Lattice Synthesis Engine for Diamond User Guide

April, 2019

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 Diamond User Guide

Lattice Synthesis Engine (LSE) is the fully-integrated synthesis tool packaged with Lattice Diamond software, custom-built for many Lattice products. Depending on the design, LSE can create better resource utilization and faster timing than other synthesis tools. LSE complements the suite of tools available in the Lattice Diamond software and provides complete and comprehensive FPGA/CPLD synthesis solutions.

LSE offers the following advantages:

- It's the built-in Lattice synthesis tool optimized for use with Lattice devices.
- Provides granular control through tool options.
- Supports industry standard Verilog (Verilog 2001 and before) and VHDL (VHDL 2008 and before) including mixed language, along with industry standard attributes and SDC constraints. Enables the user to easily synthesize existing designs using LSE.

  Note

  LSE follows the IEEE standards listed below.

  - 1076-1987 – *IEEE Standard VHDL Language Reference Manual*
  - 1076-1993 – *IEEE Standard VHDL Language Reference Manual*
  - 1076-2008 – *IEEE Standard VHDL Language Reference Manual*
  - 1364-1995 – *IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware*
  - 1364-2001 – *IEEE Standard Verilog Hardware Description Language*
  - 1364-2005 – *IEEE Standard for Verilog Hardware Description Language*

  Only synthesis constructs described by these standards are supported.

- Provides GUI tool support for constraint entry (LDC Editor) and schematic netlist viewing and analysis (Netlist Analyzer), reducing the time required for design entry and analysis.

- Offers a choice of optimization goals: Area, Balanced, and Timing enabling the user to highlight which design goals should be emphasized.

This document describes the basic features of LSE. Sections include design flow overviews, preparing HDL source files, setting constraints, inferring block primitives, setting optimization options, analyzing the synthesis report, using Netlist Analyzer, analyzing timing, simulating, designing with modules, and designing with intellectual property (IP).

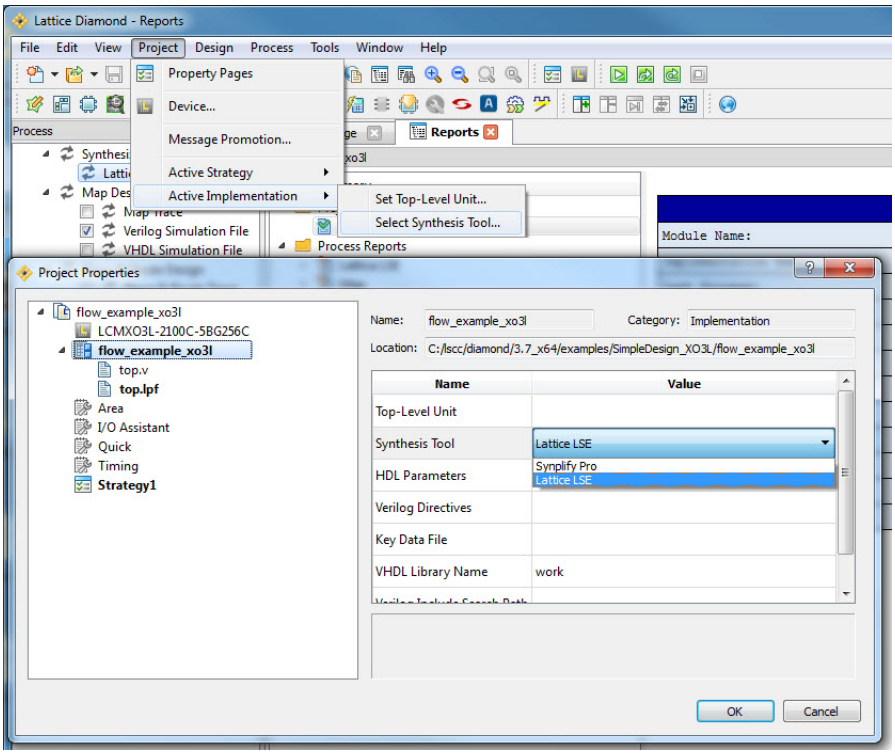
Design Flow Overview: User Interface

LSE is integrated into the Lattice Diamond software. To specify LSE as the synthesis tool:

1. Choose **Project > Active Implementation > Select Synthesis Tool**.

   The Project Properties dialog box opens with the active implementation selected, as shown in Figure 1.

Figure 1: Selecting LSE as Synthesis Tool in Lattice Diamond Software



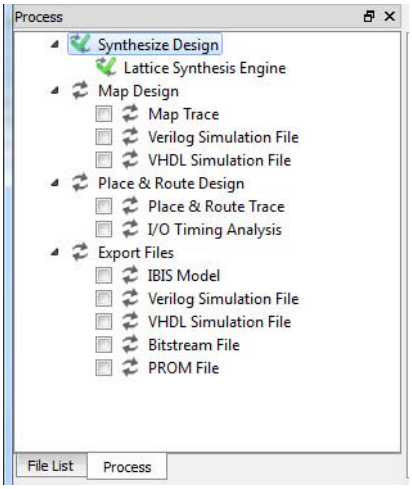
2. In the dialog box, double-click the Synthesis Tool row in the Value column.

   A menu drops down.

3. Choose **Lattice LSE**.
4. Click **OK**.
5. Start the Synthesize Design process. In the Process View (Figure 2), do one of the following:

- Select **Synthesize Design** in the Process view, and choose **Process > Run**.
- Right-click the **Synthesize Design** process and choose **Run**.
- Double-click the **Synthesize Design** process.

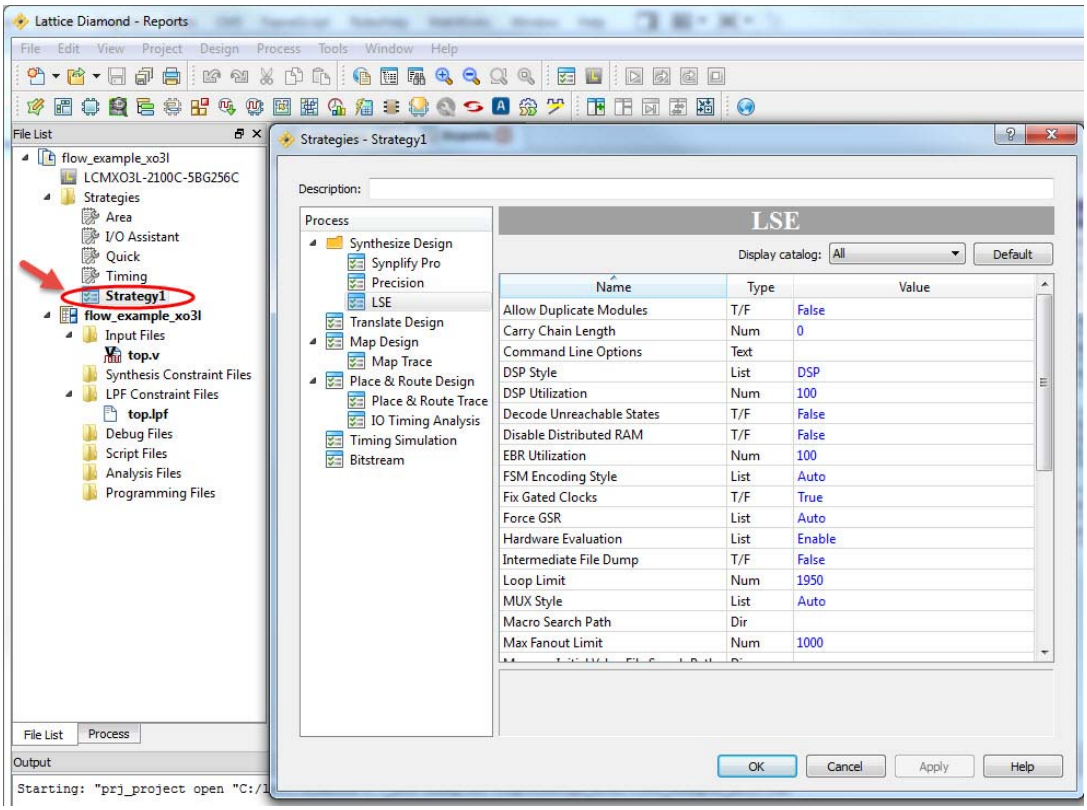
Figure 2: Diamond Software Process View



LSE Strategy Settings in Diamond

LSE strategies provide a unified view of all the options related to synthesis. LSE strategy options are listed in the LSE Strategy dialog box. Open the dialog box by double-clicking a strategy name in the File List view, as shown in Figure 3.

Figure 3: Opening the LSE Strategy Dialog Box



For information about an option, select it. A brief description appears at the bottom of the dialog box. Press F1 to open this guide and see the full description in the Diamond online help.

For more information on optimizing a strategy for area or timing, see "Optimizing LSE for Area and Speed" on page 105.

LSE Strategy Option Settings

Table 1 lists LSE strategy option settings available in the Diamond Strategy Setting dialog box.

Table 1: Strategy Option Settings

Name	Type	Value	Targeted Switch
Allow Duplicate Modules	True/False	False(Default)	-allow_duplicate_modules
Carry Chain Length	Integer	0 (not limited) (Default)	-carry_chain_length
Command line Options	Text	Text	(flags each preceded by '-')
DSP Style	List	DSP(Default)	-use_dsp
DSP Utilization	Num	100(Default)	-dsp_utilization

Table 1: Strategy Option Settings (Continued)

Name	Type	Value	Targeted Switch
Decode Unreachable States	True/False	False (Default)	-decode_unreachable_states
Disable Distributed RAM	True/False	False (Default)	
EBR Utilization	Float	100 (Default, in %)	-bram_utilization
FSM Encoding Style	String	Binary (Default) | One-Hot | Gray	-fsm_encoding_style
Fix Gated Clocks	True/False	True (Default)	-fix_gated_clocks
Force GSR	String	Yes | No | Auto (Default)	-force_gsr
Hardware Evaluation	List	Enable (Default) | Disable	-dt
Intermediate File Dump	True/False	False (Default)	-ifd
Loop Limit	Number	1950 (Default)	-loop_limit
Mux Style	String	Auto (Default) | PFU Mux | L6Mux Single | L6Mux Multiple	-mux_style
Macro Search Path	Text		-p
Max Fanout Limit	Integer	1000 (Default)	-max_fanout
Memory Initial Value File Search Path	Text		-p
Number of Critical Paths	Integer	3 (Default)	-twr_paths
Optimization Goal	String	Area | Timing | Balanced	-optimization_goal
Use LPF Created from SDC in Project	True/False	True (Default) | False	-lpf
Propagate Constants	True/False	True (Default)	-propagate_constants
Ram Style	String	Auto (Default) | Distributed | Block_ RAM | Registers	-ramstyle
Rom Style	String	Auto (Default) | ebr | Logic	-romstyle
Remove Duplicate Registers	True/False	True (Default)	- remove_duplicate_regs
Remove LOC properties	List	Off (Default) | On	-r
Resolve Mixed Drivers	True/False	False (Default)	-resolve_mixed_drivers
Resource Sharing	True/False	True (Default)	-resource_sharing
Target Frequency	Float	200 (Default, in MHz)	-frequency
Use Carry Chain	True/False	True (Default)	-use_carry_chain
Use IO Insertion	True/False	True (Default)	-use_io_insertion

Table 1: Strategy Option Settings (Continued)

Name	Type	Value	Targeted Switch
Use IO Registers	True/False	True (Default)	-use_io_reg
VHDL 2008	True/False	False (Default)	-vh2008

The following alphabetical list describes all of the strategy options associated with the LSE synthesis process.

Allow Duplicate Modules

When set to True, allows the design to keep duplicate modules. LSE issues a warning and uses the last definition of the module. Any previous definitions are ignored. The default is False, which causes an error if there are duplicate modules.

This option is equivalent to the “-allow_duplicate_modules” option in the SYNTHESIS command.

Carry Chain Length

Specifies the maximum number of carry chain cells (CCUs) that get mapped to a single carry chain. Default is 0, which is interpreted as infinite length.

This option is equivalent to the “-carry_chain_length” option in the SYNTHESIS command.

Command Line Options

Enables additional command line options for the LSE Synthesis process.

To enter a command line option:

1. In the Strategy dialog box, select **LSE** in the Process list.
2. Double-click the Value column for the Command line Options option.
3. Type in the option and its value (if any) in the text box.
4. Click **Apply**.

For detailed description on LSE command line options, see “Design Flow Overview: Command Line” on page 18.

Decode Unreachable States

When set to True, synthesis infers safe recovery logic from unreachable states in all the state machines of the design.

This option is equivalent to the “-decode_unreachable_states” option in the SYNTHESIS command.

Disable Distributed RAM

When set to True, inferred memory will not use the distributed RAM of the PFUs.

DSP Style

Specifies how DSP modules should be implemented: with DSP resources or with Logic (LUTs).

This option is equivalent to the “-use_dsp” option in the SYNTHESIS command.

DSP Utilization

Specifies the percentage of DSP sites that LSE should try to use.

This option is equivalent to the “-dsp_utilization” option in the SYNTHESIS command.

EBR Utilization

Specifies EBR utilization target setting in percent of total vacant sites. LSE will honor the setting and do the resource computation accordingly. Default is 100 (in percentage).

This option is equivalent to the “-bram_utilization” option in the SYNTHESIS command.

Fix Gated Clocks

When set to True, LSE changes standard gated clocks to forms more effective for FPGAs. Clocks are gated with AND or OR gates to conserve power, but in FPGAs such clocks cause skew and prevent global clock resources from being used. The Fix Gated Clocks option is ignored if the Optimization Goal option is set to Area. See “Optimization Goal” on page 13.

The gated clocks must be specified in the .ldc file with create_clock constraints. For more information about writing the constraints, see the following online help topics in the Diamond software:

- **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Clocks Using LDC Editor.**

- **Reference Guides > Constraints Reference Guide > Lattice Synthesis Engine (LSE) Constraints > Synopsys Design Constraints (SDC) > create_clock**.

All inputs of the gating logic must be driven by primary inputs and the gating logic must be decomposable. Instantiated primitives and black boxes are not affected. Converted clocks and the associated registers are reported in the synthesis.log file.

Force GSR

Enables (True) or disables (False) forced use of the global set/reset routing resources. When the value is Auto, the synthesis tool decides whether to use the global set/reset resources.

This option is equivalent to the “-force_gsr” option in the SYNTHESIS command.

FSM Encoding Style

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.

Note

The encoding type “gray” only works with less than or equal to four machine states. When the number of machine states is large than four, LSE will use other encoding styles and issue the following warning message:

WARNING - Gray encoding is not supported for state machines with more than four states.

Hardware Evaluation

Enables or disables the ability to temporarily test IP in a device without an IP license. If enabled, a timer is added to the design that allows unlicensed IP to function for about 4 hours in a device. If disabled, you cannot generate a bitstream if there are any unlicensed IP in the design.

You might want to disable this option to refine your design while waiting for the license. You will not be able to generate a bitstream, but you will be able to see how resources are used (without the timer) and close timing. When you get the license, you can then generate the bitstream.

Regardless of how this option is set, if there are any unlicensed IP in the design, some features of Diamond, such as gate level simulation and EPIC, are blocked.

This option is equivalent to the “-dt” option in the SYNTHESIS command.

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.

This option is equivalent to the “-ifd” option in the SYNTHESIS command.

Loop Limit

Specifies the maximum number of iterations of “for” and “while” loops in the source code. The limit is applied when the loop index is a variable, not when it is a constant. The higher the loop_limit, the longer the run time. The default value is 1950. Setting a higher value may cause stack overflow during some of the optimizations during synthesis. A lower value will be ignored and the default used instead.

This option is equivalent to the “-loop_limit” option in the SYNTHESIS command.

Macro Search Path

Allows you to specify a path (or paths) to locate physical macro files used in a given design. The software will add the specified paths to the list of directories to search when resolving file references. The option can also be used for indicating the directories containing include files that are specified in the RTL design files.

You don’t need to specify a search path if the necessary .ngo or .nmc file is in the directory containing the top-level .ngo file or if the FILE attribute in the design gives a complete path name for the file (instead of a relative path name).

The software follows the following order to search for .ngo files:

1. Current implementation directory
2. Project directory
3. Directories where the LPC or IPX source files reside
4. User-specified macro search paths

To specify a macro search path, double-click the Value box, and directly enter the path or click the **...** button to browse for one or more paths.

This option is equivalent to the “-p” option in the SYNTHESIS command.

Max Fanout Limit

Specifies the maximum fanout setting. LSE will make sure that any net in the design is not exceeding this limit. Default is 1000 fanouts. Does not apply to clock or reset network.

This option is equivalent to the “-max_fanout” option in the SYNTHESIS command.

Memory Initial Value File Search Path

Allows you to specify a path (or paths) to locate memory initialization file (.mem) used in a given design. The software will add the specified path(s) to the list of directories to search when resolving file references.

To specify a search path, double-click the Value box, and directly enter the path or click the **...** button to browse for one or more paths.

This option is equivalent to the “-p” option in the SYNTHESIS command.

MUX Style

Specifies the MUX style setting, which controls the way the macrogenerator implements the multiplexer macros.

Valid options are:

- Auto (default) - LSE looks for the best implementation for each considered macro.
- L6Mux Multiple - Generates multiplexers allowing for multiple L6Mux resources.
- L6Mux Single - Generates multiplexers allowing for the use of a single L6Mux resource.
- PFU Mux - Generates multiplexers using only PFUMux and LUT4 resources.

Note

L6Mux resources will only be inferred when driven by four LUT4 and two PFUMux devices.

This option is equivalent to the “-mux_style” option in the SYNTHESIS command.

Number of Critical Paths

Specifies the number of critical timing paths to be reported in the timing report.

This option is equivalent to the “-twr_paths” option in the SYNTHESIS command.

Optimization Goal

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

Valid options are:

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

  When Optimization Goal is set to Area, LSE honors the LDC constraints if there are any. If **Use IO Registers** is set to Auto, LSE packs input and output registers into I/O pad cells. See "Use IO Registers" on page 17..

  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.

  When Optimization Goal is set to Timing and a create_clock constraint is available in an .ldc file, LSE ignores the Target Frequency setting and uses the value from the create_clock constraint instead.

  If there are multiple clocks, and if not all the clocks use create_clock constraint, then LSE will assign 200 MHz constraint on the remaining clocks in Timing Mode.

  If Use IO Registers is set to Auto, LSE does not pack input and output registers into I/O pad cells.

- Balanced – Optimizes the design for both area and timing.

  When Optimization Goal is set to Balanced, all timing driven optimizations based on static timing analysis will run depending on LDC constraints. If Use IO Registers is set to Auto, LSE does not pack input and output registers into I/O pad cells.

The default setting depends on the device type. Smaller devices, such as MachXO and Platform Manager, default to Balanced. Larger devices—ECP5U, LatticeECP2, LatticeECP3, and LatticeXP2—default to Timing.

For more information, see "Optimizing LSE for Area and Speed" on page 568.

This option is equivalent to the "-optimization_goal" option in the SYNTHESIS command.

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.

This option is equivalent to the "-propagate_constants" option in the SYNTHESIS command.

RAM Style

Sets the type of random access memory globally to distributed, embedded block RAM, or registers.

The default is Auto which attempts to determine the best implementation, that is, the synthesis tool will map to technology RAM resources (EBR/Distributed) based on the resource availability.

This option will apply a syn_ramstyle attribute globally in the source to a module or to a RAM instance. To turn off RAM inference, set its value to Registers.

