Lattice Synthesis Engine for ispLEVER Classic User Guide

May 2015

Copyright

Trademarks

Disclaimers

Type Conventions Used in This Document

Convention	Meaning or Use
Bold	Items in the user interface that you select or click. Text that you type into the user interface.
<*Italic*>	Variables in commands, code syntax, and path names.
Ctrl+L	Press the two keys at the same time.
`Courier`	Code examples. Messages, reports, and prompts from the software.
`...`	Omitted material in a line of code.
`.` `.` `.`	Omitted lines in code and report examples.
`[ ]`	Optional items in syntax descriptions. In bus specifications, the brackets are required.
`( )`	Grouped items in syntax descriptions.
`{ }`	Repeatable items in syntax descriptions.
`|`	A choice between items in syntax descriptions.

Contents

Lattice Synthesis Engine for ispLEVER Classic

Lattice Synthesis Engine (LSE) is the integrated synthesis tool that comes with ispLEVER Classic software.

This chapter describes:

- LSE tool options
- HDL coding tips
- Attributes and directives supported by LSE

LSE is a synthesis tool custom-built for Lattice products and fully integrated with ispLEVER Classic software. Depending on the design, LSE may lead to a more compact or faster placement of the design than another synthesis tool would do.

Also, LSE offers the following advantages:

- More granular control through the tool options
- Post-synthesis Verilog netlist suitable for simulation

Selecting LSE as Synthesis Tool

Use the Select Synthesis or Simulator dialog box to select the synthesis tool. Choose **Options > Select RTL Synthesis or Simulator**. In the Select Synthesis or Simulator dialog box, choose Lattice LSE, and click OK.

Changing the LSE Tool Processes and Properties

The LSE processes and properties for current source can be changed in the Properties dialog box. To open the Properties dialog box, in the Process pane of the ispLEVER Project Navigator, right-click and choose **Properties** from the pop-up menu. Refer to the Processes section of the Project Navigator online help for more information about process and property descriptions.

This section lists all the tool options associated with LSE. The following sections describe how to set the options to optimize synthesis for either area or speed and some of the differences between LSE and Synplify Pro options.

Command Line Options

This text property enables additional command line options for the associated process.

To enter a command line, type in the command line option and its value (if any) in the text box.

Frequency

This number property specifies the global design frequency (in MHz). The default value is **200**.

This property is equivalent to the **set_option -frequency** command in Synplify Pro.

FSM Encoding

Specifies the encoding style to use with the design.

This option is equivalent to the **-fsm_encoding_style option** in the SYNTHESIS command. Valid options are **Auto**, **One-Hot**, **Gray**, and **Binary.** The default value is **Auto**, meaning that the tool looks for the best implementation.

Intermediate File Dump

If you set this to **True**, LSE will produce intermediate encrypted Verilog files. If you supply Lattice with these files, they can be decrypted and analyzed for problems. This option is good for analyzing simulation issues.

Number of Critical Paths

This number property specifies the number of critical timing paths to be reported in the timing report. The default value is 3.

This property is equivalent to the **set_option -num_critical_paths** command in Synplify Pro.

Optimization Goal

Enables LSE to optimize the design for area, speed, or both.

Valid options are:

- **Area** (default) – Optimizes the design for area by reducing the total amount of logic used for design implementation.

When Optimization Goal is set to **Area**, LSE ignores the Target Frequency setting and uses 1 MHz instead.

Note

With the **Area** setting, LSE also ignores all SDC constraints. These constraints are not used by LSE and are not added to an .lpf file for use by the later stages of implementation.

- **Timing** – Optimizes the design for speed by reducing the levels of logic.
- **Balanced** – Optimizes the design for both area and timing.

Propagate Constants

When set to **True** (default), enables constant propagation to reduce area, where possible. LSE will then eliminate the logic used when constant inputs to logic cause their outputs to be constant.

You can turn off the operation by setting this option to **False**.

Remove Duplicate Registers

Specifies the removal of duplicate registers.

When set to **True** (default), LSE removes a register if it is identical to another register. If two registers generate the same logic, the second one will be deleted and the first one will be made to fan out to the second one's destinations. LSE will not remove duplicate registers if this option is set to **False**.

