5 + 8;
```

`x` will evaluate to 8.

Special characters in numbers:

“_”	a separator to improve readability.
’x’, ’X’	unknown value.
’z’, ’Z’, ’?’	tri-state value.

Examples:

334	32 bits wide decimal number
’b101	32 bits wide binary number
3’b11	3 bits wide binary number

20’h’ff_fff	20 bits wide hexcadecimal number
10’bZ	10 bits wide all tri-state

Data Types

Verilog defines three main data types:

- net
- register
- parameter

By default these data types are scalars, but all can take an optional range specification as a means of creating a bit vector. The range expression is of the following form:

```
[<most significant bit> : <least significant bit>]
```

Some of these data types are used in the example below, along with the range expression syntax. Further details on the data types are presented in the following sections.

```
//  This design implements a Manchester Encoder
//
module manenc (clk , data , load , sdata, ready);
parameter max_count = 7;

input clk, load;
input [0:max_count] data;
output sdata, ready ;

reg sdata, ready ;
reg [2:0] count;
reg [0:max_count] sout;
reg phase;

// Phase encoding
always @ (posedge clk)
   begin
      sdata = sout[max_count] ^ phase;
      phase = ~phase ;
   end
```

```
// Shift data
always @ (posedge phase)
   begin
       if ((count == 0) & !load) begin
             sout[1 : max_count] = sout[0 : max_count - 1];
             sout[0] = 1’b0;
             ready   = 1’b1;
      end
      else if ((count == 0) & load ) begin
             sout  = data;
             count = count + 1;
             ready = 1’b0;
      end
      else if (count == max_count) begin
             sout[1 : max_count] = sout[0 : max_count - 1];
             sout[0]= 1’b0;
             count = 0;
      end
      else begin
             sout[1 : max_count] = sout[0 : max_count - 1];
             sout[0]= 1’b0;
             count = count + 1;
      end
   end
endmodule
```

Net Data Types

The net data types supported by the Exemplar synthesis tools are

- `wire`
- `tri`
- `supply0`
- `supply1`
- `wand`
- `wor`

These data types are used to represent physical connections between structural entities in the Verilog design, such as a wire between two gates, or a tristate bus. Values cannot be assigned to net data types within `always` blocks. (`tri0`, `tri1`, `triand`, `trior` and `trireg` are also net data types, but are not yet supported by the synthesis tools).

wire and tri Nets

The `wire` and `tri` net data types are identical in usage (syntax and function). The two different names are provided for design clarity. Nets driven by a single gate are usually declared as `wire` nets, as shown in "Modules" on page 2 in this chapter, while nets driven by multiple gates are usually declared as `tri` nets.

Supply Nets

The `supply1` and `supply0` net data types are used to describe the power (VCC) and ground supplies in the circuit. For example, to declare a ground net with the name GND, the following code is used:

```
supply0 GND ;
```

wand and wor Net Types

wand and wor statements result into and or logic respectively, since wired logic is not available in all technologies.

```
wor out;
out = a&b
out = c&d;
endmodule
```

Register Data Type

A register, declared with keyword `reg`, represents a variable in Verilog. Where net data types do not store values, `reg` data types do. Registers can be assigned only in an `always` block, task or function. When a variable is assigned a value in an `always` block that has a clock edge event expression (`posedge` or `negedge`), a flip-flop is synthesized by the synthesis tools. To avoid the creation of flip-flops for `reg` data types, separate the combinational logic into a different `always` block (that does not have a clock edge event expression as a trigger).

Parameter Data Type

The parameter data type is used to represent constants in Verilog. Parameters are declared by using the keyword `parameter` and a default value. Parameters can be overridden when a module is instantiated.

Declaration Local to Begin-End Block

Local declaration of registers and integers is allowed inside a named `begin-end` block.

```
input [10:0] data;
always @ (data)
begin: named_block
integer i;
   parity = 0;
   for (i = 0; i < 11; i= i + 1)
   parity = parity ^ data[i];
end //named_block
```

Array of reg and integer Declaration (Memory Declaration)

Declaration and usage of an array of registers or integers is now allowed.

```
input [0:3] address;
input [0:7] date_in;
output [0:7] data_out;
reg [0:7] data_out, mem [0:15];
always @ (address or date_in or we)
   if (we) mem [address] = date_in;
   else data_out = mem [address];
```

Continuous Assignments

A continuous assignment is used to assign values to nets and ports. The nets or ports may be either scalar or vector in nature. (Assignments to a bit select or a constant part select of a vector are also allowed.) Because nets and ports are being assigned values, continuous assignments are allowed only in the dataflow portion of the module. As such, the net or port is updated whenever the value being assigned to it changes.

Continuous assignments may be made at the same time the net is declared, or by using the `assign` statement.

Net Declaration Assignment

The net declaration assignment uses the same statement for both the declaration of the net and the continuous assignment:

```
wire [0:1]sel = selector ;
```

Only one net declaration assignment can be made to a specific net, in contrast to the continuous assignment statement, where multiple assignments are allowed.

Continuous Assignment Statement

The continuous assignment statement (`assign`) is used to assign values to nets and ports that have previously been declared.

The following example describes a circuit that loads a source vector of 4 bits on the edge of a clock (`wrclk`), and stores the value internally in a register (`intreg`) if the chip enable (`ce`) is active. One bit of the register output is put on a tristate bus (`result_int`) based on a bit selector signal (`selector`), with the bus output clocked through a final register (`result`).

```
module tri_asgn (source, ce, wrclk, selector, result) ;
input [0:3]source ;
input ce, wrclk ;
input [0:1]selector ;
output result ;
reg [0:3]intreg ;
reg result ;
// net declaration assignment
wire [0:1]sel = selector ;
tri result_int ;

// continuous assignment statement
        assign
            result_int = (sel == 2'b00)? intreg[0] : 1'bZ ,
            result_int = (sel == 2'b01)? intreg[1] : 1'bZ ,
            result_int = (sel == 2'b10)? intreg[2] : 1'bZ ,
            result_int = (sel == 2'b11)? intreg[3] : 1'bZ ;
always @(posedge wrclk)
begin
        if (ce)
        begin
                intreg = source;
                result = result_int ;
        end
end

endmodule
```

Procedural Assignments

Procedural assignments are different from continuous assignments in that procedural assignments are used to update register variables. Assignments may be made to the complete variable, or to a bit select or part select of the register variable.

Both blocking and non-blocking procedural assignments are allowed.

Blocking assignments, specified with the “=” operator, are used to designate assignments that must be executed before the execution of the statements that follow it in a sequential block. This means that the value of a register variable in a blocking assignment is updated immediately after the assignment.

Non-blocking assignments, specified with the “<=” operator, are used to schedule assignments without blocking the procedural flow. It can be used whenever register assignments within the same time step can be made without regard to order or dependence upon each other. Also, in contrast to the blocking assignment, the value of a register variable in a non-blocking assignment is updated at the end of the time step. This behavior does not affect assignments done in the dataflow environment, since assignments are done concurrently there. However, in a sequential block, such as an always block, the value of the variable in a non-blocking assignment changes only after the complete execution of the sequential block.

Refer to the *Verilog Language Reference Manual* for more information on non-blocking procedural assignments.

Always Blocks

`Always` blocks are sections of sequentially executed statements, as opposed to the dataflow environment, where all statements are executed concurrently. In an `always` block, the order of the statements DOES matter. In fact, `always` blocks resemble the sequential coding style of high level programming languages. Also, `always` blocks offer a variety of powerful statements and constructs that make them very suitable for high level behavioral descriptions.

An `always` block can be called from the dataflow area. Each `always` block is a sequentially executed program, but all `always` blocks run concurrently. In a sense, multiple `always` blocks resemble multiple programs that can run simultaneously.

Always blocks communicate with each other via variables of type reg which are declared in the module. Also, the ports and wires defined in the module can be used in the always blocks.

```
module mux_case (source, ce, wrclk, selector, result);
input [0:3]source;
input ce, wrclk;
input [0:1]selector;
output result;
reg [0:3]intreg;
reg result, result_int;

always @(posedge wrclk)
begin
        if (ce)
                intreg = source;
        result = result_int;
end

always @(intreg or selector)
        case (selector)
                2'b00: result_int = intreg[0];
                2'b01: result_int = intreg[1];
                2'b10: result_int = intreg[2];
                2'b11: result_int = intreg[3];
        endcase

endmodule
```

This example describes a circuit that can load a source vector of 4 bits, on the edge of a write clock (wrclk), store the value internally in a register (intreg) if a chip enable (ce) is active, while it produces one bit of the register constantly (not synchronized). The bit is selected by a selector signal of 2 bits, and is clocked out through the register result.

The description consists of two always blocks, one to write the value into the internal register and clock the output, and one to read from it. The two always blocks communicate via the register values intreg and result_int.

The first `always` block is a synchronous block. As is explained later, the `always` block executes only if the event expression at the event control evaluates to true. In this case, the event expression evaluates to true when a positive edge occurs on the input `wrclk` (event expression `posedge wrclk`). Each time the edge occurs, the statements inside the `always` statement are executed. In this case, the value of the input `source` is loaded into the internal variable `intreg` only if `ce` is `'1'`. If `ce` is `'0'`, `intreg` retains its value. In synthesis terms, this translates into a D flip-flop, clocked on `wrclk`, and enabled by `ce`. Also, the intermediate output `result_int` is loaded into the output `result` (a D flip-flop clocked on `wrclk`).

The second `always` block is a combinational block. In this case, the event expression evaluates to true when either `intreg` or `selector` changes. When this happens, the statements inside the `always` statement are executed, and the output `result_int` gets updated depending on the values of `intreg` and `selector`. Note that this leads to combinational behavior (essentially a multiplexer), since `result_int` only depends on `intreg` and `selector`, and each time either of these signals changes, `result_int` gets updated.

The reason for separating the two `always` blocks is to avoid the creation of a register for the variable `result_int`. `result_int` must be of `reg` data type, because it is assigned in an `always` block, but it does not need to be registered logic.

Not all constructs, or combinations of constructs, in an `always` block lead to behavior that can be implemented as logic. More information about synthesizable Verilog constructs is given in Chapter 7, “The Art of Verilog Synthesis.”

The Exemplar synthesis tools support empty `always` statements.

Note that constants on the sensitivity list have no effect in simulation or synthesis. Any kind of expression inside a sensitivity list is legal in Verilog and is accepted by the synthesis tools. For synthesis, all the leaf level identifiers of the expression are considered to be in the sensitivity list, so some simulation mismatch might be seen after synthesis.

```
always @ (inp1[0:2] or 3'b011 or {a, b}) // allowed
.........
.........
```

Module Instantiation

Module instantiation can be used to implement individual gates or cells, macros, or to add hierarchy to your design. Here is an example that generates an address for RAM and instantiates the RAM cells:

```
module scanner (reset, stop, load, clk, load_value, data) ;
input reset, stop, load, clk;
input [3:0]load_value;
output [3:0]data;
reg [4:0] addr;

//  Instantiate and connect 4 32x1-bit rams
   RAM_32x1 U0 (.a(addr), .d(load_value[0]), .we(load), .o(data[0]) );
   RAM_32x1 U1 (.a(addr), .d(load_value[1]), .we(load), .o(data[1]) );
   RAM_32x1 U2 (.a(addr), .d(load_value[2]), .we(load), .o(data[2]) );
   RAM_32x1 U3 (.a(addr), .d(load_value[3]), .we(load), .o(data[3]) );

//  Generate the address for the rams
always @(posedge clk or posedge reset)
begin
      if (reset)
            addr = 5’b0 ;
      else if (~stop )
            addr = addr + 5’b1 ;
end
endmodule

module RAM_32x1 ( a, we, d, o);
input [4:0] a;
input we, d ;
output o;
endmodule
```

For this example, if the RAM module `RAM_32x1` is a cell or macro in a library, the synthesis tools will implement that cell or macro in the output netlist. To do that, the library in which the cell or macro exists must be specified as the Input Design Technology. If no Input Design Technology is specified, the synthesis tools implement the RAM module as a black box in the output netlist, with inputs and outputs defined, but no functionality.

Note – Galileo and Leonardo use different techniques to indicate which source technology to use. Galileo uses the `-source=`*lib_name* switch. Leonardo requires that you load the source technology by using the `load_library` *lib_name* command before reading the design in the database.

The Exemplar synthesis tools support empty named port connections, e.g.,

```
nd2 x1 (.a(f), .b());
```

Parameter Override During Instantiation of Module

Parameter overriding during module instantiation (as shown in the example) is supported by the synthesis tools.

