

Introduction

This chapter documents key design methodologies and techniques for Altera® devices using the LeonardoSpectrum and Quartus II design flow. Combining HDL coding techniques, Mentor Graphics LeonardoSpectrum™ software constraints, and Quartus® II options provide the performance increase required for today's system-on-a-programmable-chip (SOC) designs.

The LeonardoSpectrum software is a mature synthesis tool supporting legacy devices and many current devices. The LeonardoSpectrum software version 2008b supports the Arria® GX, Stratix® IV, Stratix III, Stratix II, Stratix, Stratix GX, Cyclone® III, Cyclone II, Cyclone, MAX® II, MAX series, APEX™ series, FLEX® series, and ACEX® series device families. Altera recommends using the advanced Precision Synthesis software for new designs in new device families.



For more information about Precision RTL Synthesis, refer to the *Mentor Graphics Precision RTL Synthesis Support* chapter in volume 1 of the *Quartus II Handbook*.



This chapter assumes that you have set up, licensed, and are familiar with the LeonardoSpectrum software.



To obtain and license the LeonardoSpectrum software, refer to the Mentor Graphics website at www.mentor.com. For information about installing the LeonardoSpectrum software and setting up your working environment, refer to the *LeonardoSpectrum Installation Guide* and the *LeonardoSpectrum User's Manual*.

Design Flow

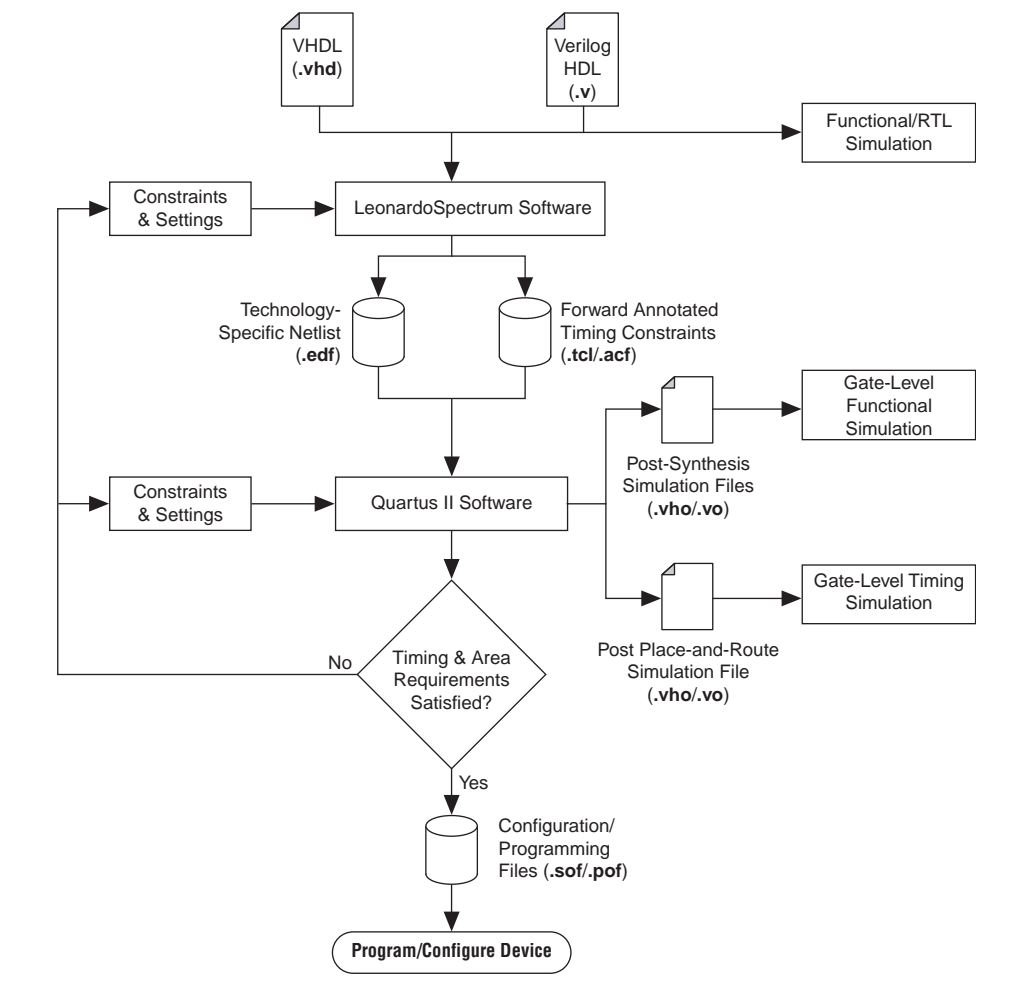
The following are basic steps in a LeonardoSpectrum-Quartus II design flow:

1. Create Verilog HDL or VHDL design files in the LeonardoSpectrum software or a text editor.
2. Import the Verilog HDL or VHDL design files into the LeonardoSpectrum software for synthesis.
3. Select a target device and add timing constraints and compiler directives to help optimize the design during synthesis.
4. Synthesize the project in the LeonardoSpectrum software.
5. Create a Quartus II project and import the technology-specific EDIF Input File (.edf) netlist and the Tcl Script File (.tcl) generated by the LeonardoSpectrum software into the Quartus II software for placement and routing, and for performance evaluation.
6. After obtaining place-and-route results that meet your requirements, configure or program the Altera device.

Figure 12–1 shows the recommended design flow using the LeonardoSpectrum and Quartus II software.

If your area and timing requirements are satisfied, use the programming files generated by the Quartus II software to program or configure the Altera device. As shown in Figure 12-1, if the area or timing requirements are not met, change the constraints in the LeonardoSpectrum software and re-run the synthesis. Repeat the process until the area and timing requirements are met. You can also use other Quartus II software options and techniques to meet the area and timing requirements.

Figure 12-1. Recommended Design Flow Using LeonardoSpectrum and Quartus II Software



The LeonardoSpectrum software supports both VHDL and Verilog HDL source files. With the appropriate license, it also supports mixed synthesis, allowing a combination of VHDL and Verilog HDL source files. After synthesis, the LeonardoSpectrum software produces several intermediate and output files. Table 12-1 lists these file extensions with a short description of each file.

Table 12-1. LeonardoSpectrum Intermediate and Output Files

File Extension(s)	File Description
.xdb	Technology-independent register transfer level (RTL) netlist file that can only be read by the LeonardoSpectrum software.
.edf	Technology-specific output netlist in electronic design interchange format (EDIF).

Table 12-1. LeonardoSpectrum Intermediate and Output Files

File Extension(s)	File Description
.acf/.tcl (1)	Forward-annotated constraint file containing constraints and assignments.

