# HDL Synthesis Coding Guidelines for Lattice Semiconductor FPGAs

**September 2012** **Technical Note TN1008**

## Introduction

Coding style plays an important role in utilizing FPGA resources. Although many popular synthesis tools have significantly improved optimization algorithms for FPGAs, it still is the responsibility of the user to generate meaningful and efficient HDL code to guide their synthesis tools to achieve the best result for a specific architecture. This application note is intended to help designers establish useful HDL coding styles for Lattice Semiconductor FPGA devices. It includes VHDL and Verilog design guidelines for both novice and experienced users.

The application note is divided into two sections. The general coding styles for FPGAs section provides an overview for effective FPGA designs. The following topics are discussed in detail:

- Hierarchical Coding
- Design Partitioning
- Encoding Methodologies for State Machines
- Coding Styles for Finite State Machines (FSM)
- Using Pipelines
- Comparing IF Statements and CASE Statements
- Avoiding Non-intentional Latches

The HDL Design with Lattice Semiconductor FPGA Devices section covers specific coding techniques and examples:

- Using the Lattice Semiconductor FPGA Synthesis Library
- Implementation of Multiplexers
- Creating Clock Dividers
- Register Control Signals (CE, LSR, GSR)
- Using PIC Features
- Implementation of Memories
- Preventing Logic Replication and Fanout
- Comparing Synthesis Results and Place and Route Results

## General Coding Styles for FPGAs

The following recommendations for common HDL coding styles will help users generate robust and reliable FPGA designs.

### Hierarchical Coding

HDL designs can either be synthesized as a flat module or as many small hierarchical modules. Each methodology has its advantages and disadvantages. Since designs in smaller blocks are easier to keep track of, it is preferred to apply hierarchical structure to large and complex FPGA designs. Hierarchical coding methodology allows a group of engineers to work on one design at the same time. It speeds up design compilation, makes changing the implementation of key blocks easier, and reduces the design period by allowing the re-use of design modules for current and future designs. In addition, it produces designs that are easier to understand. However, if the design mapping into the FPGA is not optimal across hierarchical boundaries, it will lead to lower device utilization and design performance. This disadvantage can be overcome with careful design considerations when choosing the design hierarchy. Here are some tips for building hierarchical structures:

- The top level should only contain instantiation statements to call all major blocks
- Any I/O instantiations should be at the top level
- Any signals going into or out of the devices should be declared as input, output or bi-directional pins at the top level

- Memory blocks should be kept separate from other code

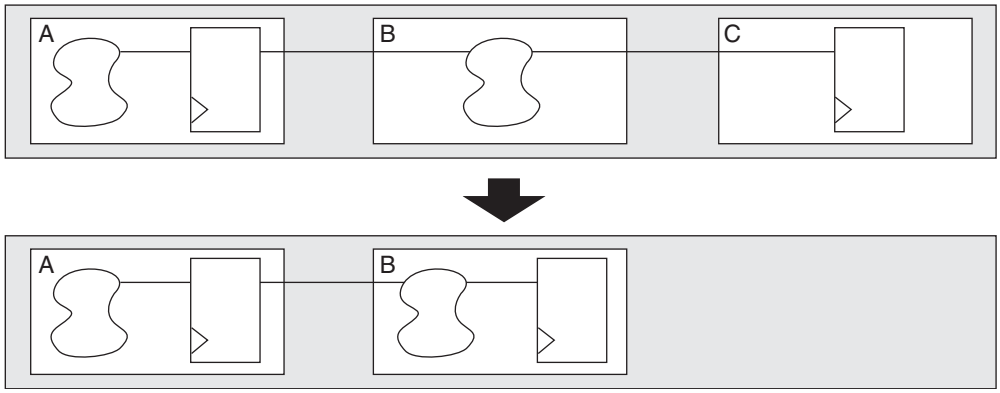
## Design Partitioning

By effectively partitioning the design, a designer can reduce overall run time and improve synthesis results. Here are some recommendations for design partitioning.

### Maintain Synchronous Sub-blocks by Registering All Outputs

It is suggested to arrange the design boundary such that the outputs in each block are registered. Registering outputs helps the synthesis tool to consider the implementation of the combinatorial logic and registers into the same logic block. Registering outputs also makes the application of timing constraints easier since it eliminates possible problems with logic optimization across design boundaries. Single clock is recommended for each synchronous block because it significantly reduces the timing consideration in the block. It leaves the adjustment of the clock relationships of the whole design at the top level of the hierarchy. Figure 15-1 shows an example of synchronous blocks with registered outputs.

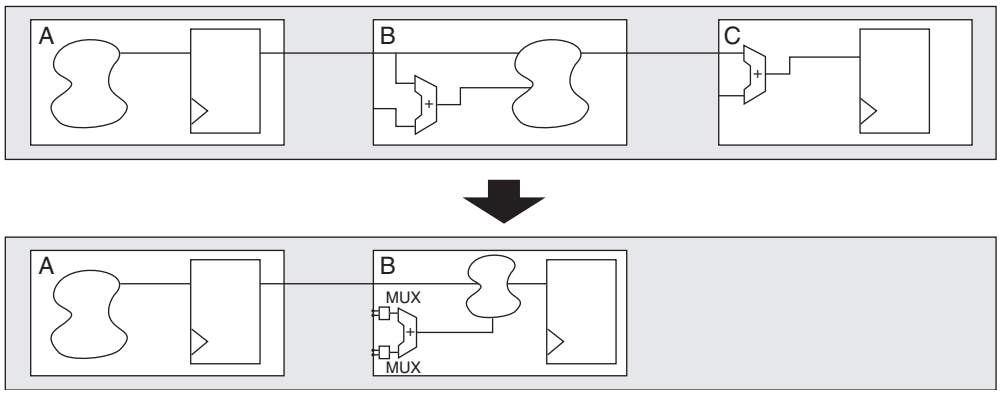
***Figure 15-1. Synchronous Blocks with Registered Outputs***




### Keep Related Logic Together in the Same Block

Keeping related logic and sharable resources in the same block allows the sharing of common combinatorial terms and arithmetic functions within the block. It also allows the synthesis tools to optimize the entire critical path in a single operation. Since synthesis tools can only effectively handle optimization of certain amounts of logic, optimization of critical paths pending across the boundaries may not be optimal. Figure 15-2 shows an example of merging sharable resource in the same block.