```
module top (a, b);
input [0:3] a;
output [0:3] b;
   do_assign #(4) name (a, b);
endmodule
module do_assign (a, b);
   parameter n = 2;
   input [0:n-1] a;
   output [0:n-1] b;
      assign b = a;
endmodule
```

Defparam Statement

When using the defparam statement, parameter values can be changed in any module instance throughout the design, provided the hierarchical name of the parameter is used.

NOTE: In the synthesis tool, the hierarchical name is restricted to single level only. This means that when the defparam statement is used, the user will be able to override any parameter value of an instance in the current module only.

Example:

```
module top (a, b);
input [0:3] a;
output [0:3] b;
wire top;

     do_assign name (a, b);
     defparam name.n = 4;
endmodule

module do_assign (a, b);
parameter n = 2;
input [0:n-1] a;
output [0:n-1] b;

     assign b = a;
endmodule
```

’unconnected_drive’ and ’nounconnected_drive’

These directives are specified as outside modules only. ’`unconnected_drive`’ takes either `pull0` or `pull1` as a parameter and causes all the unconnected input ports to be pulled down or up, according to the parameter. ’`nounconnected_drive`’ restores the normal condition (where the unconnected input ports are connected to high-Z).

```
’unconnected_drive’ pull1
module with_unconn_port (o, i);
output o;
input i;
assign o = i;
endmodule
’nounconnected_drive’
module test (i, o1, o2);
input i;
output o1, o2;
with_unconn_port I1 (o1,);    // o1 = 1
with_unconn_port I2 (o2, i); // o2 = i
endmodule
```

Operators

This section describes the operators available for use in Verilog expressions. Before discussing operators, a brief summary of the operands that the operators act on is appropriate.

Operands

An operand in an expression can be one of the following:

- Number
- Net (including bit-select and part-select)
- Register (including bit-select and part-select)
- A call to a function that returns any of the above

Bit-selects take the value of a specific bit from a vector net or register. Part-selects are a set of two or more contiguous bits from a vector net or register. For example:

```
...
wire bit_int ;
reg [0:1] part_int ;
reg [0:3] intreg;

bit_int = intreg[1] ;   // bit-select of intreg assigned to bit_int
part_int = intreg[1:2] ;// part-select of intreg assigned to part_int
...
```

The operators supported by the Exemplar synthesis tools are listed in Table 6-1.

Table 6-1 Verilog Language Operators

Operator	**Description**
+ - * /	arithmetic
< > <= >=	relational
==	logical equality
!=	logic inequality
!	logical negation
&&	logical and
||	logical or
~	bit-wise negation
&	bit-wise and
|	bit-wise inclusive or
^	bit-wise exclusive or
^~ or ~^	bit-wise equivalence
&	reduction and
|	reduction or
^	reduction xor
<<	left shift
>>	right shift
? :	conditional
{}	concatenation

Arithmetic Operators

The Exemplar synthesis tools support the following arithmetic operators:

```
+       -       *       /
```

If the bit value of any operand is ‘X’ (unknown), then the entire resulting value is ‘X’. The “/” operator is supported in the case where the divisor is a constant and a power of two.

Relational and Equality Operators

The Exemplar synthesis tools support the following relational and equality operators:

```
<       >       <=      >=      ==      !=
```

If the bit value of any operand is ‘X’ (unknown), then the entire resulting value is ‘X’.

=== and !== Operators are Treated as == and !=

=== and !== operators are treated as == and != for synthesis purposes if either one of the operands is nonconstant. If both the operands are constant, they can be used to compare metalogical values. In simulation, the difference between == and === is that one can compare metalogical characters exactly with === but not with ==. Any metalogical character causes the output of == to be unknown x. The difference between != and !== is the same.

```
module triple_eq_neq (in1, in2, O);
output [0:10] O;
input [0:2] in1, in2;
assign
   O[0] = 3'b0x0 === 3'b0x0, // output is 1
   O[1] = 3'b0x0 !== 3'b0x0, // output is 0
   O[2] = 3'b0x0 === 3'b1x0, // output is 0
   O[3] = 3'b0x0 !== 3'b1x0, // output is 1O[4]=in1===3'b0x0,
                            // LHS is non constant so this
                              // produces warning that comparison
                              // metalogical character is
                              // with zero. output is 0
   O[5] = in1 !== 3'b0x0,     // LHS is non constant so this
                              // produces warning that comparison
                              // with metalogical character is
                              // zero.output is 1,because it
                              // checks for not equality
   O[6] = in1 === 3'b010,     // normal comparison
   O[7] = in1 !== 3'b010,     // normal comparison
   O[8] = in1 === in2,        // normal comparison
   O[9] = in1 !== in2,        // normal comparison
   O[10] = 3'b00x === 1'bx;  // output is 1
endmodule
```

Logical Operators

The Exemplar synthesis tools support the following logical operators:

```
!    &&    ||
```

Bit-Wise Operators

The Exemplar synthesis tools support the following `bit_wise` operators:

```
~      &      |      ^      ^~      ~^
```

These operators perform bit-wise operations on equivalent bits in the operands.

Reduction Operators

The Exemplar synthesis tools support the following reduction operators:

```
&        |        ^
```

These operators perform reduction operations on a single operand. Operations are performed on the first and second bits of the operand, then on the result of that operation with the third bit of the operand, until the limit of the vector is reached. The result is a single bit value.

The following operators:

```
~&       ~|       ~^
```

are negations of the “&”, “|”, and “^” operators.

Shift Operators

The Exemplar synthesis tools support the following shift operators:

```
<<       >>
```

Conditional Operator

The conditional operator statement has the following syntax:

```
conditional_expression ? true_expression : false_expression
```

The result of this operation is `true_expression` if `conditional_expression` evaluates to true, and `false_expression` if false. In the following example, result is assigned the value of `intreg[0]` if `sel = 2’b00`, otherwise result is assigned `Z`:

```
...
output result ;
reg [0:3} intreg ;
wire [0:1] sel ;
      assign result = (~sel[0] && ~sel[1]) ? intreg[0] : 1’bZ ;
...
```

Concatenation

The concatenation of bits from multiple expressions is accomplished using the characters { and }. For example, the following expressions are equivalent:

```
foo = {a[4:3], 1’b0, c[1:0]} ;
foo = {a[4], a[3], 1’b0, c[1], c[0]} ;
```

For a = 5’b11010, c = 5’b10101, the result is foo = 5’b11001.

‘signed and ‘unsigned Attributes on Operators

`‘signed` and `‘unsigned` attributes change the type of a particular operator. Comparison between two bit vectors are always done unsigned, but if the functionality needs to be signed, a `‘signed` attribute can be used just after the comparator.

```
input [0:3] A, B;
output o;
assign o = A < ‘signed B; // Signed comparator.
```

Similarly, an `‘unsigned` attribute can be used to perform an unsigned operation between two integers.

The shift operators always do a logical shift. By using the `'signed` directive, they can be made to do an arithmetic shift. Arithmetic right shift shifts in the sign bit and the left shift shifts in the least significant bit (e.g., `4'b0001 << 'signed 1` produces `4'b0011`).

Operator Precedence

The operator precedence rules determine the order in which operations are performed in a given expression. Parentheses can be used to change the order in an expression. The operators supported by the synthesis tools are listed below in order from highest precedence to lowest, with operators on the same line having the same precedence.

```
+      -      !      ~      (unary)
*      /      (binary)
+      -      (binary)
<<     >>
<      >      <=     >=
==     !=
&
^      ^~     ~^
|
&&
||
? :    (ternary)
```

Statements

This section presents information on the use of `if-else`, `case` and `for` statements for specifying designs.

If-Else Statements

The `if-else` conditional construct is used to specify conditional decisions. As an example, here is the design from “Procedural Assignments,” with the multiplexer described with this construct instead of the `case` statement:

```
module mux_case (source, ce, wrclk, selector, result);
input [0:3]source;
input ce, wrclk;
input [0:1]selector;
output result;
reg [0:3]intreg;
reg result, result_int;

always @(posedge wrclk)
begin
// if statement for chip enable on register
   if (ce)
      intreg = source;
   result = result_int;
end
```

```
always @(intreg or selector)
begin
// if-else construct for multiplexer functionality
   if (sel == 2'b00)
         result_int = intreg[0] ;
   else if (sel == 2'b01)
         result_int = intreg[1] ;
   else if (sel == 2'b10)
         result_int = intreg[2] ;
   else if (sel == 2'b11)
         result_int = intreg[3] ;
end

endmodule
```

This example describes a circuit that can load a source vector of 4 bits, on the edge of a write clock (`wrclk`), store the value internally in a register (`intreg`) if a chip enable (`ce`) is active, while it produces one bit of the register constantly (not synchronized). The bit is selected by a selector signal of 2 bits, and is clocked out through the register result.

Case Statements

If many conditional clauses have to be performed on the same selection signal, a `case` statement is a better solution than the `if-else` construct. The following example describes a traffic light controller (state machine with binary encoding):

```
module traffic (clock, sensor1, sensor2,
       red1, yellow1, green1, red2, yellow2, green2);
input clock, sensor1, sensor2;
output  red1, yellow1, green1, red2, yellow2, green2;
parameter st0 = 0, st1 = 1, st2 = 2, st3 = 3,
          st4 = 4, st5 = 5, st6 = 6, st7 = 7;
reg [2:0] state, nxstate ;
reg red1, yellow1, green1, red2, yellow2, green2;

always @(posedge clock)
       state = nxstate;

always @(state or sensor1 or sensor2)
begin
       red1 = 1'b0; yellow1 = 1'b0; green1 = 1'b0;
       red2 = 1'b0; yellow2 = 1'b0; green2 = 1'b0;

       case (state)
          st0: begin
                 green1 = 1'b1;
                 red2 = 1'b1;
                 if (sensor2 == sensor1)
                    nxstate = st1;
                 else if (~sensor1 & sensor2)
                    nxstate = st2;
               end
          st1: begin
                 green1 = 1'b1;
                 red2 = 1'b1;
                 nxstate = st2;
                end
          st2: begin
                  green1 = 1'b1;
                  red2 = 1'b1;
                  nxstate = st3;
               end
          st3: begin
                  yellow1 = 1'b1;
                  red2 = 1'b1;
                  nxstate = st4;
               end
          st4: begin
                  red1 = 1'b1;
                  green2 = 1'b1;
                  if (~sensor1 & ~sensor2)
                     nxstate = st5;
                  else if (sensor1 & ~sensor2)
                     nxstate = st6;
               end
          st5: begin
                  red1 = 1'b1;
                  green2 = 1'b1;
                  nxstate = st6;
               end
          st6: begin
                  red1 = 1'b1;
                  green2 = 1'b1;
                  nxstate = st7;
               end
          st7: begin
                  red1 = 1'b1;
                  yellow2 = 1'b1;
                  nxstate = st0;
               end
        endcase
end
endmodule
```

Case Statement and Multiplexer Generation

The `case` statement, as defined by the Verilog LRM, is evaluated by order, and the first expression to match the control expression is executed (during simulation). For synthesis, this implies a priority encoding. However, in many cases the `case` statement is used to imply a multiplexer. This is true whenever the `case` conditions are mutually exclusive (the control expressions equals only one condition at any given time).

In Verilog, the case items can be non-constants also. In such a situation, the synthesis tools cannot detect that the `case` statements are parallel. Users can, however, use the global switch `-parallel_case` for Galileo or set the Tcl variable `parallel_case` to TRUE for Leonardo to inform the tool that all the `case` statements in the design a mutually exclusive.

For example, the following Verilog code:

```
case (1'b1)
      s[0]: o = a;
      s[1]: o = b;
endcase
```

results in the equation:

```
o = s[0] * a  + !s[0] * s[1] * b;
```

If parallel case is used, the following equation will be synthesized:

```
o = s[0] * a  + s[1] * b;
```

This equation is simpler than the first. For a bigger `case` statement the amount of logic reduction can be significant. This can not be determined automatically since the case items are nonconstants.

Note – The use of this option can cause simulation differences between behavioral and post-synthesis netlists.

Automatic Full Case Detection

The `casex` statement below is full case (it covers all possible values 000 to 111). The default statement is not necessary and is ignored by the synthesis tools, resulting in a warning message. The synthesis tools also do full-case detection for normal `case` and `casez` statements.

```
input [0:2] sel;
casex (sel)
   3'b10x: ...
   3'bx10: ...
   3'bx11: ...
   3'b00x: ...
   default: ....
endcase
```

The synthesis tools do full coverage analysis for the `if-then-else` structure. The following example is considered a full `if-then-else`. The last `else` is ignored and a warning is issued.

```
wire [0:1] data;
if (data == 2)
  ...........
else if (data == 1)
  ...........
else if (data == 3)
  ...........
else if (data == 0)
  ...........
else
 // Ignored for synthesis purpose
endmodule
```

Automatic Parallel Case Detection

`casex` statements are priority-encoded by definition. The Exemplar synthesis tools automatically detect parallel case and produce a warning message saying that case conditions are mutually exclusive. The following `case` statement is treated as parallel case.

```
input [0:2] sel;
casex (sel)
   3'b10x: ...
   3'bx10: ...
   3'bx11: ...
   3'b00x: ...
   default: ....
endcase
```

The synthesis tools do parallel case detection for `case` and `casez` statements. It also extracts the parallelism of a mutually exclusive `if-then-else` structure as shown below.

```
wire [0:1] data;
if (data == 2)
  ...........
else if (data == 1)
  ...........
else if (data == 3)
  ...........
else if (data == 0)
  ...........
```

casex Statement

The `casex` statement is used when comparison to only a subset of the selection signal is desired. For example, in the following Verilog code only the three least significant bits of `vect` are compared to `001`. The comparison ignores the three most significant bits.