Note to Table 12-1:

- (1) An assignment and configuration (.acf) file is created only for ACEX 1K, FLEX series, and MAX series devices. The assignment and configuration file is generated for backward compatibility with the MAX+PLUS[®] II software. A .tcl file is generated for the Quartus II software that also contains Tcl commands to create a Quartus II project.

Altera recommends that you do not use project directory names that include spaces. Some file operations in the LeonardoSpectrum software do not work correctly if the path name contains spaces.

Specify timing constraints and compiler directives for the design in the LeonardoSpectrum software, or in a constraint file (.ctr). Many of these constraints are forward-annotated in the .tcl file for use by the Quartus II software.

The LeonardoInsight[™] Schematic Viewer is an add-on graphical tool for schematic views of the technology-independent RTL netlist (.xdb) and the technology-specific gate-level results. You can use the Schematic Viewer to visually analyze and debug the design. It also supports cross-probing between the RTL and gate-level schematics, the design browser, and the source code in the HDLInventor[™] text editor.

Optimization Strategies

You can configure most general settings in the **Quick Setup** tab in the LeonardoSpectrum user interface. Other Flow tabs provide additional options, and some Flow tabs include multiple Power tabs (at the bottom of the screen) with more options. Advanced optimization options in the LeonardoSpectrum software include timing-driven synthesis, encoding style, resource sharing, and mapping I/O registers.

Timing-Driven Synthesis

The LeonardoSpectrum software supports timing-driven synthesis through user-assigned timing constraints to optimize the performance of the design. The process for setting constraints in the LeonardoSpectrum software is straightforward. Constraints such as clock frequency can be specified globally or for individual clock signals. The following sections describe how to set the various types of timing constraints in the LeonardoSpectrum software.

The timing constraints described in “**Global Power Tab**” are set in the **Constraints** Flow tab. In this tab, there are Power tabs at the bottom, such as **Global** and **Clock**, for setting various constraints.

Global Power Tab

The **Global** tab is the default Power tab in the **Constraints** Flow tab where you can specify the global clock frequency. The **Clock Frequency** on the **Quick Setup** tab is equivalent to the **Registers to Registers** delay setting. You can also specify the **Input Ports to Registers**, **Registers to Output Ports**, and **Inputs to Outputs** delays that correspond to global t_{SU} , t_{CO} , and t_{PD} requirements, respectively, in the Quartus II software. The timing diagram on this tab reflects the settings you have made.

Clock Power Tab

You can set various constraints for each clock in your design. First, select the clock name in the Clock(s) window. The clock names appear after the design is read from the **Input** Flow tab. Configure settings for that particular clock and click **Apply**. If necessary, you can also set the **Duty Cycle** to a value other than the default 50%. The timing diagram shows these settings.

If a clock has an **Offset** from the main clock, which is considered to be time “0”, this constraint corresponds to the `OFFSET_FROM_BASE_CLOCK` setting in the Quartus II software.

You can specify the pin number for the clock input pin in the **Pin Location** field. This pin number is passed to the Quartus II software for place-and-route, but does not affect synthesis in the LeonardoSpectrum software.

Input and Output Power Tabs

Configure settings for individual input or output pins in the **Input** and **Output** tabs. First, select a name in the Input Ports or Output Ports window. The names appear after the design is read from the **Input** Flow tab. Then make the setting for that pin as described below.

The **Arrival Time** setting indicates that the input signal arrives a specified time after the rising clock edge (time “0”). This setting constrains the path from the pin to the first register by including the arrival time in the total delay, and corresponds to the `EXTERNAL_INPUT_DELAY` assignment in the Quartus II software.

The **Required Time** setting indicates the maximum delay after time “0” that the output signal should arrive at the output pin. This setting directly constrains the register to output delay, and corresponds with the `EXTERNAL_OUTPUT_DELAY` assignment in the Quartus II software.

Specify the pin number for the I/O pin in the **Pin Location** field. This pin number is passed to the Quartus II software for place-and-route, but does not affect synthesis in the LeonardoSpectrum software.

Other Constraints

The following sections describe other constraints that can be set with the LeonardoSpectrum user interface and contain these topics:

- “Encoding Style”
- “Resource Sharing”
- “Mapping I/O Registers”

Encoding Style

The LeonardoSpectrum software encodes state machines during the synthesis process. To improve performance when coding state machines, separate state machine logic from all arithmetic functions and data paths. When encoded, a design cannot be re-encoded later in the optimization process. You must follow a particular VHDL or Verilog HDL coding style for the LeonardoSpectrum software to identify the state machine.

Table 12-2 shows the state machine encoding styles supported by the LeonardoSpectrum software.

Table 12-2. State Machine Encoding Styles in the LeonardoSpectrum Software

Style	Description
Binary	Generates state machines with the fewest possible flipflops. Binary state machines are useful for area-critical designs when timing is not the primary concern.
Gray	Generates state machines where only one flipflop changes during each transition. Gray-encoded state machines tend to be glitchless.
One-hot	Generates state machines containing one flipflop for each state. One-hot state machines provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than binary implementations.
Random	Generates state machines using random state machine encoding. Only use random state machine encoding when no other implementation achieves the desired results.
Auto (Default)	Implements binary or one-hot encoding, depending on the size of enumerated types in the state machine.

The **Encoding Style** setting is created in the **Input Flow** tab. It instructs the software to use a particular state machine encoding style for all state machines. The default **Auto** selection implements binary or one-hot encoding, depending on the size of enumerated types in the state machine.

 To ensure proper recognition and improve performance when coding state machines, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook* for design guidelines.

Resource Sharing

You can also enable the **Resource Sharing** setting in the **Input Flow** tab. This setting allows optimization to reduce device resources. You should generally leave this setting turned on.

Mapping I/O Registers

The **Map I/O Registers** option is located in the **Technology Flow** tab. The **Map I/O Registers** option applies to Altera FPGAs containing I/O cells (IOCs) or I/O elements (IOEs). If the option is turned on, input or output registers are moved into the device's I/O cells for faster setup or clock-to-output times.

Timing Analysis with the Leonardo-Spectrum Software

The LeonardoSpectrum software reports successful synthesis with an information message in the Transcript or Information window. Estimated device usage and timing results are reported in the Device Utilization section of this window. Figure 12-2 shows an example of a LeonardoSpectrum compilation report.

Figure 12-2. LeonardoSpectrum Compilation Report

```

*****
Device Utilization for EP20K200EQC208
*****
Resource          Used    Avail    Utilization
-----
IOs                22      136     16.18%
LCs               114     8320     1.37%
Memory Bits       0      106496    0.00%
-----

                          Clock Frequency Report

      Clock          : Frequency
-----
      clk            : 52.2 MHz
      clk2           : 149.5 MHz

                          Critical Path Report

```

The LeonardoSpectrum software estimates the timing results based on timing models. The LeonardoSpectrum software has no information about how the design is placed and routed in the Quartus II software, so it cannot report accurate routing delays. Additionally, if the design includes any black boxed Altera-specific functions, the LeonardoSpectrum software does not report timing information for these functions.