Resolve Mixed Drivers

If a net is driven by a VCC or GND and active drivers, setting this option to **True** connects the net to the VCC or GND driver.

Resource Sharing

When this true/false property is set to **True** (default), the synthesis tool uses resource sharing techniques to optimize area. With resource sharing, synthesis uses the same arithmetic operators for mutually exclusive statements; for example, with the branches of a case statement. Conversely, you can improve timing by disabling resource sharing, but at the expense of increased area.

This property is equivalent to the **set_option -resource_sharing 1 | 0** command in Synplify Pro.

Use IO Insertion

When set to **True,** LSE uses I/O insertion.

Verilog Include Search Path

A project property. LSE will use the specified search paths to search for the include files referenced in your design other than the directory of the file that specifies the include directive.

VHDL2008

When this is set to **True**, VHDL 2008 is selected as the VHDL standard for the project.

Optimizing LSE for Area and Speed

The following strategy settings for LSE can help reduce the amount of FPGA resources that your design requires or increase the speed with which it runs. Use these methods along with other, generic coding methods to optimize your design.

Minimizing area often produces larger delays, making it more difficult to meet timing requirements. Maximizing frequency often produces larger designs, making it more difficult to meet area requirements. Either goal, pushed to an extreme, may cause the place and route process to run longer or not complete routing.

To control the global performance of LSE, modify the tool options. In the Process pane of the ispLEVER Project Navigator, right-click and choose

Properties from the pop-up menu. See Table 1 for explanations and more details.

Table 1: LSE Tool Options for Area and Speed

Option	Area	Speed
FSM Encoding Style	Binary or Gray	One-Hot
Optimization Goal	Area	Timing
Remove Duplicate Registers	True	False
Resource Sharing	True	False
Frequency	<*minimum*>	

Frequency

A lower frequency target means LSE can focus more on area. A higher frequency target may force LSE to increase area. Try setting this value to about 10% higher than your minimum requirement. However, if Optimization Goal is set to Area, LSE will ignore the Target Frequency value, using a low 1 MHz target instead.

FSM Encoding Style

If your design includes large finite state machines, the Binary or Gray style may use fewer resources than One-Hot. Which one is best depends on the design. One-Hot is usually the fastest style. However, if the finite state machine is followed by a large output decoder, the Gray style may be faster.

Optimization Goal

If set to **Area**, LSE will choose smaller design forms over faster whenever possible. LSE will also ignore the Target Frequency option, using a low 1 MHz target instead. If set to **Timing**, LSE will choose faster design forms over smaller whenever possible. If you are having trouble meeting one requirement (area or speed) while optimizing for the other, try setting this option to **Balanced**.

Remove Duplicate Registers

Removing duplicate registers reduces area, but keeping duplicate registers may reduce delays.

Resource Sharing

If set to True, LSE will share arithmetic components such as adders, multipliers, and counters whenever possible.

If the critical path includes such resources, turning this option off may reduce delays. However, it may also increase delays elsewhere, possibly reducing the overall frequency.

LSE Options versus Synplify Pro

If you are moving from Synplify Pro to LSE, there are differences in the options to consider. Many of the Synplify Pro options have similar LSE options. But many also do not. See Table 2. There are numerous LSE options that have no Synplify Pro equivalents. For more information about the options, see “Changing the LSE Tool Processes and Properties” on page 2.

Table 2: Synplify Pro Tool Options and LSE Equivalents

Synplify Pro Options	LSE Equivalent	Synplify Pro Default	LSE Default
Allow Duplicate Modules	None	False	
Area	Optimization Goal	False	Balanced
Arrange VHDL Files	None	True	
Clock Conversion	None	True	
Command Line Options	Command Line Options		
Default Enum Encoding	FSM Encoding Style	Default	Auto
Disable IO Insertion	Use IO Insertion	False	True
Export Diamond Settings to Synplify Pro GUI	None	No	
Force GSR	None	False	
Frequency	Target Frequency		200
FSM Encoding	None	True	
Number of Critical Paths	Number of Critical Paths		3
Number of Start/End Points	None		
Output Netlist Format	None	None	
Output Preference File	None	True	
Pipelining and Retiming	None	Pipelining Only	
Push Tristates	None	True	
Resolved Mixed Drivers	Resolve Mixed Drivers	False	False
Resource Sharing	Resource Sharing	True	True
Update Compile Point Timing Data	None	False	
Use Clock Period for Unconstrained I/O	None	False	