- Registers – Causes an inferred RAM to be mapped to registers (flip-flops and logic) rather than the technology-specific RAM resources.
- Distributed – Causes the RAM to be implemented using the distributed RAM or PFU resources.
- Block_RAM – Causes the RAM to be implemented using the dedicated RAM resources. If your RAM resources are limited, for whatever reason, you can map additional RAMs to registers instead of the dedicated or distributed RAM resources using this attribute.

This option is equivalent to the “-ramstyle” option in the SYNTHESIS command.

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.

This option is equivalent to the “-remove_duplicate_regs” option in the SYNTHESIS command.

Remove LOC Properties

Setting this to On removes LOC properties in the synthesized design before building the Native Generic Database (.ngd) file.

Resolved 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 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 option is equivalent to the “-resource_sharing” option in the SYNTHESIS command.

ROM Style

Allows you to globally implement ROM architectures using dedicated, distributed ROM, or a combination of the two (Auto).

This applies the syn_romstyle attribute globally to the design by adding the attribute to the module or entity. You can also specify this attribute on a single module or ROM instance.

Specifying a syn_romstyle attribute globally or on a module or ROM instance with a value of:

- Auto (default) – Allows the synthesis tool to choose the best implementation to meet the design requirements for speed, size, and so on.
- Logic – Causes the ROM to be implemented using the distributed ROM or PFU resources. Specifically, the logic value will implement ROM to logic (LUT4) or ROM technology primitives (such as ROM16X1, ROM32X1, ROM64X1, and so on).
- EBR – Causes the ROM to be mapped to dedicated EBR block resources. ROM address or data should be registered to map it to an EBR block. If your ROM resources are limited, for whatever reason, you can map additional ROM to registers instead of the dedicated or distributed RAM resources using this attribute.

Infer ROM architectures using a CASE statement in your code. For the synthesis tool to implement a ROM, at least half of the available addresses in the CASE statement must be assigned a value. For example, consider a ROM with six address bits (64 unique addresses). The CASE statement for this ROM must specify values for at least 32 of the available addresses.

This option is equivalent to the “-romstyle” option in the SYNTHESIS command.

Target Frequency

Specifies the target frequency setting. This frequency applies to all the clocks in the design. If there are some clocks defined in an .ldc file, the remaining clocks will get this frequency setting. When a create_clock constraint is available in an .ldc file, LSE ignores the Target Frequency setting for that clock and uses the value from the create_clock constraint instead.

This option is equivalent to the “-frequency” option in the SYNTHESIS command.

Use Carry Chain

Turns on (True) or off (False) carry chain implementation for adders. Default is True.

This option is equivalent to the “-use_carry_chain” option in the SYNTHESIS command.

Use IO Insertion

When set to True, LSE uses IO insertion and GSR.

When set to False, LSE will generate an NGO netlist and an NGD file is not created.

This option is equivalent to the “-use_io_insertion” option in the SYNTHESIS command.

See “Creating Your Own Black Box Modules” on page 118 for more information.

Use IO Registers

When True, this option forces the synthesis tool to pack all input and output registers into I/O pad cells based on the timing requirements for the target device family. Auto, the default setting, enables this register packing if Optimization Goal is set to Area. If Optimization Goal is Timing or Balanced, Auto disables register packing.

This option is equivalent to the “-use_io_reg” option in the SYNTHESIS command.

You can also control packing on individual registers and ports. See “syn_useioff” on page 77.

Use LPF Created from SDC in Project

LSE creates a preference (.lpf) file based on the Synopsys Design Constraint (.sdc) file. (When you use LSE, SDC constraints must be in a Lattice Design Constraints (.ldc) file.) When this option is set to True, the synthesis constraints are also applied to the Map Design stage of implementation.

VHDL 2008

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

Design Flow Overview: Command Line

LSE can be run from the command line. Table 2 describes the command project options available to run LSE. Examples are provided following the table.

The command is SYNTHESIS. The '-f' option is available to simplify the command line. A project file containing all the user's desired arguments can be constructed as a text file, then passed to LSE using the –f switch. For example:

synthesis -f synth.synproj

For more information about setting up the Diamond command line environment, in the Diamond online help, refer to **Reference Manuals > Command Line Reference Guide > Command Line Basics > Setting Up the Environment to Run Command Line**.

Note

Running LSE from the command line requires an environment variable TEMP be set. Default Cygwin .bashrc unsets this variable, so user must add the following back into their .bashrc:

export TEMP=/temp

Table 2: LSE Project Options

Required Options	Parameter	Arguments	Description	Default
Optional	-s	<grade>	Target grade	
Optional	-t	<package>	Target Package	
Optional	-loop_limit	<value>	Iteration limits	
Optional	-f	<argument_filename>	Argument file	
Required	-a	"<Supported_Device_Family>"	Target Architecture	-
Optional	-d	<device>	Target Device	-
Optional	-p	<searchpath>}	Option search path.	\.
Optional	-top	<top_module_name>	Top-level module name	-

Table 2: LSE Project Options (Continued)

Required Options	Parameter	Arguments	Description	Default
Required (Only for source of this type) (Multiple arguments)	-ver	{<verilog_file.v>}	Name of input Verilog file	-
Optional (Multiple arguments)	-lib	<libname>	Include library	-
Required (Only for source of this type) (Multiple arguments)	-vhd	{<vhdl_file.vhd>}	Name of input VHDL file	-
Optional	-ngd	<ngd_file.ngd>	Name of output ngd file, option available only for Diamond devices	<top_module_name>.ngd
Optional	-ngo	{<ngo_file.ngo>}	Name of output ngo file, option available only for Diamond devices	<top_module_name>.ngo
Optional	-force_gsr	{auto | yes | no}	GSR insertion, option available only for Diamond devices	auto
Optional	-ramstyle	{ auto | distributed | block_ram | registers }	RAM style	auto
Optional	-romstyle	{ auto | ebr | logic }	ROM style	auto
Optional	-output_edif	{<filename.edf>}	Create EDIF output file	<top_module_name>.edf
Optional	-output_hdl	{<filename>}	Create HDL output file	<top_module_name>_prim.v
Optional	-sdc	{<sdc_file.sdc>}	Input SDC file	-
Optional	-lpf	{true | false}	Generate output lpf file with name: <top_module_name>_lse.lpf, option available only for Diamond devices	False
Optional	-logfile	{<synthesis_logfile>}	Name of output log file	synthesis.log

Table 2: LSE Project Options (Continued)

Required Options	Parameter	Arguments	Description	Default
Optional	-frequency	{target_frequency }	Target frequency for timing optimization, in MHz	200
Optional	-max_fanout	{max_fanout }	Maximum driver fanout	1000
Optional	-bram_utilization	{bram_utilization}	Block RAM utilization factor, percent	100
Optional	-fsm_encoding_style	{binary | one-hot | gray}	Finite State Machine encoding style	binary
Optional	-mux_style	{ auto | pfu_mux | L6Mux_single | L6Mux_multiple }	Mux implementation style	auto
Optional	-use_carry_chain	{0|1}	Use carry-chain resources	1 (use)
Optional	-carry_chain_length	{ chain_length }	Carry chain maximum length	0 (no limit)
Optional	-use_io_insertion	{ 0|1 }	Insert I/O primitives	1 (insert)
Optional	-use_io_reg	{ 0|1 }	Use I/O registers	1 (use)
Optional	-resource_sharing	{ 0|1 }	Allow resource sharing optimization	1 (allow)
Optional	-propagate_constants	{ 0|1 }	Preserve registers with constant inputs	1 (allow)
Optional	-remove_duplicate_regs	{ 0|1 }	Allow removal of duplicate registers	1 (allow)
Optional	-ip_dir	{location of IP installation}	Location of IP directory	\.
Optional	-corename	{name of IP core}	Name of IP core	-
Optional	-ertl_file	{name of encrypted RTL file}	Name of encrypted file	-
Optional	-optimization_goal	{ area | timing | balanced }	Global optimization strategy	area
Optional	-hdl_param	{<name value>}	To pass parameters/ generics to design top module	-
Optional	-h	-	Help	-
Optional	-twr_paths	{Num_paths}	Number of TRACE critical paths	3
Optional	-ifd	-	Dump intermediate files	-
Optional	-dt	-	Disable h/w timer	-
Optional	-cbn	-	Consistent Bus Naming	-

Table 2: LSE Project Options (Continued)

Required Options	Parameter	Arguments	Description	Default
Optional	-fix_gated_clocks	{0|1}	Fix Gated Clocks	1
Optional	-vh2008	-	Compile using VHDL 2008 libraries	
Optional	-key	(key dat)	Location of the key file used for decryption	=

Examples Following are a few examples of SYNTHESIS command lines and a description of what each does. Command lines only supports MachXO, MachXO2, MachXO3L, and MachXO3LF devices.

Example 1 The following command is a simple example with Verilog and VHDL file inputs.

```
synthesis -a MachXO2 -d LCMXO2-2000HC -t TQFP144 -s 5 -top
top_module_name -vhd f1.vhd f2.vhd f3.vhd -ver file1.v file2.v
-ngd file.ngd
```

Example 2 The following example illustrates the usage of a search path -p option for IP .ngo files or include files.

```
synthesis -a MachXO2 -d LCMXO2-2000HC -p D:/my_project/tmp -top
top_module_name -vhd f1.vhd f2.vhd f3.vhd -ver file1.v file2.v
-ngd file.ngd
```

Example 3 The following example shows VHDL library usage with the -lib option.

```
synthesis -a MachXO2 -d LCMXO2-2000HC -top top -lib work
-vhd top.vhd -lib my_lib -vhd ff.vhd -ngd file.ngd
```

Example 4 The following example illustrates the usage of both the -hdl_param and -optimization_goal options.

```
synthesis -a MachXO2 -d LCMXO2-2000HC -hdl_param width 7 depth
5 -optimization_goal timing -ver file1.v file2.v -ngd file.ngd
```

Example 5 This is an example of a command line with encrypted RTL for IP designs.

```
synthesis -a MachXO2 -d LCMXO2-2000HC -corename file_datapath -
ertl_file source/file_datapath_enc.vhd -ip_dir encryption -ngd
file.ngd
```

Example 6 This example shows miscellaneous commands for illustrating various syntax structures.

```
synthesis -vhd source/ora.vhd source/top.vhdl source/
anda_vhd.vhd
synthesis -ver source/anda.v source/v_top.v
synthesis -a MachXO2 -d LCMXO2-2000HC -force_gsr auto -ver
top.v mid.v prim.v -vhd count.vhd
```

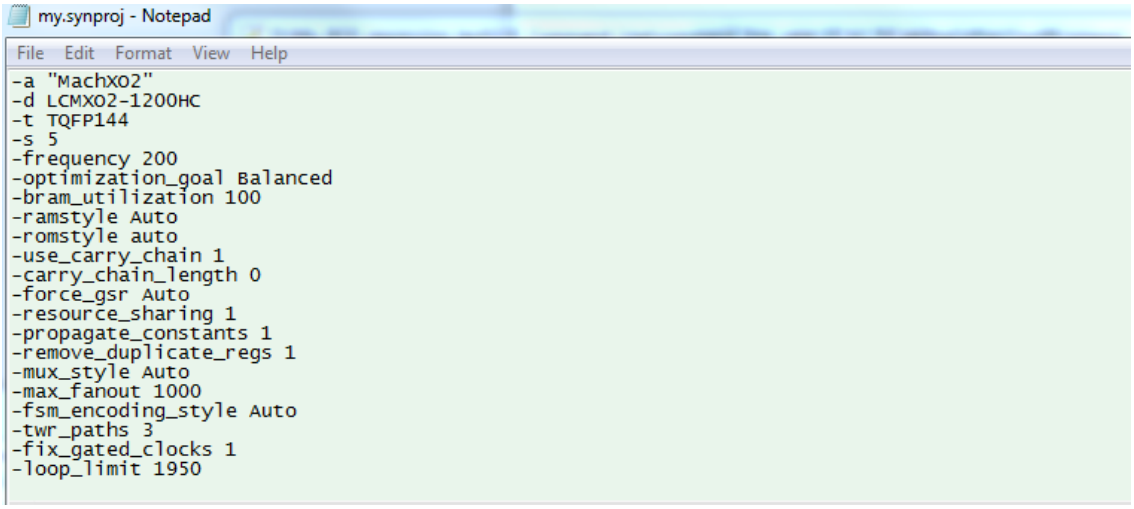
```
synthesis -lib ..\include_lib.v -ngd top.ngd -lpf top.lpf -sdc
top.ldc
```

Example 7 The following example illustrates the usage of a synthesis project file with command -f . The synthesis project file contains strategy setting, Diamond translates strategy options into command line options.

```
synthesis -f D:/my_project/my.synproj
```

Figure 4 shows the contents of the synthesis project synproj.

Figure 4: Example synproj Project



For more information about running Synthesis from the command line, in the Diamond software online help, refer to **Reference Guides > Command Line Reference Guide > Command Line Tool Usage > Running SYNTHESIS from the Command Line**.

Preparing the Input

You can create an HDL source file in Diamond Source Editor.

To create an HDL source in Source Editor:

1. From the Diamond main window, choose **File > New > File**. In the New File dialog, choose **Verilog Files** or **VHDL Files** from the Source Files list.
2. In the New File dialog, fill in the File name and Location, choose the file extension in the Ext. field.
3. Check **Add to Implementation** option if you want to add this source to the current project.
4. Click **New**.

5. In the pop up Source Editor, you can enter the text. When finished editing, click **File > Save** from the Diamond main window.

Note

You can detach Source Editor from the Diamond main window by clicking the Detach Tool icon on the upper right corner of Source Editor. If you want to attach Source Editor back to the main window, click the Attach Window icon on the upper right corner of Source Editor window, or choose Window > Attach Window from Source Editor.

For more information, in the Diamond software online help, refer to **User Guides > Entering the Design > HDL Design Entry**.

Constraint Files

LSE enables you to set Synopsys® Design Constraints (SDC), which are directly interpreted by the synthesis engine. When you use LSE, these SDC constraints are saved to a Lattice Design Constraints file (.ldc). You can create several .ldc files and select one of them to serve as the active synthesis constraint file for an implementation. You can also cause a synthesis preference file to be generated when the design is synthesized. The synthesis preferences can then be merged with the logical preference file (.lpf).

Lattice Design Constraints (LDC) Editor, as well as Source Editor, are available for creating and editing .ldc files. LDC Editor provides a spreadsheet style user interface that enables you to quickly create and edit Synopsys Design Constraints.

For more information, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints**.

Specifying Constraints and Attributes

Constraints on specific design elements are provided to LSE using the SDC constraint language. The constraints reside in the .ldc file, which can be accessed in the Diamond Synthesis Constraint Files folder in the File List pane.

In Diamond, a single constraint file can be active at any one time. The active file is used by synthesis when it is run. There can be multiple .ldc files, but only one can be active.

The .ldc file can be opened in the LDC Editor (GUI), or the text editor.

- To open the LDC Editor, double-click the .ldc file in the File List pane.
- Right-click the .ldc file in the File List pane and in the dropdown choose **Open With.** The Open With dialog box allows you to choose between the LDC Editor or the Text Editor.

For more information, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Synthesis Constraints Using LDC Editor**.

The *Timing Closure* section of the *FPGA Design Guide* focuses on timing requirements, explains timing driven FPGA implementation processes, and shows how to tackle timing issues when timing closure becomes problematic. You can access *Timing Closure* section of the *FPGA Design Guide* from the Diamond software Start Page. Or, from the Diamond software online help, refer to **User Guides > Help for Lattice Diamond**, and scroll down to *FPGA Design Guide*.

Defining Synthesis Constraints Using LDC Editor

The LDC Editor provides a GUI that enables the user to easily choose the type of constraint, and provide all the necessary constraint information (the GUI must make it clear what information is required and what is optional). Key is to provide the user access to actual design element names that will be honored by LSE (e.g. instance, port names). Moreover, it will filter the selection provided to the user based on the legal type. As an example, in a set_input_delay constraint, the user will only be given a selection of references for the clock that are considered clocks.

LDC Editor does not perform design rule check (DRC) by default. DRC check can be enabled in the Diamond software by clicking **Tools > Options > LDC Editor > General > Run DRC check before saving** and **Enable realtime DRC check**.

When the target of the constraint can not be found, or if the constraint syntax is incorrect, a warning will display and the constraint will not show in the LDC Editor.

For more information, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Synthesis Constraints Using LDC Editor**.

Defining Synthesis Constraints Using Text Editor

The .ldc file can be edited by Text Editor manually. Do not use TCL features for constraints which are not supported by LSE now. For example:

```
set board_delay 3
set_clock_groups -exclusive \
                 -group {c0_1} \
                 -group {c0_0}
```

See the following LSE constraints for syntax examples for each constraint.

Defining Clocks

The create_clock constraint creates a clock and defines its characteristics.

Note

In LSE timing, interclock domain paths are always blocked for create_clock. However, the interclock domain path is still valid for constraints such as set_false_path and set_multicycle_path.

Syntax create_clock [-name *name*] -period *period_value* [-waveform {value1 value2}] *source_object*

Arguments -name *name*

The name string specifies the name of the clock. If this parameter is not given, the name of the source object is used as the name of the clock. Virtual clocks are currently not supported.

-period *period_value*

This value is required and it specifies the clock period in nanoseconds. The value you specify is the minimum time over which the clock waveform repeats. The value specified for the period must be positive as the period of a clock must be greater than zero. The duty cycle of the clock is 50 percent.

-waveform {value1 value2}

The values are a list of edge values. Only two edges are supported. Floating values are accepted. Value1 must be less than value2, and the difference must be less than the clock period.

source_object

The source object is the object on which the clock constraint is defined. The source object can be a port object or a net object in the design. The object is obtained by using one of the get_ports or get_nets commands. If you specify a clock constraint on a source object that already has a clock, the new clock replaces the existing one. Only one source object is accepted. Wildcards are accepted as long as the resolution shows one port or net object.

Example The following example creates two clocks on ports CK1 and CK2 with a period of 6:

```
create_clock -name my_user_clock -period 6 [get_ports CK1]
create_clock -name my_other_user_clock -period 6 [get_ports CK2]
```

Example The following example creates a clock on port CK3 with a period of 7.1, and has two edges at 0 and 4.1:

```
create_clock -period 7.1 -waveform {0 4.1} [get_ports CK3]
```

For more information about defining clocks with LDC Editor, For more information, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Clocks Using LDC Editor**.

Defining Generated Clocks

The create_generated_clock creates an internally generated clock and defines its characteristics. This command is used when the clock being created is related to another clock. The generated clock will now be considered a clock when defining constraints such as set_input_delay.

Syntax create_generated_clock -source *reference_object* [-*master_clock clock_object*] [-divide_by *factor*] [-multiply_by *factor*] [-duty_cycle *value*] net_object

Arguments -source *reference_object*

The reference object is an object on which the source clock of the generated clock is defined. The source object can be a net object or a port object. The period of the generated clock is derived from the clock on the reference object using the multiply and divide factors.

-master_clock *clock_object*

If the master is defined, the master clock object becomes the source clock for the generated clock. This is an optional object used to identify a specific clock, if there is more than one clock on the source object.

-divide_by *factor*

This factor is the frequency division factor. The frequency of the generated clock is equal to the frequency of the source clock divided by this factor, if the multiply by factor is not specified. For instance, if this factor is equal to 2, the generated clock period is twice the reference clock period. Default value is 1.

-multiply_by *factor*

This factor specifies the frequency multiplication number to be used when finding the generated clock frequency. For instance, if the factor is equal to 2, the generated clock period is half the reference clock period. If both multiply_by and divide_by factors are used, the frequency is obtained by using both factors. Default value is 1.