***Figure 15-2. Merge Sharable Resource in the Same Block***




### Separate Logic with Different Optimization Goals

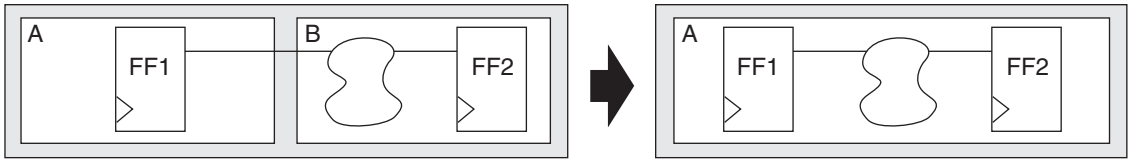
Separating critical paths from non-critical paths may achieve efficient synthesis results. At the beginning of the project, one should consider the design in terms of performance requirements and resource requirements. If there are

two portions of a block, one that needs to be optimized for area and a second that needs to be optimized for speed, they should be separated into two blocks. By doing this, different optimization strategies for each module can be applied without being limited by one another.

### Keep Logic with the Same Relaxation Constraints in the Same Block

When a portion of the design does not require high performance, this portion can be applied with relaxed timing constraints such as “multicycle” to achieve high utilization of device area. Relaxation constraints help to reduce overall run time. They can also help to efficiently save resources, which can be used on critical paths. Figure 15-3 shows an example of grouping logic with the same relaxation constraint in one block.

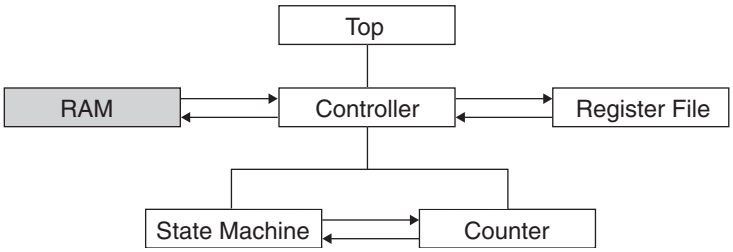
***Figure 15-3. Logic with the Same Relaxation Constraint***




### Keep Instantiated Code in Separate Blocks

It is recommended that the RAM block in the hierarchy be left in a separate block (Figure 15-4). This allows for easy swapping between the RAM behavioral code for simulation, and the code for technology instantiation. In addition, this coding style facilitates the integration of the ispLEVER® IPexpress™ tool into the synthesis process.

***Figure 15-4. Separate RAM Block***




### Keep the Number FPGA Gates at 30 to 80 PFUs Per Block

This range varies based on the computer configuration, time required to complete each optimization run, and the targeted FPGA routing resources. Although a smaller block methodology allows more control, it may not produce the most efficient design since it does not provide the synthesis tool enough logic to apply “Resource Sharing” algorithms. On the other hand, having a large number of gates per block gives the synthesis tool too much to work on and causes changes that affect more logic than necessary in an incremental or multi-block design flow.

## State Encoding Methodologies for State Machines

There are several ways to encode a state machine, including binary encoding, gray-code encoding and one-hot encoding. State machines with binary or gray-code encoded states have minimal numbers of flip-flops and wide combinatorial functions, which are typically favored for CPLD architectures. However, most FPGAs have many flip-flops and relatively narrow combinatorial function generators. Binary or gray-code encoding schemes can result in inefficient implementation in terms of speed and density for FPGAs. On the other hand, one-hot encoded state machine represents each state with one flip-flop. As a result, it decreases the width of combinatorial logic, which matches well with FPGA architectures. For large and complex state machines, one-hot encoding usually is the preferable method for FPGA architectures. For small state machines, binary encoding or gray-code encoding may be more efficient.

There are many ways to ensure the state machine encoding scheme for a design. One can hard code the states in the source code by specifying a numerical value for each state. This approach ensures the correct encoding of the state machine but is more restrictive in the coding style. The enumerated coding style leaves the flexibility of state machine encoding to the synthesis tools. Most synthesis tools allow users to define encoding styles either through

attributes in the source code or through the tool's Graphical User Interface (GUI). Each synthesis tool has its own synthesis attributes and syntax for choosing the encoding styles. Refer to the synthesis tools documentation for details about attributes syntax and values.

The following syntax defines an enumeration type in VHDL:

```
type type_name is (state1_name,state2_name,.....,stateN_name)
```

Here is a VHDL example of enumeration states:

```
type STATE_TYPE is (S0,S1,S2,S3,S4);
signal CURRENT_STATE, NEXT_STATE : STATE_TYPE;
```

The following are examples of Synplify® and LeonardoSpectrum® VHDL synthesis attributes.

Synplify:

```
attribute syn_encoding : string;
attribute syn_encoding of <signal_name> : type is "value ";
-- The syn_encoding attribute has 4 values : sequential, onehot, gray and safe.
```

LeonardoSpectrum:

```
-- Declare TYPE_ENCODING_STYLE attribute
-- Not needed if the exemplar_1164 package is used
type encoding_style is (BINARY, ONEHOT, GRAY, RANDOM, AUTO);
attribute TYPE_ENCODING_STYLE : encoding_style;
...
attribute TYPE_ENCODING_STYLE of <typename> : type is ONEHOT;
```

In Verilog, one must provide explicit state values for states. This can be done by using the bit pattern (e.g., 3'b001), or by defining a parameter and using it as the case item. The latter method is preferable. The following is an example using parameter for state values.

```
Parameter state1 = 2'h1, state2 = 2'h2;
...
current_state = state2; // setting current state to 2'h2
```

The attributes in the source code override the default encoding style assigned during synthesis. Since Verilog does not have predefined attributes for synthesis, attributes are usually attached to the appropriate objects in the source code as comments. The attributes and their values are case sensitive and usually appear in lower case. The following examples use attributes in Verilog source code to specify state machine encoding style.

Synplify:

```
Reg[2:0] state; /* synthesis syn_encoding = "value" */;
// The syn_encoding attribute has 4 values : sequential, onehot, gray and safe.
```

In LeonardoSpectrum, it is recommended to set the state machine variable to an enumeration type with enum pragma. Once this is set in the source code, encoding schemes can be selected in the LeonardoSpectrum GUI.