Table 2: Synplify Pro Tool Options and LSE Equivalents (Continued)

Synplify Pro Options	LSE Equivalent	Synplify Pro Default	LSE Default
Verilog Input	None	Verilog 2001	
VHDL 2008	None	False	

LSE has additional options that provide more granular control than Synplify Pro, including:

- Carry Chain Length

Other LSE options without Synplify Pro equivalents:

- Intermediate File Dump
- Use Carry Chain
- Use IO Registers
- Propagate Constants
- Remove Duplicate Registers

Coding Tips for LSE

If you are going to use LSE to synthesize the design, the following coding tips may help. Mostly the tips are about writing code so that blocks of memory are “inferred”: that is, automatically implemented using logic cells or block RAM (BRAM) instead of registers. There are also tips about inferring types of I/O ports and about style differences with Synplify Pro.

LSE Differences with Synplify Pro

LSE tends to apply the Verilog and VHDL specifications strictly, sometimes more strictly than other synthesis tools including Synplify Pro. Following are some coding practices that can cause problems with LSE:

- Semicolons (;) to separate ports in a Verilog module statement. For example:

```
module COUNTER (
input CLK ,
input RESET ;   // LSE error on semicolon.
output TIMEOUT
);
```

- Spaces in the location path.
- Duplicate instantiation names (due to names in generate statements).
- Module instances without instance names.
- Multiple files with the same module names. Synplify Pro will error out but LSE will not. This could cause designs in LSE to use the incorrect module.
- Global VHDL signals.
- Modules that have a port mismatch between instance and definition.

- Both ieee.std_logic_signed and unsigned packages in VHDL. When preparing VHDL code for LSE, you can include either:

  USE ieee.std_logic_signed.ALL;

  or:

  USE ieee.std_logic_unsigned.ALL;

  Code with both signed and unsigned packages could fail to synthesize because operators would have multiple definitions.

- Mismatched variable types in VHDL. A std_logic_vector signal cannot be assigned to a std_logic signal and an unsigned type cannot be assigned to a std_logic_vector signal. For example:

```
din : in  unsigned (data_width - 1 downto 0);
dout : out std_logic_vector (data_width - 1 downto 0));
...
dout <= din; // Illegal, mismatched assignment.
```

  Such mismatched assignments generate errors that stop synthesis.

About Verilog Blocking Assignments

LSE support for Verilog blocking assignments to inferred RAM and ROM, such as “ram[(addr)] = data;,” is limited to a single such assignment. Multiple blocking assignments, such as you might use for dual-port RAM (see Example of RAM with Multiple Blocking Assignments (Wrong)), or a mix of blocking and non-blocking assignments are not supported. Instead, use non-blocking assignments (<=). See Figure 2.

Figure 1: Example of RAM with Multiple Blocking Assignments (Wrong)

```
always @(posedge clka)
begin
  if (write_ena)
    ram[addra] = dina; // Blocking assignment A
  douta = ram[addra];
end
always @(posedge clkb)
begin
  if (write_enb)
    ram[addrb] = dinb; // Blocking assignment B
  doutb = ram[addrb];
end
```

Figure 2: Example Rewritten with Non-blocking Assignments (Right)

```
 always @(posedge clka)
begin
  if (write_ena)
    ram[addra] <= dina;
  douta <= ram[addra];
end
always @(posedge clkb)
```

```
begin
  if (write_enb)
    ram[addrb] <= dinb;
  doutb <= ram[addrb];
end
```

Inferring I/O

To specify types of I/O ports, follow these models.