-duty_cycle *value*

This value specifies the duty cycle in percentage of the clock period. The value can be floating point and ranges from 0 to 100. The default value is 50.

net_object

The net_object specifies the source of the clock constraint. This is usually an internal -net of the design. If you specify a clock constraint on a net that already has a clock, the new clock replaces the existing clock. Only one source is accepted. Wildcards are accepted as long as the resolution shows one net.

This command creates a generated clock in the current design at a declared net_object by defining its frequency with respect to the frequency at the reference object. The static timing analysis tool uses this information to compute and propagate the generated clock's waveform across the clock network to the clock pins of all sequential elements driven by this target

Examples The following example creates a generated clock on pin pll1/CLKOP with a period twice as long as the period at the reference port CLK:

```
create_generated_clock -divide_by 2 -source [get_ports CLK]
[get_pins pll1/CLKOP]
```

The following example creates a generated clock at the primary output of myPLL with a period 3/4 of the period at the reference pin clk:

```
create_generated_clock -divide_by 3 -multiply_by 4  -source
[get_ports clk]  [get_pins myPLL/CLK1]
```

The following example shows a clock with a duty cycle of 60 percent:

```
create_generated_clock -duty_cycle 60 -source [get_ports clk]
[get_pins myPLL/CLK1]
```

For more information about defining clocks with LDC Editor, in the Diamond software online help, refer to **User Guides > Applying Design Constraints >**

Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Generated Clocks in LDC Editor.

Defining Clock Groups

The clock_groups constraint specifies clock groups that are mutually exclusive or asynchronous with each other in a design so that the paths between these clocks are not considered during timing analysis.

Syntax set_clock_groups *-asynchronous* | *-exclusive* -group *clock_objects* [-group *clock_objects*]

Arguments -asynchronous

Specifies that the clock groups are asynchronous to each other (meanwhile, Lattice assume all clocks are asynchronous). Two clocks are asynchronous with respect to each other if they have no phase relationship at all.

-exclusive

Specifies that clocks are mutually exclusive. Only one clock group will be active at any given time.

-group *clock_object*

Specifies the clock objects in a group. If you specify only one group, it means that the clocks in that group are exclusive or asynchronous with all other clocks in the design. A default other group is created for this single group. Whenever a new clock is created, it is automatically included in this group.

Examples The following example specifies two clock ports (clka and clkb) are asynchronous to each other.

```
create_clock -period 10.000 -name clka_port [get_ports clka]
create_clock -period 10.000 -name clkb_port [get_ports clkb]
# Set clka_port and clkb_port to be mutually exclusive clocks.
set_clock_groups -asynchronous -group [get_clocks clka_port] -
group [get_clocks clkb_port]
# The previous line is equivalent to the following two
commands.
set_false_path -from [get_clocks clka_port] -to [get_clocks
clkb_port]
set_false_path -from [get_clocks clkb_port] -to [get_clocks
clka_port]
```

The following example specifies four clock constraints that won't be active at the same time:

For more information about defining clocks with LDC Editor, in the Diamond software online help, refer to **User Guides > Applying Design Constraints >**

```
create_clock -period 10.000 -name clka_port [get_ports clka]
create_clock -period 10.000 -name clkb_port [get_ports clkb]
create_clock -period 10.000 -name clkc_port [get_ports clkc]
set_clock_groups -exclusive -group [get_clocks {clka_port
clkb_port}] -group [get_clocks clkc_port]
```

Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Clock Groups in LDC Editor.

Setting Input Delays

The set_input_delay constraint defines the arrival time of an input relative to a clock.

Syntax set_input_delay *delay_value* [-max |-min] -clock *clock_object input_port_object*

Arguments *delay_value*

Specifies the arrival time in nanoseconds that represents the amount of time for which the signal is available at the specified input after a clock edge.

-max

Specifies that the delay value is the maximum delay.

-min

Specifies that the delay value is the minimum delay.

-clock *clock_object*

Specifies the clock reference to which the specified input delay is related. This is a mandatory argument.

input_port_object

Provides one or more input ports in the current design to which delay_value is assigned. You can also use the keyword “all_inputs” to include all input ports.

Example The following example sets an input delay of 1.2 ns for port data1 relative to the rising edge of CLK1:

```
set_input_delay 1.2 -clock [get_clocks CLK1] [get_ports data1]
```

Example The following example sets an input delay of 1.2 ns minimum and 1.5 ns maximum for port data1 relative to the rising edge of CLK1:

```
set_input_delay 1.2 -min -clock [get_clocks CLK1] [get_ports
data1]
set_input_delay 1.5 -max -clock [get_clocks CLK1] [get_ports
data1]
```

For more information about setting input and output delays with LDC Editor, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Setting Input and Output Delays Using LDC Editor**.

Setting Output Delays

The set_output_delay constraint defines the output delay of an output relative to a clock.

Syntax set_output_delay *delay_value* [-max |-min] -clock *clock_object output_port_object*

Arguments *delay_value*

Specifies the amount of time from a reference clock to a primary output port.

-max

Specifies that the delay value is the maximum delay.

-min

Specifies that the delay value is the minimum delay.

-clock *clock_object*

Specifies the clock reference to which the specified output delay is related. This is a mandatory argument.

output_port_object

Provides one or more (by wildcard) output ports in the current design to which delay_value is assigned. You can also use the keyword “all_outputs” to include all output ports.

Example The following example sets an output delay of 1.2 ns for all outputs relative to clki_c:

```
set_output_delay 1.2 -clock [get_clocks CLK1] [get_ports OUT1]
set_output_delay 1.2 -clock [get_clocks CLK1] [all_outputs]
```

Example The following example sets an output delay of 1.2 ns minimum and 1.5 ns maximum for port data1 relative to the rising edge of CLK1:

```
set_output_delay 1.2 -min -clock [get_clocks CLK1] [get_ports
data1]
set_output delay 1.5 -max -clock [get_clocks CLK1] [get_ports
data1]
```

For more information about setting input and output delays with LDC Editor, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Setting Input and Output Delays Using LDC Editor**.

Defining Minimum Delay Paths

The set_min_delay constraint specifies the maximum delay for the timing paths.

Syntax set_max_delay *delay_value* [-from *from port_object or cell_object*] [-to *to port_object or cell_object*]

Arguments *delay_value*

Specifies a floating point number in nanoseconds that represents the required maximum delay value for specified paths.

If the path ending point is on a sequential device, the tool includes library setup time in the computed delay.

-from *from port_object or cell_object*

Specifies the timing path start point. A valid timing start point is a clock, a primary input, a combinational logic cell, or a sequential cell (clock pin).

-to *to port_object or cell_object*

Specifies the timing path end point. A valid timing end point is a primary output, a combinational logic cell, or a sequential cell (data pin)

Examples The following example sets a maximum delay by constraining all paths from ff1a:CLK to ff2e:D with a delay less than or equal to 5 ns:

```
set_max_delay 5 -from [get_cells ff1a] -to [get_cells ff2e]
```

For more information about defining delay paths with LDC Editor, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Delay Paths Using LDC Editor**.

Defining Maximum Delay Paths

The set_max_delay constraint specifies the maximum delay for the timing paths.

Syntax set_max_delay *delay_value* [-from *from port_object or cell_object*] [-to *to port_object or cell_object*]

Arguments *delay_value*

Specifies a floating point number in nanoseconds that represents the required maximum delay value for specified paths.

If the path ending point is on a sequential device, the tool includes library setup time in the computed delay.

-from *from port_object or cell_object*

Specifies the timing path start point. A valid timing start point is a clock, a primary input, a combinational logic cell, or a sequential cell (clock pin).

-to *to port_object or cell_object*

Specifies the timing path end point. A valid timing end point is a primary output, a combinational logic cell, or a sequential cell (data pin)

Examples The following example sets a maximum delay by constraining all paths from ff1a:CLK to ff2e:D with a delay less than or equal to 5 ns:

```
set_max_delay 5 -from [get_cells ff1a] -to [get_cells ff2e]
```

For more information about defining delay paths with LDC Editor, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Delay Paths Using LDC Editor**.

Defining False Paths

The set_false_path constraint identifies paths that are considered false and excluded from timing analysis.

Syntax set_false_path [-from *from port_object or cell_object*] [-to *to port_object or cell_object*]

or

set_false_path [-through *through_net_object*]

Arguments -from *from port_object or cell_object*

Specifies the timing path start point. A valid timing starting point is a clock, a primary input, a combinational logic cell, or a sequential cell (clock-pin).

-to *to port_object or cell_object*

Specifies the timing path end point. A valid timing end point is a primary output, a combinational logic cell, or a sequential cell (data-pin).

-through *through_net_object*

Specifies a net through which the paths should be blocked.

Examples The following example specifies all paths from clock pins of the registers in clock domain clk1 to data pins of a specific register in clock domain clk2 as false paths:

```
set_false_path –from [get_ports clk1] –to [get_cells reg_2]
```

The following example specifies all paths through the net UO/sigA as false:

```
set_false_path –through [get_nets UO/sigA]
```

For more information about defining delay paths with LDC Editor, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Delay Paths Using LDC Editor**.

Defining Multicycle Paths

The set_multicycle_path constraint defines a path that takes multiple clock cycles.

Syntax set_multicycle_path *ncycles* [-from *from net_object or cell_object*] [-to *to net_object or cell_object*]

Arguments *ncycles*

Specifies a value that represents the number of cycles the data path must have for setup check. The value is relative to the ending point clock and is defined as the delay required for arrival at the ending point.

-from *from net_object or cell_object*

Specifies the timing path start point. A valid timing start point is a sequential cell (clock pin) or a clock net (signal). You can also use the keyword “all_registers” to include all registers’ clock inputs.

-to *to net_object or cell_object*

Specifies the timing path end point. A valid timing end point is a sequential cell (data-pin) or a clock-net (signal). You can also use the keyword “all_registers” to include all registers’ data inputs.

Example The following example sets all paths between reg1 and reg2 to 3 cycles for setup check. Hold check is measured at the previous edge of the clock at reg2.

```
set_multicycle_path 3 –from [get_cells reg1] –to [get_cells
reg2]
```

For more information about defining delay paths with LDC Editor, in the Diamond software online help, refer to **User Guides > Applying Design Constraints > Using SDC Constraints > Applying Lattice Synthesis Engine Constraints > Defining Delay Paths Using LDC Editor**.

Setting Up Attributes

This section describes the Synplify Lattice Attributes that are supported by the Lattice Synthesis Engine (LSE). These attributes 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 Map and Place & Route.

All HDL attributes have priority over Strategy settings.

black_box_pad_pin

This attribute specifies pins on a user-defined black box module. The pins are defined as I/O pads that are visible 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. This attribute must be used in conjunction with the syn_black_box attribute.

Verilog Syntax object /* synthesis syn_black_box black_box_pad_pin = "portList" */ ;

where object is a module declaration, and portList is a spaceless, comma-separated list of the black box port names that are I/O pads.

Verilog Example

```
module black_box_pad_pin2(
                    input[4:0] in1,
                    input[4:0] in2,
                    input clk,
                    output[4:0] q
                    )/*          synthesis          syn_black_box
                    black_box_pad_pin="in1(4:0),q" */;

  reg [4:0] q;
  always @(posedge clk)
    begin
       q <= in1 + in2;
    end
endmodule

module black_box_pad_pin_instan(
  input[4:0] in1,
  input[4:0] in2,
  input[4:0] in3,
  input clk,
  output[5:0] q_out
  );

  wire [4:0] q;
  reg [5:0] q_out;
  black_box_pad_pin2 test_123(
                           .in1(in1),
                           .in2(in2),
                           .clk(clk),
                           .q(q)
                           );

  always @(posedge clk)
    begin
       q_out <= q + in3;
    end
endmodule
```

VHDL Syntax attribute black_box_pad_pin of object : architecture is "portList" ;

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

VHDL Example

```
entity BBDLHS is
  port (D: in std_logic;
    E: in std_logic;
    GIN : in std_logic_vector(2 downto 0);
    Q : out std_logic );
end;

architecture BBDLHS_behav of BBDLHS is
end bl_box_behav;
attribute syn_black_box : boolean;
attribute syn_black_box of BBDLHS_behav : architecture is true;
attribute black_box_pad_pin : string;
attribute black_box_pad_pin of BBDLHS_behav : architecture is
"GIN(2:0),Q";
```

full_case

Directive. For Verilog designs only. When used with a case, casex, or casez statement, this directive indicates that all possible values have been given, and that no additional hardware is needed to preserve signal values.

Verilog Syntax object /* synthesis full_case */

Verilog Example

```
module full_case1 (q, in1, in2, in3, in4, sel);
  output q;
  input in1, in2, in3, in4;
  input [3:0] sel;
  reg q;
  always @(sel or in1 or in2 or in3 or in4)
    begin
      casez (sel) /* synthesis full_case */
        4'b11??: q = in4;
        4'b?1??: q = in3;
        4'b???1: q = in1;
        4'b??1?: q = in2;
        default: q = 'bx;
      endcase
    end
endmodule
```

GSR

This attribute specifies the use of the global set/reset routing resources. Allows the user to specify which portions of the design are to be altered in the way they respond to the GSR reset signal. Unless specified otherwise, all design elements will respond to the global reset signal if it is present.

Available values:

- ENABLED – This is the default value on most library elements. This value allows the software to determine the final value and will be overridden by the parent hierarchy if the parent has a value of anything other than ENABLED.
- DISABLED – This prevents the hierarchy or element from responding to the GSR value. It cannot be changed by the parent's value.
- FORCEENABLE – This forces the hierarchy or element to respond to the GSR value. It cannot be changed by the parent's value.
- IPENABLE – This forces the hierarchy or element to respond to the GSR value when IP is being used in evaluation mode. It cannot be changed by the parent's value. This value is only for internal Lattice IP to use. It should never be used within a design itself.

Verilog Syntax object /* synthesis GSR = {ENABLED | DISABLED | FORCEENABLE | IPENABLE} */ ;

Verilog Example

```
`timescale 1 ns / 1 ns
module top (reg_q, rotate_q, a, b, r_l, clk, rst)/* synthesis
GSR = "ENABLED" */;
output [7:0] reg_q, rotate_q;
input [7:0] a, b;
input clk, rst, r_l;

sub1 reg8_1 (.clk(clk), .data(a), .q(reg_q), .rst(rst));
sub2 rotate_1 (rotate_q, b, clk, r_l, rst);
endmodule
```

VHDL Syntax attribute gsr of object : objectType is ENABLED | DISABLED | FORCEENABLE | IPENABLE;

VHDL Example

```
architecture archtest of test is
attribute gsr : string;
attribute gsr of archtest : architecture is “ENABLED”;
```

loc

The loc attribute specifies pin locations for Lattice I/Os, instances, and registers, and forward-annotates them to the place-and-route tool. Refer to the Lattice databook for valid pin location values. If the attribute is on a bus, the software writes out bit-blasted constraints for forward-annotation.

Verilog Syntax object /* synthesis loc = "pinLocations" */ ;

In the syntax, pin_locations is a spaceless, comma-separated list of pin locations.

Verilog Example

```
I/O pin location
input [3:0]DATA0 /* synthesis loc="p10,p12,p11,p15" *;
Register pin location
reg data_in_ch1_buf_reg3 /* synthesis loc="R40C47" */;
Vectored internal bus
reg [3:0] data_in_ch1_reg /*synthesis loc =
"R40C47,R40C46,R40C45,R40C44" */;
```

VHDL Syntax attribute loc of object : objectType is "pinLocations" ;

In the syntax, pinLocations is a spaceless, comma-separated list of pin locations.

VHDL Example

```
entity mycomp is port(DATA0 : in std_logic_vector (3 downto 0);
                .
                .
                .
                );
attribute loc : string;
attribute loc of DATA0 : signal is "p10,p12,p11,p15";
```

parallel_case

Directive. For Verilog designs only. Forces a parallel-multiplexed structure rather than a priority-encoded structure. This is useful because case statements are defined to work in priority order, executing (only) the first statement with a tag that matches the select value.

If the select bus is driven from outside the current module, the current module has no information about the legal values of select, and the software must create a chain of disabling logic so that a match on a statement tag disables all following statements. However, if you know the legal values of select, you can eliminate extra priority-encoding logic with the parallel_case directive. In the following example, the only legal values of select are 4'b1000, 4'b0100, 4'b0010, and 4'b0001, and only one of the tags can be matched at a time. Specify the parallel_case directive so that tag-matching logic can be parallel and independent, instead of chained.

Note

Designers should be aware that it is possible for the priority of overlapping cases in post-synthesis simulation to mismatch with the priority behavior in RTL simulation when using this pragma.

Verilog Syntax You specify the directive as a comment immediately following the select value of the case statement.

object /* synthesis parallel_case */

where object is a case, casex or casez statement declaration.

Verilog Example

```
module parallel_case1 (q, in1, in2, in3, in4, sel);
  output q;
  input in1, in2, in3, in4;
  input [3:0] sel;
  reg q;
  always @(sel or in1 or in2 or in3 or in4)
    begin
      casez (sel) /* synthesis parallel_case */
        4'b11??: q = in4;
        4'b?1??: q = in3;
        4'b???1: q = in1;
        4'b??1?: q = in2;
        default: q = 'bx;
      endcase
    end
endmodule
```

If the select bus is decoded within the same module as the case statement, the parallelism of the tag matching is determined automatically, and the parallel_case directive is unnecessary.

syn_black_box

This attribute specifies that a Verilog module or VHDL architecture declaration is for a black box. Only the module's interface is 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. However, the syn_black_box attribute cannot be used with the top-level module or architecture of a design. Additionally, the syn_black_box attribute is not supported for instances in Verilog or components in VHDL.

This attribute has an implicit Boolean value of 1 or true.

If any of the ports are I/O pads, add the black_box_pad_pin attribute. See "black_box_pad_pin" on page 35.

Verilog Syntax object /* synthesis syn_black_box */ ;

where *object* is a module declaration.

Verilog Example

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

VHDL Syntax attribute syn_black_box of object : architecture is true ;

where *object* is an architecture name. Data type is Boolean.

VHDL Example

```
entity bl_box is
  port (data : in std_logic_vector (7 downto 0);
    clk : in std_logic;
    out : out std_logic);
end;

architecture bl_box_behav of bl_box is
end bl_box_behav;
attribute syn_black_box : boolean;
attribute syn_black_box of bl_box_behav : architecture is true;
```

syn_encoding

This attribute specifies the encoding style for a finite state machine (FSM), overriding the default LSE encoding. The default encoding is based on the number of states in the FSM. This attribute takes effect only when LSE infers an FSM. This attribute has no effect when syn_state_machine is 0, which blocks inference of an FSM.

Values for syn_encoding are as follows:

- sequential – More than one bit of the state register can change at a time, but because more than one bit can be hot, the value must be decoded to determine the state. For example: 000, 001, 010, 011, 100
- onehot – Only two bits of the state register change (one goes to 0; one goes to 1) and only one of the state registers is hot (driven by a 1) at a time. For example: 0000, 0001, 0010, 0100, 1000
- gray – Only one bit of the state register changes at a time, but because more than one bit can be hot, the value must be decoded to determine the state. For example: 000, 001, 011, 010, 110

  There can be no more than four states for gray encoding. If the FSM has more than four states, LSE switches to sequential encoding.
- safe – If the state machine enters an invalid state, additional logic will drive the state machine into its reset state. The design must have a defined reset state.

  Safe encoding can be combined with either sequential or onehot encoding (not with gray encoding) as in:

  ```
  syn_encoding = "safe,onehot"
  ```

  If the safe value is given by itself, it combines with the encoding method of a preceding syn_encoding statement or the default method.