```
casex (vect)
       6’bXXX001 : <statement> ;
// this statement is executed if vect[2:0] = 3’b001
endcase
```

For more information on comparisons to X and Z, refer to Chapter 8, “Verilog and Synthesis of Logic.”

casez Supported

`casez` is used in Verilog to specify “don't care” bits of the case tags. The `’z’`s in the case tags are not compared when a comparison between the case expression `sel` and the tags is done.

```
...
casez (sel)
   3'b10z: ...
   3'bz10: ...
   3'bz11: ...
   3'b00z: ...
   default: ....
endcase
```

’case’ and ’default’ Statements

The Exemplar synthesis tools allow the default statement to appear anywhere in a `case`, `casez`, or `casex` statement, and supports the `case` statement with only one default entry.

for Statements

`for` loops are used for repetitive operations on vectors. In the following example, each bit of an input signal is ANDed with a single bit enable signal to produce the result:

```
...
input clk ;
reg [4:0] input_signal, result ;
reg enable ;

always @ (posedge clk)
    for (i = 0; i < 5; i = i + 1)
            result[i] = enable & input_signal[i] ;
...
```

`for` loops are supported on if they are bounded by constants.

Disable Statement

The disable statement disables a named block or a task. Disabling of one block from another block is supported only if the second block is contained in the first one. Below is an example of disabling a named block.

```
module add_up_to (up_to_this, the_out);
input [0:3] up_to_this;
output the_out;
reg [0:7] the_out;
integer i;
   always @ (up_to_this)
begin: blk
      the_out = 0;
      for (i = 0; i < 16; i = i + 1)
      begin
         the_out = the_out + i;
         if (i == up_to_this) disable blk;
      end
end
endmodule

//Below  is an example of disabling a task.
module add_up_to (up_to_this, the_out);
input [0:3] up_to_this;
output the_out;
reg [0:7] the_out;
   always @ (up_to_this)
begin
      add_upto_this (up_to_this, the_out);
end
```

```
task  add_upto_this;
input [0:3] up_to_this;
output [0:7] the_out;
integer i;
begin
        the_out = 0;
        for (i = 0; i < 16; i = i + 1)
        begin
                the_out = the_out + i;
                if (i == up_to_this) disable add_upto_this;
        end
end
endtask
endmodule
```

forever, repeat, while and Generalized Form of for Loop

`forever`, `repeat`, `while`, and the generalized form of the `for` loop are supported as long as they are bounded by constants. In the following `forever` example, the system counts the number of 1s in the input vector.

Note – The `forever` loops only twice (which can be determined during compilation).

```
module forever_example (in, out);
input [0:1] in;
output out;
reg [0:1] out;

always @ (in)
begin:label
integer tmpcount;
reg [0:1] in_tmp;
```

```
always @ (in)
begin:label
integer tmpcount;
reg [0:1] in_tmp;

    out = 0;
    in_tmp = in;
    tmpcount = 0;
    forever
    begin
        if (in_tmp[1])
            out = out + 1;
        in_tmp = in_tmp >> 1;
        tmpcount = tmpcount +1;
        if (tmpcount == 2) disable label;
    end
end
endmodule
```

```
module repeat_example (i, o);
input i;
output o;
reg o;

always @ (i)
begin
    o = i;
repeat (4'b1011)
    o = ~o; // o = ~i
end
endmodule
```

Note – If any loop construct is NOT bound by constants, the synthesis tools issue the "iteration limit reached" error.

Functions and Tasks

Pieces of Verilog can be grouped together in functions and tasks, which can then be used as subprograms in the Verilog code. This is useful for repeated code, or for readability of the main module.

Tasks and functions appear similar, but are used in different ways. A task is a subprogram with inputs and outputs, and replaces any piece of verilog code in a module. Expressions in a task can be both combinational and sequential.

Functions have only inputs and returns a value by its name. Functions are purely combinational.

Functions

Functions are defined inside a module and can be freely used once they are defined. Functions are always used in an expression, behavioral or dataflow:

```
assign y = func(a,b);
```

or

```
x = func(z);
```

An example of a function is given below.

```
module calculator ( a, b, clk, s, operator );
    input [7:0] a, b;
    input clk;
    input [1:0] operator;
    output [7:0] s;
    reg [7:0] s;
    parameter ADD = 2’b00, SUB = 2’b01, MUL = 2’b10;

function [15:0] mult;
    input [ 7:0] a, b ;
    reg [15:0] r;
    integer i;
begin
    if (a[0] == 1)
            r = b;
    else
            r = 0;
    for (i = 1; i < 7; i = i + 1) begin
            if (a[i] == 1 )
                r = r + b << i ;
    end
    mult = r;
end
endfunction

always @ (posedge clk)
begin
    case (operator)
            ADD: s = a + b ;
            SUB: s = a - b ;
            MUL: s = mult(a,b);
    endcase
end
endmodule
```

Tasks

Tasks are always displayed as statements:

```
my_task(a,b,c,d);
```

The Exemplar synthesis tools support empty tasks.

An example of a task is presented below.

```
task demux ( state, load, bait, enable, ready, write, read );
input [2:0] state;
output load, bait, enable, ready, write, read;
parameter  LOAD = 3’b000, WAIT = 3’b100, ENAB = 3’b110,
          READ = 3’b111, WRIT = 3’b011, STRO = 3’b001;

   case (state)
        LOAD:
          {state, load, bait, enable, ready, write, read} = 6’b100000;
        WAIT:
          {state, load, bait, enable, ready, write, read} = 6’b010000;
        ENAB:
          {state, load, bait, enable, ready, write, read} = 6’b001000;
       READ:
          {state, load, bait, enable, ready, write, read} = 6’b000100;
       WRIT:
          {state, load, bait, enable, ready, write, read} = 6’b000010;
       STRO:
          {state, load, bait, enable, ready, write, read} = 6’b000001;
   endcase
endtask
```

Inout Ports in Task

The Exemplar synthesis tools support inout ports in a `task` statement. Any value passed through inout ports can be used and modified inside the `task`.

```
module inoutintask (i, o1, o2);
input i;
output o1, o2;
reg r, o1, o2;
task T ;
inout io;
output o;
begin
   o = io;
   io = ~io;
end
endtask
always @ (i)
begin
   r = i;
   T (r, o1); // o1 = i, r = ~i
   o2 = r;   // o2 = ~i;
end
endmodule
```

Access of Global Variables from Functions and Tasks

Global variables can be accessed for both reading and writing.

```
module x (clk, reset, i1, i2, o);
input clk, reset, i1, i2;
output o;
reg o;
reg [0:1] state;

 task T; //without any port
begin
    case (state)
      2'b00: o = i1;
      2'b01: o = i2;
      2'b10: o = ~i1;
      2'b11: o = ~i2;
    endcase
    state = state + 1; // next state
end
endtask

always @ (posedge clk or posedge reset)
if (reset) begin
    state = 0;
    o = 0;
end
 else T;
endmodule
```

System Task Calls

The Exemplar synthesis tools accept system task calls. System task calls are ignored, and a warning is issued.

System Function Calls

The Exemplar synthesis tools accept system function calls. The value 0 is assumed for system function calls, and a warning is issued.

Initial Statement

The Exemplar synthesis tools accept `initial` statements. The actual value is ignored.

Compiler Directives

Verilog supports a large list of compiler directives. Most of them are useful for simulation, but are meaningless for synthesis purposes. A few directives are supported by the synthesis tools, and those directives have to do with macro substitution and conditional compilation. Following is a list of these directives:

```
‘define
‘ifdef
‘else
‘endif
‘include
‘signed
‘unsigned
‘unconnected_drive
‘nounconnected_drive
```

Note – The symbol `exemplar` is predefined by the synthesis tools.

Therefore, the statement:

```
‘ifdef exemplar
```

will always be true, and the else part will always be false. This is useful if some parts need to be excluded from synthesis, but used by simulation or other tools. For example:

```
‘ifdef exemplar
// do nothing here when running simulator
‘else
initial
// do all initialization here. This will be ignored by the synthesis
tools.
‘endif
```

The Art of Verilog Synthesis 7

This chapter explains how particular logic constructs can be synthesized with Verilog restrictions taken into account.

Registers, Latches, and Resets

Verilog synthesis produces registers and combinational logic at the RTL level. All combinational behavior around the registers is, unless prohibited by the user, optimized automatically. Hence, the style of coding combinational behavior, like `if-then-else` and `case` statements, has little affect on the final circuit result, but the style of coding sequential behavior has significant impact on your design.

This section shows how sequential behavior is produced with Verilog, so that you understand why registers are generated at certain places and why not in others.

Most examples explain the generation of these modules with short Verilog descriptions in an `always` block.

Level-Sensitive Latch

This first example describes a level-sensitive latch:

```
...
input input_foo, ena ;
reg output_foo ;
...
always @ (ena or input_foo)
          if (ena)
                output_foo = input_foo ;
...
```

The sensitivity list is required, and indicates that the `always` block is executed whenever the signals `ena` or `input_foo` change. Also, since the assignment to the register `output_foo` is hidden in a conditional clause, `output_foo` cannot change (preserves its old value) if `ena` is `0`. If `ena` is `1`, `output_foo` is immediately updated with the value of `input_foo`, whenever that changes. This is the behavior of a level-sensitive latch.

In technologies where level-sensitive latches are not available, the Exemplar synthesis tools translate the initially generated latches to the gate equivalent of the latch, using a combinational loop.

Edge-Sensitive Flip-flops

An edge triggered flip-flop is generated from a Verilog description if a variable assignment is executed only on the leading (or only on the trailing) edge of another variable. For that reason, the condition under which the assignment is done must include an edge-detecting construct. There are a number of edge detecting attributes in Verilog. The two most commonly constructs are `posedge` and `negedge`.

The `posedge` construct detects transitions (is true) for 0 to 1. The `negedge` construct detects transitions from 1 to 0.

Here is one example of the posedge construct, used in the condition clause in an always block. The synthesis tools generate an edge-triggered flip-flop out of this behavior, with output_foo updated only if clk shows a leading edge.

```
....
input input_foo, clk ;
reg output_foo ;
....
always @ (posedge clk)
       output_foo = input_foo ;
....
```

If the posedge construct is not in the sensitivity list of the always block, a warning is issued that input_foo is not on the sensitivity list.

Synchronous Sets and Resets

All conditional assignments to variable output_foo inside the if clause translate into combinational logic in front of the D-input of the flip-flop. For instance, we can make a synchronous reset on the flip-flop by doing a conditional assignment to output_foo:

```
...
input input_foo, clk, reset ;
reg output_foo ;
...
always @ (posedge clk)
       if (reset)
           output_foo = 1'b0 ;
       else
           output_foo = input_foo ;
...
```

Variables reset and input_foo should not be included on the sensitivity list executing this block should not occur when they change.

Asynchronous Sets and Resets

If we want the reset signal to have immediate effect on the output, but still let the assignment to `output_foo` from `input_foo` only happen on the leading clock edge, we require the behavior of an asynchronous reset.

```
...
input input_foo, clk, reset ;
reg output_foo ;
...
always @ (posedge clk or posedge reset)
      if (reset)
         output_foo = 1'b0 ;
      else
         output_foo = input_foo ;
...
```

Now `reset` HAS TO BE on the sensitivity list. If it is not there, Verilog semantics require that the `always` block will not execute if `reset` changes. It will execute only if a positive change in `clk` is detected.

Asynchronous set and reset can both be used. This results in combinational logic driving the set and reset input of the flip-flop of the target signal. The following code fragment shows the structure of such a process:

```
always @(<edge of clock> or <edge_of_asynchronous_signals> )
      if (<asynchronous_signal>)
             <asynchronous signal_assignments>
      else if (<asynchronous_signal>)
             <asynchronous signal_assignments>
            ...
      else
             <synchronous signal_assignments>
```

There can be several asynchronous `else if` clauses, but the synchronous assignments have to be the last one in the `if` clause. A flip-flop is generated for each signal that is assigned in the synchronous signal assignment. The asynchronous clauses result in combinational logic that drives the set and reset inputs of the flip-flops.

Clock Enable

It is also possible to specify an enable signal in a process. Some technologies (specifically Xilinx) have a special enable pin on their basic flip-flop. The synthesis tools recognize the function of the enable from the Verilog description and generates a flip-flop with an enable signal from the following code fragment:

```
...
input input_foo, clk, enable ;
reg output_foo ;
...
always @ (posedge clk)
       if (enable)
          output_foo = input_foo ;
...
```

If an enable pin does not exist in the target technology a multiplexer is generated in front of the data input of the flip-flop.

Assigning I/O Buffers from Verilog

There are three ways to assign I/O buffers to your design from Verilog:

- Run the synthesis tools in “chip” mode
- Use the `buffer_sig` command
- Use component instantiation in Verilog of the buffer you require.

The `buffer_sig` command or the direct component instantiation will overwrite any default buffer assignment that the synthesis tools would do in “chip” mode.

The `buffer_sig` command is implemented differently for Galileo and Leonardo. For Galileo, you put the command in the control file. For Leonardo, you use the `buffer_sig` procedure.

These approaches can be used together by specifying certain I/O buffers in the Verilog source description and others in the control file, with the remaining buffers assigned automatically by the synthesis tools. The order the buffers are inserted in the design is important:

1. Components in the Verilog source are instantiated from the source technology.
2. Buffers are added by using the `buffer_sig` command from the target technology.
3. Terminals without identifiable I/O gates have buffers inserted from the target technology.

In all cases, the names of the original I/O terminals are preserved.

Automatic Assignment Using Chip Mode

The easiest way of assigning buffers is to run the synthesis tools in chip mode. (This is the default.) This automatically assigns appropriate input, output, tristate, or bidirectional buffers to the ports in your module definition. For example,