Verilog

Open Drain:

```
output <port>;
wire <output_enable>;
assign <port> = <output_enable> ? 1'b0 : 1'bz;
```

Bidirectional:

```
inout <port>;
wire <output_enable>;
wire <output_driver>;
wire <input_signal>;
assign <port> = <output_enable> ? <output_driver> : 1'bz;
assign <input_signal> = <port>;
```

VHDL

Tristate:

```
library ieee;
use ieee.std_logic_1164.all;
entity <tbuf> is
port (
  <enable> : std_logic;
  <input_sig> : in std_logic_vector (1 downto 0);
  <output_sig> : out std_logic_vector (1 downto 0));
end tbuf2;
architecture <port> of <tbuf> is
begin
  <output_sig> <= <input_sig> when <enable> = '1' else "ZZ";
end;
```

Open Drain:

```
library ieee;
use ieee.std_logic_1164.all;
entity <od> is
port (
  <enable> : std_logic;
  <output_sig> : out std_logic_vector (1 downto 0));
end od2;
architecture <port> of <od> is
begin
  <output_sig> <= "00" when <enable> = '1' else "ZZ";
end;
```

Bidirectional:

```
library ieee;
use ieee.std_logic_1164.all;
entity <bidir> is
port (
  <direction> : std_logic;
  <input_sig> : in std_logic_vector (1 downto 0);
  <output_sig> : out std_logic_vector (1 downto 0);
  <bidir_sig> : inout std_logic_vector (1 downto 0));
end bidir2;
architecture <port> of <bidir> is
begin
  <bidir_sig> <= <input_sig> when <direction> = '0' else "ZZ";
  <output_sig> <= <bidir_sig>;
end;
```

Event Inside an Event

Do not code an event within another event such as shown below:

Figure 3: Event within an Event (Wrong)

```
always begin :main
  guess = 0;
  @(posedge clk or posedge rst);
  if (rst) disable main;
  while(1) begin
    while(!result ) begin
      guess = 0;
      while(!result ) begin
        @(posedge clk or posedge rst);
        if (rst) disable main;
      end
      @(posedge clk or posedge rst);
      if (rst) disable main;
    end
    while(result) begin
      guess = 1;
      while(result) begin
        @(posedge clk or posedge rst);
        if (rst) disable main;
      end
      @(posedge clk or posedge rst);
      if (rst) disable main;
    end
  end
end
```

HDL Attributes and Directives

This section describes the Synplify Lattice attributes and directives that are supported by LSE. These attributes and directives are directly interpreted by the engine and influence the optimization or structure of the output netlist. Traditional HDL attributes, such as UGROUP, are also compatible with LSE and are passed into the netlist to direct place and route.

black_box_pad_pin

Directive. Specifies pins on a user-defined black-box component as I/O pads that are visible to the environment outside of the black box. If there is more than one port that is an I/O pad, list the ports inside double-quotes ("), separated by commas (,), and without enclosed spaces.

Verilog Syntax

```
object /* synthesis syn_black_box black_box_pad_pin =
"portList" */ ;
```

where portList is a spaceless, comma-separated list of the names of the ports on black boxes that are I/O pads.

Figure 4: Verilog Example

```
module BBDLHS(D,E,GIN,GOUT,PAD,Q)
/* synthesis syn_black_box black_box_pad_pin="GIN[2:0],Q" */;
```

VHDL Syntax

```
attribute black_box_pad_pin of object : objectType is
"portList" ;
```

where object is an architecture or component declaration of a black box. Data type is string; portList is a spaceless, comma-separated list of the black-box port names that are I/O pads.

Figure 5: VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;
package my_components is
component BBDLHS
   port (D: in std_logic;
         E: in std_logic;
         GIN : in std_logic_vector(2 downto 0);
         Q : out std_logic );
end component;

attribute syn_black_box : boolean;
attribute syn_black_box of BBDLHS : component is true;
attribute black_box_pad_pin : string;
attribute black_box_pad_pin of BBDLHS : component is
"GIN(2:0),Q";
end package my_components;
```

syn_black_box

Directive. Specifies that a module or component is a black box with only its interface defined for synthesis. The contents of a black box cannot be optimized during synthesis. A module can be a black box whether it is empty or not. This directive has an implicit Boolean value of 1 or true.

Verilog Syntax

```
object /* synthesis syn_black_box */ ;
```

where *object* is a module declaration.

Figure 6: Verilog Example

```
module bl_box(out,data,clk) /* synthesis syn_black_box */;
```

VHDL Syntax

```
attribute syn_black_box of object : objectType is true ;
```

where *object* is a component declaration, label of an instantiated component to define as a black box, architecture, or component. Data type is Boolean.

Figure 7: VHDL Example