Verilog Syntax Object /* synthesis syn_encoding = "value" */;

Where object is an enumerated type and value is from the list above.

Verilog Example

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

entity syn_state_machine2 is
  port(
    clk : in std_logic;
    reset: in std_logic;
    en   : in std_logic;
    q  : out std_logic_vector(1 downto 0)
    );

end entity;

architecture behave of syn_state_machine2 is
  type state_type is(state0,state1,state2,state3);
  signal state,next_state:state_type;
  attribute syn_state_machine : boolean;
  attribute syn_state_machine of behave : architecture is true;
  attribute syn_encoding : string;
  attribute syn_encoding of state,next_state: signal is
"binary";
begin
  process(clk,reset)
  begin
    if reset = '1' then
      state <= state0;
    elsif clk'event and clk = '1' then
      state <= next_state;
    end if;
  end process;
  process(state)
  begin
    case state is
      when state0 =>
        if (en = '1') then
           q <= "00";
        end if;
        next_state <= state1;
      when state1=>
        if (en = '1') then
           q <= "01";
        end if;
        next_state <= state2;
      when state2 =>
        if (en = '1') then
           q <= "10";
        end if;
        next_state <= state3;
      when state3 =>
        if (en = '1') then
           q <= "11";
        end if;
        next_state <= state0;
    end case;
  end process;
end behave;
```

VHDL Syntax attribute syn_encoding of object: objectType is "value";

Where object is an enumerated type and value is from the list above.

VHDL Example

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

entity syn_encoding1 is
  port(
    clk : in std_logic;
    reset: in std_logic;
    en   : in std_logic;
    q  : out std_logic_vector(1 downto 0)
    );
end entity;

architecture behave of syn_encoding1 is
  signal state : std_logic_vector(3 downto 0);
  constant state0 : std_logic_vector(3 downto 0) := "1000";
  constant state1 : std_logic_vector(3 downto 0) := "0100";
  constant state2 : std_logic_vector(3 downto 0) := "0010";
  constant state3 : std_logic_vector(3 downto 0) := "0001";
  attribute syn_encoding : string;
  attribute syn_encoding of state : signal is "safe,onehot";
begin
  process(clk,reset,en)
  begin
    if reset = '1' then
      state <= state0;
      q <= "00";
    elsif clk'event and clk = '1' then
      case state is
        when state0 =>
          if (en = '1') then
              q <= "00";
          end if;
          state <= state1;
        when state1=>
          if (en = '1') then
              q <= "01";
          end if;
          state <= state2;
        when state2 =>
          if (en = '1') then
              q <= "10";
          end if;
          state <= state3;
        when state3 =>
          if (en = '1') then
              q <= "11";
          end if;
          state <= state0;
        when others => null;
      end case;
    end if;
  end process;
end behave;
```

syn_force_pads

This attribute prevents unused ports from being optimized away to allow I/O pad insertion on the unused port. This attribute is not supported at the global level. Instead, set the use_io_insertion option to control I/O insertion globally.

This attribute is supported in the rtl, and it will override the global use_io_insertion option setting on the given input, output, or bidir port.

For example, in the following Verilog file, the syn_force_pads attribute can be set to 1 on an unused input port (dataz), and it will not be optimized away, regardless of the use_io_insertion global setting.

Verilog syntax object /* synthesis syn_force_pads = {1 | 0} */;

where object is port declaration.

Verilog Example

```
`define DSIZE 9
`define OSIZE 18

module multp9x9(dataout, dataax, dataay, dataz, clk, rst, ce);
   output [`OSIZE-1:0] dataout;
   input [`DSIZE:0] dataz /* synthesis syn_force_pads = 1*/;
   input [`DSIZE-1:0] dataax, dataay;
   input clk, rst, ce;
   reg  [`DSIZE-1:0] dataax_reg, dataay_reg;

   reg [`OSIZE-1:0] dataout;
   wire [`OSIZE-1:0] dataout_tmp ;
   assign dataout_tmp = dataax_reg * dataay_reg;

   always @(posedge clk or posedge rst)
   begin
     if (rst)
     begin
         dataax_reg <= 0;
         dataay_reg <= 0;
         dataout <= 0;
     end
     else if (ce == 1'b1)
     begin
         dataax_reg <= dataax;
         dataay_reg <= dataay;
         dataout <= dataout_tmp;
     end
   end
endmodule
```

To force I/O pads to be inserted for input ports that do not drive logic, follow the guidelines below.

- To force I/O pad insertion on an individual port, set the syn_force_pads attribute on the port with a value to 1. To disable I/O insertion for a port, set the attribute on the port with a value of 0.

Enable this attribute to preserve user-instantiated pads, insert pads on unconnected ports, insert bi-directional pads on bi-directional ports instead of converting them to input ports, or insert output pads on unconnected outputs.

If you do not set the syn_force_pads attribute, the synthesis design optimizes any unconnected I/O buffers away.

VHDL syntax Attribute syn_force_pads of object: objectType is "true | false" ;

VHDL Example

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

entity multp9x9 is
  port ( dataout : out std_logic_vector(17 downto 0);
      dataax, dataay: in std_logic_vector( 8 downto 0);
           dataz : in std_logic_vector(8 downto 0);
           clk,rst,ce: in std_logic
    );
  attribute syn_force_pads : string;
  attribute syn_force_pads of dataz : signal is "true";

end multp9x9;

architecture rtl of multp9x9 is

signal dataax_reg,dataay_reg: std_logic_vector(8 downto 0);
signal dataout_tmp: std_logic_vector(17 downto 0);

begin

  dataout_tmp <= dataax_reg * dataay_reg;
  process (clk, rst)
  begin
      if rst = '1' then
           dataax_reg <= (others => '0');
           dataay_reg <= (others => '0');
           dataout <= (others => '0');
      elsif clk'event and clk = '1' then
        if ce = '1' then
           dataax_reg <= dataax;
           dataay_reg <= dataay;
           dataout <= dataout_tmp;
      end if;
      end if;
  end process;

end rtl;
```

syn_hier

This attribute allows you to control the amount of hierarchical transformation that occurs across boundaries on module or component instances during optimization. This attribute cannot be applied globally. The user must set this attribute on the selective modules to stop cross-boundary optimizations.

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.

Verilog Syntax 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.

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.

VHDL Example

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

syn_insert_pad

This attribute removes an existing I/O buffer from a port or net when I/O buffer insertion is enabled.

The syn_insert_pad attribute is used when the use_io_insertion global option is enabled (when I/O buffers are automatically inserted) to allow users to selectively remove an individual buffer from a port or net.

It can also be used to force an I/O buffer to be inserted on a specific port or net, if the use_io_insertion global option is disabled.

- Setting the attribute to 0 on a port or net removes the I/O buffer (or prevents an I/O buffer from being automatically inserted, if the use_io_insertion global option is enabled).

- Setting the attribute to 1 on a port or net forces an I/O buffer to be inserted if the use_io_insertion global option is disabled.

Verilog Syntax object /* synthesis syn_insert_pad = {1 | 0} */;

where *object* is a port or net declaration.

Verilog Example

```
`define OSIZE 16
`define DSIZE 8

module mac8x8 (dataout, x, y, clk, rst);
 output [`OSIZE:0] dataout;
 input [`DSIZE-1:0] x, y;
 input clk;
 input rst /* synthesis syn_insert_pad = 0 */;
 reg  [`OSIZE:0] dataout;
 reg [`DSIZE-1:0] x_reg, y_reg;
 wire [`OSIZE-1:0] multout ;
 wire [`OSIZE:0] sum_out;

 assign multout = x_reg * y_reg;
 assign sum_out = multout + dataout;

 always @(posedge clk or posedge rst)
 begin
   if (rst)
   begin
      x_reg = 0;
      y_reg = 0;
      dataout = 0;
   end
   else
   begin
      x_reg = x;
      y_reg = y;
      dataout = sum_out;
   end
 end
endmodule
```

In the previous example, the input port labeled “rst” will not have an input buffer connected to it in the technology-mapped netlist after LSE completes.

VHDL Syntax attribute syn_insert_pad of object : objectType is "true | false" ;

VHDL Example

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

entity register_en_reset is
          generic (
                    width : integer := 8
                     );
  port (
    datain  : in std_logic_vector(width-1 downto 0);
    clk     : in std_logic;
    enable  : in std_logic;
    reset   : in std_logic;
    dataout : out std_logic_vector(width-1 downto 0)
    );
  attribute syn_insert_pad : string;
  attribute syn_insert_pad of reset : signal is "false";


end register_en_reset;

architecture lattice_behav of register_en_reset is

begin
    process (clk,reset)
    begin
      if (reset = '1') then
                      dataout <= (others => '0');
      elsif (rising_edge(clk) and enable = '1') then
        dataout <= datain;
      end if;
    end process;
end lattice_behav;
```

syn_keep

This attribute 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 (/).

Verilog Example

```
module syn_keep1(
        input a,
        input b,
        input clk,
        output q1,
        output q2);
  reg temp1;
  reg temp2;
  reg q1;
  reg q2;
  wire or_result;
  wire keep1/* synthesis syn_keep=1 */;
  wire keep2/* synthesis syn_keep=1 */;

  always @(posedge clk)
    begin
      temp1 = a;
      temp2 = b;
    end
  assign or_result = (temp1 | temp2);
  assign keep1 = or_result;
  assign keep2 = or_result;
  always@(posedge clk)
    begin
      q1 = keep1;
      q2 = keep2;

    end
endmodule
```

VHDL Syntax attribute syn_keep of object : objectType is true ;

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

VHDL Example

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

entity syn_keep1 is
  port(
    a : in std_logic;
    b : in std_logic;
    clk: in std_logic;
    q1: out std_logic;
    q2: out std_logic
    );
end entity;

architecture behave of syn_keep1 is
  signal temp1 : std_logic;
  signal temp2 : std_logic;
  signal keep1 : std_logic;
  signal keep2 : std_logic;
  signal or_result : std_logic;
  attribute syn_keep: boolean;
  attribute syn_keep of keep1,keep2: signal is true;
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      temp1 <= a;
      temp2 <= b;
    end if;
  end process;

  or_result <= (temp1 or temp2);
  keep1 <= or_result;
  keep2 <= or_result;

  process(clk)
  begin
    if clk'event and clk = '1' then
      q1 <= keep1;
      q2 <= keep2;
    end if;
  end process;

end behave;
```

syn_maxfan

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

Verilog Syntax object /* synthesis syn_maxfan = "value" */ ;

Note

LSE will take integer values for non-integral values to syn_maxfan attribute.

For example, syn_maxfan value of 5.1 will be truncated to 5.

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" ;

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;
```

See Also "Optimizing LSE for Area and Speed" on page 568

syn_multstyle

This attribute specifies whether the multipliers are implemented as dedicated hardware blocks or as logic.

syn_multstyle Values block_mult | logic

Value Description Default block_mult Implements the multipliers as dedicated hardware blocks (Lattice: DSP blocks)

This attribute only applies to families that use DSP blocks on the device. To override this behavior, specify a value of logic.

Verilog Syntax input net /* synthesis syn_multstyle = "block_mult | logic" */;

Verilog Example

```
module syn_multstyle1(
            input [7:0] in1,
            input [7:0] in2,
            input rst,
            input clk,
            output [15:0] result);

  reg [7:0] temp1,temp2;
  reg [15:0] result;
  wire [15:0] product /*synthesis syn_multstyle = "logic"*/;

  always@(posedge clk ,negedge rst)
    begin
      if(!rst)
        begin
          temp1 = 'b0;
          temp2 = 'b0;
        end
      else
        begin
          temp1  = in1;
          temp2  = in2;
        end
    end

  assign product = temp1*temp2;

  always@(posedge clk, negedge rst)
    begin
      if(!rst)
        begin
          result  = 'b0;
        end
      else
        begin
          result = product;
        end
      end
endmodule
```

VHDL Syntax attribute syn_multstyle of instance : signal is "block_mult | logic";

VHDL Example

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

entity syn_pipeline2 is
  port(
     inA1: in std_logic_vector(3 downto 0);
     inA2: in std_logic_vector(3 downto 0);
     inB1: in std_logic_vector(3 downto 0);
     inB2: in std_logic_vector(3 downto 0);
     clk : in std_logic;
     sum : out std_logic_vector(7 downto 0)
    );
end entity;

architecture behave of syn_pipeline2 is
  signal temp1,temp2,temp3,temp4: std_logic_vector(3 downto 0);
  signal sum_s : std_logic_vector(7 downto 0);
  attribute syn_pipeline : string;
  attribute syn_pipeline of sum_s: signal is "true";
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      temp1 <= inA1;
      temp2 <= inA2;
      temp3 <= inB1;
      temp4 <= inB2;
      sum_s <= (temp1*temp3) + (temp2*temp4);
    end if;
  end process;

  process(clk)
  begin
    if clk'event and clk = '1' then
      sum <= sum_s;
    end if;
  end process;
end behave;
```

syn_noprune

This attribute prevents instance optimization for black-box modules (including technology-specific primitives) with unused output ports. This attribute is not a global attribute. It works on the component basis. The user must set the attribute on the instance.

Verilog Syntax object /* synthesis syn_noprune = 1 */ ;

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

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 a component.

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 noprune : component 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_pipeline

This attribute permits registers to be moved to improve timing. Specifies that registers that are outputs of Multipliers/Adders can be moved to improve timing. Depending on the criticality of the path, the tool moves the output register to the input side.

Verilog Syntax object /* synthesis syn_pipeline = {1 | 0} */ ;

where *object* is a register declaration.

The value of 0 (or false) indicates pipelining for the specified register is disabled, which means the register position in the design is fixed.

The value of 1 (or true) indicates pipelining for the specified register is allowed, which means the register may be moved if it helps improve timing.

LSE will identify registers that are candidates for possible pipelining based on running RTL timing analysis. It may identify some candidate registers, or it may determine there are none that are suitable.

If LSE decides no candidate registers for pipelining exist, if the user sets the syn_pipeline attribute to “1” on a specific register in the RTL to force pipelining for that register, that attribute will not be honored.

If global pipelining is enabled for a design, and given one or more registers that LSE has identified as possible candidates for pipelining, the user may prevent these registers from being pipelined by setting synthesis attribute syn_pipeline=0 for each of those registers in the RTL.

Verilog Example

```
module pipeline (a,b,c,d,clk,out);

input [3:0] a,b,c,d;
input clk;
output[7:0]out;

reg[7:0]out,out1 /* synthesis syn_pipeline = 0 */;
reg[3:0] a_temp,b_temp,c_temp,d_temp;

always @(posedge clk)
begin
   a_temp <= a;
   b_temp <= b;
   c_temp <= c;
   d_temp <= d;
   out1 <= (a_temp * b_temp) +(c_temp * d_temp);
   out <= out1;
end
endmodule
```

In the previous example, the registers labeled “out1” will not be moved to the input side of the adder to improve timing.

VHDL Syntax attribute syn_pipeline of object : objectType is {true|false} ;

VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity syn_pipeline_exp is
port (CLK_0 : in std_logic;
    A_IN : in std_logic_vector(3 downto 0);
    B_IN : in std_logic_vector(3 downto 0);
    RST : out std_logic_vector(7 downto 0)
    );
end syn_pipeline_exp;

architecture rtl of syn_pipeline_exp is
signal A_REGSTR : std_logic_vector(3 downto 0);
signal B_REGSTR : std_logic_vector(3 downto 0);
signal TMP : std_logic_vector(7 downto 0);
signal TMP1 : std_logic_vector(7 downto 0);
signal TMP2 : std_logic_vector(7 downto 0);
attribute syn_pipeline : string;
attribute syn_pipeline of TMP1 : signal is "true";

begin
  process(CLK_0)
  begin
    if (CLK_0'event and CLK_0 = '1') then
      TMP <= A_REGSTR * B_REGSTR;
      A_REGSTR <= A_IN;
      B_REGSTR <= B_IN;
      TMP1 <= TMP;
      TMP2 <= TMP1;
      RST <= TMP2;
    end if;
  end process;

end rtl;
```

syn_preserve

This attribute prevents sequential optimizations such as constant propagation and inverter push-through from removing the specified register. The syn_encoding attribute is not honored if there is a syn_preserve attribute on any of the state machine registers.

Verilog Syntax object /* synthesis syn_preserve = 1 */ ;

where object is a register definition signal or a module.

Verilog Example

```
module syn_preserve1(
          input[3:0]in1,
          input[3:0]in2,
          input[3:0]in3,
          input clk,
          output [7:0] result,
          output [3:0] sum
          )/* synthesis syn_preserve= 1*/;

reg [7:0] result/*synthesis syn_multstyle = "EBR"*/;
reg [3:0] temp1,temp2,temp3;
reg [3:0] sum;

always@(posedge clk)
  begin
    temp1 = in1 & in2;
    temp2 = !temp1;
    temp3 = temp1 & temp2;
    result = temp3*in3;
    sum = temp3 + in3;
  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.

VHDL Example

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

entity syn_preserve2 is
  port(
     in1: in std_logic_vector(3 downto 0);
     in2: in std_logic_vector(3 downto 0);
     in3: in std_logic_vector(3 downto 0);
     clk: in std_logic;
     result: out std_logic_vector(7 downto 0);
     sum   : out std_logic_vector(3 downto 0)
    );
end entity;

architecture behave of syn_preserve2 is
  signal temp1,temp2,temp3 : std_logic_vector(3 downto 0);
  attribute syn_preserve : boolean;
  attribute syn_preserve of behave: architecture is true;
  attribute syn_multstyle : string;
  attribute syn_multstyle of result: signal is "EBR";
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      temp1 <= in1 and in2;
      temp2 <= not temp1;
      temp3 <= temp1 and temp2;
      result <= temp3*in3;
      sum <= temp3 + in3;
    end if;
  end process;
end behave;
```

syn_ramstyle

The syn_ramstyle attribute specifies the implementation to use for an inferred RAM. You apply syn_ramstyle globally, to a module, or to a RAM instance. To turn off RAM inference, set its value to registers.

The following values can be specified globally or on a module or RAM instance:

- registers – Causes an inferred RAM to be mapped to registers (flip-flops and logic) rather than the technology-specific RAM resources.
- distributed – Causes the RAM to be implemented using the distributed RAM or PFU resources.
- block_ram – Causes the RAM to be implemented using the dedicated RAM resources. If your RAM resources are limited, you can use this attribute to map additional RAMs to registers instead of the dedicated or distributed RAM resources.
- no_rw_check (some modes, but all technologies). – You cannot specify this value alone. Without no_rw_check, the synthesis tool inserts bypass logic around the RAM to prevent the mismatch. If you know your design does not read and write to the same address simultaneously, use no_rw_check to eliminate bypass logic. Use this value only when you cannot simultaneously read and write to the same RAM location and you want to minimize overhead logic.

Verilog Syntax object /* synthesis syn_ramstyle = "string" */ ;

where object is a register definition (reg) signal. The data type is string.

Verilog Example

```
module ram4 (datain,dataout,clk);
output [31:0] dataout;
input clk;
input [31:0] datain;
reg [7:0] dataout[31:0] /* synthesis syn_ramstyle="block_ram" */;
```

VHDL Syntax attribute syn_ramstyle of object : objectType is "string" ;

where object is a signal that defines a RAM or a label of a component instance. Data type is string.

VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;
entity ram4 is
   port (d : in std_logic_vector(7 downto 0);
         addr : in std_logic_vector(2 downto 0);
         we : in std_logic;
         clk : in std_logic;
         ram_out : out std_logic_vector(7 downto 0) );
end ram4;
library synplify;
architecture rtl of ram4 is
type mem_type is array (127 downto 0) of std_logic_vector (7
downto 0);
signal mem : mem_type; -- mem is the signal that defines the
RAM
attribute syn_ramstyle : string;
attribute syn_ramstyle of mem : signal is "block_ram";
```

syn_replicate

This attribute controls replication. The synthesis tool can automatically replicate registers during optimization. This attribute disables replication either globally or on a per register basis.

Verilog Syntax object /* synthesis syn_replicate = 1 | 0 */;

Verilog Example

```
module syn_replicate1 (en1,en2,clk,in1,in2,q);
  input en1,en2;
  input clk;
  input [6:0]in1,in2;
  output [6:0]q;
  reg [6:0]q;
  reg enc /* synthesis syn_maxfan = 1 syn_replicate = 1*/;

  always @(posedge clk)
    begin
      enc = en1 & en2;
    end

  always @(posedge clk)
    begin
      if (enc)
        q = in1;

      else
        q = in2;
    end
endmodule
```

VHDL Syntax attribute syn_replicate of object : objectType is true | false ;

VHDL Example

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

entity syn_replicate2 is
  port(
     en1: in std_logic;
     en2: in std_logic;
     clk: in std_logic;
     in1: in std_logic_vector(6 downto 0);
     in2: in std_logic_vector(6 downto 0);
     q:  out std_logic_vector(6 downto 0)
    );
end entity;

architecture behave of syn_replicate2 is
  signal enc : std_logic;
  attribute syn_maxfan: integer;
  attribute syn_maxfan of behave : architecture is 1;
  attribute syn_replicate: boolean;
  attribute syn_replicate of enc : signal is false;
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      enc <= (en1 and en2);
    end if;
  end process;

  process(clk)
  begin
    if enc = '1' then
      q <= in1;
    else
      q <= in2;
    end if;
  end process;
end behave;
```

syn_romstyle

This attribute allows you to implement ROM architectures using dedicated or distributed ROM. Infer ROM architectures using a CASE statement in your code.

For the synthesis tool to implement a ROM, at least half of the available addresses in the CASE statement must be assigned a value. For example, consider a ROM with six address bits (64 unique addresses). The case statement for this ROM must specify values for at least 32 of the available addresses. You can apply the syn_romstyle attribute globally to the design by adding the attribute to the module or entity.

The following values can be specified globally on a module or ROM instance:

- auto – Allows the synthesis tool to chose the best implementation to meet the design requirements for speed, size, etc.
- logic – Causes the ROM to be implemented using the distributed ROM or PFU resources.
- EBR – Causes the ROM to be implemented using the dedicated ROM resources. If your ROM resources are limited, you can use this attribute to map additional ROM to registers instead of the dedicated or distributed RAM resources.

Verilog Syntax object /* syn_romstyle = "auto(default) | EBR | logic" */ ;

Verilog Example

```
reg [8:0] z /* synthesis syn_romstyle = "EBR" */;
```

VHDL Syntax attribute syn_romstyle of object : object_type is "auto(default) | EBR | logic" ;

VHDL Example

```
signal z : std_logic_vector(8 downto 0);
attribute syn_romstyle : string;
attribute syn_romstyle of z : signal is "logic";
```

syn_srlstyle

This attribute determines how to implement the sequential shift components.

Verilog Syntax object /* synthesis syn_srlstyle = "string",

where string can take one of the following values:

registers: seqShift register components are implemented as registers.

distributed: seqShift register components are implemented as distributed RAM.

block_ram: seqShift register components are implemented as block RAM

If the attribute value set by the user cannot be honored (for example, the user sets the attribute value to "block_ram", however, the selected device does not contain enough available EBR blocks to implement the shift register), LSE will display a message to indicate this.

" | registers | distributed | |block_ram" */ ;

In the above syntax, *object* is a register declaration.

Verilog Example The following example implements seqShift components as distributed memory with any required fabric logic.

```
module test_srl(clk, enable, dataIn, result, addr);
input clk, enable;
input [3:0] dataIn;
input [3:0] addr;
output [3:0] result;
reg [3:0] regBank[15:0]
  /* synthesis syn_srlstyle="distributed" */;
integer i;
always @(posedge clk) begin
  if (enable == 1) begin
    for (i=15; i>0; i=i-1) begin
      regBank[i] <= regBank[i-1];
    end
  regBank[0] <= dataIn;
  end
end
assign result = regBank[addr];
endmodule
```

The following example implements a seqShift for 16x256 bits wide and serial in and serial out register using syn_srlstyle set to block_ram.

VHDL Syntax attribute syn_srlstyle of object : signal is

" registers | distributed |block_ram " ;

In the above syntax, object is a register.

```
// shift left register with 16X256 bits width and serial in and
serial out
module test(clock, arst, sr_en, shiftin, shiftout);
parameter sh_len=16;
parameter sh_width=256;
parameter ARESET_VALUE = {(sh_width){1'b0}};
input clock,arst,sr_en;
input [sh_width-1:0] shiftin;
output [sh_width-1:0] shiftout;
integer i;
reg [sh_width-1:0] sreg [sh_len-1:0] /* synthesis
syn_srlstyle="block_ram" */;
  always @(posedge clock or posedge arst)
  begin
    if(arst)
      begin
        for(i = 0;i <= sh_len-1;i = i+1)
        sreg[i] <= ARESET_VALUE ;
      end
    else
  begin
    if(sr_en)
    begin
      sreg[0] <= shiftin;
      for(i=sh_len-1;i>0;i=i-1)
        sreg[i] <= sreg[i-1];
      end
    end
  end
assign shiftout = sreg[sh_len-1];
endmodule
```

Verilog Example The example below implements seqShift components as distributed memory primitives:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity d_p is
  port (clk : in std_logic;
    data_out : out std_logic_vector(127 downto 0));
  end d_p;

architecture rtl of d_p is
type dataAryType is array(3 downto 0) of
  std_logic_vector(127 downto 0);
signal h_data_pip_i : dataAryType;
attribute syn_srlstyle : string;
attribute syn_srlstyle of h_data_pip_i : signal
is "distributed";
begin
  process (Clk)
  begin
    if (Clk'Event And Clk = '1') then
      h_data_pip_i <= (h_data_pip_i(2 DOWNTO 0)) &
      h_data_pip_i(3);
    end if;
  end process;
data_out <= h_data_pip_i(0);
end rtl;
```

syn_sharing

Directive. Enables or disables the sharing of operator resources during the compilation stage of synthesis.

The syn_sharing directive controls resource sharing during the compilation stage of synthesis. This is a compiler-specific optimization that does not affect the mapper; this means that the mapper might still perform resource sharing optimizations to improve timing, even if syn_sharing is disabled.

If you disable resource sharing globally, you can use the syn_sharing directive to turn on resource sharing for specific modules or architectures.

Verilog Syntax object /* synthesis syn_sharing=”on | off” */ ;

Verilog Example

```
module syn_sharing1 (
          input [7:0] inA1,
          input [7:0] inA2,
          input [7:0] inB1,
          input [7:0] inB2,
          input clk,
          input sel1,
          input sel2,
          input rst,
          output [15:0] product1,
          output [15:0] product2
          )/*synthesis syn_sharing = 1*/;

  reg [15:0] product1,product2;
  wire [15:0] temp1,temp2;
  assign temp1 = inA1*inB1;
  assign temp2 = inA2*inB2;
  always@(posedge clk)
    begin
      if(sel1)
        begin
          if (sel2)
            product1 = temp1;
          else
            product1 = temp2;
        end
      else
        begin
          if (sel2)
            product2 = temp1;
          else
            product2 = temp2;
        end
    end
endmodule
```

VHDL Syntax attribute syn_sharing of object : objectType is “true | false” ;

VHDL Example

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

entity syn_sharing2 is
  port(
    inA1 : in std_logic_vector(7 downto 0);
    inA2 : in std_logic_vector(7 downto 0);
    inB1 : in std_logic_vector(7 downto 0);
    inB2 : in std_logic_vector(7 downto 0);
    clk  : in std_logic;
    sel1 : in std_logic;
    sel2 : in std_logic;
    rst  : in std_logic;
    product1 : out std_logic_vector(15 downto 0);
    product2 : out std_logic_vector(15 downto 0)
    );
end entity;

architecture behave of syn_sharing2 is
 signal temp1,temp2: std_logic_vector(15 downto 0);
 attribute syn_sharing : boolean;
 attribute syn_sharing of behave : architecture is false;
begin
  temp1 <= inA1*inB1;
  temp2 <= inA2*inB2;
  process(clk)
  begin
    if clk'event and clk ='1' then
      if sel1 = '1' then
        if sel2 = '1' then
          product1 <= temp1;
        else
          product1 <= temp2;
        end if;
      else
        if sel2 = '1' then
          product2 <= temp1;
        else
          product2 <= temp2;
        end if;

      end if;
    end if;
  end process;
end behave;
```

syn_state_machine

This attribute enables/disables state-machine optimization on individual state registers in the design. To extract some state machines, use this attribute with a value of 1 on just those individual state-registers to be extracted. If there are state machines in your design that you do not want extracted, use syn_state_machine with a value of 0 to override extraction on just those individual state registers.

All state machines are usually detected during synthesis. However, on occasion there are cases in which certain state machines are not detected. You can use this attribute to declare those undetected registers as state machines.

The syn_sharing attribute only can be used in architecture. The syn_sharing attribute cannot be used in entity.

Verilog Syntax object /* synthesis syn_state_machine = 0 | 1 */ ;

where *object* is a state register. Data type is Boolean: 0 does not extract an FSM, 1 extracts an FSM.

Verilog Example

```
module syn_state_machine1 (clk, reset, en, q);
  input clk, reset, en;
  output[1:0]q;
  reg q;
  reg [3:0] state,next_state /* synthesis syn_state_machine = 0
*/;
  parameter state0 = 4'b1000;
  parameter state1 = 4'b0100;
  parameter state2 = 4'b0010;
  parameter state3 = 4'b0001;
  always @(posedge clk or posedge reset)
    begin
      if (reset)
          state <= state0;
      else
        state <= next_state;
    end

  always @(state)
    begin
      case (state)
        state0:
          begin
            if (en == 1)
              q <= 2'b00;
            next_state <= state1;
          end

        state1:
          begin
            if (en == 1)
              q <= 2'b01;
            next_state <= state2;
          end
        state2:
          begin
            if (en == 1)
              q <= 2'b10;
            next_state <= state3;
          end
        state3:
          begin
            if (en == 1)
              q <= 2'b11;
            next_state <= state0;
          end
      endcase
    end
endmodule
```

VHDL Syntax attribute syn_state_machine of object : objectType is true | false ;

where *object* is a signal that holds the value of the state machine.

VHDL Example

```
attribute syn_state_machine of current_state: signal is true;
```

Following is the source code used for the example in the previous figure.

```
library ieee;
use ieee.std_logic_1164.all;
entity syn_statemachine_exp is
port (CLK_0, RESET, IN1 : in std_logic;
                  OUT1 : out std_logic_vector (2 downto 0)
      );
end syn_statemachine_exp;

architecture behave of syn_statemachine_exp is
type ST_VALS is (STATE0, STATE1, STATE2, STATE3);
signal STATE, NXT_ST: ST_VALS;
attribute syn_state_machine : boolean;
attribute syn_state_machine of STATE : signal is true;

begin
  process (CLK_0, RESET)
  begin
    if RESET = '1' then
      STATE <= STATE0;
    elsif rising_edge(CLK_0) then
      STATE <= NXT_ST;
    end if;
  end process;

  process (STATE, IN1)
  begin
    case STATE is
      when STATE0 =>
        OUT1 <= "000";
        if IN1 = '1' then NXT_ST <= STATE1;
        else NXT_ST <= STATE0;
        end if;
      when STATE1 =>
        OUT1 <= "001";
        if IN1 = '1' then NXT_ST <= STATE2;
        else NXT_ST <= STATE1;
        end if;
      when STATE2 =>
        OUT1 <= "010";
        if IN1 = '1' then NXT_ST <= STATE3;
        else NXT_ST <= STATE2;
        end if;
      when others =>
        OUT1 <= "XXX"; NXT_ST <= STATE0;
    end case;
  end process;

end behave;
```

syn_use_carry_chain

This attribute is 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 ;

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;
```

syn_useenables

This attribute controls the use of clock enables on registers in the design. Usually exploiting clock enables on registers is beneficial. However, there are timing closure situations where clock enable routing causes timing violations. This is one reason why the user may want to stop the use of clock enable on the register.

Verilog Syntax: object /* synthesis syn_useenables = "0 | 1" */;

Verilog Example

```
module syn_useenable1 (Din1,Din2,en,clk,Dout);
  input [7:0] Din1, Din2;
  input clk,en;
  output [7:0] Dout;
  reg [7:0] temp1;
  reg [7:0] Dout /* synthesis syn_useenables = 0*/;
  always@(posedge clk)
    begin
      temp1 <= Din1 & Din2;
    end
  always @(posedge clk)
    begin
      if(en)
        Dout <= temp1;
    end
endmodule
```

VHDL Syntax:

attribute syn_useenables of object : objectType is "true | false";

VHDL Example

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

entity syn_useenable2 is
  port(
     Din1 : in std_logic_vector(7 downto 0);
     Din2 : in std_logic_vector(7 downto 0);
     clk  : in std_logic;
     en   : in std_logic;
     Dout : out std_logic_vector(7 downto 0)
    );
end entity;

architecture behave of syn_useenable2 is
  signal temp1 : std_logic_vector(7 downto 0);
  attribute syn_useenables: boolean;
  attribute syn_useenables of Dout: signal is false;
begin
  process(clk)
  begin
    if clk'event and clk = '1' then
      temp1 <= Din1 and Din2;
    end if;
  end process;

  process(clk)
  begin
    if clk'event and clk = '1' then
      if en = '1' then
        Dout <= temp1;
      end if;
    end if;
  end process;
end behave;
```

syn_useioff

This attribute overrides the default behavior to pack registers into I/O pad cells based on timing requirements for the target Lattice families. Attribute syn_useioff is Boolean-valued: 1 enables (default) and 0 disables register packing. You can place this attribute on an individual register or port or apply it globally. When applied globally, the synthesis tool packs all input, output, and I/O registers into I/O pad cells. When applied to a register, the synthesis tool packs the register into the pad cell; and when applied to a port, it packs all registers attached to the port into the pad cell.

The syn_useioff attribute can be set on the following ports:

- top-level port
- register driving the top-level port
- lower-level port, if the register is specified as part of the port declaration

Verilog Syntax object /*synthesis syn_useioff = {1 | 0} */ ;

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

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

To use this attribute on individual ports, apply it to individual port declarations.

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

VHDL Syntax attribute syn_useioff of object : objectType is true | false ;

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_useioff : boolean;
attribute syn_useioff of archtest : architecture is true;
```

translate_off/translate_on

This attribute allows you to synthesize designs originally written for use with other synthesis tools without needing to modify source code. All source code that is between these two attributes is ignored during synthesis.

Verilog Syntax /* pragma translate_off */
/* pragma translate_on */

Verilog Example

```
module real_time (ina, inb, out);
input ina, inb;
output out;
/* synthesis translate_off */
realtime cur_time;
/* synthesis translate_on */
assign out = ina & inb;
endmodule
```

VHDL Syntax pragma translate_off
pragma translate_on

VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;
entity adder is
   port (a, b, cin:in std_logic;
         sum, cout:out std_logic );
end adder;
architecture behave of adder is
signal a1:std_logic;
--synthesis translate_off
constant a1:std_logic:='0';
--synthesis translate_on
begin
   sum <= (a xor b xor cin);
   cout <= (a and b) or (a and cin) or (b and cin); end behave;
```

Inferring Block Primitives

This section describes inferring block primitives, including Memory and DSP Blocks.

Inferring Memory

This section describes inferring memory, including inferring RAM, RAM with Synchronous Read, Inferring Dual-Port RAM, Inferring ROM, Initializing Inferred RAM, and Creating Memory Initialization File.

Inferring RAM

The basic inferred RAM is synchronous. It can have synchronous or asynchronous reads and can be either single- or dual-port. You can also set initial values. Other features, such as resets and clock enables, can be added as desired. The following text lists the rules for coding inferred RAM. Following that, Figure 5 on page 79 (Verilog) and Figure 6 on page 80 (VHDL) show the code for a simple, single-port RAM with asynchronous read.

To code RAM to be inferred, do the following:

- Define the RAM as an indexed array of registers.
- To control how the RAM is implemented (with distributed or block RAM), consider adding the syn_ramstyle attribute. "syn_ramstyle" on page 60.
- Control the RAM with a clock edge and a write enable signal.
- For synchronous reads, see "Inferring RAM with Synchronous Read" on page 81.
- For single-port RAM, use the same address bus for reading and writing.
- For dual-port RAM, pseudo and true, see "Inferring Dual-Port RAM" on page 83.
- If desired, assign initial values to the RAM as described in "Initializing Inferred RAM" on page 87.