```
module buffer_example (inp, outp, inoutp) ;
input inp ;
output outp ;
inout inoutp;
endmodule
```

generates an `INPUT_BUFFER` for `inp`, and an `OUTPUT_BUFFER` for `outp`. `outp` becomes a `TRISTATE_BUFFER` if it was assigned in the following fashion:

```
tri outp ;
      assign outp = ena ? inp : 1'bZ
```

The above example also holds for buses. The sections “Tristate Buffers” on page 8 and “Bidirectional Buffers” on page 10 in this chapter provide more details on how to generate tristate buffers and bidirectional buffers from Verilog.

Manual Assignment Using the Control File

Special buffers, e.g., <gate>, can be assigned using the control file. The command

```
BUFFER_SIG <gate> clk
```

where <gate> is the name of a gate on the target technology, connects signal clk to the input of the external clock buffer <gate>. An intermediate node called clk_manual appears on CLOCK_BUFFER's output. Gates specified in the control file are searched for in the target technology library.

Use of the control file together with chip mode, to manually control only critical buffers, is accepted procedure when using the synthesis tools.

Buffer Assignment Using Component Instantiation

It is also possible to instantiate buffers in the Verilog source file with component instantiation. In particular, if you want a specific input or output buffer to be present on a specific input or output, component instantiation is a very powerful method:

```
module special_buffer_example (inp, clk, outp, inoutp) ;
input inp, clk ;
output outp ;
inout inoutp ;
wire intern_in, intern_out, io_control ;

   OUTPUT_FF A1(.c(clk), .d(intern_out), .t(io_control),.o(inoutp));
   INPUT_BUFFER A2(.i(inp), .o(intern_in)) ;

endmodule
```

In this example, component instantiation forces an OUTPUT_FF buffer (complex I/O output/flip-flop buffer) on the bidirectional pin inoutp. Also an input buffer INPUT_BUFFER is specified to pick up the value from inp to be used internally.

In the case of component instantiation of I/O buffers, a source technology must be specified to assure that the synthesis tools take the instantiated I/O buffer from the right library. If no source library is specified, an error is issued. If the source technology is specified, the components are instantiated from this library, which automatically gives them the right functionality. The synthesis tools recognize that the I/O pin is properly buffered, and does not add default buffers around it.

Tristate Buffers

Tristate buffers and bidirectional buffers (covered in the next section) are very easy to generate from a Verilog description.

Example 1:

```
// conditional expression
assign o1 = oe1 ? d1 : 1’bz;
assign x = oe2 ? d2 : 1’bz;
assign o1 = x;

// if statement
always @ (oe3 or d3)
   if (oe3)
      o2 = d3;
   else
      o2 = 1’bz;

// case statement
always @ (oe4 or d4)
   case (oe4)
      default : o2 = 1’bz;
      1’b1    : o2 = d4;
   endcase
```

Example 2:

```
module tristate (input_signal, ena, output_signal) ;
input input_signal, ena ;
output output_signal ;

      assign output_signal = ena ? input_signal :  1’bz ;

endmodule
```

Note that in the conditional clause of the assign statement, both `input_signal` and `ena` can be full expressions. The Exemplar synthesis tools generate combinational logic driving the input or the enable of the tristate buffer for these expressions.

However, it is illegal to use the `'z'` value in an expression. It is also illegal to use the `'z'` value in any form inside a clocked `always` block.

Example 3:

```
assign  output_signal = input_signal & 1'bz;
```

Normally, simultaneous assignment to one signal in Verilog is not allowed for synthesis, since it would cause data conflicts. However, if a conditional `'Z'` is assigned in each assignment, simultaneous assignment resembles multiple tristate buffers driving the same bus.

```
module tristate_example_2 (input_signal_1, input_signal_2, ena1, ena2,
output_signal) ;
input input_signal_1, input_signal_2, ena1, ena2 ;
output output_signal ;

      assign output_signal = ena1 ? input_signal_1 : 1'bz ;
      assign output_signal = ena2 ? input_signal_2 : 1'bz ;

endmodule
```

You can still introduce a data conflict with these simultaneous assignments to `output_signal`, by making both `ena_1` and `ena_2 1'b1`. The synthesis tools do not check for a possible bus conflict. Make sure that you can never have that possibility by carefully generating the enable signals for the tristate conditions.

These examples show assignments to outputs. However, it is certainly possible to do the assignments to an internal wire as well. This might be used for generating buses, and is discussed in "Buses" on page 10 in this chapter.

If the target technology does not have any internal three-state drivers, Galileo can transform the three-state buffers into regular logic with the `-tristate` option. Leonardo performs this transformation when the `tristate_map` variable is set to TRUE.

Bidirectional Buffers

Bidirectional I/O buffers can be coded in Verilog as follows:

```
module bidirectional (bidir_port, ena, ...) ;
input ena ;
inout bidir_port ;

        assign bidir_port = ena ? internal_output : 1'bZ ;
        assign internal_input = bidir_port ;
        ...
        // use internal_input
        ...
        // generate internal_output
endmodule
```

The difference with the previous examples is that in this case, the output itself is used again internally. For that reason, the port `bidir_port` is declared to be inout.

The enable signal `ena` could also be generated inside the module instead of being a primary input as in this example.

The synthesis tools select a suitable bidirectional buffer from the target technology library. If there is no bidirectional buffer available, it selects a combination of a tristate buffer and an input buffer.

Buses

The examples given above all use single bits as signals. In reality, buses (arrays of bits with tristatable (multiple) drivers) are often used. Buses are used both internally to the design and as I/O. For internal tristate buses, the bus signal should be declared as a `tri` net.

The following example describes a circuit that loads a source vector of 4 bits on the edge of a clock (`wrclk`), and stores the value internally in a register (`intreg`) if the chip enable (`ce`) is active. One bit of the register output is put on a tristate bus (`result_int`), based on a 2-bit selector signal (`selector`), with the bus output clocked through a final register (`result`). For more information, refer to “Continuous Assignments” on page 9.

```
module tri_asgn (source, ce, wrclk, selector, result) ;
input [0:3]source ;
input ce, wrclk ;
input [0:1]selector ;
output result ;
reg [0:3]intreg ;
reg result ;
wire [0:1]sel = selector ;
tri result_int ;

// assignment to internal tristate bus
      assign
            result_int = (~sel[0] && ~sel [1]) ? intreg[0] : 1’bZ ,
            result_int = (sel[0] && ~sel [1])  ? intreg[1] : 1’bZ ,
            result_int = (~sel[0] && sel [1])  ? intreg[2] : 1’bZ ,
            result_int = (sel[0] && sel [1])   ? intreg[3] : 1’bZ ;

always @(posedge wrclk)
begin
      if (ce)
         intreg = source;
      result = result_int ;
end

endmodule
```

In the following example of a tristate bus used for output, a source is loaded into a register (`tbuf_in`) whose output is a set of tristate buffers.

```
module tri_bus (d, clk, en, tbuf_out) ;
parameter n = 8 ;
parameter triZ = 8'bZ ;
input [(n-1):0] d ;
input clk, en ;
output [(n-1):0] tbuf_out ;
reg [(n-1):0] tbuf_in ;

assign tbuf_out = en ? tbuf_in : triZ ;

always @ (posedge clk)
   tbuf_in = d ;

endmodule
```

State Machines

There are basically two forms of state machines, Mealy machines and Moore machines. In a Moore machine, the outputs do not directly depend on the inputs, only on the present state. In a Mealy machine, the outputs depend directly on the present state and the inputs.

In general, a description of a state machine consists of descriptions of the state transitions, the output functions and a register function. Because of the register function, an `always` block in Verilog is an appropriate way to describe a state machine. `if-else-if` or `case` statements in an `always` block perform the state transition and output function descriptions.

In the following sections, the DRAM interface state machine shown in Figure 7-1 is used to illustrate state machine design using Verilog.

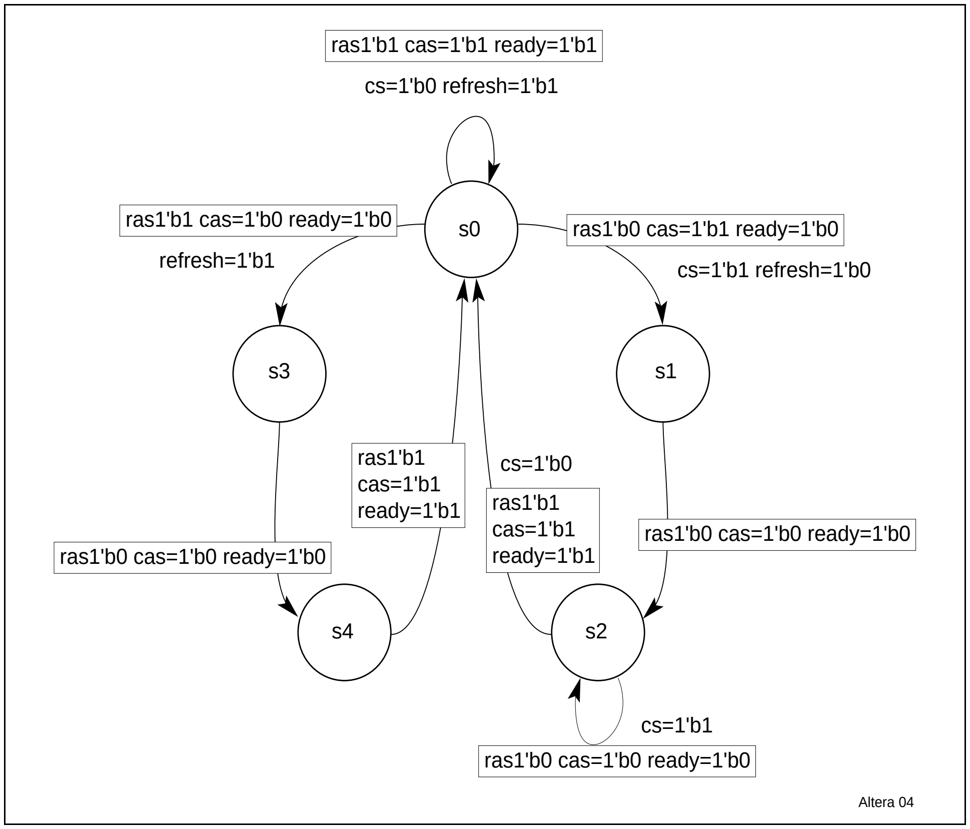
ras1'b1 cas=1'b1 ready=1'b1
cs=1'b0 refresh=1'b1
s0
ras1'b1 cas=1'b0 ready=1'b0
refresh=1'b1
ras1'b0 cas=1'b1 ready=1'b0
cs=1'b1 refresh=1'b0
s3
s1
ras1'b1
cas=1'b1
ready=1'b1
cs=1'b0
ras1'b1
cas=1'b1
ready=1'b1
ras1'b0 cas=1'b0 ready=1'b0
ras1'b0 cas=1'b0 ready=1'b0
s4
s2
cs=1'b1
ras1'b0 cas=1'b0 ready=1'b0
Altera 04

Figure 7-1 DRAM Interface with Refresh

Moore Machines

An example of a Moore machine is:

```
module moore (clk, cs, refresh, ras, cas, ready) ;
input clk, cs, refresh ;
output ras, cas, ready ;

parameter s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4 ;
reg [2:0] present_state ;
reg ras, cas, ready ;

always @ (posedge clk)
begin
      case (present_state)
            s0 : begin
                  if (refresh)
                        present_state = s3 ;
                  else if (cs)
                        present_state = s1 ;
                  else
                        present_state = s0 ;
                 end
            s1 : begin
                  present_state = s2 ;
                 end
            s2 : begin
                  if (~cs)
                        present_state = s0 ;
                  else
                        present_state = s2 ;
                 end
            s3 : begin
                  present_state = s4 ;
                 end
            s4 : begin
                  present_state = s0 ;
                 end
```

```
                default : begin
                      present_state = s0 ;
                    end
        endcase
end
always @ (present_state)
begin
        case (present_state)
                s0 : begin
                      ras = 1'b1 ; cas = 1'b1 ; ready = 1'b1 ;
                    end
                s1 : begin
                      ras = 1'b0 ; cas = 1'b1 ; ready = 1'b0 ;
                    end
                s2 : begin
                      ras = 1'b0 ; cas = 1'b0 ; ready = 1'b0 ;
                    end
                s3 : begin
                      ras = 1'b1 ; cas = 1'b0 ; ready = 1'b0 ;
                    end
                s4 : begin
                      ras = 1'b0 ; cas = 1'b0 ; ready = 1'b0 ;
                    end
                default : begin
                      ras = 1'bX ; cas = 1'bX ; ready = 1'bX ;
                    end
        endcase
end
endmodule
```

There are two `always` blocks in the state machine description. The first is synchronized with the clock `clk` and describes the state transitions. This block depends on the present state and the inputs. The second is not synchronized, but it reacts immediately if there is a change in `present_state`. This second `always` block describes the functions of the outputs depending on the present state. The split into two processes is not absolutely necessary. The same functional behavior can be generated by merging the two always blocks into one. However, the logic that is generated is somewhat different, as explained below.

Below is exactly the same Moore machine description, but this time it consists of only one always block. In the first description, the outputs `ras`, `cas` and `ready` were assigned in an asynchronous (not clocked) always block as a function of `present_state`. They therefore appear as purely combinational logic. In the description below, the same outputs are generated in a clocked always block. Therefore, the outputs `ras`, `cas` and `ready` appear at the Q-output of flip-flops with the combinational logic computing the value of these signals at the D-inputs of the same flip-flops.

The subtle differences between the two descriptions result in trading off timing behavior and logic circuitry. The first description builds a circuit where the outputs ripple through logic after the clock edge. In the second description, the outputs change glitch-free at the clock-edge, and are stable immediately after that, but at the cost of an additional flip-flop for each output.