Final timing results are generated by the Quartus II software and are reported separately in the Transcript or Information window if the **Run Integrated Place and Route** option is turned on. Refer to “[Integration with the Quartus II Software](#)” for more information.

Exporting Designs Using NativeLink Integration

You can use NativeLink® integration to integrate the LeonardoSpectrum software and the Quartus II software with a single GUI for both the synthesis and place-and-route operations. You can run the Quartus II software from within the LeonardoSpectrum software GUI with NativeLink integration or you can run the LeonardoSpectrum software from within the Quartus II software GUI for device families supported in the Quartus II software.

Generating Netlist Files

The LeonardoSpectrum software generates an **.edif** netlist file readable as an input file in the Quartus II software for place-and-route. Select the **.edif** file option name in the **Output Flow** tab. The **.edif** netlist file is also generated if the **Auto** option is turned on in the **Output Flow** tab.

Including Design Files for Black Boxed Modules

If the design has black boxed megafunctions, be sure to include the MegaWizard™ Plug-In Manager-generated custom megafunction variation design file in the Quartus II project directory, or add it to the list of project files for place-and-route.

Passing Constraints with Scripts

The LeonardoSpectrum software can write out a `.tcl` file called `<project name>.tcl`. This file contains commands to create a Quartus II project along with constraints and other assignments. To output a Tcl script, turn on the **Write Vendor Constraint Files** option in the **Output Flow** tab.

To create and compile a Quartus II project using the `.tcl` file generated from the LeonardoSpectrum software, perform the following steps in the Quartus II software:

1. Place the `.edif` netlist files and Tcl scripts in the same directory.
2. On the View menu, point to **Utility**, and click **Tcl Console** to open the Quartus II Tcl Console.
3. Type `source <path>/<project name>.tcl` at a Tcl Console command prompt.
4. On the File menu, click **Open Project** to open the new project. On the Processing menu, click **Start Compilation**.

Integration with the Quartus II Software

You can launch the Quartus II software from within the LeonardoSpectrum software with the **Place And Route** section in the **Quick Setup** tab. Turn on the **Run Integrated Place and Route** option to start the compilation using the Quartus II software to show the fitting and performance results. You can also run the place-and-route software by turning on the **Run Quartus** option on the **Physical Flow** tab and clicking **Run PR**.

To use integrated place-and-route software, on the Options menu, point to **Place and Route Path** and click **Tools**. Specify the location of the Quartus II software executable file (browse to `<Quartus II software installation directory>/bin`).

Guidelines for Altera Megafunctions and LPM Functions

Altera provides parameterizable megafunctions ranging from simple arithmetic units, such as adders and counters, to advanced phase-locked loop (PLL) blocks, multipliers, and memory structures. These functions are performance-optimized for Altera devices. Megafunctions include the library of parameterized modules (LPM), device-specific megafunctions such as PLLs, LVDS, and digital signal processing (DSP) blocks, intellectual property (IP) available as Altera MegaCore® functions, and IP available through the Altera Megafunction Partners Program (AMPPsm).



Some IP cores require that you synthesize them in the LeonardoSpectrum software. Refer to the user guide for the specific IP.

There are two methods for handling megafunctions in the LeonardoSpectrum software: inference and instantiation.

The LeonardoSpectrum software supports inferring some of the Altera megafunctions, such as multipliers, DSP functions, and RAM and ROM blocks. The LeonardoSpectrum software supports all Altera megafunctions through instantiation.

Instantiating Altera Megafunctions

There are two methods of instantiating Altera megafunctions in the LeonardoSpectrum software. The first and least common method is to directly instantiate the megafunction in the Verilog HDL or VHDL code. The second method, to maintain target technology awareness, is to use the MegaWizard Plug-In Manager in the Quartus II software to set up and parameterize a megafunction variation. The megafunction wizard creates a wrapper file that instantiates the megafunction. The advantage of using the megafunction wizard in place of the instantiation method is the megafunction wizard properly sets all the parameters and you do not need the library support required in the direct instantiation method. This is referred to as black box methodology.



Altera recommends using the MegaWizard Plug-In Manager to ensure that the ports and parameters are set correctly.



When directly instantiating megafunctions, see the Quartus II Help for a list of the ports and parameters.

Inferring Altera Memory Elements

The LeonardoSpectrum software can infer memory blocks from Verilog HDL or VHDL code. When the LeonardoSpectrum software detects a RAM or ROM from the style of the RTL code at a technology-independent level, it then maps the element to a generic module in the RTL database. During the technology-mapping phase of synthesis, the LeonardoSpectrum software maps the generic module to the most optimal primitive memory cells, or Altera megafunction, for the target Altera technology.



For more information about inferring RAM and ROM megafunctions, including examples of VHDL and Verilog HDL code, see the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Inferring RAM

The LeonardoSpectrum software supports RAM inference for various device families. The following are the restrictions for the LeonardoSpectrum software to successfully infer RAM in a design:

- The write process must be synchronous
- The read process can be asynchronous or synchronous depending on the target Altera architecture
- Resets on the memory are not supported

Table 12-3 shows a summary of the minimum memory sizes and minimum address widths for inferring RAM in various device families.

Table 12-3. Inferring RAM Summary

Devices	RAM Primitive	Minimum RAM Size	Minimum Address Width
Stratix Series and Cyclone Series	altsyncram	2 bits	1 bit
APEX Series, Excalibur and Mercury	altdpram	64 bits	4 bits
FLEX 10KE and ACEX 1K	altdpram	128 bits	5 bits

To disable RAM inference, set the `extract_ram` and `infer_ram` variables to “false.” On the Tools menu, click **Variable Editor** to enter the value “false” when synthesizing in the user interface with the **Advanced Flow** tabs, or add the commands `set extract_ram false` and `set infer_ram false` to your synthesis script.

Inferring ROM

You can implement ROM behavior in HDL source code with CASE statements or specify the ROM as a table. The LeonardoSpectrum software infers both synchronous and asynchronous ROM depending on the target Altera device. For example, memory for the Stratix series devices must be synchronous to be inferred.

To disable ROM inference, set the `extract_rom` variable to “false.” To enter the value “false” when synthesizing in the user interface with the **Advanced Flow** tabs, on the Tools menu, click **Variable Editor**, or add the commands `set extract_rom false` to your synthesis script.