LeonardoSpectrum:

```
Parameter /* exemplar enum <type_name> */ s0 = 0, s1 = 1, s2 = 2, s3 = 3, S4 = 4;
Reg [2:0] /* exemplar enum <type_name> */ present_state, next_state ;
```

In general, synthesis tools will select the optimal encoding style that takes into account the target device architecture and size of the decode logic. One can always apply synthesis attributes to override the default encoding style if necessary.

## Coding Styles for FSM

A finite state machine (FSM) is a hardware component that advances from the current state to the next state at the clock edge. As mentioned in the Encoding Methodologies for State Machines section, the preferable scheme for FPGA architectures is one-hot encoding. This section discusses some common issues encountered when constructing state machines, such as initialization and state coverage, and special case statements in Verilog.

### General State Machine Description

Generally, there are two approaches to describe a state machine. One is to use one process/block to handle both state transitions and state outputs. The other is to separate the state transition and the state outputs into two different process/blocks. The latter approach is more straightforward because it separates the synchronous state registers from the decoding logic used in the computation of the next state and the outputs. This will make the code easier to read and modify, and makes the documentation more efficient. If the outputs of the state machine are combinatorial signals, the second approach is almost always necessary because it will prevent the accidental registering of the state machine outputs.

The following examples describe a simple state machine in VHDL and Verilog. In the VHDL example, a sequential process is separated from the combinatorial process. In Verilog code, two *always* blocks are used to describe the state machine in a similar way.

**VHDL Example for State Machine**

```
. . .
architecture lattice_fpga of dram_refresh is
   type state_typ is (s0, s1, s2, s3, s4);
   signal present_state, next_state : state_typ;
begin
   -- process to update the present state
   registers: process (clk, reset)
   begin
      if (reset='1') then
         present_state <= s0;
      elsif clk'event and clk='1' then
         present_state <= next_state;
      end if;
   end process registers;

   -- process to calculate the next state & output
   transitions: process (present_state, refresh, cs)
   begin
      ras <= '0'; cas <= '0'; ready <= '0';
      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;
         when others =>
            ras <= '0'; cas <= '0'; ready <= '0';
            next_state <= s0;
      end case;
   end process transitions;
. . .
```

**Verilog Example for State Machine**

```
. . .
parameter s0 = 0, s1 = 1, s2 = 2, s3 = 3, s4 = 4;

reg [2:0] present_state, next_state;
reg ras, cas, ready;

// always block to update the present state
always @ (posedge clk or posedge reset)
begin
   if (reset) present_state = s0;
   else present_state = next_state;
end

// always block to calculate the next state & outputs
always @ (present_state or refresh or cs)
begin
   next_state = s0;
   ras = 1'bX; cas = 1'bX; ready = 1'bX;
   case (present_state)
      s0 : 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
      s1 : begin
              next_state = s2; ras = 1'b0; cas = 1'b0; ready = 1'b0;
           end
      s2 : 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
      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
. . .
```

### Initialization and Default State

A state machine must be initialized to a valid state after power-up. This can be done at the device level during power up or by including a reset operation to bring it to a known state. For all Lattice Semiconductor FPGA devices, the Global Set/Reset (GSR) is pulsed at power-up, regardless of the function defined in the design source code. In the above example, an asynchronous reset can be used to bring the state machine to a valid initialization state. In the same manner, a state machine should have a default state to ensure the state machine will not go into an invalid state if not all the possible combinations are clearly defined in the design source code. VHDL and Verilog have different syntax for default state declaration. In VHDL, if a CASE statement is used to construct a state machine, "When Others" should be used as the last statement before the end of the statement, If an IF-THEN-ELSE statement is used, "Else" should be the last assignment for the state machine. In Verilog, use "default" as the last assignment for a CASE statement, and use "Else" for the IF-THEN-ELSE statement.

**When Others in VHDL**

```
...
architecture lattice_fpga of FSM1 is
   type state_typ is (deflt, idle, read, write);
   signal next_state : state_typ;
begin
process(clk, rst)
begin
   if (rst='1') then
      next_state <= idle; dout <= '0';
   elsif (clk'event and clk='1') then
      case next_state is
         when idle =>
            next_state <= read; dout <= din(0);
         when read =>
            next_state <= write; dout <= din(1);
         when write =>
            next_state <= idle; dout <= din(2);
         when others =>
            next_state <= deflt; dout <= '0';
      end case;
   end if;
end process;
...
```

**Default Clause in Verilog**

```
...
// Define state labels explicitly
parameter deflt=2'bxx;
parameter idle =2'b00;
parameter read =2'b01;
parameter write=2'b10;

reg [1:0] next_state;
reg dout;

always @(posedge clk or posedge rst)
   if (rst) begin
      next_state <= idle;
      dout <= 1'b0;
   end
   else begin
      case(next_state)
         idle: begin
                dout <= din[0]; next_state <= read;
               end
         read: begin
                dout <= din[1]; next_state <= write;
               end
         write: begin
                dout <= din[2]; next_state <= idle;
               end
         default: begin
                   dout <= 1'b0; next_state <= deflt;
                  end
```

### Full Case and Parallel Case Specification in Verilog

Verilog has additional attributes to define the default states without writing it specifically in the code. One can use "full_case" to achieve the same performance as "default". The following examples show the equivalent representations of the same code in Synplify. LeonardoSpectrum allows users to apply Verilog-specific options in the GUI settings.

```
…
case (current_state)  // synthesis full_case
   2'b00 : next_state <= 2'b01;
   2'b01 : next_state <= 2'b11;
   2'b11 : next_state <= 2'b00;
…
```

```
…
case (current_state)
   2'b00 : next_state <= 2'b01;
   2'b01 : next_state <= 2'b11;
   2'b11 : next_state <= 2'b00;
   default : next_state <= 2bx;
```

"Parallel_case" makes sure that all the statements in a case statement are mutually exclusive. It is used to inform the synthesis tools that only one case can be true at a time. The syntax for this attribute in Synplify is as follows:

```
// synthesis parallel_case
```

## Using Pipelines in the Designs

Pipelining can improve design performance by restructuring a long data path with several levels of logic and breaking it up over multiple clock cycles. This method allows a faster clock cycle by relaxing the clock-to-output and setup time requirements between the registers. It is usually an advantageous structure for creating faster data paths in register-rich FPGA devices. Knowledge of each FPGA architecture helps in planning pipelines at the

beginning of the design cycle. When the pipelining technique is applied, special care must be taken for the rest of the design to account for the additional data path latency. The following illustrates the same data path before (Figure 15-5) and after pipelining (Figure 15-6).