```
module moore_example_2 (clk, cs, refresh, reset, ras, cas, ready) ;
input clk, cs, refresh, reset ;
output ras, cas, ready ;

parameter s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4 ;

reg [2:0] present_state ;
reg ras, cas, ready ;

always @ (posedge clk or posedge reset)
begin
      if (reset)  // asynchronous reset
      begin
         present_state = s0 ;
         ras = 1'b1 ; cas = 1'b1 ; ready = 1'b1 ;
      end
      else
      begin
         case (present_state)
            s0 :
               if (refresh)
               begin
                  present_state = s3 ;
                  ras = 1'b1; cas = 1'b0 ; ready = 1'b0 ;
               end
            else if (cs)
            begin
                present_state = s1 ;
                ras = 1'b0; cas = 1'b1 ; ready = 1'b0 ;
            end
            else
            begin
                present_state = s0 ;
                ras = 1'b1; cas = 1'b1 ; ready = 1'b1 ;
            end
        s1 :
            begin
                present_state = s2 ;
                ras = 1'b0 ; cas = 1'b0 ; ready = 1'b0 ;
            end
        s2 :
            begin
                if (~cs)
                begin
                    present_state = s0 ;
                    ras = 1'b1; cas = 1'b1 ; ready = 1'b1 ;
                end
                else      // cs = 1'b1
                begin
                    present_state = s2 ;
                    ras = 1'b0; cas = 1'b0 ; ready = 1'b0 ;
                end
            end
        s3 :
            begin
                present_state = s4 ;
                ras = 1'b0 ; cas = 1'b0 ; ready = 1'b0 ;
            end
        s4 :
            begin
                present_state = s0 ;
                ras = 1'b1 ; cas = 1'b1 ; ready = 1'b1 ;
            end
```

```
                    default:
                       begin
                          present_state = s0 ;
                          ras = 1'bX ; cas = 1'bX ; ready = 1'bX ;
                       end
              endcase
        end
end
endmodule
```

Note – This example also added an asynchronous reset to the design.

Mealy Machines

So far, we have shown a number of examples of Moore machines. In a Mealy machine, outputs depend on both the present state and the inputs. Below is the state machine again, but now in a Mealy machine form. Notice that the behavior changes slightly, since the inputs affect the outputs immediately, without waiting for the new state to be generated.

In the Moore machine example, it was possible to merge the two processes into one, synchronized with a clock, since all activity was happening on the clock edge. In this Mealy machine example, however, the outputs are updated even when there is no clock edge. Thus, in this case, it is not possible to merge the two processes into one.

A Mealy machine is, in general, described with two always blocks, where one block does all combinational functionality and the other just updates the present state with the next state, on the clock edge.

This code shows an example of a Mealy machine.

```
module mealy (clk, cs, refresh, ras, cas, ready) ;
input clk, cs, refresh ;
output ras, cas, ready ;

parameter s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4 ;

reg [2:0] present_state, next_state ;
reg ras, cas, ready ;

always @ (posedge clk)
begin
       // always block to update the present state
                     present_state = next_state ;
end

always @ (present_state or refresh or cs)
begin
       // always block to calculate the next state and the outputs
       next_state = s0 ;
       ras = 1’bX ; cas = 1’bX ; ready = 1’bX ;
       case (present_state)
              s0 : begin
                     if (refresh)
                     begin
                        next_state = s3 ;
                        ras = 1’b1 ; cas = 1’b0 ; ready = 1’b0 ;
                     end
                     else if (cs)
                     begin
                        next_state = s1 ;
                        ras = 1’b0 ; cas = 1’b1 ; ready = 1’b0 ;
                     end
                     else
                     begin
                        next_state = s0 ;
                        ras = 1’b1 ; cas = 1’b1 ; ready = 1’b1 ;
                     end
                   end
```

```
                s1 : begin
                       next_state = s2 ;
                       ras = 1’b0 ; cas = 1’b0 ; ready = 1’b0 ;
                     end
                s2 : begin
                       if (~cs)
                       begin
                          next_state = s0 ;
                          ras = 1’b1 ; cas = 1’b1 ; ready = 1’b1 ;
                       end
                       else
                       begin
                          next_state = s2 ;
                          ras = 1’b0 ; cas = 1’b0 ; ready = 1’b0 ;
                       end
                     end
                s3 : begin
                       next_state = s4 ;
                       ras = 1’b1 ; cas = 1’b0 ; ready = 1’b0 ;
                     end
                s4 : begin
                       next_state = s0 ;
                       ras = 1’b0 ; cas = 1’b0 ; ready = 1’b0 ;
                     end
      endcase
end
endmodule
```

Combinational loops can be generated easily (and are in most cases unwanted) in a Mealy machine description. If nothing is assigned to a signal in one or more cases (for instance because you do not care what the value is going to be), Verilog semantics require that the value of the signal is preserved. In an asynchronized `always` block as the one shown above, this means that synthesis must generate a combinational loop or a level-sensitive latch to preserve the value.

For more information on how to avoid unwanted loops, refer to “Operators” on page 17.

Issues in State Machine Design

This section discusses several issues regarding the design of synthesizable state machines:

- State encoding
- One-hot encoding
- Initialization of the state machine
- Power-up conditions
- Semantics of the case statement

State Encoding

States must be explicitly specified by the user. This can be done by explicitly using the bit pattern (e.g., `3’b101`), or by defining a parameter (e.g., `parameter s3 = 3’b101`) and using the parameter as the case item.

One-Hot Encoding

In order to achieve a different style of encoding, for example a one-hot encoding, a slightly different style of Verilog is required. As an example, here is the Verilog description for a one-hot encoded state machine with the same functionality as the example shown above.

```
module one_hot_mealy (clk, cs, refresh, reset, ras, cas, ready) ;
input clk, cs, refresh, reset ;
output ras, cas, ready ;

reg [4:0] present_state, next_state ;
reg ras, cas, ready ;

always @ (posedge clk)
begin
      // always block to update the present state
      if (reset)
         present_state = 5'b00001 ;
      else
         present_state = next_state ;
end

always @ (present_state or refresh or cs)
begin
      // always block to calculate the next state and the outputs
      next_state = 5'b00000 ;
      ras = 1'bX ; cas = 1'bX ; ready = 1'bX ;

      if  (present_state[0])
      begin
         if (refresh)
         begin
            next_state = 5'b01000 ;
            ras = 1'b1 ; cas = 1'b0 ; ready = 1'b0 ;
         end
         else if (cs)
         begin
            next_state = 5'b00010 ;
            ras = 1'b0 ; cas = 1'b1 ; ready = 1'b0 ;
```

```
                    end
                    else
                    begin
                       next_state = 5'b00001 ;
                       ras = 1'b1 ; cas = 1'b1 ; ready = 1'b1 ;
                    end
              end
        if (present_state[1])
        begin
           next_state = 5'b00100 ;
           ras = 1'b0 ; cas = 1'b0 ; ready = 1'b0 ;
        end
        if (present_state[2])
        begin
           if (~cs)
           begin
              next_state = 5'b00001 ;
              ras = 1'b1 ; cas = 1'b1 ; ready = 1'b1 ;
           end
           else
           begin
              next_state = 5'b00100 ;
              ras = 1'b0 ; cas = 1'b0 ; ready = 1'b0 ;
           end
        end
        if (present_state[3])
        begin
           next_state = 5'b10000 ;
           ras = 1'b1 ; cas = 1'b0 ; ready = 1'b0 ;
        end
        if (present_state[4])
        begin
           next_state = 5'b00001 ;
           ras = 1'b0 ; cas = 1'b0 ; ready = 1'b0 ;
        end
end
endmodule
```

Some key points from this one-hot state machine are:

- The `case` statement should not be used for one-hot state machine design. When the `casex` statement is used for state comparisons, the comparisons must be done on only one bit of the state vector. If the whole vector is used for comparison, then full binary encoding logic is synthesized. Also, the `case` statement needs to be compiled as `parallel_case`.
- The `else if` construct should not be used to do the state comparisons, since that introduces additional constraints on the values of each state. Using `else if` means that this code is only entered if the all previous conditions are false. In the case of one-hot encoding, it is certain that all previous conditions are false already.

This state machine description works fine, as long as the machine can never appear in a state with more than one `'1'` in the state vector. In order to assure that condition, the need for a reset becomes inevitable in the one-hot case. The use of resets is discussed in greater detail in the next section.

Initialization and Power-Up Conditions

In synthesis, if the total number of states is not a power of two, the state signal can power-up in a state that has not been defined, if binary encoding is used. In this situation, it is essential that the Verilog description does an assignment to the output variables and the state variable under all conditions.

This can be done in two ways:

- Do a default assignment to the outputs and state variable before the `case` statement that updates the state machine. This method is used in the first Moore and the Mealy machine examples from the previous sections. It assures that outputs and state variable always get a value assigned regardless of the state of the state machine.
- Do the default assignment in the `default` clause of the `case` statement, as was shown in the second Moore machine example. This has the same effect; outputs and states always get a value regardless of the state of the machine.

If you do not do a default assignment, the state machine could power-up in a undefined state. Verilog semantics require that if there is no assignment to a signal, the previous value has to be preserved. In case the state transitions are defined in an asynchronous always block, latches would be generated by the synthesis tools to preserve the state value.

If one-hot or another state encoding is used, the number of undefined states could be even larger. Consider that in one-hot encoding, the specification of the state machine has to rely on the fact that only one single state bit of the state vector is 1. That means that the designer has to provide a special feature that takes care of the power-up conditions.

One possibility might be to include a special detection function that sets the state to a valid one the moment it occurs in a invalid one. However, it would require too much logic to implement this functionality, making the use of one-hot encoding unattractive. In most cases, it is much more cost effective to include the possibility of a reset function. The reset can be defined to be synchronous or asynchronous, depending on what you want. The details about implementing resets are given in the section “Registers, Latches, and Resets” on page 1 in this chapter.

Arithmetic and Relational Logic

This section gives an overview of how arithmetic logic is generated from Verilog, what the synthesis tools do with it and how to avoid getting into combinational explosion with large amounts of arithmetic behavior.

In general, logic synthesis is very powerful in optimizing random combinational behavior, but has problems with logic which is arithmetic in nature. Often special precautions have to be taken into consideration to avoid ending up with inefficient logic or excessive run times. Alternatively, macros may be used to implement these functions (see “Technology-Specific Macros” on page 29 in this chapter).

The synthesis tools support the operators “+”, “-”, “==”, “!=”, “<”, “>”, “>>”, “<<”, “*”,“/”, “<=”, and “>=”.

If you use these operators to calculate compile time constants, there is no restriction or problem in using them. For example, the following division does not result in a any logic, but replaces signal `foo` with a `constant 3’d133`.