Inferring Multipliers and DSP Functions

Some Altera devices include dedicated DSP blocks optimized for DSP applications. The following Altera megafunctions are used with DSP block modes:

- `LPM_MULT`
- `ALTMULT_ACCUM`
- `ALTMULT_ADD`

You can instantiate these megafunctions in the design or have the LeonardoSpectrum software infer the appropriate megafunction by recognizing a multiplier, multiplier-accumulator (MAC), or multiplier-adder in the design. The Quartus II software maps the functions to the DSP blocks in the device during place-and-route.



For more information about inferring multipliers and DSP functions, including examples of VHDL and Verilog HDL code, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Simple Multipliers

The `LPM_MULT` megafunction implements the DSP block in the simple multiplier mode. The following functionality is supported in this mode:

- The DSP block includes registers for the input and output stages, and an intermediate pipeline stage
- Signed and unsigned arithmetic is supported

Multiplier Accumulators

The `ALTMULT_ACCUM` megafunction implements the DSP block in the multiply-accumulator mode. The following functionality is supported in this mode:

- The DSP block includes registers for the input and output stages, and an intermediate pipeline stage
- The output registers are required for the accumulator
- The input and pipeline registers are optional

- Signed and unsigned arithmetic is supported



If the design requires input registers to be used as shift registers, use the black box method to instantiate the ALTMULT_ACCUM megafunction.

Multiplier Adders

The LeonardoSpectrum software can infer multiplier adders and map them to either the two-multiplier adder mode or the four-multiplier adder mode of the DSP blocks. The LeonardoSpectrum software maps the HDL code to the correct ALTMULT_ADD function.

The following functionality is supported in these modes:

- The DSP block includes registers for the input and output stages, and an intermediate pipeline stage
- Signed and unsigned arithmetic is supported, but support for the Verilog HDL “signed” construct is limited

Controlling DSP Block Inference

Device features, such as dedicated DSP blocks, multipliers, multiply-accumulators, and multiply-adders can be implemented in DSP blocks or in logic. You can control this implementation through attribute settings in the LeonardoSpectrum software.

As shown in Table 12-4, attribute settings in the LeonardoSpectrum software control the implementation of the multipliers in DSP blocks or logic at the signal block (or module), and project level.

Table 12-4. Attribute Settings for DSP Blocks in the LeonardoSpectrum Software (Note 1)

Level	Attribute Name	Value	Description
Global	extract_mac (2)	TRUE	All multipliers in the project mapped to DSP blocks.
		FALSE	All multipliers in the project mapped to logic.
Module	extract_mac (3)	TRUE	Multipliers inside the specified module mapped to DSP blocks.
		FALSE	Multipliers inside the specified module mapped to logic.
Signal	dedicated_mult	ON	LPM inferred and multipliers implemented in DSP block.
		OFF	LPM inferred, but multipliers implemented in logic by the Quartus II software.
		LCELL	LPM not inferred, and multipliers implemented in logic by the LeonardoSpectrum software.
		AUTO	LPM inferred, but the Quartus II software automatically maps the multipliers to either logic or DSP blocks based on the Quartus II software place-and-route.

Notes to Table 12-4:

- (1) The extract_mac attribute takes precedence over the dedicated_mult attribute.
- (2) For devices with DSP blocks, the extract_mac attribute is set to “true” by default for the entire project.
- (3) For devices with DSP blocks, the extract_mac attribute is set to “true” by default for all modules.

Global Attribute

You can set the global attribute `extract_mac` to control the implementation of multipliers in DSP blocks for the entire project. You can set this attribute using the script interface. The script command is:

```
set extract_mac <value>
```

Module Level Attributes

You can control the implementation of multipliers inside a module or component by setting attributes in the Verilog HDL source code. The attribute used is `extract_mac`. Setting this attribute for a module affects only the multipliers inside that module. The command is:

```
//synthesis attribute <module name> extract_mac <value>
```

The Verilog HDL and VHDL code samples in [Example 12-1](#) and [Example 12-2](#) show how to use the `extract_mac` attribute.

Example 12-1. Using Module Level Attributes in Verilog HDL Code

```
module mult_add ( dataa, datab, datac, datad, result);  
//synthesis attribute mult_add extract_mac FALSE  
// Port Declaration  
input [15:0] dataa;  
input [15:0] datab;  
input [15:0] datac;  
input [15:0] datad;  
  
output [32:0] result;  
  
// Wire Declaration  
wire [31:0] mult0_result;  
wire [31:0] mult1_result;  
  
// Implementation  
// Each of these can go into one of the 4 mults in a  
// DSP block  
assign mult0_result = dataa * `signed datab;  
//synthesis attribute mult0_result preserve_signal TRUE  
  
assign mult1_result = datac * datad;  
  
// This adder can go into the one-level adder in a DSP  
// block  
assign result = (mult0_result + mult1_result);  
  
endmodule
```

Example 12-2. Using Module Level Attributes in VHDL Code

```

library ieee ;
USE ieee.std_logic_1164.all;

USE ieee.std_logic_arith.all;

entity mult_acc is
  generic (size : integer := 4) ;
  port (
    a: in std_logic_vector (size-1 downto 0) ;
    b: in std_logic_vector (size-1 downto 0) ;
    clk : in std_logic;
    accum_out: inout std_logic_vector (2*size downto 0)
  ) ;
  attribute extract_mac : boolean;
  attribute extract_mac of mult_acc : entity is FALSE;
end mult_acc;

architecture synthesis of mult_acc is
  signal a_int, b_int : signed (size-1 downto 0);
  signal pdt_int : signed (2*size-1 downto 0);
  signal adder_out : signed (2*size downto 0);

begin
  a_int <= signed (a);
  b_int <= signed (b);
  pdt_int <= a_int * b_int;
  adder_out <= pdt_int + signed(accum_out);
  process (clk)
  begin
    if (clk'event and clk = '1') then
      accum_out <= std_logic_vector (adder_out);
    end if;
  end process;
end synthesis ;

```

Signal Level Attributes

You can control the implementation of individual LPM_MULT multipliers by using the `dedicated_mult` attribute, as shown below:

```
//synthesis attribute <signal_name> dedicated_mult <value>
```



The `dedicated_mult` attribute is only applicable to signals or wires; it is not applicable to registers.

Table 12-5 shows the supported values for the `dedicated_mult` attribute.

Table 12-5. Values for the `dedicated_mult` Attribute (Part 1 of 2)

Value	Description
ON	LPM inferred and multipliers implemented in DSP block.
OFF	LPM inferred and multipliers synthesized, implemented in logic, and optimized by the Quartus II software. (1)
LCELL	LPM not inferred and multipliers synthesized, implemented in logic, and optimized by the LeonardoSpectrum software. (1)

Table 12-5. Values for the dedicated_mult Attribute (Part 2 of 2)

Value	Description
AUTO	LPM inferred but the Quartus II software maps the multipliers automatically to either the DSP block or logic based on resource availability.