***Figure 15-5. Before Pipelining***

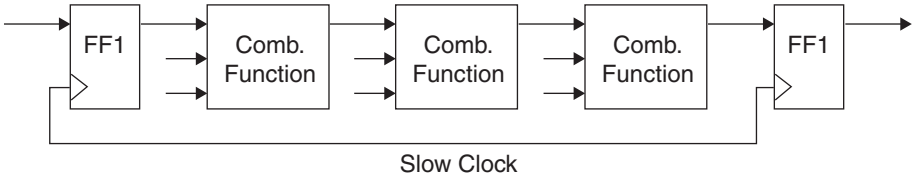
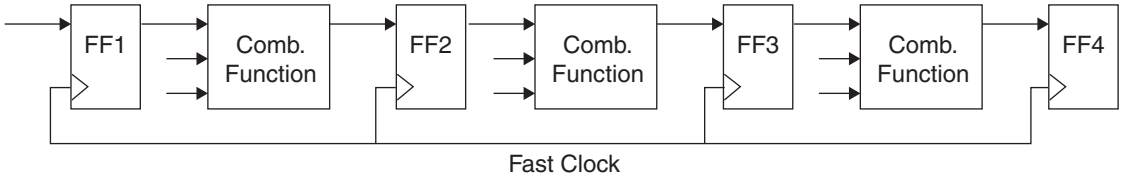



***Figure 15-6. After Pipelining***




Before pipelining, the clock speed is determined by the clock-to-out time of the source register, the logic delay through four levels of combinatorial logic, the associated routing delays, and the setup time of the destination register. After pipelining is applied, the clock speed is significantly improved by reducing the delay of four logic levels to one logic level and the associated routing delays, even though the rest of the timing requirements remain the same. It is recommended to check the Place and Route timing report to ensure that the pipelined design gives the desired performance.

## Comparing IF statement and CASE statement

CASE and IF-THEN-ELSE statements are common for sequential logic in HDL designs. The IF-THEN-ELSE statement generally generates priority-encoded logic, whereas the CASE statement implements balanced logic. An IF-THEN-ELSE statement can contain a set of different expressions while a Case statement is evaluated against a common controlling expression. Both statements will give the same functional implementation if the decode conditions are mutually exclusive, as shown in the following VHDL codes.

```
-- Case Statement — mutually exclusive conditions
process (s, x, y, z)
begin
   O1 <= '0';
   O2 <= '0';
   O3 <= '0';
   case (s) is
      when "00" => O1 <= x;
      when "01" => O2 <= y;
      when "10" => O3 <= z;
   end case;
end process;
```

```
-- If-Then-Else — mutually exclusive conditions
process (s, x, y, z)
begin
   O1 <= '0';
   O2 <= '0';
   O3 <= '0';
   if s = "00" then O1 <= x;
   elsif s = "01" then O2 <= y;
   elsif s = "10" then O3 <= z;
   end if;
end process;
```

However, the use of If-Then-Else construct could be a key pitfall to make the design more complex than necessary, because extra logic are needed to build a priority tree. Consider the following examples:

```
--A: If-Then-Elese Statement: Complex O3 Equations
process(s1, s2, s3, x, y, z)
begin
   O1 <= '0';
   O2 <= '0';
   O3 <= '0';
   if s1 = '1' then
      O1 <= x;
   elsif s2 = '1' then
      O2 <= y;
   elsif s3 = '1' then
      O3 <= z;
   end if;
end process;
```

```
--B: If-Then-Else Statement: Simplified O3 Equation
process (s1, s2, s3, x, y, z)
begin
   O1 <= '0';
   O2 <= '0';
   O3 <= '0';
   if s1 = '1' then
      O1 <= x;
   end if;
   if s2 = '1' then
      O2 <= y;
   end if;
   if s3 <= '1' then
      O3 <= z;
   end if;
 end process;
```

If the decode conditions are not mutually exclusive, IF-THEN-ELSE construct will cause the last output to be dependent on all the control signals. The equation for O3 output in example A is:

```
O3 <= z and (s3) and (not (s1 and s2));
```

If the same code can be written as in example B, most of the synthesis tools will remove the priority tree and decode the output as:

```
O3 <= z and s3;
```

This reduces the logic requirement for the state machine decoder. If each output is indeed dependent of all of the inputs, it is better to use a CASE statement since CASE statements provide equal branches for each output.

### Avoiding Non-intentional Latches

Synthesis tools infer latches from incomplete conditional expressions, such as an IF-THEN-ELSE statements without an Else clause. To avoid non-intentional latches, one should specify all conditions explicitly or specify a default assignment. Otherwise, latches will be inserted into the resulting RTL code, requiring additional resources in the device or introducing combinatorial feedback loops that create asynchronous timing problems. Non-intentional latches can be avoided by using clocked registers or by employing any of the following coding techniques:

- Assigning a default value at the beginning of a process
- Assigning outputs for all input conditions
- Using else, (when others) as the final clause

Another way to avoid non-intentional latches is to check the synthesis tool outputs. Most of the synthesis tools give warnings whenever there are latches in the design. Checking the warning list after synthesis will save a tremendous amount of effort in trying to determine why a design is so large later in the Place and Route stage.

## HDL Design with Lattice Semiconductor FPGA Devices

The following section discusses the HDL coding techniques utilizing specific Lattice Semiconductor FPGA system features. This kind of architecture-specific coding style will further improve resource utilization and enhance the performance of designs.

### Lattice Semiconductor FPGA Synthesis Library

The Lattice Semiconductor FPGA Synthesis Library includes a number of library elements to perform specific logic functions. These library elements are optimized for Lattice Semiconductor FPGAs and have high performance and utilization. The following are the classifications of the library elements in the Lattice Semiconductor FPGA Synthe-

sis Library. The definitions of these library elements can be found in the *Reference Manuals* section of the ispLEVER on-line help system.