```
...
integer largest ;
integer divider ;
assign largest = 800 ;
asign divider = 6 ;
assign foo <= largest / divider ;
...
```

If you are not working with constant operands, arithmetic logic is generated.

The operator “+” generates an adder. The number of bits of the adder depends on the size of the operands. If you use integers, a 32 bit adder is generated. If you add vectors and integers, the size of the adder is defined to the range of the vector in bits. For example:

```
...
integer a, b, c ;
assign c = a + b ;
...
```

generates a 32-bit adder but:

```
...
input [7:0] a ;
output [7:0] c ;
integer b ;
assign c = a + b ;
...
```

generates an 8-bit adder.

If one of the operands is a constant, initially a full-sized adder is still generated but logic minimization eliminates much of the logic inside the adder because half of the inputs of the adder are constant.

The operator “-” generates a subtracter. Same remarks as with the “+” operator.

The operator “*” generates a multiplier. Multiplication by a constant power of two is implemented as a shift operation. In all other cases ModGen (generic or technology specific) is required to implement the multiplier.

The operator “/” generates a divider. Only division by a power of two is supported, hence no logic here, only shifting the non-constant operand.

The operators “==”, “!=”, “<”, “>”, “>>”, “<<”, “<=”, and “>=” generate comparators with the appropriate functionality. Same remarks apply as for the “+” operator.

- Operations on integers are done in twos-complement implementation.

All arithmetic behavior is translated into logic functions and is part of the logic optimization process. The result is that depending on area and timing criteria and constraints set, the final logic circuit can include, for example, carry lookahead or ripple carry adder implementation. If the design is getting large, run-time and memory requirements increase rapidly. Some large designs can run forever without any improvement, if any solution is produced at all. The reason is that the logic synthesis optimization algorithms try too many possible circuit implementations from the exponentially growing search space. Good design practices are needed to help avoid this problem.

Below are some guidelines that have helped users to achieve a good synthesis result.

Module Generation

When arithmetic and relational logic are used for a specific Verilog design, the Exemplar synthesis tools provide a method to synthesize technology specific implementations for these operations. Generic modules (for bit-sizes > 2) have been developed for many of the CPLDs supported by the Exemplar synthesis tools to make the most efficient technology specific implementation for arithmetic and relational operations.

For Galileo, use the `-modgen=`*modgen_library* option to include a module generation library of the specified technology and infer the required arithmetic and relational operations of the required size from a design. For Leonardo, use the `modgen_read` *modgen_library* command to load the module generation library into the HDL database. Since these modules have been designed optimally for a target technology, the synthesis result is, in general, smaller and/or faster and takes less time to compile.

You may define your own module generator for a specific technology.

Resource Sharing and Common Subexpression Elimination

The synthesis tools automatically do CSE. For the following example, it will create only one adder `(a+b)` and use it for both the `if` conditions. For bigger expressions user need to use parentheses properly to direct the synthesis tool for CSE, e.g., `y = a+(b-c), z = d+(b-c), (b-c)` is shared.

```
...
reg a, b, c, d ;

always @ (a or b)
begin
        if ( a+b == c ) //This adder will be shared
                ...
        else if ( a+b == d) // with this one.
                ...
        else
                ...
end ;
...
```

Comparator Design

Often, applications involve a counter that counts up to an input signal value, and if it reaches that value, some actions are needed and the counter is reset to 0.

```
 ...
        begin
                if (count == input_signal)
                        ...
                        count = 0 ;
                else
                        count = count + 1 ;
        end ;
...
```

In this example the synthesis tools build an incrementer and a full-size comparator that compares the incoming signal with the counter value. It is usually better to preset the counter to the `input_signal` and count down, until zero is reached.

```
...
        begin
                if (count == 0)
                        ...
                        count = input_signal ;
                else
                        count = count - 1 ;
        end ;
...
```

Now, one decrementer is needed plus a comparison to a constant (0). Since comparisons to constants are a lot cheaper to implement, this new behavior is much easier to synthesize, and results in a smaller circuit.

Even better results can be obtained with the use of hard macros and soft macros of the target technology, as well as the use of hierarchy in the design. The following two sections explain this in more detail.

Technology-Specific Macros

In many cases, the target technology library includes a number of hard macros and soft macros that perform specific arithmetic logic functions. These macros are optimized for the target technology and have high performance.

With the Exemplar synthesis tools, it is possible to use component instantiation of soft macros or hard macros in the target technology. An added benefit is that the time needed for optimization of the whole circuit can be significantly reduced since the synthesis tools do not have to optimize the implementation of the dedicated functions any more.

Suppose you want to add two 8 bit vectors, and there is an 8 bit adder macro available in your target technology. You could use the “+” operator to add these two vectors. The alternative is to define a component that has the same name and inputs and outputs as the hard macro you want to use. Instantiate the component in your Verilog description and connect the inputs and output to the their appropriate signals. The synthesis tools instantiate the hard macro without having to bother with the complicated optimization of the internal logic implemented by the macro.

This speeds up the optimization process considerably. In the netlist produced by the synthesis tools, the macro appears as a "black box" that the downstream place and route tools recognize.

If your arithmetic functions cannot be expressed in hard macros or soft macros immediately (for instance if you need a 32 bit adder, but only have an 8 bit adder macro), you could write a Verilog description that instantiates the appropriate number of these macros.

Synthesis Directives

parallel_case and full_case directives

`parallel_case` and `full_case` directives are allowed as synthesis directive on case by case basis. The synthesis tool detects the true full and parallel cases automatically. However, there are cases (like onehot encoded state machine) that are not inherently parallel/full, but the environment guarantees that the case statement is parallel and/or full. In such a condition the following two synthesis directives are very useful.

```
input [0:3] inp_state;
// example of onehot encoded machine
case (1'b1) // exemplar parallel_case full_case
  inp_state[0]: .......
  inp_state[1]: .......
  inp_state[2]: .......
  inp_state[3]: .......
endcase
```

translate_off and translate_on directives

`translate_off` and `translate_on` synthesis directives are allowed to comment out a portion of code that you may want to retain for some purpose other than synthesis.

```
// code for synthesis
// exemplar translate_off
$display (.....); // not for synthesis
// exemplar translate_on
// code for synthesis
endmodule
```

enum directive

The `enum` synthesis directive is supported for user convenience when trying out different encoding on a state machine. With the synthesis directive, the synthesis tool becomes sensitive to the global state encoding switch (`-encoding`), and the enumeration values are encoded according to the setting of that option (`onehot`, `gray`, `binary,` or `random`).

Using the `enum` synthesis directive, a set of parameters can be treated as enumerated values; resources like wire and reg can be declared as that enumerated type. The synthesis tool puts some restrictions on these enumerated types. Elements are allowed with enumerated objects areas in the following instances:

> In case statements: The enum type of case expression should match with the case tags. For comparison of the enumerated types with each other, assigning enumerated types to each other (type should match).

These objects are treated as strongly typed so they cannot be mixed with the object of any other type. Any boolean or arithmetic operations are considered to be in error for enumerated objects. The synthesis tool gives an appropriate error when any one of these rules is violated. In such cases, you may not use the `enum` synthesis directive.

The encoding style of the enumeration can be selected from boolean (default), `onehot,gray,` or `random` using the global `-encoding` option on the synthesis tool mainline, or using the state encoding selection on the Verilog input options dialog of the user interface.

```
module state_mc (clk, reset, o, i1, i2, i_state);
input clk, reset, i1, i2;
output o, i_state;
reg o;
parameter [0:1] /* exemplar enum ee1 */S0=1,S1=2,S2=3,S3=0;
reg [0:1] /* exemplar enum ee1 */ state;
assign i_state = (state == S1 | state == S3); // legal.
always @ (posedge clk or posedge reset)
if (reset) begin
  o = 0;
  state = S0; // Note state = 1, will cause a type mismatch
error
end
else
    case (state) // No need of full and parallel case
      S0: begin o = i1;  state = S1; end
      S1: begin o = ~i1; state = S2; end
      S2: begin o = i2;  state = S3; end
      S3: begin o = ~i2; state = S0; endNote case tag 0:
would cause type
mismatch error
endcase
endmodule
```

State and S0, S1, S2, S3 are of `enum` type ee1. They cannot be used for any boolean or arithmetic operation. Bit or part select from state or its values is also considered an error. Enumerated type module parts are not allowed.

attribute directive

The user can set some simple attributes on signals/instances to enhance the synthesis efficiency of the Exemplar synthesis tool. For example, by setting the `modgen_select` attribute to `fastest` on a signal on a critical path of a design, the user can improve the timing performance of the design. The synthesis of this directive is as follows:

```
// exemplar attribute <object_name><attribute_name><attribute_value>
```

```
//example
module  expr (a, b, c, out1, out2);
input [0:15] a, b, c;
output [0:15] out1, out2;

  assign out1 = a + b;
  assign out2 = b + c;

// exemplar atribute out1 modgen_sel fastest
endmodule
```

Verilog and Synthesis of Logic 8

Verilog is a language that has been developed for simulation purposes. Synthesis was not an issue in the development of the language. As a result, there are a number of Verilog constructs that cannot be synthesized. There has been very little written that explains which constructs cannot be synthesized into logic circuits and why.

This chapter provides explanations on why certain Verilog constructs cannot be synthesized into logic circuits and what changes have to be made to reach the intended behavior to obtain a synthesizable Verilog description.

Some obvious restrictions of the language are first presented, followed by a list summarizing Verilog syntax and semantic restrictions for the Exemplar synthesis tools. In addition, some guidelines are presented that should enable you to write Verilog that is easy to synthesize and give you a feeling for synthesis complexity problems you might introduce when you write your Verilog design.

Comparing With X and Z

Consider the Verilog modeling case where an if clause should be entered if a part of a vector has a particular value. The rest of the vector does not really matter. You might want to write this as follows:

```
if (vect == 6’bXXX001) begin ...
```

The user intention is to do a comparison to `001` (the right most three bits) and forget about the left three bits. However, Verilog defines comparison on vectors as the AND of comparison of each individual element. Also, comparison of two elements is only true if both elements have exactly the same value. This means that in order for this condition to be true, the three left most bits have to be 'X'. But in logic synthesis, a bit can only be `'0'` or `'1'`, so the condition is always be false. In fact, this condition is not doing what was intended for simulation as well, since if any of the left most three bits does not have the value 'X' explicitly, the result is false.

However, comparison to 'X' is allowed using the `casex` construct. This is implemented in the following manner:

```
casex (vect)
      6'bXXX001 :  <statement> ;
endcase
```

In this case, only the three least significant bits of vect are compared to “001". The comparison ignores the three most significant bits.

Variable Indexing of Bit Vectors

The Exemplar synthesis tools support variable indexing of a vector. The limitation is that only variable indexing of the form 'bit select' is supported. Or more specifically, variable indexing of the form 'part select' is not supported because it is not a synthesizable construct.

The semantics of variable indexing varies depending on whether the variable indexing is done on the left hand side of an assignment or on the right hand side of the assignment. The right-hand side variable indexing generates a multiplexer controlled by the index. The left-hand variable indexing generates a de-multiplexer controlled by the index. set of decoders enabling. The following example shows both examples.

```
module tryit (input_bus, in_bit, control_input, output_bus, out_bit);
input [3:0] input_bus ;
input  [1:0] control_input ;
input  in_bit ;
output [3:0] output_bus ;
output out_bit ;

reg [1:0] control_input ;
reg [3:0] input_bus, output_bus ;
reg in_bit, out_bit ;

always @ (control_input or input_bus or in_bit)
begin
   out_bit = input_bus [control_input] ;
   output_bus [control_input] = in_bit ;
end
endmodule
```

Syntax and Semantic Restrictions

This section provides a summary of the syntax and semantic restrictions of the Exemplar synthesis tools' Verilog HDL parser.

Unsupported Verilog Features

- UDP primitives
- `specify` block
- `real` variables and constants
- `initial` statement
- `tri0`, `tri1`, `tri1`, `tri1`, `tri1`, net types
- `time` data type

- Named events and event triggers
- The following gates: `pulldown`, `pullup`, `nmos`, `mmos`, `pmos`, `rpmos`, `cmos`, `rcmos`, `tran`, `rtran`, `tranif0`, `rtranif0`, `tranif1`, `rtranif1`
- `wait` statements
- Parallel block, `join` and `for`.
- System task enable and system function call
- `force` statement
- `release` statement
- Blocking assignment with event control
- Named port specification (not to be confused with passing arguments by name, which is supported)
- Concatenation in port specification
- Bit selection in port specification
- Procedural assign and de-assign

Supported Verilog Features (Limited in Usage)

- Edge triggers on sensitivity list must be single bit variables.
- Indexing of parameters is not allowed.
- Loops must be bounded by constants.

Supported Verilog Features (Ignored by Exemplar Synthesis)

- Delay and delay control.
- ’vectored’ declaration.

Introduction to Module Generation 9

Arithmetic and relational logic, commonly known as data path logic, has traditionally been difficult to synthesize with logic synthesis software. This is especially true for FPGAs, where each target technology has a different way to optimally utilize resources.

Exemplar Logic's *Module Generation* capability provides VHDL and Verilog HDL designers with a mechanism to overload data path operators, such as "+", "-" and ">", with technology-specific implementations.

This chapter introduces the concept of *Module Generation* and describes how to make optimal use of this feature of the Exemplar synthesis tools. Chapter 10, "Using Module Generation," focuses on how to use *Module Generation* to improve performance for VHDL and Verilog HDL design files. Chapter 11, "User-Defined Module Generators," provides a detailed description of how to create your own module generators.

Module Generation provides a mechanism that matches behavioral operators like "+", "-", and ">", with pre-designed implementations. This allows designers to describe logic in a purely behavioral fashion, while making optimal use of technology-specific hard or soft macros. As an example, consider the following VHDL statement:

```
signal a, b, s : std_logic_vector(n downto 0);
s <= a + b;
```

When implementing this VHDL statement in an FPGA architecture, designers would like to utilize vendor-provided adder hard macros, dependent on the size of `n`.

In HDLs, the user can explicitly instantiate a desired component (using component instantiation in VHDL or module instantiation in Verilog).

Three drawbacks exist with using component/module instantiation:

- The design methodology is no longer behavioral.
- The HDL source becomes technology dependent.
- Component instantiation is not allowed in operator or function definitions.

However, if neither component/module instantiation nor module generation is used, the synthesis tools generate logic without any knowledge of an optimal implementation for the target technology. This typically produces sub-optimal results.

Module Generation solves this problem by matching certain data path operators with pre-designed implementations from a side library. Whenever a supported operator is encountered in the source design, a technology-specific module generation library is consulted for a matching implementation. If an implementation is found, it is used in the network. If no technology dependent implementation is found, the synthesis tools default to a generic logic implementation, which is applicable for a CMOS gate array implementation, for the operator (ripple carry for the above adder).

Figure 9-1 shows the general flow of data in the Exemplar Synthesis Tool/Module Generation environment. After the HDL source code is successfully parsed, it is passed on to an inference engine that matches supported operators (like addition) with preferred implementations in the module generation library.

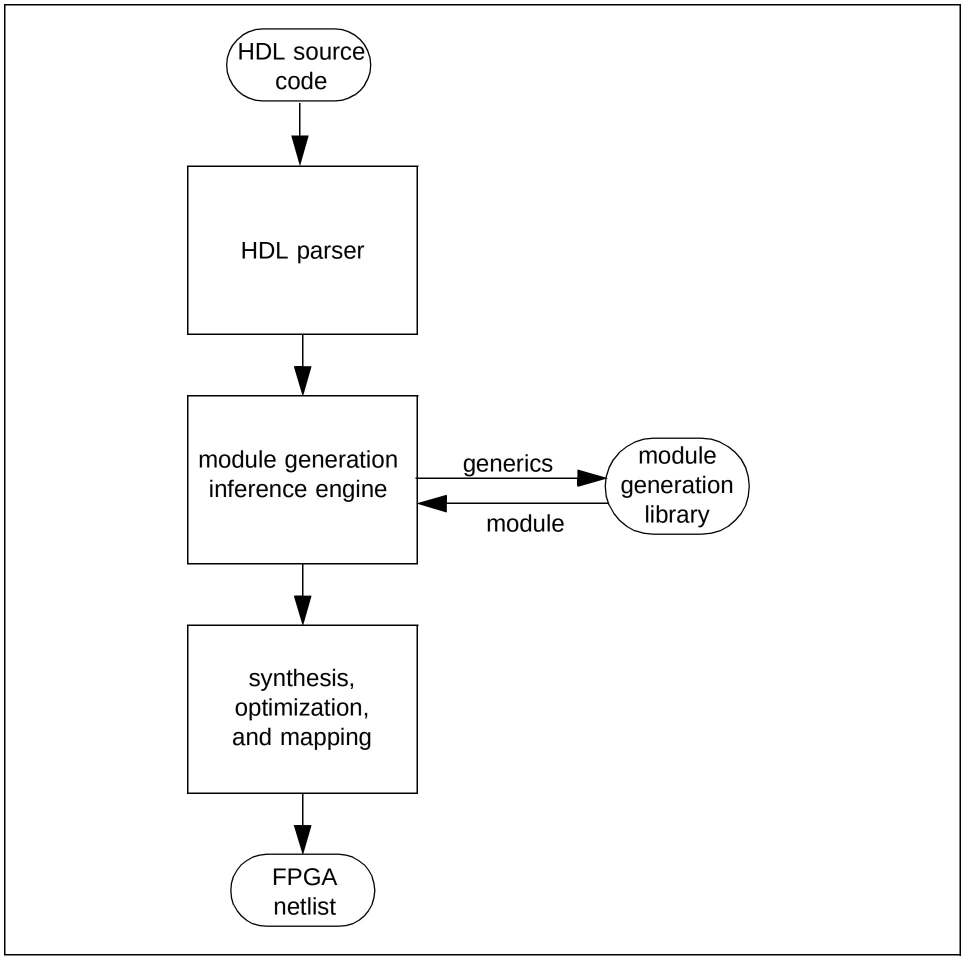
HDL source code
HDL parser
module generation inference engine
generics
module
module generation library
synthesis, optimization, and mapping
FPGA netlist

Figure 9-1 Exemplar Synthesis Tool/Module Generation Environment

As examples of the benefits of *Module Generation*, Figure 9-2 presents the average area reduction achieved when Module Generation is used for synthesis targeting FPGAs, while Figure 9-3 presents the average delay reduction achieved.

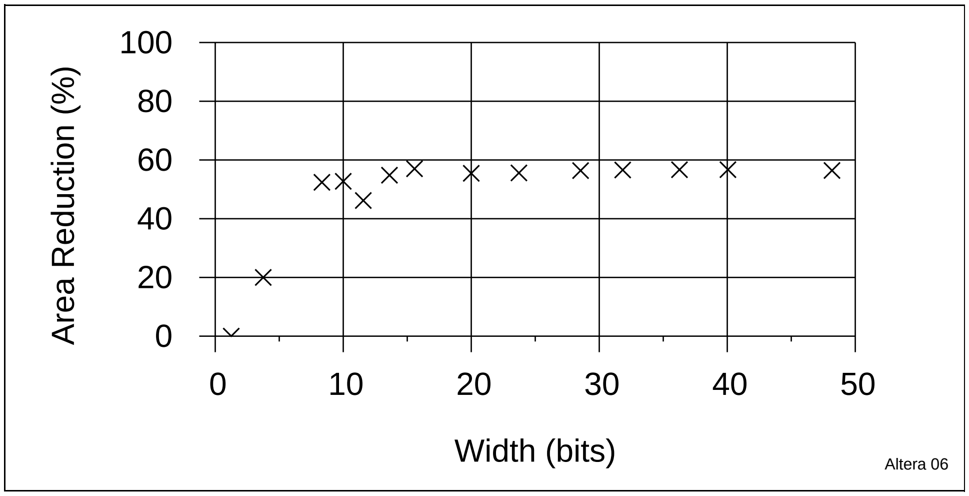

Figure 9-2 Using Module Generation Results in Area Reduction When Adders Are Required

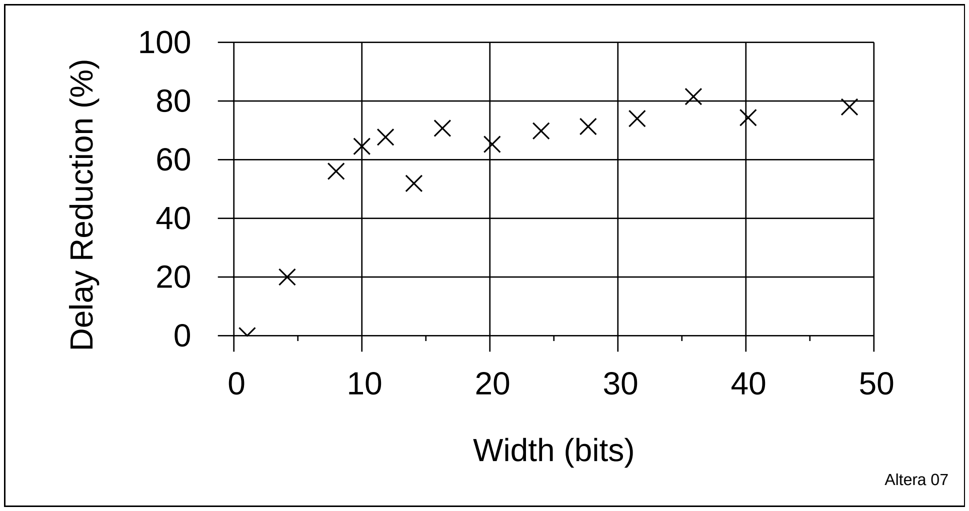

Figure 9-3 Using Module Generation Results In Delay Reduction When Adders Are Required

Using Module Generation 10

This chapter presents information on the use of *Module Generation*: which operators are supported, using *Module Generation* with the Exemplar synthesis tools, and invoking *Module Generation* from both VHDL and Verilog design sources. It focuses on using *Module Generation* for the technologies that are supported in the synthesis tools.

Supported Technologies

A list of currently supported technologies is presented in the Release Notes accompanying this manual. Also, performance information for the module generators are presented in the appropriate chapter in the *Synthesis and Technology Guide*. These data show how *Module Generation* implementation improves area or timing for arithmetic and relational operations, as compared to random logic implementation.

Supported Operators

The following operations are recognized by the synthesis tools for matching with module generation libraries:

Verilog	**VHDL ’87**	**Operation**
"+"	"+"	addition
"-"	"-"	binary subtraction, unary negation
"+ 1"	"+ 1"	increment
"- 1"	"- 1"	decrement
"=="	"="	equal
"!="	"\="	not equal
">"	">"	greater than
"=>"	"=>"	greater than or equal
"<"	"<"	less than
"<="	"<="	less than or equal
"*"	"*"	multiplication
"/"	"/"	division
N/A	"**"	power
"%"	"mod"	modulo
N/A	"rem"	remainder
N/A	"abs"	absolute value

Verilog	VHDL ’93	Operation
">>"	"sra"	shift right logical
"<<"	"sla"	shift left logical
N/A	"sra"	shift right arithmetic
N/A	"sla"	shift left arithmetic
N/A	"rol"	rotate left
"!="	"ror"	rotate right
">"	">"	greater than
"=>"	"=>"	greater than or equal
"<"	"<"	less than
"<="	"<="	less than or equal
"*"	"*"	multiplication
"/"	"/"	division
N/A	"**"	power

From VHDL, the synthesis tools recognize these operations for operators on the predefined type `integer`. It also recognizes these operations from operators for the `bit_vector` and `std_logic_vector` types, as long as the package `exemplar` or `numeric_std` package is included with a `use` clause. For Verilog HDL, the synthesis tools recognize these operations from all (predefined) supported operators in the Verilog HDL language.

Counters and RAMs

Both Leonardo and Galileo can recognize counter and RAM behavior in a VHDL or Verilog HDL description and infer module generators. Counters are positive edge-triggered with optional clock enable and/or count enable, asynchronous clear and/or set, synchronous clear, and synchronous load. Up, down, and up-down counters are supported. The following example is recognized as an 8-bit loadable down-counter with asynchronous clear and clock enable:

Example

```
library ieee, exemplar;
use ieee.std_logic_1164.all;
use exemplar.exemplar_1164.all;

entity cnt_dn_ac_sl_en is
   port (clk, clk_en, aclear, sload: in std_logic;
      data: in std_logic_vector(7 downto 0);
      q: out std_logic_vector(7 downto 0));
end cnt_dn_ac_sl_en;

architecture ex of cnt_dn_ac_sl_en is
   signal q_int: std_logic_vector(q'range);
begin
   process (clk, aclear)
      begin
         if (aclear = '1') then
          q_int <= (q_int'range => '0');
       elsif (clk'event and clk'last_value = '0' and clk = '1') then
          if (clk_en = '1') then
             if (sload = '1') then
               q_int <= data;
             else
               q_int <= q_int - "1";
             end if;
          end if;
       end if;
      end process;
   q <= q_int;
end ex;
```

Counter and RAM Inferencing and Module Generation

There are two basic types of RAM Module Generators: a single-port RAM with separate input and output data lines, and a single-port RAM with bidirectional data lines. Both of these RAM types support synchronous or asynchronous read and write operation. Synchronous writes use a positive edge-triggered clock to latch the write-enable, address, and data signals. The inferencing process distinguishes between RAMs that perform the read operation with an address that is clocked or not clocked with the write clock.

The RAM output signals may also be latched by the same or a different positive edge-triggered clock. The following two VHDL examples demonstrate the difference between synchronous RAMs that do or do not clock the read address with the write clock. The first example, `ram_example1`, does clock the read address, while the second example, `ram_example2`, does not clock the read address.

Most technologies only support one of these types. In addition, particular technology Modgen libraries may not contain module generators for all types of RAMs recognized by Leonardo and Galileo. Information concerning which types are supported by a particular technology can be found in the Leonardo Synthesis and Technology Guide and the Galileo Synthesis and Technology Guide.

Example 1

```
library ieee, exemplar;
use ieee.std_logic_1164.all;
use exemplar.exemplar_1164.all;

entity ram_example1 is
     port (data: in std_logic_vector(7 downto 0);
         address: in std_logic_vector(5 downto 0);
          we, inclock, outclock: in std_logic;
          q: out std_logic_vector(7 downto 0));
end ram_example1;

architecture ex1 of ram_example1 is
     type mem_type is array (63 downto 0) of
          std_logic_vector (7 downto 0);
     signal mem: mem_type;
begin
    l0: process (inclock, outclock, we, address) begin
          if (inclock = '1' and inclock'event) then
               if (we = '1') then
                    mem(evec2int(address)) <= data;
              end if;
          end if;
          if (outclock = '1' and outclock'event) then
              q <= mem(evec2int(address));
          end if;
     end process;
end ex1;
```

Example 2