Note to Table 12-5:

- (1) Although both dedicated_mult=OFF and dedicated_mult=LCELLS result in logic implementations, the optimized results in these two cases may differ.



Some signals for which the dedicated_mult attribute is set might get synthesized away by the LeonardoSpectrum software due to design optimization. In such cases, if you want to force the implementation, the signal is preserved from being synthesized away by setting the preserve_signal attribute to “true.”

The extract_mac attribute must be set to “false” for the module or project level when using the dedicated_mult attribute.

Example 12-3 and Example 12-4 are samples of Verilog HDL and VHDL codes, respectively, using the dedicated_mult attribute.

Example 12-3. Signal Attributes for Controlling DSP Block Inference in Verilog HDL Code

```

module mult (AX, AY, BX, BY, m, n, o, p);
input [7:0] AX, AY, BX, BY;
output [15:0] m, n, o, p;
wire [15:0] m_i = AX * AY; // synthesis attribute m_i dedicated_mult ON
// synthesis attribute m_i preserve_signal TRUE
//Note that the preserve_signal attribute prevents
// signal m_i from getting synthesized away
wire [15:0] n_i = BX * BY; // synthesis attribute n_i dedicated_mult OFF
wire [15:0] o_i = AX * BY; // synthesis attribute o_i dedicated_mult AUTO
wire [15:0] p_i = BX * AY; // synthesis attribute p_i dedicated_mult
LCELL
// since n_i , o_i , p_i signals are not preserved,
// they may be synthesized away based on the design
assign m = m_i;
assign n = n_i;
assign o = o_i;
assign p = p_i;
endmodule
    
```

Example 12-4. Signal Attributes for Controlling DSP Block Inference in VHDL Code

```

library ieee ;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_signed.all;
ENTITY mult is

PORT( AX,AY,BX,BY: IN
std_logic_vector (17 DOWNTO 0);
m,n,o,p: OUT
std_logic_vector (35 DOWNTO 0));
attribute dedicated_mult: string;
attribute preserve_signal : boolean
END mult;
ARCHITECTURE struct of mult is

signal m_i, n_i, o_i, p_i : unsigned (35 downto 0);
attribute dedicated_mult of m_i:signal is "ON";
attribute dedicated_mult of n_i:signal is "OFF";
attribute dedicated_mult of o_i:signal is "AUTO";
attribute dedicated_mult of p_i:signal is "LCELL";

begin

m_i <= unsigned (AX) * unsigned (AY);
n_i <= unsigned (BX) * unsigned (BY);
o_i <= unsigned (AX) * unsigned (BY);
p_i <= unsigned (BX) * unsigned (AY);

m <= std_logic_vector(m_i);
n <= std_logic_vector(n_i);
o <= std_logic_vector(o_i);
p <= std_logic_vector(p_i);
end struct;

```

Guidelines for Using DSP Blocks

In addition to the guidelines mentioned earlier in this section, use the following guidelines while designing with DSP blocks in the LeonardoSpectrum software:


- To access all the control signals for the DSP block, such as `sign A`, `sign B`, and `dynamic addnsub`, use the black box technique.
- While performing signed operations, ensure that the specified data width of the output port matches the data width of the expected result. Otherwise, the sign bit might be lost or data might be incorrect because the sign is not extended. For example, if the data widths of input A and B are `width_a` and `width_b`, respectively, the maximum data width of the result can be $(width_a + width_b + 2)$ for the four-multipliers adder mode. Thus, the data width of the output port should be less than or equal to $(width_a + width_b + 2)$.
- While using the accumulator, the data width of the output port should be equal to or greater than $(width_a + width_b)$. The maximum width of the accumulator can be $(width_a + width_b + 16)$. Accumulators wider than this are implemented in logic.

- If the design uses more multipliers than are available in a particular device, you might get a no fit error in the Quartus II software. In such cases, use the attribute settings in the LeonardoSpectrum software to control the mapping of multipliers in your design to DSP blocks or logic.

Block-Based Design with the Quartus II Software


The incremental compilation and LogicLock™ block-based design flows enable users to design, optimize, and lock down a design one section at a time. You can independently create and implement each logic module into a hierarchical or team-based design. With this method, you can preserve the performance of each module during system integration and have more control over placement of your design. To maximize the benefits of the incremental compilation or LogicLock design methodology in the Quartus II software, you can partition a new design into a hierarchy of netlist files during synthesis in the LeonardoSpectrum software.

You can create different netlist files with the LeonardoSpectrum software for different sections of a design hierarchy. When you have different netlist files, it means that each section is independent of the others. When synthesizing the entire project, only portions of a design that have been updated have to be re-synthesized when you compile the design. You can make changes, optimize, and re-synthesize your section of a design without affecting other sections.

-  For more information about incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about the LogicLock feature, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Hierarchy and Design Considerations

You must plan your design's structure and partitioning carefully to use incremental compilation and LogicLock features effectively. Optimal hierarchical design practices include partitioning the blocks at functional boundaries, registering the boundaries of each block, minimizing the I/O between each block, separating timing-critical blocks, and keeping the critical path within one hierarchical block.

-  For more recommendations for hierarchical design partitioning, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

To ensure the proper functioning of the synthesis tool, you can apply the LogicLock option in the LeonardoSpectrum software only to modules, entities, or netlist files. In addition, each module or entity should have its own design file. It is difficult to maintain incremental synthesis if two different modules are in the same design file (but are defined as being part of different regions), because both regions have to be recompiled when you change one of the modules or entities.

If you use boundary tri-states in a lower-level block, the LeonardoSpectrum software pushes (or “bubbles”) the tri-states through the hierarchy to the top-level to take advantage of the tri-state drivers on the output pins of the Altera device. Because bubbling tri-states require optimizing through hierarchies, lower-level tri-states are not supported with a block-level design methodology. You should use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

If the hierarchy is flattened during synthesis, logic is optimized across boundaries, preventing you from making LogicLock assignments to the flattened blocks. Altera recommends preserving the hierarchy when compiling the design. In the **Optimize** command of your script, use the **Hierarchy Preserve** command, or in the user interface, select **Preserve** in the **Hierarchy** section on the **Optimize** Flow tab.

If you are compiling your design with a script, you can use an alternative method for preventing optimization across boundaries. In this case, use the **Auto** hierarchy setting and set the `auto_dissolve` attribute to false on the instances or views that you want to preserve (that is, the modules with LogicLock assignments) using the following syntax:

```
set_attribute -name auto_dissolve -value false \  
    .work.<block1>.INTERFACE
```

This alternative method flattens your design according to the `auto_dissolve` limits, but does not optimize across boundaries where you apply the attribute as described.



For more details about LeonardoSpectrum attributes and hierarchy levels, refer to the LeonardoSpectrum documentation in the Help menu.