Figure 5: Simple, Single-Port RAM in Verilog

```
module ram (din, addr, write_en, clk, dout);
  parameter addr_width = 8;
  parameter data_width = 8;
  input [addr_width-1:0] addr;
  input [data_width-1:0] din;
  input write_en, clk;
  reg [data_width-1:0] mem [(1<<addr_width)-1:0];
    // Define RAM as an indexed memory array.

  always @(posedge clk) // Control with a clock edge.
  begin
    if (write_en) // And control with a write enable.
      mem[(addr)] <= din;
  end
  assign dout = mem[addr];
endmodule
```

Figure 6: Simple, Single-Port RAM in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ram is
generic (
  addr_width : natural := 8;
  data_width : natural := 8);
port (
  addr : in  std_logic_vector (addr_width - 1 downto 0);
  write_en : in  std_logic;
  clk : in  std_logic;
  din : in  std_logic_vector (data_width - 1 downto 0);
  dout : out std_logic_vector (data_width - 1 downto 0));
end ram;

architecture rtl of ram is
  type mem_type is array ((2** addr_width) - 1 downto 0) of
    std_logic_vector(data_width - 1 downto 0);
  signal mem : mem_type;
    -- Define RAM as an indexed memory array.
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then --Control with clock edge
      if (write_en = '1') then -- Control with a write enable.
        mem(conv_integer(addr)) <= din;
      end if;
    end if;
  end process;
  dout <= mem(conv_integer(addr));
end rtl;
```

Inferring RAM with Synchronous Read

For synchronous reads, add a register for the read address or for the data output. Load the register inside the procedure or process that is controlled by the clock. See the following examples. They show the simple RAM of Figure 5 on page 79 (for Verilog) and Figure 6 on page 80 (for VHDL) modified for synchronous reads. Changes are in bold text.

Verilog Examples

Figure 7: RAM with Registered Output in Verilog

```
module ram (din, addr, write_en, clk, dout);
  parameter addr_width = 8;
  parameter data_width = 8;
  input [addr_width-1:0] addr;
  input [data_width-1:0] din;
  input write_en, clk;
  output [data_width-1:0] dout;
  reg [data_width-1:0] dout; // Register for output.
  reg [data_width-1:0] mem [(1<<addr_width)-1:0];

  always @(posedge clk)
  begin
    if (write_en)
      mem[(addr)] <= din;
    dout = mem[addr]; // Output register controlled by clock.
  end
endmodule
```

Figure 8: RAM with Registered Read Address in Verilog

```
module ram (din, addr, write_en, clk, dout);
  parameter addr_width = 8;
  parameter data_width = 8;
  input [addr_width-1:0] addr;
  input [data_width-1:0] din;
  input write_en, clk;
  output [data_width-1:0] dout;
  reg [data_width-1:0] raddr; // Register for read address.
  reg [data_width-1:0] mem [(1<<addr_width)-1:0];

  always @(posedge clk)
  begin
    if (write_en)
    begin
      mem[(addr)] <= din;
    end
    raddr <= addr; // Read addr. register controlled by clock.
  end
  assign dout = mem[raddr];
endmodule
```

VHDL Examples

Figure 9: RAM with Registered Output in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ram is
generic (
  addr_width : natural := 8;
  data_width : natural := 8);
port (
  addr : in  std_logic_vector (addr_width - 1 downto 0);
  write_en : in  std_logic;
  clk : in  std_logic;
  din : in  std_logic_vector (data_width - 1 downto 0);
  dout : out std_logic_vector (data_width - 1 downto 0));
end ram;

architecture rtl of ram is
  type mem_type is array ((2** addr_width) - 1 downto 0) of
    std_logic_vector(data_width - 1 downto 0);
  signal mem : mem_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (write_en = '1') then
        mem(conv_integer(addr)) <= din;
      end if;
    end if;
    dout <= mem(conv_integer(addr));
      -- Output register controlled by clock.
  end process;
end rtl;
```

Figure 10: RAM with Registered Read Address in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ram is
generic (
  addr_width : natural := 8;
  data_width : natural := 8);
port (
  addr : in  std_logic_vector (addr_width - 1 downto 0);
  write_en : in  std_logic;
  clk : in  std_logic;
  din : in  std_logic_vector (data_width - 1 downto 0);
  dout : out std_logic_vector (data_width - 1 downto 0));
end ram;

architecture rtl of ram is
  type mem_type is array ((2** addr_width) - 1 downto 0) of
    std_logic_vector(data_width - 1 downto 0);
  signal mem : mem_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (write_en = '1') then
        mem(conv_integer(addr)) <= din;
      end if;
      raddr <= addr;
        -- Read address register controlled by clock.
    end if;
  end process;
  dout <= mem(conv_integer(raddr));
end rtl;
```

Inferring Dual-Port RAM

For dual-port RAM, pseudo or true:

- Use two address buses.
- If the design does not simultaneously read and write the same address, add the syn_ramstyle attribute with the no_rw_check value to minimize overhead logic.
- If writing in Verilog, use non-blocking assignments.

The following examples are based on the simple RAM of Figure 5 on page 79 (for Verilog) and Figure 6 on page 80 (for VHDL).

Verilog Examples

Figure 11: Pseudo Dual-Port RAM in Verilog

```
module ram (din, write_en, waddr, wclk, raddr, rclk, dout);
  parameter addr_width = 8;
  parameter data_width = 8;
  input [addr_width-1:0] waddr, raddr;
  input [data_width-1:0] din;
  input write_en, wclk, rclk;
  output [data_width-1:0] dout;
  reg [data_width-1:0] dout;
  reg [data_width-1:0] mem [(1<<addr_width)-1:0]
    /* synthesis syn_ramstyle = "no_rw_check" */ ;

  always @(posedge wclk) // Write memory.
  begin
    if (write_en)
      mem[waddr] <= din; // Using write address bus.
  end
  always @(posedge rclk) // Read memory.
  begin
    dout <= mem[raddr]; // Using read address bus.
  end
endmodule
```

Figure 12: True Dual-Port RAM in Verilog

```
module ram (dina, write_ena, addra, clka, douta,
  dinb, write_enb, addrb, clkb, doutb);
  parameter addr_width = 8;
  parameter data_width = 8;
  input [addr_width-1:0] addra, addrb;
  input [data_width-1:0] dina, dinb;
  input write_ena, clka, write_enb, clkb;
  output [data_width-1:0] douta, doutb;
  reg [data_width-1:0] douta, doutb;
  reg [data_width-1:0] mem [(1<<addr_width)-1:0]
    /* synthesis syn_ramstyle = "no_rw_check" */ ;

  always @(posedge clka) // Using port a.
  begin
    if (write_ena)
      mem[addra] <= dina; // Using address bus a.
    douta <= mem[addra];
  end
  always @(posedge clkb) // Using port b.
  begin
    if (write_enb)
      mem[addrb] <= dinb; // Using address bus b.
    doutb <= mem[addrb];
  end
endmodule
```

VHDL Examples

Figure 13: Pseudo Dual-Port RAM in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ram is
generic (
  addr_width : natural := 8;
  data_width : natural := 8);
port (
  write_en : in  std_logic;
  waddr : in  std_logic_vector (addr_width - 1 downto 0);
  wclk : in  std_logic;
  raddr : in  std_logic_vector (addr_width - 1 downto 0);
  rclk : in  std_logic;
  din : in  std_logic_vector (data_width - 1 downto 0);
  dout : out std_logic_vector (data_width - 1 downto 0));
end ram;

architecture rtl of ram is
  type mem_type is array ((2** addr_width) - 1 downto 0) of
    std_logic_vector(data_width - 1 downto 0);
  signal mem : mem_type;
  attribute syn_ramstyle: string;
  attribute syn_ramstyle of mem: signal is "no_rw_check";
begin
  process (wclk) -- Write memory.
  begin
    if (wclk'event and wclk = '1') then
      if (write_en = '1') then
        mem(conv_integer(waddr)) <= din;
          -- Using write address bus.
      end if;
    end if;
  end process;
  process (rclk) -- Read memory.
  begin
    if (rclk'event and rclk = '1') then
      dout <= mem(conv_integer(raddr));
        -- Using read address bus.
    end if;
  end process;
end rtl;
```

Figure 14: True Dual-Port RAM in VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity ram is
generic (
  addr_width : natural := 8;
  data_width : natural := 8);
port (
  addra : in  std_logic_vector (addr_width - 1 downto 0);
  write_ena : in  std_logic;
  clka : in  std_logic;
  dina : in  std_logic_vector (data_width - 1 downto 0);
  douta : out std_logic_vector (data_width - 1 downto 0);
  addrb : in  std_logic_vector (addr_width - 1 downto 0);
  write_enb : in  std_logic;
  clkb : in  std_logic;
  dinb : in  std_logic_vector (data_width - 1 downto 0);
  doutb : out std_logic_vector (data_width - 1 downto 0));
end ram;

architecture rtl of ram is
  type mem_type is array ((2** addr_width) - 1 downto 0) of
    std_logic_vector(data_width - 1 downto 0);
  signal mem : mem_type;
  attribute syn_ramstyle: string;
  attribute syn_ramstyle of mem: signal is "no_rw_check";
begin
  process (clka) -- Using port a.
  begin
    if (clka'event and clka = '1') then
      if (write_ena = '1') then
        mem(conv_integer(addra)) <= dina;
          -- Using address bus a.
      end if;
      douta <= mem(conv_integer(addra));
    end if;
  end process;
  process (clkb) -- Using port b.
  begin
    if (clkb'event and clkb = '1') then
      if (write_enb = '1') then
        mem(conv_integer(addrb)) <= dinb;
          -- Using address bus b.
      end if;
      doutb <= mem(conv_integer(addrb));
    end if;
  end process;
end rtl;
```

Inferring ROM

To code ROM to be inferred, do the following:

- Define the ROM with a case statement or equivalent if statements.

- Assign constant values, all of the same width.
- Assign values for at least 16 addresses or half of the address space, whichever is greater. For example, if the address has 6 bits, the address space is 64 words, and at least 32 of them must be assigned values.
- To control how the ROM is implemented (with distributed or block ROM), consider adding the syn_romstyle attribute. See "syn_romstyle" on page 1339.

Figure 15: ROM Inferred with Case Statement in Verilog

```
module rom(data, addr);
  output [3:0] data;
  input [4:0] addr;
  always @(addr) begin
    case (addr)
       0 : data = 'h4;
       1 : data = 'h9;
       2 : data = 'h1;
       ...
      15 : data = 'h8;
      16 : data = 'h1;
      17 : data = 'h0;
      default : data = 'h0;
    endcase
  end
endmodule
```

Figure 16: ROM Inferred with If Statement in VHDL

```
entity rom is
port (addr : in std_logic_vector(4 downto 0);
  data : out std_logic_vector(3 downto 0) );
end rom;

architecture behave of rom is
begin
  process(addr)
  begin
    if    addr =  0 then data <= "0100";
    elsif addr =  1 then data <= "1001";
    elsif addr =  2 then data <= "0001";
    ...
    elsif addr = 15 then data <= "1000";
    elsif addr = 16 then data <= "0001";
    elsif addr = 17 then data <= "0000";
    else                 data <= "0000";
    end if;
  end process;
end behave;
```

Initializing Inferred RAM

The following examples show how to infer RAM.

Example1 To specify RAM initial contents, initialize the signal describing the memory array in Verilog code using initial statements as shown in the following coding example.

Figure 17: Initializing Block RAM Verilog Coding Example

```
module v_rams_20a (clk, we, addr, di, do);
input clk;
input we;
input [5:0] addr;
input [19:0] di;
output [19:0] do;
reg [19:0] ram [63:0];
reg [19:0] do;
initial begin
ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
ram[57] = 20'h00300; ram[56] = 20'h08602; ram[55] = 20'h02310;
ram[54] = 20'h0203B; ram[53] = 20'h08300; ram[52] = 20'h04002;
ram[51] = 20'h08201; ram[50] = 20'h00500; ram[49] = 20'h04001;
ram[48] = 20'h02500; ram[47] = 20'h00340; ram[46] = 20'h00241;
ram[45] = 20'h04002; ram[44] = 20'h08300; ram[43] = 20'h08201;
ram[42] = 20'h00500; ram[41] = 20'h08101; ram[40] = 20'h00602;
ram[39] = 20'h04003; ram[38] = 20'h0241E; ram[37] = 20'h00301;
ram[36] = 20'h00102; ram[35] = 20'h02122; ram[34] = 20'h02021;
ram[33] = 20'h00301; ram[32] = 20'h00102; ram[31] = 20'h02222;
ram[30] = 20'h04001; ram[29] = 20'h00342; ram[28] = 20'h0232B;
ram[27] = 20'h00900; ram[26] = 20'h00302; ram[25] = 20'h00102;
ram[24] = 20'h04002; ram[23] = 20'h00900; ram[22] = 20'h08201;
ram[21] = 20'h02023; ram[20] = 20'h00303; ram[19] = 20'h02433;
ram[18] = 20'h00301; ram[17] = 20'h04004; ram[16] = 20'h00301;
ram[15] = 20'h00102; ram[14] = 20'h02137; ram[13] = 20'h02036;
ram[12] = 20'h00301; ram[11] = 20'h00102; ram[10] = 20'h02237;
 ram[9] = 20'h04004;  ram[8] = 20'h00304;  ram[7] = 20'h04040;
 ram[6] = 20'h02500;  ram[5] = 20'h02500;  ram[4] = 20'h02500;
 ram[3] = 20'h0030D;  ram[2] = 20'h02341;  ram[1] = 20'h08201;
 ram[0] = 20'h0400D;
end
always @(posedge clk)
begin
if (we)
ram[addr] <= di;
do <= ram[addr];
end
endmodule
```

Example2 To initialize RAM from values contained in an external file, use a $readmemb or $readmemh system task in Verilog code.

Figure 18: Initializing Block RAM (External Data File) Verilog Coding Example

```
module ram_int1(input clk, we, input[31:0] in1, input
[3:0]addr, output[31:0]  out);
reg[31:0]  mem [15:0]/*synthesis syn_ramstyle= "block_ram"*/ ;
reg [31:0]  out1;
initial
begin
 $readmemh("data.dat", mem);
end
always @(posedge clk)
begin
if (we)
 mem[addr] <= in1;
else
 out1 <= mem[addr];
end
assign out = out1;
endmodule
```

Example 3 To specify RAM initial contents, initialize the RAM in the VHDL code with signal declarations or with variable declarations.

Figure 19: Initializing VHDL Rams with Signal Declarations

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity w_r2048x8 is
port (
 clk : in std_logic;
 adr : in std_logic_vector(10 downto 0);
 di : in std_logic_vector(7 downto 0);
 we : in std_logic;
 dout : out std_logic_vector(7 downto 0));
end;
architecture arch of w_r2048x8 is
-- Signal Declaration --
type MEM is array(0 to 2047) of std_logic_vector (7 downto 0);
signal memory : MEM := (
 "00000000"
,"01111000"
,"10110011"
,"01111000"
,"10011011"
,"11111111"
,"10011011"
,"01111000"
,"10110011"
,"01111000"
,"00000000"
,"10000111"
,"01001100"
,"10000111"
,"01100100"
,"00000000"
,"01100100"
,"10000111"
,"01001100"
,"10000111"
,"11111111"
,"01111000"
,"10110011"
,"01111000"
,"10011011"
,"11111111"
,others => (others => '0'));
begin
process(clk)
begin
 if rising_edge(clk) then
  if (we = '1') then
   memory(conv_integer(adr)) <= di;
  end if;
   dout <= memory(conv_integer(adr));
 end if;
end process;
end arch;
```

Example 4 To specify RAM initial contents, initialize the RAM in the VHDL code with readline and read task in VHDL code.

Figure 20: Initializing Block RAM (External Data File) VHDL Coding Example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
entity rams_20c is
port(clk : in std_logic;
we : in std_logic;
addr : in std_logic_vector(5 downto 0);
din : in std_logic_vector(31 downto 0);
dout : out std_logic_vector(31 downto 0));
end rams_20c;
architecture syn of rams_20c is
type RamType is array(0 to 63) of bit_vector(31 downto 0);
impure function InitRamFromFile (RamFileName : in string)
return RamType is
FILE RamFile : text is in RamFileName;
variable RamFileLine : line;
variable RAM : RamType;
begin
for I in RamType'range loop
readline (RamFile, RamFileLine);
read (RamFileLine, RAM(I));
end loop;
return RAM;
end function;
signal RAM : RamType := InitRamFromFile("rams_20c.data");
begin
process (clk)
begin
if clk'event and clk = '1' then
if we = '1' then
RAM(conv_integer(addr)) <= to_bitvector(din);
end if;
dout <= to_stdlogicvector(RAM(conv_integer(addr)));
end if;
end process;
end syn;
```

Creating Memory Initialization File

If the RAM initialization file is an external data file, there are two ways to point to the initialization file.

1. Assign the RAM initial file name in the VHDL or Verilog code and set the location in the “memory initial value file search path” in the LSE project file or the Diamond project Strategy setting. For example:

   ```
   Initial begin
    $readmemh("data.dat", mem);
   End
   ```

The “memory initial value file search path” is an absolute path, i.e. “C: /design/undesigned/source"

2. Assign the RAM initial file name in the VHDL or Verilog code and its relative path. For example:

```
Initial begin
 $readmemh("../source/data.dat", mem);
end
```

Inferring Lattice DSP Blocks Using Behavioral HDL

LSE uses the DSP feature efficiently. LSE packs multipliers, registers, adders, subtractors, and accumulators to DSP blocks. During the inference LSE checks all the design rule checks (DRCs). Based on the feasibility, LSE will pack these primitives to DSP blocks as best as possible.

The following examples include MULT, MULTADDSUB, MULTADDSUBSUM, and MULTACC.

MULT9X9

9X9 multiplier with/without output registers.

MULT18X18

18X18 multiplier with/without output registers.

MULT36X36

36X36 multiplier with/without output registers.

Following are Verilog and VHDL code examples for MULT.

Figure 21: MULT - Verilog without Register

```
`timescale 1 ns / 1 ns

module mult(a,b,c);

parameter A_WIDTH = 9;
parameter B_WIDTH = 9;

input  [(A_WIDTH - 1):0] a;
input  [(B_WIDTH - 1):0] b;
output  [(A_WIDTH + B_WIDTH - 1):0] c;

assign c = a*b;

endmodule
```

Figure 22: MULT - Verilog with Register

```
`timescale 1 ns / 1 ns

module mult_reg(clk,a,b,c,rst);

parameter A_WIDTH = 9;
parameter B_WIDTH = 9;

input rst;
input clk;
input  [(A_WIDTH - 1):0] a;
input  [(B_WIDTH - 1):0] b;
output  [(A_WIDTH + B_WIDTH - 1):0] c;
reg [(A_WIDTH + B_WIDTH - 1):0] c;



always @ (posedge clk)
  begin
    if (rst)
      c <= 18'b0;
    else
      c <= a*b;
  end



endmodule
```

Figure 23: MULT - VHDL without Register

```
library ieee;
use ieee.std_logic_1164.all;
--use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;



entity Mult is
generic (data_width_a: integer := 9;
         data_width_b: integer := 9
        );
 port  (  a    : in   std_logic_vector(data_width_a-1 downto
0);
          b    : in   std_logic_vector(data_width_b-1 downto
0);
          q    : out
std_logic_vector(data_width_a+data_width_b-1 downto 0)
        );
end Mult;


architecture rtl of Mult is
  --attribute syn_multstyle : string ;
  --attribute syn_multstyle of q : signal is "dsp" ;
begin

   q <= a * b;

end rtl;
```

Figure 24: MULT - VHDL with Register

```
library ieee;
use ieee.std_logic_1164.all;
--use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;



entity Mult is
generic (data_width_a: integer := 9;
         data_width_b: integer := 9
        );
 port  (  a    : in   std_logic_vector(data_width_a-1 downto
0);
          b    : in   std_logic_vector(data_width_b-1 downto
0);
      clk  : in std_logic;
      rst  : in std_logic;
          q    : out
std_logic_vector(data_width_a+data_width_b-1 downto 0)
        );
end Mult;


architecture rtl of Mult is
   --attribute syn_multstyle : string ;
  -- attribute syn_multstyle of q : signal is "dsp" ;
  signal q_s,temp
:std_logic_vector(data_width_a+data_width_b-1 downto 0);
begin
  q_s <= a * b;
  process(clk,rst)
  begin
    if rst = '1' then
       q <= (others=>'0');
    elsif clk'event and clk = '1' then
      q <= q_s;
    end if;


  end process;

end rtl;
```

MULTADDSUB

Two multipliers driving adder/subtractor.

Following are Verilog and VHDL code examples for MULTADDSUB.

Figure 25: MULTADDSUB - Verilog without Register

```
`timescale 1 ns / 1 ns

module multaddsub(a,b,c,q);

parameter A_WIDTH = 9;
parameter B_WIDTH = 9;

input  [(A_WIDTH - 1):0] a;
input  [(B_WIDTH - 1):0] b;
input  [(A_WIDTH + B_WIDTH - 1):0] c;
output  [(A_WIDTH + B_WIDTH - 1):0] q;


assign q = a*b+c;


endmodule
```

Figure 26: MULTADDSUB - Verilog with Register

```
`timescale 1 ns / 1 ns

module Multaddsub_reg(clk,a,b,c,q,arst);

parameter A_WIDTH = 9;
parameter B_WIDTH = 9;

input arst;
input clk;
input  [(A_WIDTH - 1):0] a;
input  [(B_WIDTH - 1):0] b;
input  [(A_WIDTH + B_WIDTH - 1):0] c;
output  [(A_WIDTH + B_WIDTH - 1):0] q;


reg [(A_WIDTH + B_WIDTH - 1):0] reg_tmp_c;
assign q = reg_tmp_c;

always @(posedge clk,posedge arst)
begin
    if(arst)
    begin
        reg_tmp_c     <= 0;
    end
    else
    begin
        reg_tmp_c     <= a*b+c;
    end
```

Figure 27: MULTADDSUB - VHDL without Register

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

entity Multaddsub is
generic (data_width_a: integer := 9;
         data_width_b: integer := 9;
         data_width_c: integer := 18;
         product_width: integer :=18);
 port  (  a    : in   std_logic_vector(data_width_a-1 downto
0);
          b    : in   std_logic_vector(data_width_b-1 downto
0);
          c    : in   std_logic_vector(data_width_c-1 downto
0);
          q    : out  std_logic_vector(product_width-1 downto
0)  );
end Multaddsub;


architecture rtl of Multaddsub is

  --attribute syn_multstyle : string ;
  --attribute syn_multstyle of q : signal is "dsp" ;

begin

   q <=  a*b + c;

end rtl;
```