- Logic gates and LUTs
- Comparators, adders, subtractors
- Counters
- Flip-flops and latches
- Memory, 4E-specific memory (block RAM function)
- Multiplexors
- Multipliers
- All I/O cells, including I/O flip-flops
- PIC cells
- Special cells, including PLL, GSR, boundary scan, etc.
- FPSC elements

IPepxress, a parameterized module complier optimized for Lattice FPGA devices, is available for more complex logic functions. IPexpress supports generation of library elements with a number of different options such as PLLs and creates parameterized logic functions such as PFU and EBR memory, multipliers, adders, subtractors, and counters. IPexpress accepts options that specify parameters for parameterized modules such as data path modules and memory modules, and produces a circuit description with Lattice Semiconductor FPGA library elements. Output from IPexpress can be written in EDIF, VHDL, or Verilog. In order to use synthesis tools to utilize the Lattice FPGA architectural features, it is strongly recommended to use IPexpress to generate modules for source code instantiation. The following are examples of Lattice Semiconductor FPGA modules supported by IPexpress:

- PLL
- Memory implemented in PFU:
  – Synchronous single-port RAM, synchronous dual-port RAM, synchronous ROM, synchronous FIFO
- Memory implemented with EBR:
  – Quad-port Block RAM, Dual-Port Block RAM, Single-Port Block RAM, ROM, FIFO
- Other EBR based Functions
  – Multiplier, CAM
- PFU based functions
  – Multiplier, adder, subtractor, adder/subtractor, linear feedback shifter, counter
- MPI/System Bus

IPexpress is especially efficient when generating high pin count modules as it saves time in manually cascading small library elements from the synthesis library. Detailed information about IPexpress and its user guide can be found in the ispLEVER help system.

## Implementing Multiplexers

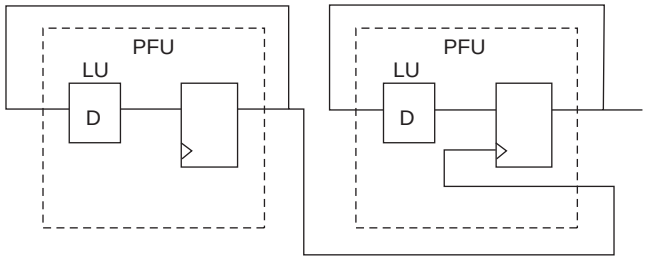
The flexible configurations of LUTs can realize any 4-, 5-, or 6-input logic function like 2-to-1, 3-to-1 or 4-to-1 multiplexers. Larger multiplexers can be efficiently created by programming multiple 4-input LUTs. Synthesis tools camn automatically infer Lattice FPGA optimized multiplexer library elements based on the behavioral description in the HDL source code. This provides the flexibility to the Mapper and Place and Route tools to configure the LUT mode and connections in the most optimum fashion.

```
16:1 MUX
…
process(sel, din)
   begin
      if    (sel="0000") then muxout <= din(0);
      elsif (sel="0001") then muxout <= din(1);
      elsif (sel="0010") then muxout <= din(2);
      elsif (sel="0011") then muxout <= din(3);
      elsif (sel="0100") then muxout <= din(4);
      elsif (sel="0101") then muxout <= din(5);
      elsif (sel="0110") then muxout <= din(6);
      elsif (sel="0111") then muxout <= din(7);
      elsif (sel="1000") then muxout <= din(8);
      elsif (sel="1001") then muxout <= din(9);
      elsif (sel="1010") then muxout <= din(10);
      elsif (sel="1011") then muxout <= din(11);
      elsif (sel="1100") then muxout <= din(12);
      elsif (sel="1101") then muxout <= din(13);
      elsif (sel="1110") then muxout <= din(14);
      elsif (sel="1111") then muxout <= din(15);
      else muxout <= '0';
      end if;
   end process;
…
```

## Clock Dividers

There are two ways to implement clock dividers in Lattice Semiconductor FPGA devices. The first is to cascade the registers with asynchronous clocks. The register output feeds the clock pin of the next register (Figure 15-7). Since the clock number in each PFU is limited to two, any clock divider with more than two bits will require multiple PFU implementations. As a result, the asynchronous daisy chaining implementation of clock divider will be slower due to the inter-PFU routing delays. This kind of delays is usually ambiguous and inconsistent because of the nature of FPGA routing structures.

***Figure 15-7. Daisy Chaining of Flip-flops***



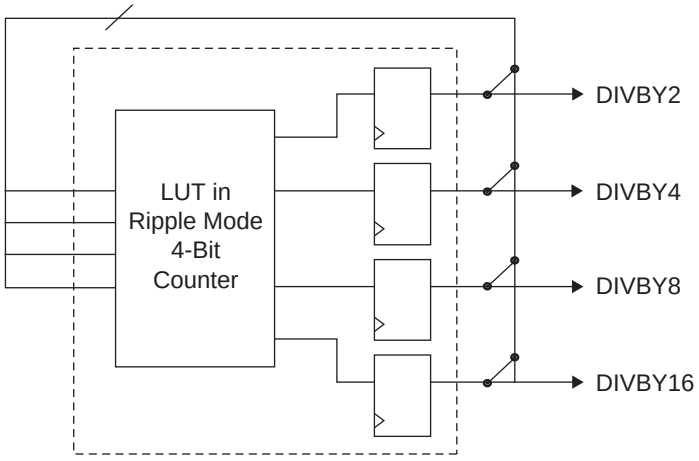
The following are the HDL representations of the design in Figure 15-7.

```
-- VHDL Example of Daisy Chaining FF
...
-- 1st FF to divide Clock in half
CLK_DIV1: process(CLK, RST)
begin
   if (RST='1') then
      clk1 <= '0';
   elsif (CLK'event and CLK='1') then
      clk1 <= not clk1;
   end if;
end process CLK_DIV1;

-- 2nd FF to divide clock in half
CLK_DIV2: process(clk1, RST)
begin
   if (RST='1') then
      clk2 <= '0';
   elsif (clk1'event and clk1='1') then
      clk2 <= not clk2;
   end if;
end process CLK_DIV2;
```

```
//Verilog Example of Daisy Chaining FF
...
always @(posedge CLK or posedge RST)
begin
   if (RST)
      clk1 = 1'b0;
   else
      clk1 = !clk1;
end

always @(posedge clk1 or posedge RST)
begin
   if (RST)
      clk2 = 1'b0;
   else
      clk2 = !clk2;
end

...
```

The preferable way is to fully employ the PFU's natural “Ripple-mode”. A single PFU can support up to 8-bit ripple functions with fast carry logic. Figure 15-8 is an example of 4-bit counter in PFU “Ripple Mode”. In Lattice Semiconductor FPGA architectures, an internal generated clock can get on the clock spine for small skew clock distribution, further enhancing the performance of the clock divider.