```
library ieee, exemplar;
use ieee.std_logic_1164.all;
use exemplar.exemplar_1164.all;

entity ram_example2 is
   port (data: in std_logic_vector(7 downto 0);
         address: in std_logic_vector(5 downto 0);
         we, inclock, outclock: in std_logic;
         q: out std_logic_vector(7 downto 0));
   end ram_example2;

architecture ex2 of ram_example2 is
   type mem_type is array (63 downto 0) of
         std_logic_vector (7 downto 0);
   signal mem: mem_type;
   signal address_int: std_logic_vector(5 downto 0);
begin
   l0: process (inclock, outclock, we, address) begin
         if (inclock = '1' and inclock'event) then
            address_int <= address;
               if (we = '1') then
                  mem(evec2int(address)) <= data;
              end if;
        end if;
        if (outclock = '1' and outclock'event) then
            q <= mem(evec2int(address_int));
        end if;
   end process;
end ex2;
```

Using Module Generation With Exemplar Synthesis Tools

Specifying Module Generation Library

Module Generation is invoked by including a module generation library during logic synthesis.

From the command line for Galileo, use the `-modgen=`*modgen_library* option to include a module generation library of the specified technology and infer the required arithmetic and relational operations of the required size from a user VHDL design. For Leonardo, use the `modgen_read` *modgen_library* command to load the module generation library into the HDL database. Since these modules have been designed optimally for a target technology, the synthesis result is, in general, smaller and/or faster and takes less time to compile.

The module generation library can have any name, without an extension. All the module generator files provided by Exemplar Logic are named *lib_base_name*`.vhd`, where *lib_base_name* is the technology library base name. These files can be found in the directory `$EXEMPLAR/data/modgen`. Since the directory is in the search path for the synthesis tools, if you specify a module generation library, the synthesis tools will read the file with the matching technology name. These files are encrypted.

The Exemplar synthesis tools do not validate the generator. If, for instance, an Actel technology is specified as the target technology, but accidentally a Xilinx module generation library is specified, Xilinx macros will appear in the output netlist.

Area/Delay Trade-offs Attributes

Implementations of area and delay trade-offs may vary between module generator packages. Galileo will choose a smaller or faster implementation, depending on the area/delay switch in the GUI, or `-area` versus `-delay` option in the command line. With Leonardo, the method for choosing between smaller and faster implementations is to use the `-area` or `-delay` options to the `optimize` command.

Specific implementations can be configured in the VHDL file through attributes on specific signals. The attribute `modgen_sel` is used for this purpose. `modgen_sel` is an attribute of enumerated type `modgen_select`, with four values: `smallest`, `small`, `fast`, `fastest`. This attribute controls which implementation of a module generator is used. By default, the synthesis tools use `small` if the global optimization criteria is `-area`. The synthesis tools choose `fast` if the `-delay` switch is set. The user can overwrite these defaults by specifying the attribute `modgen_sel` on a target signal or variable that is driven by an expression that calls module generators. Here is an example:

```
type modgen_select is (smallest, small, fast, fastest) ;
attribute modgen_sel : modgen_select ;
signal a,b,c,s : bit_vector (7 downto 0) ;
attribute modgen_sel of s:signal is smallest ;
...
s <= a + b + c ;
```

In this example, for both adders that drive `s`, *Module Generation* will choose the smallest implementation possible. In essence, the `modgen_sel` attribute is passed to the module generator inference engine where a different implementation, other than the default, is selected.

The type `modgen_select` and the attribute `modgen_sel` are declared in the packages `exemplar` and `exemplar_1164`. Hence, if you use one of these packages, declaring them is not required in the user code.

Disabling Module Generation

Once the `-modgen` option is specified, *Module Generation* is enabled for all arithmetic and relational operators in the design. *Module Generation* can be switched off for all operator calls driving a particular signal, by setting the boolean `use_modgen` to FALSE.

```
attribute use_modgen : boolean ;
signal a,b,c,s : bit_vector (7 downto 0) ;
attribute use_modgen of s:signal is FALSE ;
--
s <= a + b + c ;
```

In this case, for both adders that drive `s`, *Module Generation* is disabled and the adders will be implemented in random logic. Disabling *Module Generation* for specific signals or variables can be useful when large portions of the operators can be eliminated during the boolean optimization and synthesis process. This often happens for user defined type-transformation functions, where the operators implement simulation behavior, but for synthesis the function should implement a simple set of wires. Using *Module Generation* for such function would generate a large amount of arithmetic logic when it is not required. The attribute `use_modgen` is defined in the `exemplar` and `exemplar_1164` packages. If one of these packages is used, declaring the attribute is not required in the user code.

Counter and RAM Extraction

In Galileo, counters and RAMs are recognized and extracted by default. In Leonardo, the `pre_optimize` command with the `-extract` option must be executed.

Verilog Usage

Verilog usage of *Module Generation* is completely straightforward. *Module Generation* will infer the arithmetic and relational operators from Verilog descriptions and implement them accordingly.

User-Defined Module Generators 11

Apart from the module generators that have been developed by Exemplar to support the standard FPGA technologies, a user can build his/her own module generator.

The purpose of this chapter is to set guidelines and boundary conditions on how to use the module generation environment to produce user-defined module generators with the intended functionality.

Module generators are described in VHDL, regardless of the actual HDL input design language.

User-defined module generators, as opposed to using overloaded functions, allow the use of technology specific macros (with component instantiation) for operators in VHDL or Verilog HDL.

The Module Generator Boundary

Since all operators in VHDL are defined for various sized vectors and integers, each module generator description for a particular operator should be an entity with generics.

Only one generic affects the amount of inputs and outputs that have to be generated. This is the integer generic `size`. The amount of inputs and outputs generated by a modgen description should exactly match the amount required by `size`. Any discrepancy will be labeled as an error. Of course, the functionality inside the modgen description is the responsibility of the modgen description designer. It is relatively easy to let a `"+"` in VHDL work as a `"-"` with this amount of freedom.

Since the function of some operators is defined both for unsigned integers (or vectors) and for signed integers, a boolean generic `signed` is supplied to indicate that a signed or unsigned function needs to be generated.

Table 11-1 on page 3 states which VHDL operators are supported in the *Module Generation* environment, which generics are required, how many inputs are needed for each (of the two) parameters of the operator and how many outputs should be generated.

Note that the generic `signed` is not required for arithmetic operations. The reason is that there is no difference between signed and unsigned arithmetic functions if the input parameters and the output all have the same `size`, and thus the carry bit is not used. The synthesis tools will make sure that this always happens.

In general, the module description should have two input vectors (one for each parameter of the operator it represents), and one output vector.

Table 11-1 Supported Operators, Their Module Generators And An Overview Of Boundary Conditions For Correct Matching Of Operators And Module Generation

VHDL'87	Modgen	Required	# of Input Bits		
Operator	**Module Name**	**Generics**	**par.1**	**par. 2**	**# of Output Bits**
"+"	modgen_add	size	size	size	size
"-"	modgen_sub	size	size	size	size
"-"	modgen_umin	size	size	n/a	size
"+ 1"	modgen_inc	size	size	n/a	size
"- 1"	modgen_dec	size	size	n/a	size
"*"	modgen_mult	size	size	size	size
"/"	modgen_div	size	size	size	size
"="	modgen_eq	size	size	size	1 bit
"/="	modgen_ne	size	size	size	1 bit
"<"	modgen_lt	size, signed	size	size	1 bit
">"	modgen_gt	size, signed	size	size	1 bit
"<="	modgen_le	size, signed	size	size	1 bit
"=>"	modgen_ge	size, signed	size	size	1 bit
"**"	modgen_power	size	size	size	size
"mod"	modgen_mod	size	size	size	size
"rem"	modgen_rem	size	size	size	size
"abs"	modgen_abs	size	size	n/a	size

VHDL '93	Modgen	Required	# of Input Bits		
Operator	**Module Name**	**Generics**	**par.1**	**par. 2**	**# of Output Bits**
"sll"	sll	size	size	size	size
"srl"	srl	size	size	size	size
"sla"	sra	size	size	size	size
"sra"	sra	size	size	size	size
"ror"	ror	size	size	size	size
"rol"	rol	size	size	size	size

As an example, the entity VHDL description for a module generator that implements a "<=" operator should look like this:

```
entity modgen_le is
       generic (
              size : integer := 8 ;
              signed : boolean := FALSE
       ) ;
       port (
              x, y : std_logic_vector (size-1 downto 0) ;
              result : out std_logic
       ) ;
end modgen_le ;
```

Below are some important facts to keep in mind when defining module generators:

- The initial assignments to both `signed` and `size` are optional. These two generics are required for the `"<="` operator and therefore are always inferred by the synthesis tools for each call of a `"<="` operator in VHDL.
- The types of the ports should represent arrays of bit values or single bit values. The type `std_logic_vector` for vector types and `std_logic` for bit values are advised because they comply with the IEEE 1164 standard type definitions. Make sure you include the IEEE 1164 package in your description. Use the following statement before each new entity:

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

- The names of the ports can be chosen freely. The associations are order dependent. The first input port (`x` in this example) will be associated with the parameter on the left of the operator. The second port mentioned in the port interface list will be associated with the parameter on the right of the operator.
- The output port mentioned (there can be only one) will be associated with the result of the operator function.

- The ’weight’ of the bits in a port which is a vector is also order dependent. The LEFT most bit in the array range definition of the port is the MSB. In this example, `x` is defined with a range `size-1 downto 0` and therefore `x(size-1)` is MSB, and `x(0)` is LSB. If the range would have been defined as `(0 to size-1)`, `x(0)` would have been MSB.
- If signed operation is required (signed is TRUE), the *Module Generation* environment expects the MSB bit to be the sign bit, and the bit next to it will be the new MSB.

Module Generator Contents

The VHDL entity for a module generator is relatively fixed for each module generator, as shown in the previous section. This is needed to provide a guaranteed interface between the module generators and VHDL operators they implement.

The contents of the module generators (the VHDL architecture) is completely left up to the user. You can use all VHDL constructs as long as they do not violate the VHDL synthesis restrictions.

Typically, component instantiations of technology specific macros will be used in the module generators. Some guidelines should be considered when making module generators:

1. Make sure that the module generator has a definition for each generic ’size’ that could be used from a user HDL description.
2. The synthesis tools do not check the functionality of the module generator. It would be fairly easy to implement subtractor functionality for the `modgen_add` module generator. In that case, each `"+"` operator in VHDL will build a subtractor circuit. Make sure you verify the module generators for each generic size they could implement.
3. If you use operators inside a module generator description, the synthesis tools will NOT try to infer a module generator for these. Instead, the default random-logic implementation for the operator will be chosen. This prevents infinite recursion from occurring (module generators calling themselves). It also allows the user to utilize a specific implementation operator for just a few sizes, and rely on the default implementation for all others.

Below is an example of a module generator that implements an `ADDER8` hard-macro if the size of the required adder is between 4 and 8.

```
library ieee ;
use ieee.std_logic_1164.all ;    -- Include IEEE 1164 type
                             -- definition
library exemplar ;
use exemplar.exemplar_1164.all ; -- Include functions ’extend’, "+"
                              -- etc.
entity modgen_add is
   generic (size : integer) ;
   port (x, y : std_logic_vector (size-1 downto 0) ;
            o : out std_logic_vector (size-1 downto 0)) ;
end modgen_add ;

architecture exemplar of modgen_add is
   -- Declare the Hard Macro
   component ADDER8
      port (a, b: in std_logic_vector(7 downto 0);
            add: in std_logic;
            s: out std_logic_vector(7 downto 0);
            ofl: out std_logic);
   end component;
   -- Declare internally used signals
   signal intern_a, intern_b, intern_o :
          std_logic_vector (7 downto 0) ;
   constant pwr : std_logic := ’1’ ;
```

```
--ADDER8 hard macro example (cont.)
   begin

      l1 : if size>=4 and size <=8 generate
         -- Adjust the inputs to the size of the hard macro
         intern_a <= extend (x,8) ;
         intern_b <= extend (y,8) ;

         -- Instantiate the Hard Macro
         i1 : ADDER8 port map (a=>intern_a, b=>intern_b,
                                add=>pwr,
         s=>intern_o, ofl=>OPEN) ;

         -- For the output :pick-up the LSB bits from the hard macro
         o <= intern_o (size-1 downto 0) ;

      end generate ;

      -- Default "+" for all other sizes :
      l2 : if size<4 or size>8 generate
      o <= x + y ;
      end generate ;

   end exemplar ;
```

This is the description of a full definition of a module generator that instantiates an `ADDER8` hard macro (generic name, not from any specific library, used for this example) for adders between 4 and 8 bits. A default implementation (random logic) is provided for all sizes of adders that should not generate a hard macro.

Usage

To include a module generator description into Galileo, use the `-modgen=`*modgen_library* option to include a module generation library of the specified technology and infer the required arithmetic and relational operations of the required size from a user VHDL design. For Leonardo, use the `modgen_read` *modgen_library* command to load the module generation library into the HDL database.

The search path for these files is:

1. The current working directory
2. The `$EXEMPLAR/data/modgen` directory
3. The `$EXEMPLAR/data` directory

Multiple module generator files can be included. If there is an overlap of operators in two included files, the operator from the last included file will be resolved. In any case, for each operator resolved, Galileo reports the file that was used. Therefore, it will be clear which operator has been resolved from which modgen file.

VHDL Index

Verilog Index

Modgen Index