```
architecture top of top-entity is
component ram4
   port (myclk : in bit;
         opcode : in bit_vector(2 downto 0);
         a, b : in bit_vector(7 downto 0);
         rambus : out bit_vector(7 downto 0) );
end component;
attribute syn_black_box : boolean;
attribute syn_black_box of ram4: component is true;
```

syn_encoding

Directive for VHDL designs. Defines how enumerated data types are implemented. The type of implementation affects the performance and device utilization.

VHDL Syntax

```
attribute syn_encoding of object : objectType is "value" ;
```

Where *object* is an enumerated type and value is one of the following: default, sequential, onehot, or gray.

Figure 8: VHDL Example

```
package testpkg is
type mytype is (red, yellow, blue, green, white,
    violet, indigo, orange);
attribute syn_encoding : string;
attribute syn_encoding of mytype : type is "sequential";
end package testpkg;
library IEEE;
```

```
use IEEE.std_logic_1164.all;
use work.testpkg.all;
entity decoder is
   port (sel : in std_logic_vector(2 downto 0);
   color : out mytype );
end decoder;
architecture rtl of decoder is
begin
   process(sel)
   begin
      case sel is
         when "000" => color <= red;
         when "001" => color <= yellow;
         when "010" => color <= blue;
         when "011" => color <= green;
         when "100" => color <= white;
         when "101" => color <= violet;
         when "110" => color <= indigo;
         when others => color <= orange;
      end case;
   end process;
end rtl;
```

syn_hier

Attribute. Allows you to control the amount of hierarchical transformation that occurs across boundaries on module or component instances during optimization.

syn_hier Values

The following value can be used for syn_hier:

hard – Preserves the interface of the design unit with no exceptions. This attribute affects only the specified design units.

```
object /* synthesis syn_hier = "value" */ ;
```

where *object* can be a module declaration and value can be any of the values described in syn_hier Values. Check the attribute values to determine where to attach the attribute.

Figure 9: Verilog Example

```
module top1 (Q, CLK, RST, LD, CE, D)
   /* synthesis syn_hier = "hard" */;
```

VHDL Syntax

```
attribute syn_hier of object : architecture is "value" ;
```

where *object* is an architecture name and value can be any of the values described in syn_hier Values. Check the attribute values to determine the level at which to attach the attribute.

Figure 10: VHDL Example

```
architecture struct of cpu is
attribute syn_hier : string;
attribute syn_hier of struct: architecture is "hard";
```

syn_keep

Directive. Keeps the specified net intact during optimization and synthesis.

Verilog Syntax

```
object /* synthesis syn_keep = 1 */ ;
```

where *object* is a wire or reg declaration. Make sure that there is a space between the object name and the beginning of the comment slash (/).

Figure 11: Verilog Example

```
module example2(out1, out2, clk, in1, in2);
output out1, out2;
input clk;
input in1, in2;
wire and_out;
wire keep1 /* synthesis syn_keep=1 */;
wire keep2 /* synthesis syn_keep=1 */;
reg out1, out2;
assign and_out=in1&in2;
assign keep1=and_out;
assign keep2=and_out;
always @(posedge clk)begin;
   out1<=keep1;
   out2<=keep2;
end
endmodule
```

VHDL Syntax

```
attribute syn_keep of object : objectType is true ;
```

where *object* is a single or multiple-bit signal.

Figure 12: VHDL Example

```
entity example2 is
   port (in1, in2 : in bit;
         clk : in bit;
         out1, out2 : out bit );
end example2;
architecture rt1 of example2 is
attribute syn_keep : boolean;
signal and_out, keep1, keep2: bit;
attribute syn_keep of keep1, keep2 : signal is true;
begin
and_out <= in1 and in2;
keep1 <= and_out;
keep2 <= and_out;
   process(clk)
   begin
```

```
      if (clk'event and clk = '1') then
         out1 <= keep1;
         out2 <= keep2;
      end if;
   end process;
end rt1;
```

syn_maxfan

Attribute. Overrides the default (global) fan-out guide for an individual input port, net, or register output.

Verilog Syntax

```
object /* synthesis syn_maxfan = "value" */ ;
```

Figure 13: Verilog Example