Creating a Design with Multiple .edif Files

The first stage of a hierarchical design flow is to generate multiple **.edif** files, so that you can take advantage of the incremental compilation flows in the Quartus II software. If the whole design is in one **.edif** file, changes in one block affect other blocks because of possible node name changes. You can generate multiple **.edif** files either by using the LogicLock option in the LeonardoSpectrum software, or by manually using a black box methodology on each block that you want to be part of a LogicLock region.

After you have created multiple **.edif** files with one of these methods, you must create the appropriate Quartus II project(s) to place-and-route the design.

Generating Multiple .edif Files Using the LogicLock Option

This section describes how to generate multiple **.edif** files using the LogicLock option in the LeonardoSpectrum software.

When synthesizing a top-level design that includes LogicLock regions, perform the following general steps:

1. Read in the Verilog HDL or VHDL source files.
2. Add LogicLock constraints.
3. Optimize and write output netlist files, or choose **Run Flow**.

To set the correct constraints and compile the design, perform the following steps in the LeonardoSpectrum software:

1. On the Tools menu, switch to the **Advanced Flow** tab instead of the **Quick Setup** tab.
2. Set the target technology and speed grade for the device on the **Technology Flow** tab.
3. Open the input source files on the **Input Flow** tab.
4. Click **Read** on the **Input Flow** tab to read the source files but not begin optimization.
5. Select the **Module Power** tab located at the bottom of the **Constraints** Flow tab.
6. Click on a module to be placed in a LogicLock region in the **Modules** section.
7. Turn on the **LogicLock** option.
8. Type the desired LogicLock region name in the text field under the **LogicLock** option.
9. Click **Apply**.
10. Repeat steps 6-9 for any other modules that you want to place in LogicLock regions.




In some cases, you are prompted to save your LogicLock and other non-global constraints in a Constraints File (.ctr) when you click anywhere off the **Constraints Flow** tab. The default name is *<project name>.ctr*. This file is added to your **Input** file list, and must be manually included later if you recreate the project.

The command written into the LeonardoSpectrum Information or Transcript Window is the Tcl command that gets written into the .ctr file. The format of the "path" for the module specified in the command should be *work.<module>.INTERFACE*. To ensure that you don't see an optimized version of the module, do not perform a **Run Flow** on the **Quick Setup** tab prior to setting LogicLock constraints. Always use the **Read** command, as described in step 4.

11. Continue making any other settings as required on the **Constraints** tab.
12. Select **Preserve** in the **Hierarchy** section on the **Optimize** tab to ensure that the hierarchy names are not flattened during optimization.
13. Continue making any other settings as required on the **Optimize** tab.
14. Run your synthesis flow with each Flow tab, or click **Run Flow**.

Synthesis creates an .edif file for each module that has a LogicLock assignment in the **Constraints** Flow tab. You can now use these files with the incremental compilation flows in the Quartus II software.

 You might occasionally see multiple `.edif` files and LogicLock commands for the same module. An “unfolded” version of a module is created when you instantiate a module more than once and the boundary conditions of the instances are different. For example, if you apply a constant to one instance of the block, it might be optimized to eliminate unneeded logic. In this case, the LeonardoSpectrum software must create a separate module for each instantiation (unfolding). If this unfolding occurs, you see more than one `.edif` file, and each EDIF file has a LogicLock assignment to the same LogicLock region. When you import the `.edif` files to the Quartus II software, the `.edif` files created from the module are placed in different LogicLock regions. Any optimizations performed in the Quartus II software using the LogicLock methodology must be performed separately for each `.edif` netlist file.

Creating a Quartus II Project for Multiple `.edif` Files Including LogicLock Regions


The LeonardoSpectrum software creates `.tcl` files that provide the Quartus II software with the appropriate LogicLock assignments, creating a region for each `.edif` file along with the information to set up a Quartus II project.

The `.tcl` file contains the commands shown in [Example 12-5](#) for each LogicLock region. This example is for module `taps` where the name `taps_region` was typed as the LogicLock region name in the **Constraints** Flow tab in the LeonardoSpectrum software.

Example 12-5. Tcl File for Module Taps with `taps_region` as LogicLock Region Name

```
project add_assignment {taps} {taps_region} {} {}
  {LL_AUTO_SIZE} {ON}
project add_assignment {taps} {taps_region} {} {}
  {LL_STATE} {FLOATING}
project add_assignment {taps} {taps_region} {} {}
  {LL_MEMBER_OF} {taps_region}
```

These commands create a LogicLock region with Auto-Size and Floating-Origin properties. This flexible LogicLock region allows the Quartus II Compiler to select the size and location of the region.

 For more information about Tcl commands, refer to the [TCL Scripting](#) chapter in volume 2 of the *Quartus II Handbook*.

You can use the following methods to import the `.edif` file and corresponding `.tcl` file into the Quartus II software:


- Use the `.tcl` file that is created for each `.edif` file by the LeonardoSpectrum software. This method allows you to generate multiple Quartus II projects, one for each block in the design. Each designer in the project can optimize their block separately in the Quartus II software and preserve their results. Altera recommends this method for bottom-up incremental and hierarchical design methodologies because it allows each block in the design to be treated separately. Each block can be brought into one top-level project with the import function.

or

- Use the `<top-level project>.tcl` file that contains the assignments for all blocks in the project. This method allows the top-level designer to import all the blocks into one Quartus II project. You can optimize all modules in the project at once in a top-down design flow. If additional optimization is required for individual blocks, each designer can use their `.edif` file to create a separate project at that time. You would then have to add new assignments to the top-level project using the import function.

In both methods, use the following steps to create the Quartus II project, import the appropriate LogicLock assignments, and compile the design:

1. Place the `.edif` and `.tcl` files in the same directory.
2. On the View menu, point to **Utility Windows** and click **Tcl Console** to open the **Quartus II Tcl Console**.
3. Type `source <path>/<project name>.tcl` ↵.
4. To open the new completed project, on the File menu, click **Open Project**. Browse to and select the project name, and click **Open**.

 For more information about importing a design using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about importing LogicLock assignments, see the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

Generating Multiple `.edif` Files Using Black Boxes

This section describes how to manually generate multiple `.edif` files using the black box technique. The manual flow was supported in older versions of the LeonardoSpectrum software. The manual flow is discussed here because some designers want more control over the project for each submodule.

To create multiple `.edif` files in the LeonardoSpectrum software, create a separate project for each module and top-level design that you want to maintain as a separate `.edif` file. Implement black box instantiations of lower-level modules in your top-level project.

When synthesizing the projects for the lower-level modules and the top-level design, use the following general guidelines.

For lower-level modules:

- Turn off **Map IO Registers** for the target technology on the **Technology** Flow tab.
- Read the HDL files for the modules. Modules may include black box instantiations of lower-level modules that are also maintained as separate `.edif` files.
- Add constraints.
- Turn off **Add I/O Pads** on the **Optimize** Flow tab.