Figure 28: MULTADDSUB - VHDL with Register

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

entity Multaddsub_reg is
generic (data_width_a: integer := 9;
         data_width_b: integer := 9;
         data_width_c: integer := 18;
         product_width: integer :=18);
 port  (  a    : in   std_logic_vector(data_width_a-1 downto
0);
          b    : in   std_logic_vector(data_width_b-1 downto
0);
          c    : in   std_logic_vector(data_width_c-1 downto
0);
      clk  : in   std_logic;
      arst : in   std_logic;
          q    : out  std_logic_vector(product_width-1 downto
0)  );
end Multaddsub_reg;


architecture rtl of Multaddsub_reg is

  --attribute syn_multstyle : string ;
  --attribute syn_multstyle of q : signal is "dsp" ;

begin

  process(clk,arst)
  begin
    if arst = '1' then
      q <= (others => '0');
    elsif clk'event and clk = '1' then
      q <=  a*b + c;
    end if;
  end process;
end rtl;
```

MULTADDSUBSUM

Four multipliers driving two adders/subtractors driving one adder.

Following are Verilog and VHDL code examples for MULTADDSUBSUM.

Figure 29: MULTADDSUBSUM - Verilog without Register

```
module Multaddsubsum(out, ina, inb, inc, ind, ine, inf, ing,
inh);
  parameter Data_width_in = 9;
  parameter Data_width_out = 19;
  output [Data_width_out-1:0] out;
  input [Data_width_in-1:0] ina, inb, inc, ind, ine, inf,
ing, inh;
  wire[Data_width_out-2:0] prod4, prod1, prod2, prod3;
  wire[Data_width_out-2:0] wireSum;
  wire[Data_width_out-2:0] wireSum1;



  assign prod1 = ina*inb;
  assign prod2 = inc*ind;
  assign prod3 = ine*inf;
  assign prod4 =  ing*inh;
  assign wireSum = prod1 + prod2;
  assign wireSum1 = prod3 + prod4;
  assign out = wireSum+ wireSum1;




endmodule
```

Figure 30: MULTADDSUBSUM - Verilog with Register

```
module Multaddsubsum_reg(out, ina, inb, inc, ind, ine, inf,
ing, inh,clk4, clk1, clk2, clk3);

  parameter Data_width_in = 9;
  parameter Data_width_out = 19;
  output [Data_width_out-1:0] out;
  input [Data_width_in-1:0] ina, inb, inc, ind, ine, inf,
ing, inh;
  inputclk1, clk2, clk3, clk4;
  wire[Data_width_out-2:0] prod4, prod1, prod2, prod3;
  wire[Data_width_out-2:0] wireSum;
  wire[Data_width_out-2:0] wireSum1;
  reg    [Data_width_in-1:0] inaReg , indReg, ineReg, inhReg;
  //reg    [17:0] outReg;


  assign prod1 = inaReg*inb;
  assign prod2 = inc*indReg;
  assign prod3 = ineReg*inf;
  assign prod4 =  ing*inhReg;
  assign wireSum = prod1 + prod2;
  assign wireSum1 = prod3 + prod4;
  assign out = wireSum+ wireSum1;


always @(posedge clk1 )
begin

  inaReg = ina;
end

always @(posedge clk2 )
begin

  indReg = ind;
end

always @(posedge clk3 )
begin

  ineReg = ine;
end

always @(posedge clk4 )
begin

  inhReg = inh;
end
endmodule
```

Figure 31: MULTADDSUBSUM - VHDL without Register

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

entity Multaddsubsum is
  generic (data_width_a   : integer := 9;
       data_width_b   : integer := 9;
       data_width_c   : integer := 9;
       data_width_d   : integer := 9
       );
   port  (  a    : in   std_logic_vector(data_width_a-1
downto 0);
        b    : in   std_logic_vector(data_width_b-1 downto
0);
        c    : in   std_logic_vector(data_width_c-1 downto
0);
        d    : in   std_logic_vector(data_width_d-1 downto
0);
        e    : in   std_logic_vector(data_width_a-1 downto
0);
        f    : in   std_logic_vector(data_width_b-1 downto
0);
        g    : in   std_logic_vector(data_width_c-1 downto
0);
        h    : in   std_logic_vector(data_width_d-1 downto
0);
       -- sum  : out
std_logic_vector(data_width_a+data_width_c downto 0);
        q    : out
std_logic_vector(data_width_a+data_width_c-1 downto 0));
end Multaddsubsum;

architecture rtl of Multaddsubsum is

  --attribute syn_multstyle : string ;
  --attribute syn_multstyle of q : signal is "dsp" ;
  signal sum_s1,sum_s2   :
std_logic_vector(data_width_a+data_width_c-1 downto 0);

begin

  sum_s1 <= a*b + c*d;
  sum_s2 <= e*f + g*h;
  q <= sum_s1 + sum_s2;

end rtl;
```

Figure 32: MULTADDSUBSUM - VHDL with Register

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

entity Multaddsubsum is
  generic (data_width_a   : integer := 9;
       data_width_b   : integer := 9;
       data_width_c   : integer := 9;
       data_width_d   : integer := 9
       );
   port  (  a    : in   std_logic_vector(data_width_a-1
downto 0);
        b    : in   std_logic_vector(data_width_b-1 downto
0);
        c    : in   std_logic_vector(data_width_c-1 downto
0);
        d    : in   std_logic_vector(data_width_d-1 downto
0);
        e    : in   std_logic_vector(data_width_a-1 downto
0);
        f    : in   std_logic_vector(data_width_b-1 downto
0);
        g    : in   std_logic_vector(data_width_c-1 downto
0);
        h    : in   std_logic_vector(data_width_d-1 downto
0);
        clk  : in   std_logic;
        arst  : in   std_logic;
       -- sum  : out
std_logic_vector(data_width_a+data_width_c downto 0);
        q    : out
std_logic_vector(data_width_a+data_width_c-1 downto 0));
end Multaddsubsum;

architecture rtl of Multaddsubsum is

  --attribute syn_multstyle : string ;
  --attribute syn_multstyle of q : signal is "dsp" ;
  signal sum_s1,sum_s2   :
std_logic_vector(data_width_a+data_width_c-1 downto 0);

begin

  sum_s1 <= a*b + c*d;
  sum_s2 <= e*f + g*h;
  --q <= sum_s1 + sum_s2;

  process(clk,arst)
  begin
    if arst = '1' then
      q <= (others=>'0');
    elsif clk'event and clk = '1' then
      q <= sum_s1 + sum_s2;
    end if;
  end process;

end rtl;
```

MULTACC

One or two multipliers driving accumulator.

Following are Verilog and VHDL code examples for MULTACC.

Figure 33: MULTACC - Verilog without Register

```
`timescale 1 ns / 1 ns

module multacc(a,b,q);

parameter A_WIDTH = 9;
parameter B_WIDTH = 9;

input unsigned [(A_WIDTH - 1):0] a;
input unsigned [(B_WIDTH - 1):0] b;
output unsigned [(A_WIDTH + B_WIDTH - 1):0] q;



 assign q = a*b+q;

endmodule
```

Figure 34: MULTACC - Verilog with Register

```
`timescale 1 ns / 1 ns

module multacc_unsign_8_8(clk,a,b,q,set);

parameter A_WIDTH = 9;
parameter B_WIDTH = 9;

input set;
input clk;
input unsigned [(A_WIDTH - 1):0] a;
input unsigned [(B_WIDTH - 1):0] b;
output unsigned [(A_WIDTH + B_WIDTH - 1):0] q;


reg [(A_WIDTH + B_WIDTH - 1):0] reg_tmp_c;
assign q = reg_tmp_c;

always @(posedge clk)
begin
    if(set)
    begin
        reg_tmp_c     <= 0;
    end
    else
    begin
        reg_tmp_c     <= a*b+q;
    end
end
```

Figure 35: MULTACC - VHDL without Register

```
library ieee;
use ieee.std_logic_1164.all;
--use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;



entity Multacc is
  generic (data_width_a: integer := 9;
       data_width_b: integer := 9;
       product_width: integer :=19);
   port  (
      a    : in   std_logic_vector(data_width_a-1 downto 0);
      b    : in   std_logic_vector(data_width_b-1 downto 0);
      q     : out  std_logic_vector(product_width-1 downto 0)
      );
end Multacc;


architecture rtl of Multacc is

  --attribute syn_multstyle : string ;
  --attribute syn_multstyle of q : signal is "dsp" ;
  signal q_s : std_logic_vector(product_width-1 downto
0):=(others=>'0');
  signal q_s1 : std_logic_vector(data_width_a+data_width_b-1
downto 0):=(others=>'0');
begin

  q_s1 <= a*b;
  q <= q_s;
  q_s <= ('0'&q_s1) + q_s;


end rtl;
```

Figure 36: MULTACC - VHDL with Register

```
library ieee;
use ieee.std_logic_1164.all;
--use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;



entity Multacc_reg is
  generic (data_width_a: integer := 9;
       data_width_b: integer := 9;
       product_width: integer :=19);
   port  (  clk,rst : in  std_logic;
        a    : in   std_logic_vector(data_width_a-1 downto
0);
        b    : in   std_logic_vector(data_width_b-1 downto
0);
      q    : out  std_logic_vector(product_width-1 downto 0)
);
end Multacc_reg;


architecture rtl of Multacc_reg is

  --attribute syn_multstyle : string ;
  --attribute syn_multstyle of q : signal is "dsp" ;
  signal q_s : std_logic_vector(product_width-1 downto
0):=(others=>'0');
  signal q_s1 : std_logic_vector(data_width_a+data_width_b-1
downto 0):=(others=>'0');
begin

   q_s1 <= a*b;
   q <= q_s;
   process(clk,rst)
   begin
   if rst = '0' then
    q_s <= (others => '0');
   elsif clk'event and clk = '1' then
       q_s <= ('0'&q_s1) + q_s;
   end if;
   end process;