```
module test (registered_data_out, clock, data_in);
output [31:0] registered_data_out;
input clock;
input [31:0] data_in /* synthesis syn_maxfan=1000 */;
reg [31:0] registered_data_out /* synthesis syn_maxfan=1000 */;
```

VHDL Syntax

```
attribute syn_maxfan of object : objectType is "value" ;
```

Figure 14: VHDL Example

```
entity test is
   port (clock : in bit;
         data_in : in bit_vector(31 downto 0);
         registered_data_out: out bit_vector(31 downto 0) );
attribute syn_maxfan : integer;
attribute syn_maxfan of data_in : signal is 1000;
```

syn_noprune

Directive. Prevents instance optimization for black-box modules (including technology-specific primitives) with unused output ports.

Verilog Syntax

```
object /* synthesis syn_noprune = 1 */ ;
```

where object is a module declaration or an instance. The data type is Boolean.

Figure 15: Verilog Example

```
module top(a1,b1,c1,d1,y1,clk);
output y1;
input a1,b1,c1,d1;
input clk;
wire x2,y2;
reg y1;
```

```
syn_noprune u1(a1,b1,c1,d1,x2,y2) /* synthesis syn_noprune=1 */
;

always @(posedge clk)
   y1<= a1;

endmodule
```

VHDL Syntax

```
attribute syn_noprune of object : objectType is true ;
```

where the data type is boolean, and object is an architecture, a component, or a label of an instantiated component.

Figure 16: VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;
entity top is
   port (a1, b1 : in std_logic;
         c1,d1,clk : in std_logic;
         y1 :out std_logic );
end ;
architecture behave of top is
component noprune
port (a, b, c, d : in std_logic;
      x,y : out std_logic );
end component;
signal x2,y2 : std_logic;
attribute syn_noprune : boolean;
attribute syn_noprune of u1 : label is true;
begin
   u1: noprune port map(a1, b1, c1, d1, x2, y2);
   process begin
      wait until (clk = '1') and clk'event;
      y1 <= a1;
   end process;
end;
```

syn_preserve

Directive. Prevents sequential optimization such as constant propagation, inverter push-through, and FSM extraction.

Verilog Syntax

```
object /* synthesis syn_preserve = 1 */ ;
```

where object is a register definition signal or a module.

Figure 17: Verilog Example

```
module syn_preserve (out1,out2,clk,in1,in2)/* synthesis
syn_preserve=1 */;
output out1, out2;
input clk;
input in1, in2;
```

```
reg out1;
reg out2;
reg reg1;
reg reg2;
always@ (posedge clk)begin
reg1 <= in1 &in2;
reg2 <= in1&in2;
out1 <= !reg1;
out2 <= !reg1 & reg2;
end
endmodule
```

VHDL Syntax

```
attribute syn_preserve of object : objectType is true ;
```

where object is an output port or an internal signal that holds the value of a state register or architecture.

Figure 18: VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;
entity simpledff is
   port (q : out std_logic_vector(7 downto 0);
         d : in std_logic_vector(7 downto 0);
         clk : in std_logic );

-- Turn on flip-flop preservation for the q output
attribute syn_preserve : boolean;
attribute syn_preserve of q : signal is true;
end simpledff;
architecture behavior of simpledff is
begin
   process(clk)
   begin
      if rising_edge(clk) then
   -- Notice the continual assignment of "11111111" to q.
         q <= (others => '1');
      end if;
   end process;
end behavior;
```

syn_use_carry_chain

Attribute. Used to turn on or off the carry chain implementation for adders.

Verilog Syntax

```
object synthesis syn_use_carry_chain = {1 | 0} */ ;
```

Verilog Example

To use this attribute globally, apply it to the module.

```
module test (a, b, clk, rst, d) /* synthesis
syn_use_carry_chain = 1 */;
```

VHDL Syntax

```
attribute syn_use_carry_chain of object : objectType is true |
false ;
```

Figure 19: VHDL Example

```
architecture archtest of test is
signal temp : std_logic;
signal temp1 : std_logic;
signal temp2 : std_logic;
signal temp3 : std_logic;
attribute syn_use_carry_chain : boolean;
attribute syn_use_carry_chain of archtest : architecture is
true;
```