***Figure 15-8. Use PFU “Ripple Mode”***




Here are the HDL representations of the design in Figure 15-8.

```
-- VHDL : “RippleMode” Clock Divider
...

COUNT4: process(CLK, RST)
begin
   if (RST='1') then
      cnt <= (others=>'0');
   elsif (CLK'event and CLK='1') then
      cnt <= cnt + 1;
   end if;
end process COUNT4;

DIVBY4  <= cnt(1);
DIVBY16 <= cnt(3);
```

```
//Verilog : “RippleMode” Clock Divider
...
always @(posedge CLK or posedge RST)
begin
   if (RST)
      cnt = 4'b0;
   else
      cnt = cnt + 1'b1;
end

assign DIVBY4  = cnt[1];
assign DIVBY16 = cnt[3];

...
```

## Register Control Signals

The general-purpose latches/FFs in the PFU are used in a variety of configurations depending on device family. For example, the Lattice EC, ECP, SC and XP family of devices clock, clock enable and LSR control can be applied to the registers on a slice basis. Each slice contains two LUT4 lookup tables feeding two registers (programmed asto be in FF or Latch mode), and some associated logic that allows the LUTs to be combined to perform functions such as LUT5, LUT6, LUT7 and LUT8. There is control logic to perform set/reset functions (prgorammable as synchronous/asynchronous), clock select, chip-select and wider RAM/ROM functions. The ORCA Series 4 family of devices clock, clock enable and LSR control can be applied to the registers on a nibble-wide basis. When writing design codes in HDL, keep the architecture in mind to avoid wasting resources in the device. Here are several points for consideration:

- If the register number is not a multiple of 2 or 4 (dependent on device family), try to code the registers in a way that all registers share the same clock, and in a way that all registers share the same control signals.
- Lattice Semiconductor FPGA devices have multiple dedicated Clock Enable signals per PFU. Try to code the asynchronous clocks as clock enables, so that PFU clock signals can be released to use global low-skew clocks.
- Try to code the registers with Local synchronous Set/Reset and Global asynchronous Set/Reset

For more detailed architecture information, refer to the Lattice Semiconductor FPGA data sheets.

### Clock Enable

Figure 15-9 shows an example of gated clocking. Gating clock is not encouraged in digital designs because it may cause timing issues such as unexpected clock skews. The structure of the PFU makes the gating clock even more undesirable since it will use up all the clock resources in one PFU and sometimes waste the FF/ Latches resources in the PFU. By using the clock enable in the PFU, the same functionality can be achieved without worrying about timing issues as only one signal is controlling the clock. Since only one clock is used in the PFU, all related logic can be implemented in one block to achieve better performance. Figure 15-10 shows the design with clock enable signal being used.

***Figure 15-9. Asynchronous: Gated Clocking***

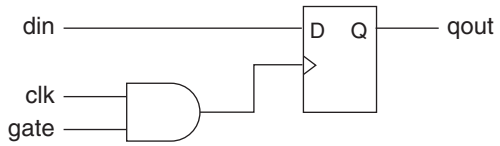
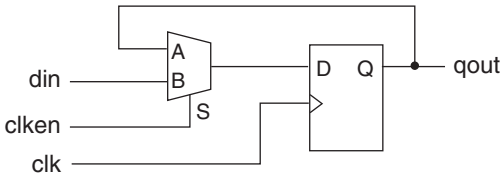



***Figure 15-10. Synchronous: Clock Enabling***




The VHDL and Verilog coding for Clock Enable are as shown in Figure 15-10.

```
-- VHDL example for Clock Enable

...
Clock_Enable: process(clk)
begin
   if (clk'event or clk='1') then
      if (clken='1') then
         qout <= din;
      end if;
   end if;
end process Clock_Enable;
```

```
// Verilog example for Clock Enable

...
always @(posedge clk)
   qout <= clken ? din : qout;

...
```

The following are guidelines for coding the Clock Enable in Lattice Semiconductor FPGAs:

- Clock Enable is only supported by FFs, not latches.
- Nibble wide FFs and slices inside a PFU share the same Clock Enable
- All flip-flops in the Lattice Semiconductor FPGA library have a positive clock enable signal
- In the ORCA Series 4 architecture, the Clock Enable signal has the higher priority over synchronous set/reset by default. However, it can be programmed to have the priority of synchronous LSR over the priority of Clock Enable. This can be achieved by instantiating the library element in the source code. For example, the library element FD1P3IX is a flip-flop that allows synchronous Clear to override Clock Enable. Users can also specify the priority of generic coding by setting the priority of the control signals differently. The following examples demonstrate coding methodologies to help the synthesis tools to set the higher priority of Clock Enable or synchronous LSR.

```
-- VHDL Example of CE over Sync. LSR
...
COUNT8: process(CLK, GRST)
begin
   if (GRST = '1') then
      cnt <= (others => '0');
   elsif (CLK'event and CLK='1') then
   -- CE Over LSR: Clock Enable has higher priority
      if (CKEN = '1') then
            cnt <= cnt + 1;
      elsif (LRST = '1') then
            cnt <= (others =>'0');
      end if;
   end if;
end process COUNT8;
```

```
// Verilog Example of CE over Sync. LSR
...
always @(posedge CLK or posedge GRST)
begin
   if (GRST)
      cnt = 4'b0;
   else
      if (CKEN)
         cnt = cnt + 1'b1;
      else if (LRST)
         cnt = 4'b0;
end...
```

```
-- VHDL Example of Sync. LSR Over CE
...
COUNT8: process(CLK, GRST)
begin
   if (GRST = '1') then
      cnt <= (others => '0');
   elsif (CLK'event and CLK='1') then
   -- LSR over CE: Sync. Set/Reset has higher priority
      if (LRST = '1') then
         cnt <= (others => '0');
      elsif (CKEN = '1') then
         cnt <= cnt + 1;
      end if;
```

```
// Verilog Example of Sync. LSR Over CE
...
always @(posedge CLK or posedge GRST)
begin
   if (GRST)
      cnt = 4'b0;
   else if (LRST)
      cnt = 4'b0;
   else if (CKEN)
      cnt = cnt + 1'b1;
end
...
```