For the top-level design:

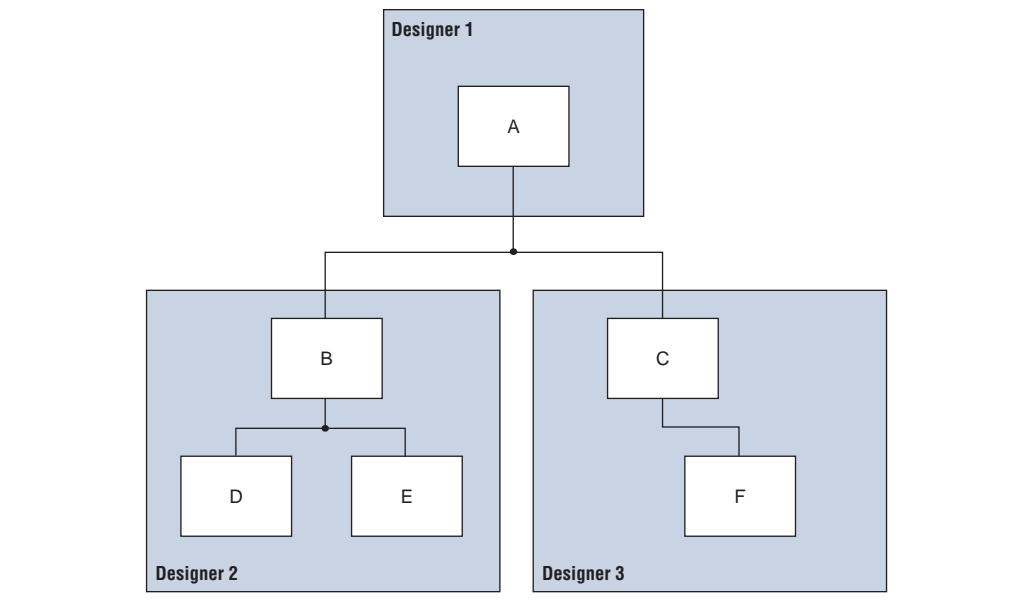
- Turn on **Map IO Registers** if you want to implement input and/or output registers in the IOEs for the target technology on the **Technology** Flow tab.

- Read the HDL files for the top-level design.
 - Black box lower-level modules in the top-level design.
- Add constraints (clock settings should be made at this time).

The following sections describe examples of black box modules in a block-based and team-based design flow.

In [Figure 12-3](#), the top-level design A is assigned to one engineer (designer 1), while two-engineers work on the lower levels of the design. Designer 2 works on B and its submodules D and E, while designer 3 works on C and its submodule F.

Figure 12-3. Block-Based and Team-Based Design Example



One netlist is created for the top-level module A, another netlist is created for B and its submodules D and E, and another netlist is created for C and its submodule F. To create multiple `.edif` files, perform the following steps:

1. Generate an `.edif` file for module C. Use `C.v` and `F.v` as the source files.
2. Generate an `.edif` file for module B. Use `B.v`, `D.v`, and `E.v` as the source files.
3. Generate a top-level `.edif` file `A.v` for module A. Ensure that your black box modules B and C were optimized separately in steps 1 and 2.

Black Box Methodology in Verilog HDL

Any design block that is not defined in the project, or included in the list of files to be read for a project, is treated as a black box by the software. In Verilog HDL, you must also provide an empty module declaration for the module that you plan to treat as a black box.

[Example 12-6](#) shows an example of the `A.v` top-level file. If any of your lower-level files also contain a black-boxed lower-level file in the next level of hierarchy, follow the same procedure.

Example 12-6. Verilog HDL Top-Level File Black Boxing Example

```
module A (data_in,clk,e,ld,data_out);
    input data_in, clk, e, ld;
    output [15:0] data_out;

    reg [15:0] cnt_out;
    reg [15:0] reg_a_out;

    B U1 ( .data_in (data_in),.clk (clk), .e(e), .ld (ld),
        .data_out(cnt_out) );

    C U2 ( .d(cnt_out), .clk (clk), .e(e), .q (reg_out));
    // Any other code in A.v goes here.

endmodule

// Empty Module Declarations of Sub-Blocks B and C follow here.
// These module declarations (including ports) are required for
blackboxing.

module B (data_in,e,ld,data_out );
    input data_in, clk, e, ld;
    output [15:0] data_out;
endmodule

module C (d,clk,e,q );
    input d, clk, e;
    output [15:0] q;
endmodule
```



Previous versions of the LeonardoSpectrum software required an attribute statement `//exemplar attribute U1 NOOPT TRUE`, which instructed the software to treat the instance U1 as a black box. This attribute is no longer required, although it is still supported in the software.

Black Boxing in VHDL

Any design block that is not defined in the project, or included in the list of files to be read for a project, is treated as a black box by the software. In VHDL, a component declaration is required for the black box which is normal for any other block in the design.

Example 12-7 shows an example of the **A.vhd** top-level file. If any of your lower-level files also contain a black boxed lower-level file in the next level of hierarchy, follow the same procedure.

Example 12-7. VHDL Top-Level File Black Boxing Example

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY A IS
PORT ( data_in : IN INTEGER RANGE 0 TO 15;
      clk : IN STD_LOGIC;
      e : IN STD_LOGIC;
      ld : IN STD_LOGIC;
      data_out : OUT INTEGER RANGE 0 TO 15
);
END A;

ARCHITECTURE a_arch OF A IS

COMPONENT B PORT(
  data_in : IN INTEGER RANGE 0 TO 15;
  clk : IN STD_LOGIC;
  e : IN STD_LOGIC;
  ld : IN STD_LOGIC;
  data_out : OUT INTEGER RANGE 0 TO 15
);
END COMPONENT;

COMPONENT C PORT(
  d : IN INTEGER RANGE 0 TO 15;
  clk : IN STD_LOGIC;
  e : IN STD_LOGIC;
  q : OUT INTEGER RANGE 0 TO 15
);
END COMPONENT;

-- Other component declarations in A.vhd go here

signal cnt_out : INTEGER RANGE 0 TO 15;
signal reg_a_out : INTEGER RANGE 0 TO 15;
BEGIN
CNT : C
PORT MAP (
  data_in => data_in,
  clk => clk,
  e => e,
  ld => ld,
  data_out => cnt_out
);

REG_A : D
PORT MAP (
  d => cnt_out,
  clk => clk,
  e => e,
  q => reg_a_out
);