end rtl;
```

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. (For other synthesis tools, see those tools' documentation.) Use these methods along with other, generic coding methods to optimize your design. Also, consider using the predefined Area or Timing strategies.

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 strategy settings. Choose **Project > Active Strategy > LSE Settings**. In the Strategy dialog box, set the following options, which are found in Synthesize Design > LSE. See the following text for explanations and more details.

Table 3: LSE Strategy Settings for Area and Speed

Option	Area	Speed
FSM Encoding Style	Binary or Gray	One-Hot
Max Fanout Limit	<*maximum*>	<*minimum*>
Optimization Goal	Area	Timing
Remove Duplicate Registers	True	False
Resource Sharing	True	False
Target Frequency	<*minimum*>	
Use IO Registers	Auto or True	Auto or False

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.

Max Fanout Limit A larger fanout limit means less duplicated logic and fewer buffers. A lower fanout limit may reduce delays. The default is 1000, which is essentially unlimited fanout. Select a balanced fanout constraint. A large constraint creates nets with large fanouts, and a low fanout constraint results in replicated logic. You can use this in conjunction with the syn_replicate attribute. See “syn_replicate” on page 1337. To minimize area, don’t lower this value any more than needed to meet other requirements. To minimize speed, try much lower values, such as 50.

You can change the fanout limit for portions of the design by using the syn_maxfan attribute. See “syn_maxfan” on page 1325. Set Max Fanout Limit to meet your most demanding requirement. Then add syn_maxfan to help other requirements.

Optimization Goal If set to Area, LSE will choose smaller design forms over faster whenever possible.

If set to Timing, LSE will choose faster design forms over smaller whenever possible. If a create_clock constraint is available in an .ldc file, LSE ignores the Target Frequency setting and uses the value from the create_clock constraint instead.

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.

Target 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. If Optimization Goal is set to Timing and a create_clock constraint is available in an .ldc file, LSE will use the value from the create_clock constraint instead.

Use IO Registers If set to True, LSE will pack all input and output registers into I/O pad cells. Register packing reduces area but adds delays.

Auto, the default setting, enables this register packing if Optimization Goal is set to Area. If Optimization Goal is Timing or Balanced, Auto disables register packing.

You can also control packing on individual registers. See “syn_useioff” on page 1352. Set Use IO Registers to meet your most demanding requirement. Then add syn_useioff to help other requirements.

Specifying Optimization Options

This section describes options provided by LSE to optimize your design.

Preserving Objects from Optimization

Nets can be removed or collapsed during optimization. Attributes can be used to retain a net for synthesis implementations such as simulation. Duplicate registers are removed in synthesis. Use attributes to preserve logic for simulation or analysis.

Setting Fanout Limits

You can use the Max Fanout Limit strategy to specify the maximum fanout setting. LSE will make sure that any net in the design is not exceeding this limit. Default is 1000 fanouts. This option is equivalent to the “-max_fanout” option in the SYNTHESIS command. See “Max Fanout Limit” on page 12.

Sharing Resources

You can use the Resource Setting strategy 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.See “Resource Sharing” on page 16.

Inserting I/Os

LSE uses I/O insertion and GSR to optimize designs. For more information on this strategy, see “Use IO Insertion” on page 17

Optimizing State Machines

You can use the FSM Encoding Style strategy to optimize state machines. Valid options are auto, one-hot, gray, and binary. The default value is auto, meaning that the tool looks for the best implementation. See “FSM Encoding Style” on page 11

Working with Gated Clocks

The Fix Gated Clocks strategy can change standard gated clocks to forms more effective for FPGAs. See “Fix Gated Clocks” on page 10

Analyzing the Synthesis Report

Lattice Diamond generates log files for all project activities. The log files contain processing information, as well as error and warning messages. If you run processes, reports are generated.

Viewing Logs and Reports

A log file is displayed in the Output frame as a process is running. A scroll bar can be used to scroll up and down in the information.

Errors are displayed in red. Warnings are displayed in orange. There are also information messages. These messages are also displayed in the Warning, Error, and Info views. These views may not automatically be visible in your Diamond main window. To turn on the views, choose **View > Show Views >** <*view*>. A check mark indicates the view is displayed.

Viewing Reports The Reports view displays reports for the major processes.

There are two panes in the Reports view. The left pane lists the Design Summary information including the report types. The reports in detail are displayed in the right pane.

Type of Report	Description
Project Summary	Lists the summary information of the project including module name, synthesis tool chosen, implementation name, strategy name, target device, device family, device type, package type, performance grade, operating conditions, logic preference file, software product version, project file name, and location.
Process Reports	Lists the synthesis, map, place and route, signal/pad, and bitstream reports in HTML format.
Analysis Reports	Lists the trace and timing reports.
Tool Reports	Lists the I/O SSO analysis, hierarchy parsing, PIO DRC, and ECO Editor reports. Also has a log of Tcl commands used in recent sessions.
Messages	Lists the implementation messages and user defined filters for the messages.

In the Design Summary pane, there is the report icon  . If a report has been generated, the icon appears as  . If the report is not the most recent version, the icon appears as  . To view the contents of the entire report, click on the report to be viewed. The entire report is then displayed in the right pane of the Reports view. Use the scroll bar to navigate through the report. Some of the reports are divided into sections (for example, Map, Place & Route, and Signal/Pad). Expand the report listing to display the sections in a list. Choose the desired section. The whole report will be displayed with the selected section displayed at the top of the right pane of the Reports view.

You can navigate the reports quickly by using the Find function (right-click in the right pane of the Reports view and choose **Find in Text**).

Other Reports The Synthesize Design stage produces reports that do not appear in the Reports view. You can find these reports in the implementation folder. In the File List view, right-click the implementation name and choose **Open Containing Folder**. A window will open showing the contents of the folder. All of these reports can be read with a text editor.

One of the reports is a detailed description of the device resources that will be used by the design. This report is much more detailed than the synthesis report in the Reports view. The report includes the resources used by each module of the design. Similar information can also be found in the Hierarchy view. For Synplify Pro, look for <*top_module*>.areasrr; for Lattice Synthesis Engine, look for <*top_module*>.arearep.

Cross-Probing from Reports to Schematics

While studying one of the timing or trace reports you might want to see where a module or port is in the design. You can cross-probe, or jump, from the LSE timing report and from the place & route trace report to a schematic view of the design.

To cross-probe from the LSE timing report to a schematic view:

1. After running synthesis with LSE, open the Reports view.
2. In the Design Summary column, click **LSE Timing Report**. It's under Analysis Reports.
3. Select text that has the name of one or more module instances or ports of interest.
4. Right-click and choose **Filter in Netlist Analyzer**.

   Netlist Analyzer opens with the technology netlist view of the selected objects. For more information, see "Analyzing Using Netlist Analyzer" on page 112.

To cross-probe from the place & route trace report to a schematic view:

1. After running place & route of the design, choose **Tools > Synplify Pro for Lattice**.
2. In Synplify Pro, click the **Implementation Directory** tab.
3. Find the **.twr** file and double-click it.

   A text editor opens in Synplify Pro with the report.
4. Find the name of an instance or port of interest and select it.
5. Right-click in the selected name and choose one of the following:
   - **Filter in Analyst** to see the item by itself
   - **Select in Analyst** to find the item in the full schematic
6. If a suitable Analyst view is not open, a dialog box asks if you want to open one. Click **Yes**.
7. Go to the Analyst schematic view to see the item.

Navigating Messages/Warnings

If an error or a warning results from the specific line in an HDL source file, you can easily go to that line to edit the source file.

To navigate errors and warnings:

- In the Reports, Output, Error, or Warning view, double-click the line describing the error or warning.

- In the Reports, Output, Error, or Warning view, right-click the message and choose **Locate in > Text Editor**. If the command is dimmed, there is no link to a source file. Depending on the message, more than one tool may be available to view the source. Choose the one you want to use.

Finding Results Your default text editor opens with the appropriate HDL source file at the line number specified in the error or warning message. You can then modify the file to debug your design.

After you load a design in Diamond, you can find the information you need via the following ways.

- In the active Reports view, choose **Edit > Find**. You can type the desired text into the Find field at the bottom-left of the Reports view window. The first occurrence of the desired text will be found and highlighted in the right side of the window for you. Click **Next** or **Previous** to find more. And check the **Case Sensitive** option if needed for the search. While typing in the text, the Find field will be automatically colored if no occurrence of the text is found.
- In the active Source Editor, after choosing **Edit > Find**, you will get the Find and Replace dialog box. You can enter the text you want to search in the Find What field and start a search. Use the **Find Next** command to find more. If you want to replace the current find, you can use the Replace tab of the dialog box. Check **Match Case**, **Match Whole Word**, **Search Up**, **Regular Expression** options as needed. Use the **Replace** or **Replace All** command to replace the text found.
- If you want to find information without loading the files, you can choose **Edit > Find in Files** from Diamond main window. In the pop-up Find In Files dialog box, type the text you want to find, specify the search path and search filters, and check the desired options: **Search subdir**, **Include hidden files**, **Match case**, **Match whole word**, and **Regular expressions**. Press **Find**. The results will be displayed in the Find Results frame. Double-click any of the findings from the Find Results frame to open the associated source file in the associated editor. For example, if the finding is in a log file, the log file will be opened in the Reports view with the first finding appears on the first line.
- You can search the Output log by clicking in the Output view (in the text, not on the tab) and then pressing **Ctrl**-**F**. This opens a basic text search dialog box at the top of the Output view. You can type the text in the Find text field and start a search.

Find Results View The Find Results view may not be displayed automatically in your Diamond main window. To turn on the Find Results view, select **View > Show Views > Find Results**. A check mark indicates the frame is displayed.

The Find Results view can be detached from the main window by clicking the detaching icon on the upper-right corner of the view. After detaching, you can double-click on the title bar of the view to get it back to the main window.

For more information about messages, in the Diamond software online help, refer to **User Guides > Managing Projects > Viewing Logs and Reports**.

Analyzing Using Netlist Analyzer

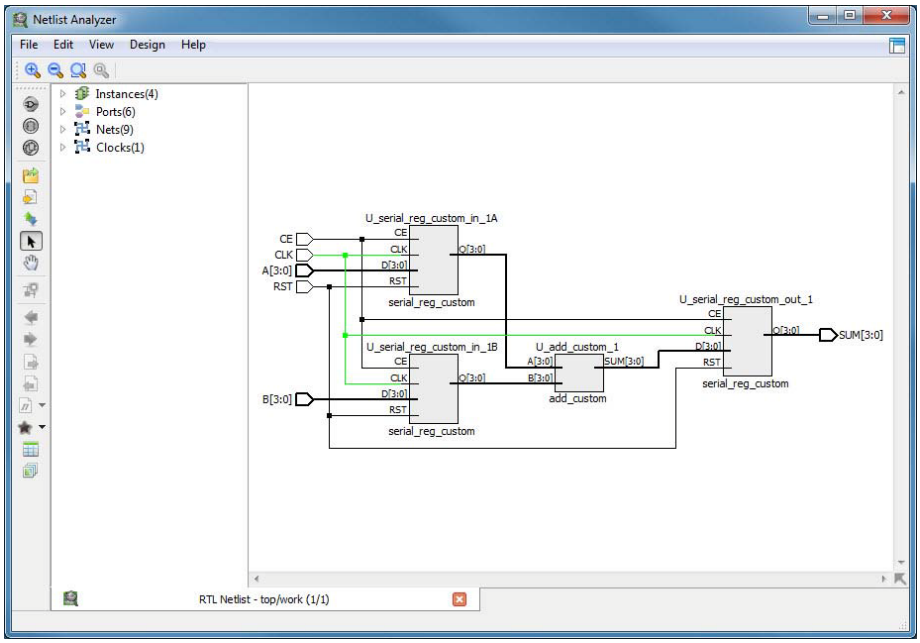
Netlist Analyzer works with LSE to produce schematic views of your design while it is being implemented. Use the schematic views to better understand the hierarchy of the design and how the design is being implemented.

To start Netlist Analyzer:

1. Synthesize the design with LSE.
2. Choose **Tools > Netlist Analyzer**.

The Netlist Analyzer window opens with the RTL netlist showing. as shown in Figure 37.

Figure 37: Netlist Analyzer



The Netlist Analyzer clock tree, shown in Figure 38, is displayed along with the design tree, which lists all the clock nets along with their drivers. The clock tree feature helps locate the clock signal and analyze the clock network of the design.

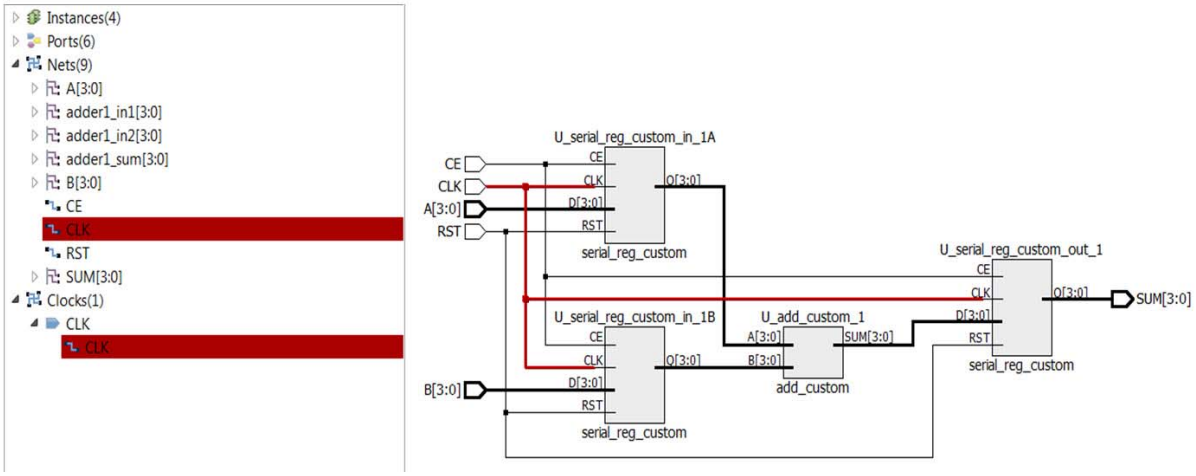
The clock tree has three levels:

- The first level is a node named “Clocks” with a number denoting the number of the clock signals in the design.
- The second level includes all the drivers of the clock signals,
- The third level are the child signals associated with the clock net

The objects in the clock tree show hierarchical names as their name. Selection is synchronized among the Netlist Analyzer design tree, schematic view and the clock tree, as shown in Figure 39.

Figure 38: Clock Tree View of Netlist Analyzer

Figure 39: Synchronization among Netlist Analyzer Views



About Netlist Analyzer Views The Netlist Analyzer window has four parts:

- Tool bar provides buttons for various functions.
- Netlist browser provides nested lists of module instances, ports, nets, and clocks.
- Schematic view shows a schematic of the design. Depending on the size of the design, the schematic may be made of multiple sheets.
- Mini-map, which is a miniature view of the sheet, helps you pan and zoom in the schematic view.

Netlist Analyzer can have multiple schematics open. The open schematics are shown on tabs along the bottom of the window.

Bold lines are buses. Green lines are clock signals.

For more information about Netlist Analyzer, in the Diamond software online help, refer to **User Guides > Managing Projects > Analyzing a Design > About Netlist Analyzer**.

Simulating the Synthesis Output

LSE generates a post-synthesis netlist file in Verilog format. The file is generated after running the Verilog Simulation File process in Diamond. The file name is <*design*>_prim.v. This file is a structural netlist of the synthesized design, and differs from the original RTL used as input for synthesis. The file is also a post-synthesis source simulation file for functional simulation of primitive gate-level logic.

Typically, this netlist is used for gate-level simulation, to verify synthesis results. Some designers prefer to simulate before and after synthesis, and also after place-and-route. This approach helps to isolate the stage of the design process where a problem occurred.

The Verilog output file is for functional simulation only. When you input stimulus into a simulator for functional simulation, use a cycle time for the stimulus of 1,000 time ticks.

Simulation flow For post-synthesis simulation, the designer needs a Verilog simulation library, a <*design*>_prim.v file, and a testbench file.

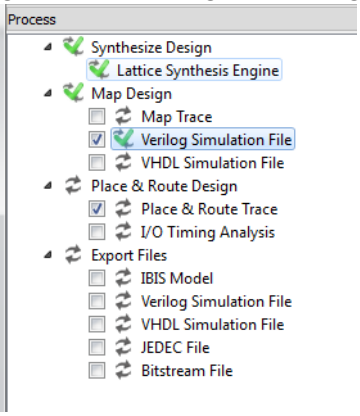
Method 1 Using Diamond Simulation Wizard.

Note

For more information on Simulation Wizard, refer to the *Lattice Diamond User Guide* or the Diamond online help topic **User Guides > Simulating the Design > Simulation in Diamond > Creating a New Simulation Project in Diamond**,

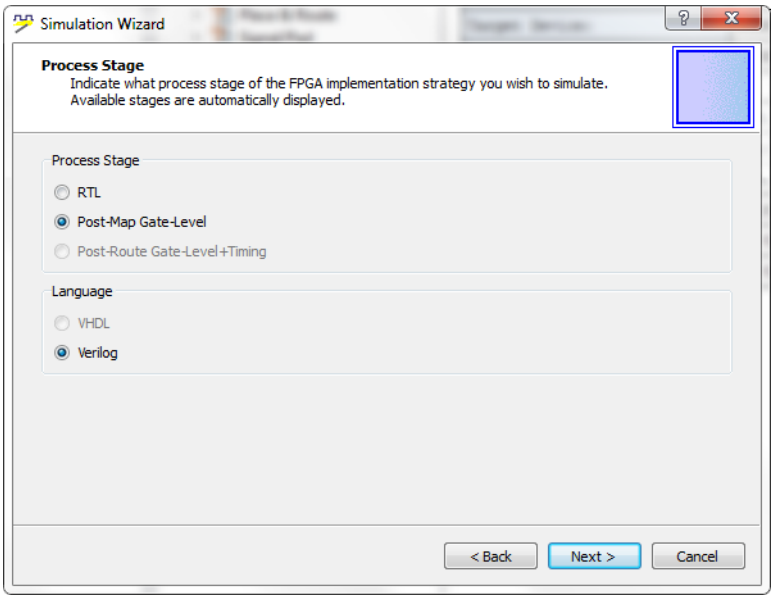
1. Run the Verilog Simulation File process, as shown in Figure 40.

Figure 40: Running the Verilog Simulation File Process



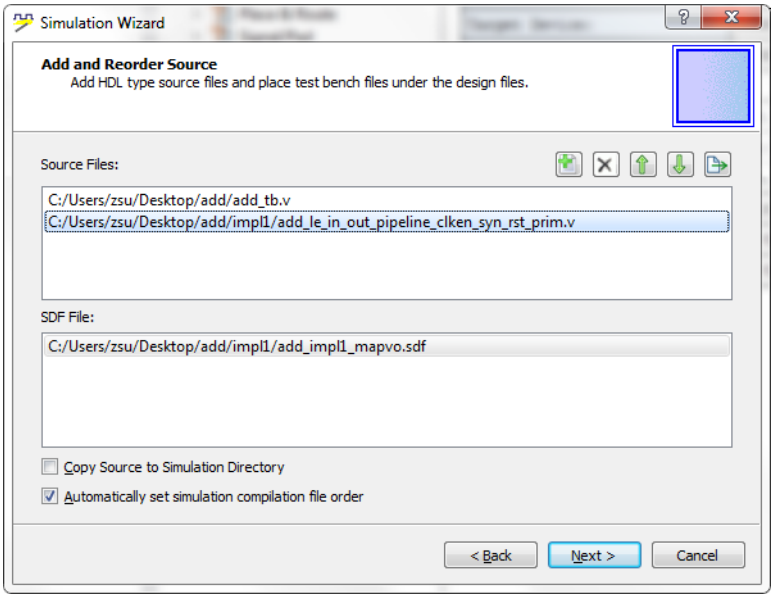
2. From your project directory, remove or rename the _mapvo.vo (post-map simulation file).
3. Run the Choose **Tools > Simulation Wizard** or click the Simulation Wizard button on the toolbar.
4. In the Simulation Wizard, specify a **Project Name**, **Project Location**, and **Simulator,** and then click **Next**.
5. In the Simulation Wizard, choose Post-Map Gate-Level, and choose Verilog as the language, as shown in Figure 41. Click **Next**.

Figure 41: Simulation Wizard Process Stage



6. In the Simulation Wizard, add the _prim.v and the testbench into Simulation Wizard, as shown in Figure 42
7. Continue with the simulation.

Figure 42: Simulation Wizard Add and Reorder Source



Method 2 Using a third-party simulation tool such as ModelSim:

1. Compile the simulation library with running cmpl_lib.tcl from the Command Line. For more information, in the Diamond software online help, refer to **Reference Guides > Command Line Reference Guide > Command Line Tool Usage > Running cmpl_lib.tcl from the Command Line**.
2. Compile the Verilog simulation library.
3. Add the compiled library and _prim.v, testbench file into third-part simulation tool and run the simulation.

Designing with Modules/IP

Modules are functional bits of design that can be re-used wherever that function is needed. Creating such modules with hardware design languages is common practice. To help your design along, Lattice Semiconductor provides a variety of modules for common functions. They are optimized for Lattice device architectures and can be customized. Use these modules to speed your design work and to get the most effective results.

Lattice Semiconductor's modules come in a variety of forms:

- IPexpress provides a variety of functions ranging from the most basic, such as arithmetic and memory, to much more complex functions. With IPexpress these modules can be extensively customized. They can be created as part of a specific project or as a library for multiple projects. For more information, in the Diamond software online help, refer to **User Guides > Entering the Design > Designing with Modules > Creating IPexpress Modules and IP**.

However, many of these modules can also be used with PMI (see next item). To decide which method to use, in the Diamond software online help, refer to **User Guides > Entering the Design > Designing with Modules > Use PMI or IPexpress**?

- PMI (Parameterized Module Instantiation) is an alternate way to use some of the modules that come with IPexpress. With PMI, instead of using IPexpress, you directly instantiate a module into your HDL and customize it by setting parameters in the HDL. You may find this easier than using IPexpress if your design requires many variations of the same module. To decide which method to use, in the Diamond software online help, refer to **User Guides > Entering the Design > Designing with Modules > Use PMI or IPexpress**?
- Clarity Designer provides modules similar to those from IPexpress but for ECP5. As with IPexpress, with Clarity Designer you can customize these modules. Clarity Designer also helps you connect these modules to each other and place the PCS and DDR modules in the device's architecture. For more information, in the Diamond software online help, refer to **User Guides > Entering the Design > Creating Clarity Designer Modules**.
- LatticeMico32 microprocessors and LatticeMico8 microcontrollers are exceptions in that they are not customized with IPexpress. LatticeMico32 and LatticeMico8 have their own development environment. To design with LatticeMico System, in the Diamond software online help, refer to **User Guides > Entering the Design > Designing with LatticeMico Platforms**.
- Reference designs provide you with a starting point on creating your own modules. Lattice Reference Designs are available in Verilog and VHDL, and can be downloaded from the Lattice Web site: www.latticesemi.com/ip.
- Lattice library primitives are very basic functions, such as logic gates and flip-flops. They can be directly instantiated as HDL into designs. But this is an advanced technique and should usually be avoided. For more information, see "Designing with Lattice Library Primitives" on page 120.

Of course you can also create your own modules and that is fully supported too. In fact, Diamond supports creating your own black-box modules. See "Creating Your Own Black Box Modules" on page 118.

Using IPexpress Modules

Below are the basic steps of using IPexpress modules and IP. For details of performing these steps, see the following topics.

1. Start running IPexpress. It can be started from Diamond's Tools menu after you open your design project. If you want to create a library of configured modules or IP, IPexpress can be opened as a stand-alone tool to create a library of modules.
2. If you want to use a Lattice IP that's not visible, it must be downloaded and installed first. This can be done from IPexpress.

3. Customize the module/IP. These modules and IP can be extensively customized for your design. The options may range from setting the width of a data bus to selecting features in a communications protocol. At a minimum you need to specify the design language to use for the output.
4. Generate the module/IP and bring its .ipx file into your project. Prior to generating the module/IP, select the option “Import IPX to Diamond Project.” This will then automatically bring the .ipx file into your project after the generation step completes. If you do not select this option, then after generation, add the .ipx file to your project as you would with any other source file (such as a Verilog or VHDL file). If using IPexpress standalone, there is no project to automatically add the .ipx file.
5. Instantiate the module/IP into the project’s design. An HDL instance template is generated during the generation step to simplify this step.
6. IPexpress modules and IP can be further modified or updated later. After the .ipx file has been added to the Diamond project, it is visible in the project’s file list. Double-clicking the .ipx file brings up the module/IP’s configuration dialog box where changes can be made and the generation process repeated.

Using Clarity Modules

Clarity Designer is a tool within the Lattice Diamond software environment that addresses the need to be able to generate and plan multiple blocks together. Clarity Designer is used for configuration of blocks, building the connections between blocks, and planning the resources used by the PCS and DDR blocks in the design. For device families supported by Clarity Designer, IPexpress functionality is accomplished along with functionality for building and planning. The IPexpress tool is disabled when using a device family supported by Clarity Designer. Device families that are not supported yet by Clarity Designer still require the use of IPexpress. Clarity Designer is currently only available for the ECP5 device family. A comparison chart between IPexpress and Clarity Designer features is shown in Table 4.

Creating Your Own Black Box Modules

In some cases, you may not want to distribute HDL source code because of the risk of changes or of exposing proprietary information. So, Lattice Semiconductor offers a compiled Native Generic Object (NGO) netlist format as an alternative to HDL.

Advantages and Disadvantages An NGO netlist has the following advantages over HDL source code:

- Hides details of internal logic
- Easy to distribute
- Optimized to meet timing or area requirements
- Optional grouping and floorplan constraints

Table 4: IPexpress versus Clarity Designer

		IPExpress	Clarity Designer
Configuration / Generation			
	Modules	Yes	Yes
	IP	Yes	Yes
	Download IP	Yes	Yes
Building			
	Rule checking	No	Yes
	Generate Connectivity	No	Yes
	Connection Assistance	No	Yes
	Design Reuse	No	Yes
Planning (PCS & DDR)			
	Pre-Synthesis	No	Yes
	Placement Assistance	No	Yes
	Rule checking	No	Yes
	Graphical usage	No	Yes

On the other hand, an NGO netlist:

- May not be portable across all device families
- Cannot be parameterized

Overview of the Black Box Process The following are the basic steps for creating and using a black box module. For details about performing these steps, see the topics listed under “See Also.”

1. Create an NGO netlist.

   Start with a design project just for the module and add some attributes. Then run the synthesis and Translate Design processes.

2. Create support files.

   In addition to the NGO netlist, users of the black box module will need additional information such as declaration and instantiation templates, a data sheet, a simulation model, and timing attributes.

Note

NGO blocks will be given a unique name which is constructed by appending the parameter value to the name of the NGO module as in the following example.

For the following module:

```
module my_add_sub (DataA, DataB, Add_Sub, Result);
parameter bit_width = 6;
...
endmodule
```

After generating the NGO file, the ngo must be renamed with the parameter value appended to the module name as shown below:

“my_add_sub_6.ngo”

3. Instantiate the module.

   Following the instructions from the module's data sheet, copy the declaration and instantiation templates into your design project.

Designing with Lattice Library Primitives

Any Lattice library primitive described in the FPGA Libraries Reference Guide can be instantiated as a Verilog module or VHDL component in your RTL design. This sort of "gate-level" design can be error-prone and should be limited to a small number of primitives if attempted at all. In general, Lattice recommends you rely on IPexpress to generate modules that are built with Lattice library primitives.

To minimize the amount of code overhead required to design with a library primitive, Lattice provides a Verilog and VHDL synthesis header library file for each major FPGA device family. Refer to the Lattice Synthesis Header Libraries topic for details. Typically the module is treated as a "black box" which causes the synthesis tool to pass instances of the library primitive into the target netlist untouched.

Global signals for global set/reset (GSR), power-up reset (PUR), tri-state all (TSALL), and the internal oscillator (OSCA, OSCC, OSCD, OSCE, OSCF) can be used within structural models built with Lattice library primitives. For more information, see How to Use the Global Set/Reset (GSR) Signal, How to Use the Tristate Interface (TSALL) Global Signal, and How to Use the Internal Oscillator.

The FPGA Libraries Reference Guide contains descriptions, pinouts, and schematic diagrams of all library primitives for Lattice FPGA libraries. For more information, in the Diamond software online help, refer to **Reference Guides > FPGA Libraries Reference Guide**.

Revision History

Date	Diamond Software Version	Description
April, 2019	3.11	Updated "Use IO Insertion" on page 17.
February, 2016	3.7	Initial release of document.

Index