**SET / Reset**

There are two types of set/reset functions in Lattice Semiconductor FPGAs: Global (GSR) and Local (LSR). The GSR signal is asynchronous and is used to initialize all registers during configuration. It can be activated either by an external dedicated pin or from internal logic after configuration. The local SET/Reset signal may be synchronous or asynchronous. GSR is pulsed at power up to either set or reset the registers depending on the configuration of the device. Since the GSR signal has dedicated routing resources that connect to the set and reset pin of the flip-flops, it saves general-purpose routing and buffering resources and improves overall performance. If asynchronous reset is used in the design, it is recommended to use the GSR for this function, if possible. The reset signal can be forced to be GSR by the instantiation library element. Synthesis tools will automatically infer GSR if all

registers in the design are asynchronously set or reset by the same wire. The following examples show the correct syntax for instantiating GSR in the VHDL and Verilog codes.

```
-- VHDL Example of GSR Instantiation
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity gsr_test is
          port (rst, clk: in std_logic;
          cntout : out std_logic_vector(1 downto 0));
end gsr_test;

architecture behave of gsr_test is
          signal cnt : std_logic_vector(1 downto 0);
begin

          u1: GSR port map (gsr=>rst);

          process(clk, rst)
          begin
                    if rst = '1' then
                              cnt <= "00";
                    elsif rising_edge (clk) then
                              cnt <= cnt + 1;
                    end if;
          end process;
          cntout <= cnt;
end behave;
```

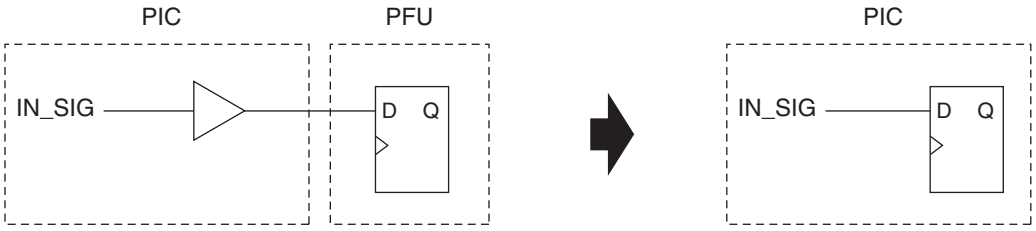
```
// Verilog Example of GSR Instantiation

module gsr_test(clk, rst, cntout);

input clk, rst;
output[1:0] cntout;

reg[1:0] cnt;

GSR u1 (.GSR(rst));

always @(posedge clk or negedge rst)
begin
          if (!rst)
                    cnt = 2'b0;
          else
                    cnt = cnt + 1;
end

assign cntout = cnt;

endmodule
```

## Use PIC Features

### Using I/O Registers/Latches in PIC

Moving registers or latches into Input/Output cells (PIC) may reduce the number of PFUs used and decrease routing congestion. In addition, it reduces setup time requirements for incoming data and clock-to-output delay for output data, as shown in Figure 15-11. Most synthesis tools will infer input registers or output registers in PIC if possible. Users can set synthesis attributes in the specific tools to turn off the auto-infer capability. Users can also instantiate library elements to control the implementation of PIC resource usage.

***Figure 15-11. Moving FF into PIC Input Register***




**Before Using Input Register** **After Using Input Register**

***Figure 15-12. Moving FF into PIC Output Register***


**Before Using Output Register** **After Using Output Register**

### Inferring Bi-directional I/O

Users can either structurally instantiate the bi-directional I/O library elements, or behaviorally describe the I/O paths to infer bi-directional buffers. The following VHDL and Verilog examples show how to infer bi-directional I/O buffers.

```
-- Inferring Bi-directional I/O in VHDL

library ieee;
use ieee.std_logic_1164.all;

entity bidir_infer is
   port(A, B   : inout std_logic;
        dir    : in std_logic);
end bidir_infer;

architecture lattice_fpga of bidir_infer is
begin
   B <= A when (dir='1') else 'Z';
   A <= B when (dir='0') else 'Z';
end lattice_fpga
```

```
// Inferring Bi-directional I/O in Verilog

module bidir_infer (A, B, DIR);
inout A, B;
input DIR;

assign B = (DIR)  ? A : 1'bz;
assign A = (~DIR) ? B : 1'bz;

endmodule
```

### Specifying I/O Types and Locations

Users can either assign I/O types and unique I/O locations in the Preference Editor or specify them as attributes in the VHDL or Verilog source code. The following examples show how to add attributes in the Synplify and Leonardo-Spectrum synthesis tool sets. For a complete list of supported attributes, refer to the HDL Attributes section of the ispLEVER on-line help system.

-- VHDL example of specifying I/O type and location attributes for Synplify & Leonardo