-- Any other code in A.vhd goes here

END a_arch;

```



Previous versions of the LeonardoSpectrum software required the attribute statement `noopt of C: component is TRUE`, which instructed the software to treat the component C as a black box. This attribute is no longer required, although it is still supported in the software.

After you have completed the steps outlined in this section, you have a different **.edif** netlist file for each block of code. You can now use these files for incremental compilation flows in the Quartus II software.

Creating a Quartus II Project for Multiple **.edif** Files


The LeonardoSpectrum software creates a **.tcl** file for each **.edif** file, which provides the Quartus II software with the information to set up a project.

As in the previous section, there are two different methods for bringing each **.edif** file and corresponding **.tcl** file into the Quartus II software:

- Use the **.tcl** file that is created for each **.edif** file by the LeonardoSpectrum software. This method generates multiple Quartus II projects, one for each block in the design. Each designer in the project can optimize their block separately in the Quartus II software and preserve their results. Designers should create a LogicLock region for each block; the top-level designer should then import all the blocks and assignments into the top-level project. Altera recommends this method for bottom-up incremental and hierarchical design methodology because it allows each block in the design to be treated separately; each block can be imported into one top-level project.

or

- Use the *<top-level project>.tcl* file that contains the information to set up the top-level project. This method allows the top-level designer to create LogicLock regions for each block and bring all the blocks into one Quartus II project. Designers can optimize all modules in the project at once in a top-down design flow. If additional optimization is required for individual blocks, each designer can take their **.edif** file and create a separate Quartus II project at that time. New assignments would then have to be added to the top-level project manually or through the import function.

 For more information about importing designs using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. For more information about importing LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

In both methods, use the following steps to create the Quartus II project and compile the design:

1. Place the **.edif** and **.tcl** files in the same directory.
2. On the View menu, point to **Utility Windows** and click **Tcl Console**. The Quartus II Tcl Console appears.
3. At a Tcl prompt, type `source <path>/<project name>.tcl` ↵.
4. On the File menu, click **Open Project**. In the New Project window, browse to and select the project name. Click **Open**.
5. To create LogicLock assignments, on the Assignments menu, click **LogicLock Regions Window**.
6. On the Processing menu, click **Start Compilation**.

Incremental Synthesis Flow

If you make changes to one or more submodules, you can manually create new projects in the LeonardoSpectrum software to generate a new **.edif** netlist file when there are changes to the source files. Alternatively, you can use incremental synthesis to generate a new netlist for the changed submodule(s). To perform incremental synthesis in the LeonardoSpectrum software, use the script described in this section to reoptimize and generate a new **.edif** netlist file for only the affected modules using the LeonardoSpectrum top-level project. This method applies only when you are using the **LogicLock** option in the LeonardoSpectrum software.

Modifications Required for the **LogicLock_Incremental.tcl** Script File

There are three sets of entries in the file that must be modified before beginning incremental synthesis. The variables in the **.tcl** file are surrounded by angle brackets (< >).

1. Add the list of source files that are included in the project. You can enter the full path to the file or just the file name if the files are located in the working directory.
2. Indicate which modules in the design have changed. These modules are the **.edif** files that are regenerated by the LeonardoSpectrum software. These modules contain a LogicLock assignment in the original compilation.



Obtain the LeonardoSpectrum software path for each module by looking at the **.ctr** file that contains the LogicLock assignments from the original project. Each LogicLock assignment is applied to a particular module in the design.

3. Enter the target device family using the appropriate device keyword. The device keyword is written into the Transcript or Information window when you select a target Technology and click **Load Library** or **Apply** on the **Technology** Flow tab in the graphical user interface.

Example 12-8 shows the **LogicLock_Incremental.tcl** file for the incremental synthesis flow. You must modify the **.tcl** file before you can use it for your project.

Example 12-8. LogicLock_Interface.tcl Script File for Incremental Synthesis

```
#####
### LogicLock Incremental Synthesis Flow ###
#####

## You must indicate which modules have changed (based on the source files
## that have changed) and provide the complete path to each module

## You must also specify the list of design files and the target Altera
## technology being used

# Read the design source files.
read <list of design files separated by spaces (such as block1.v block2.v)>

# Get the list of modified modules in bottom-up "depth first search" order
# where the lower-level blocks are listed first (these should be modules
# that had LogicLock assignments and separate EDIF netlist files in the
# first pass and had their source code modified)

set list_of_modified_modules {.work.<block2>.INTERFACE .work.<block1>.INTERFACE}

foreach module $list_of_modified_modules {
    set err_rc [regexp {\.(.*)\.(.*)\.(.*)} $module unused lib module_name arch]
    present_design $module

    # Run optimization, preserving hierarchy. You must specify a technology.
    optimize -ta <technology> -hierarchy preserve

    # Ensure that the lower-level module is not optimized again when
    # optimizing higher-level modules.
    dont_touch $module
}

foreach module $list_of_modified_modules {
    set err_rc [regexp {\.(.*)\.(.*)\.(.*)} $module unused lib module_name arch]
    present_design $module
    undont_touch $module
    auto_write $module_name.edf
    # Ensure that the lower-level module is not written out in the EDIF file
    # of the higher-level module.
    noopt $module
}

```

Running the Tcl Script File in LeonardoSpectrum

When you have modified the Tcl script, as described in [“Modifications Required for the LogicLock_Incremental.tcl Script File” on page 12–24](#), you can compile your design using the script.

You can run the script in batch mode at the command line prompt using the following command:

```
spectrum -file <Tcl_file> ↵
```

To run the script from the interface, on the File menu, click **Run Script**, then browse to your .tcl file and click **Open**.

The LogicLock incremental design flow uses module-based design to help you preserve performance of modules and have control over placement. By tagging the modules that require separate .edif files, you can make multiple .edif files for use with the Quartus II software from a single LeonardoSpectrum software project.

Conclusion

Advanced synthesis is an important part of the design flow. Taking advantage of the Mentor Graphics LeonardoSpectrum software and the Quartus II design flow allows you to control how your design files are prepared for the Quartus II place-and-route process, as well as to improve performance and optimize a design for use with Altera devices. The methodologies outlined in this chapter can help optimize a design to achieve performance goals and save design time.

Referenced Documents

This chapter references the following documents:


- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*
- *LeonardoSpectrum Installation Guide* and the *LeonardoSpectrum User's Manual*.
- *Mentor Graphics Precision RTL Synthesis Support* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 12-6 shows the revision history of this chapter.

Table 12-6. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	<ul style="list-style-type: none"> ■ No change to content. ■ Chapter 12 was previously Chapter 11 in software release 8.1. 	Updated for the Quartus II 9.0 software release.
November 2008 v8.1.0	<ul style="list-style-type: none"> ■ Changed to 8-1/2" x 11" page size. ■ Updated Table 12-3. 	Updated for the Quartus II 8.1 software release.
May 2008 v8.0.0	Updated date and part number and added hypertext links.	—

 For previous versions of the *Quartus II Handbook*, refer to the [Quartus II Handbook Archive](#).