Chapter 3
The Art of VHDL Synthesis

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.

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.

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, LeonardoSpectrum translates 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. LeonardoSpectrum recognizes 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`. LeonardoSpectrum supports both attributes.

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 ;
```

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. LeonardoSpectrum issues 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. LeonardoSpectrum 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, as long as all of the control paths inside the process contain at least one 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 ;
```

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. Multiple `wait` statements per process are also supported as long as all of the statements have the same `wait until` clause.

LeonardoSpectrum does 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 LeonardoSpectrum to instantiate the required flip-flop or D-latch. There are various forms of these procedures available, including versions with asynchronous preset and clear.

Assigning I/O Buffers From VHDL

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

- Run LeonardoSpectrum 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 LeonardoSpectrum does in "chip" mode.

It is important to realize that if you specify buffer names in the VHDL source, LeonardoSpectrum checks the source technology library to find the buffer you requested. If you specify buffers in the control file, LeonardoSpectrum checks 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.

- Use the `-chip` batch mode option on the command line.
- Use the `-chip` option with the `optimize` command in the interactive shell. 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.

Manual Assignment Using The BUFFER_SIG Property

Special buffers, e.g. clock buffers, can be assigned using the `buffer_sig` property. This can be done with the `BUFFER_SIG` command. Here is an example:

```
BUFFER_SIG CLOCK_BUFFER clk
```

For LeonardoSpectrum, 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` is 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 rtl 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 rtl ;
```

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.

LeonardoSpectrum 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, LeonardoSpectrum 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 rtl of three-state is
begin
   output_signal <= input_signal when ena = '1' else
'Z' ;
end rtl ;
```

In the when clause, both `input_signal` and the condition `ena='1'` can be full expressions. LeonardoSpectrum generates 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 rtl 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 rtl ;
```

LeonardoSpectrum does 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 buses 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, then use one of the following methods:

- Transform the three-state buffers into regular logic with the `-tristate` batch mode option.
- Set the `tristate_map` variable is set to `TRUE` in the interactive shell.

Bidirectional Buffers

Bidirectional I/O buffers will be created by LeonardoSpectrum 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 rtl 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 rtl ;
```

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.

LeonardoSpectrum selects 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.

Buses

The examples in the previous sections all use single bits as signals. In reality, buses are often used: arrays of bits with (multiple) three-state drivers. In this case, the type of the bus signal

should be std_logic_vector. All examples given still apply for buses, 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 (7 downto 0) ;
      ena_1, ena_2 : in std_logic ;
      output_signal : out std_logic_vector(7 downto
0)
            ) ;
end three-state ;

architecture rtl 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 rtl ;
```

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

As with single three-state drivers, buses can be an internal signal, or ports. Similarly, buses 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 rtl 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 rtl ;
```

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).

Graphical state-machine entry tools often generate state machine descriptions that do not always assign values to the outputs under all conditions. LeonardoSpectrum will issue a warning 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 LeonardoSpectrum does state encoding on the enumeration type of the state machine, 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, an asynchronous reset is used, but a synchronous reset would be possible. Registers, Latches, and Reset explains more about how to specify resets on registers in VHDL.

Encoding Methods

LeonardoSpectrum has several methods to control encoding for state machines that use an enumerated type for the declaration of the states. In this chapter, the section Enumerated Types discusses state encoding methods in detail.

Arithmetic And Relational Logic

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.

LeonardoSpectrum supports 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 LeonardoSpectrum Synthesis and Technology Manual 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.

By default, LeonardoSpectrum generates "random" logic for all pre-defined operators. Alternatively, if a module generator for a particular target technology is supplied, LeonardoSpectrum 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 LeonardoSpectrum to make the most efficient technology specific implementation for arithmetic and relational operations. Use the following:

- Use the batch mode option `-modgen=`*`modgen_library`* 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.
- Use the interactive shell `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.

Resource Sharing

LeonardoSpectrum performs 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;
```

LeonardoSpectrum automatically performs 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. LeonardoSpectrum has 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, LeonardoSpectrum builds 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 LeonardoSpectrum does automatic inference of technology specific macros from standard (technology independent) arithmetic and relational operators when Module Generation is used. 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 LeonardoSpectrum, 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) ;
```

LeonardoSpectrum synthesizes this component as a black-box, since there is no entity/architecture description. The black box appears 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 following to load the source library into the design database:

- Batch mode option, `-source=fpgax`.
- Interactive shell, `load_library fpgax` command.

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.

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, LeonardoSpectrum creates 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 resembles 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 creates the multiplexer function:

```
signal vec : std_logic_vector (15 downto 0) ;
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, LeonardoSpectrum creates the same multiplexers as they do for the variable-indexed array.

LeonardoSpectrum fully supports 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.

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

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY async_sevenseg IS
   PORT(
      addr : IN     unsigned (3 DOWNTO 0);
      data : OUT    unsigned (6 DOWNTO 0)
   );

END async_sevenseg ;

ARCHITECTURE rtl OF async_sevenseg IS

SUBTYPE seven_segment IS unsigned(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

BEGIN

data <= hex_to_7 (to_integer(addr)) ;

END rtl;
```

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.

LeonardoSpectrum 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.