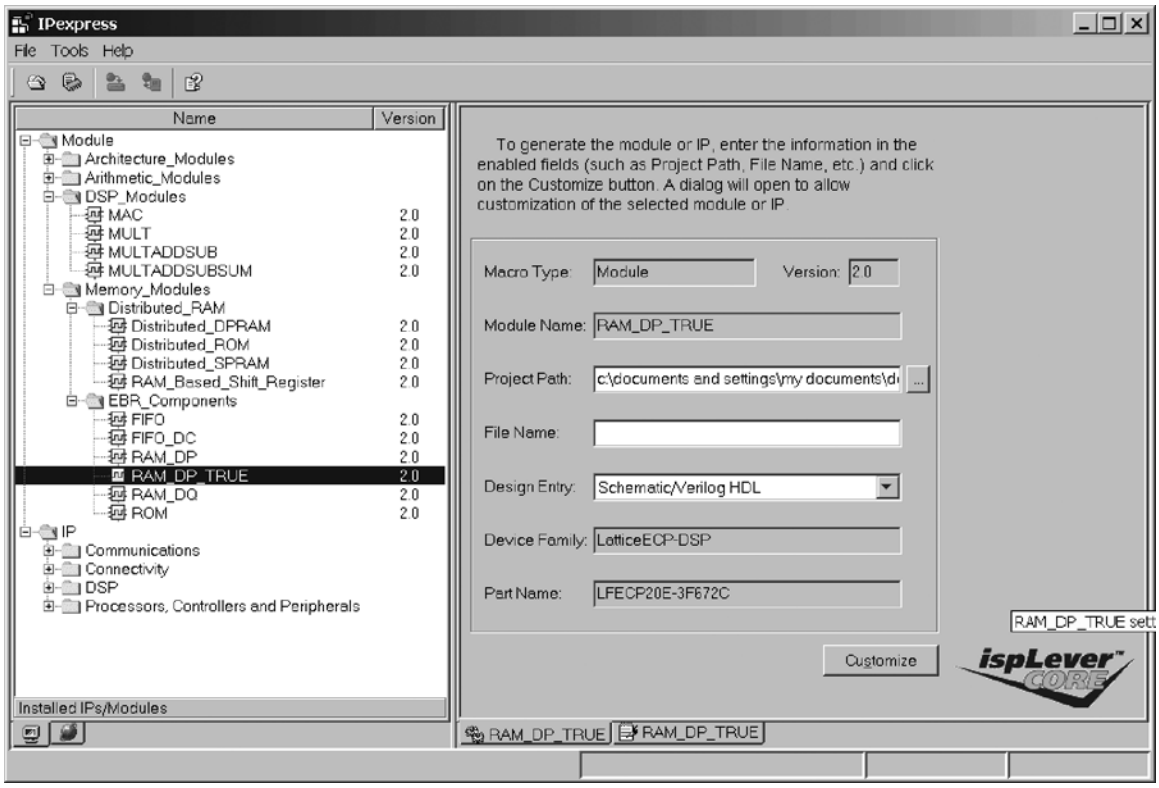
```
entity cnt is
     port(clk: in std_logic;
          res: out std_logic);
     attribute LEVELMODE: string:
     attribute LEVELMODE of clk : signal is “SSTL2”;
     attribute LOC of clk : signal is “V2”;
     attribute LEVELMODE of res : signal is “SSTL2”;
     attribute LOC of res : signal is “V3”;
end entity cnt;
```

-- Verilog example of specifying I/O type and location attributes for Synplify & Leonardo

```
module cnt(clk,res);

     input clk /* synthesis LEVELMODE=”SSTL2” LOC=”V2”*/;
     output res /* synthesis LEVELMODE=”SSTL2” LOC=”V3” */;

     ...

     // exemplar begin
          // exemplar attribute clk LEVELMODE SSTL2
          // exemplar attribute clk LOC V2
          // exemplar attribute res LEVELMODE SSTL2
          // exemplar attribute res LOC V3
     // exemplar end

endmodule
```

## Implementation of Memories

Although an RTL description of RAM is portable and the coding is straightforward, it is not recommended because the structure of RAM blocks in every architecture is unique. Synthesis tools are not optimized to handle RAM implementation and thus generate inefficient netlists for device fitting. For Lattice Semiconductor FPGA devices, RAM blocks should be generated through IPexpress as shown in the following screen shot.



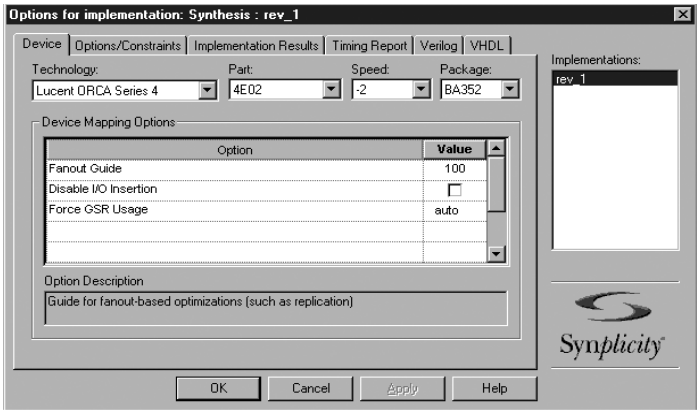

When implementing large memories in the design, it is recommended to construct the memory from the Enhanced Block RAM (EBR) components found in every Lattice Semiconductor FPGA device. When implementing small memories in the design, it is recommended to construct the memory from the resources in the PFU. The memory utilizing resources in the PFU can also be generated by IPexpress.

Lattice Semiconductor FPGAs support many different memory types including synchronous dual-port RAM, synchronous single-port RAM, synchronous FIFO and synchronous ROM. For more information on supported memory types per FPGA architecture, please consult the Lattice Semiconductor FPGA data sheets.

## Preventing Logic Replication and Limited Fanout

Lattice Semiconductor FPGA device architectures are designed to handle high signal fanouts. When users make use of clock resources, there will be no hindrance on fanout problems. However, synthesis tools tend to replicate logic to reduce fanout during logic synthesis. For example, if the code implies Clock Enable and is synthesized with speed constraints, the synthesis tool may replicate the Clock Enable logic. This kind of logic replication occupies more resources in the devices and makes performance checking more difficult. It is recommended to control the logic replication in synthesis process by using attributes for high fanout limit.

In the Synplicity® project GUI, under the Implementation Options => Devices tab, users can set the Fanout Guide value to 1000 instead of using the default value of 100. This will guide the tool to allow high fanout signals without replicating the logic. In the LeonardoSpectrum tool project GUI, under Technology => Advanced Settings, users can set the Max Fanout to be any number instead of the default value “0”.




## Use ispLEVER Project Navigator Results for Device Utilization and Performance

Many synthesis tools give usage reports at the end of a successful synthesis. These reports show the name and the number of library elements used in the design. The data in these reports do not represent the actual implementation of the design in the final Place and Route tool because the EDIF netlist will be further optimized during Mapping and Place and Route to achieve the best results. It is strongly recommended to use the MAP report and the PAR report in the ispLEVER Project Navigator tool to understand the actual resource utilization in the device. Although the synthesis report also provides a performance summary, the timing information is based on estimated logic delays only. The Place & Route TRACE Report in the ispLEVER Project Navigator gives accurate performance analysis of the design by including actual logic and routing delays in the paths.

# Technical Support Assistance

Hotline: 1-800-LATTICE (North America)
+1-503-268-8001 (Outside North America)
e-mail: techsupport@latticesemi.com
Internet: www.latticesemi.com

# Revision History

| Date | Version | Change Summary |
|---|---|---|
| — | — | Previous Lattice releases. |
| September 2012 | 02.2 | Updated document with new corporate logo. |