

Introduction

As programmable logic devices (PLDs) increase in density and complexity, it is essential for PLD vendors and EDA companies to provide designers with the software and methodology required to make the design task as efficient as possible. As the demand for performance increases, designers must construct their designs for maximum logic optimization.

Achieving better performance in FLEX® 10K devices is possible by using VHDL and Verilog HDL coding techniques, Exemplar Logic Leonardo Spectrum software constraints, and MAX+PLUS® II software options. Using these tools can help you streamline your design, optimize it for Altera® FLEX 10K devices, and increase the speed of an already high-performance programmable logic family. This application note documents the design methodology and techniques for achieving better performance in FLEX 10K devices using the Altera/Exemplar Logic design flow.



You should have a basic understanding of the FLEX 10K architecture, Altera's MAX+PLUS II development system, and Exemplar Logic's Leonardo Spectrum synthesis tool before reading this document.



For information on how to use the Leonardo Spectrum software with the MAX+PLUS II software, refer to the MAX+PLUS II Altera Commitment to Cooperative Engineering Solutions (ACCESSSM) Key Guidelines, which are available in the Altera Technical Support (AtlasSM) section of the Altera web site at <http://www.altera.com>, and on the *MAX+PLUS II Programmable Logic Development Software CD-ROM* (versions 8.2 and higher).

Contents

This application note provides information on the following topics:

Design Flow	2
Effective HDL Design Techniques.....	3
HDL Design Methodology	4
Design Partitioning.....	8
Combinatorial Logic.....	10
Sequential Logic.....	10
Internally Generated Gated Clocks.....	12
State Machine Synthesis.....	13

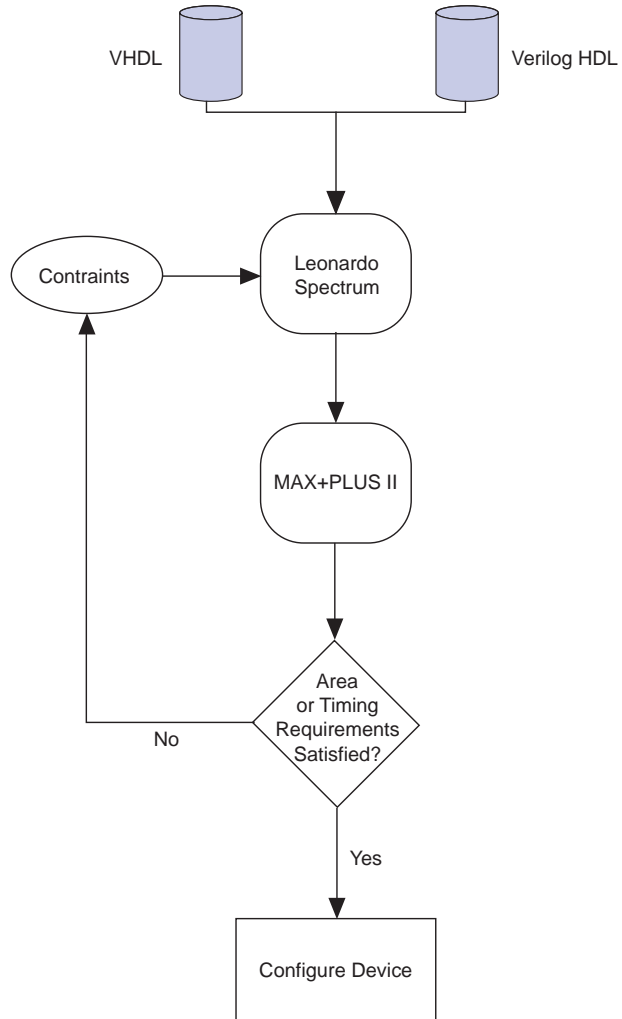
Module Generation.....	15
Inferring Memory Blocks.....	19
Asynchronous Feedback Loops.....	20
Setting Leonardo Spectrum Constraints.....	20
Clock Constraints.....	21
Input Arrival Time.....	22
Output Required Time.....	23
Multi-Cycle Path Constraints.....	24
False Path Constraints.....	24
Purely Combinatorial Designs.....	25
Mixed Designs.....	25
Optimization Strategies.....	27
Area Optimization Strategy.....	28
Timing Optimization Strategy.....	28
Altera-Specific Optimization.....	29
MAX+PLUS II Options for High Performance.....	34
Synthesis Style.....	34
Using the Fast I/O Logic Option.....	35
Timing-Driven Compilation.....	36
Pin Locking.....	37
Leonardo Spectrum Levels.....	38
Leonardo Spectrum Level 1.....	38
Leonardo Spectrum Level 2.....	38
Leonardo Spectrum Level 3.....	39
Conclusion.....	39

Design Flow

The Leonardo Spectrum software accepts hierarchical Verilog HDL or VHDL designs and can map these designs to any Altera FLEX architecture. In addition, you can use the Leonardo Spectrum software to target designs from other PLD or FPGA vendors to Altera FLEX 10K devices.

The design flow begins with the creation of a hardware description language (HDL) design that is imported into the Leonardo Spectrum software for synthesis. The Leonardo Spectrum software then translates the input design into intermediate data structures and performs FLEX-specific optimization. This optimization process, called *fanin-limited optimization*, limits the number of inputs to combinatorial functions. Next, the Leonardo Spectrum software maps the design into look-up tables (LUTs) and attaches a `lut_function` equation to each LUT. After synthesis, the Leonardo Spectrum software creates an EDIF file to be imported into the MAX+PLUS II software for place-and-route.

Figure 1 shows the recommended design flow when using the Leonardo Spectrum software and the MAX+PLUS II software.

Figure 1. Recommended Design Flow

Effective HDL Design Techniques

By practicing good HDL design techniques, you can streamline your design, optimize logic, reduce logic delay, and improve overall performance. The following sections describe how to achieve these results.



The procedures outlined in the following sections use specific Leonardo Spectrum shell commands. However, you can also access all of these commands through the toolbar or pull-down menus in the Leonardo Spectrum software.

HDL Design Methodology

Choosing the appropriate design methodology can significantly improve performance and speed up your design cycle. In general, your design can be structural or behavioral. Most HDL-based designs use either a top-down or bottom-up design methodology. In top-down designs, a single optimization is applied to the top level of the design; in bottom-up designs, optimizations are performed on individual sub-blocks and then the design is stitched together. This section discusses the merits of both methodologies and provides general guidelines to improve the quality of your synthesis results.

Top-Down Design Methodology

In a top-down design methodology, the design starts with the function of the root (or the top-level block). The design is then partitioned into a set of lower-level primitives or blocks until the leaf-nodes (or the bottom) of the design is reached. For guidelines on how to break up, or partition, designs efficiently, see [“Design Partitioning” on page 8](#).

The following sections provide guidelines for optimizing top-down designs for Leonardo Spectrum synthesis.

Optimize Area-Critical Blocks

When creating a top-down design for synthesis in the Leonardo Spectrum software, individual hierarchy blocks should not exceed 50,000 gates. You can limit the block size and optimize results by flattening the sub-blocks. Although preserving the hierarchy of smaller sub-blocks achieves faster optimizations, flattening the sub-blocks usually provides higher performance and a smaller area. The following steps explain how to optimize a top-down design’s area-critical blocks using the Leonardo Spectrum software.

1. Read in the entire design and perform an area optimization using the following command.

```
optimize -target flex10 -area -chip -effort ←
```

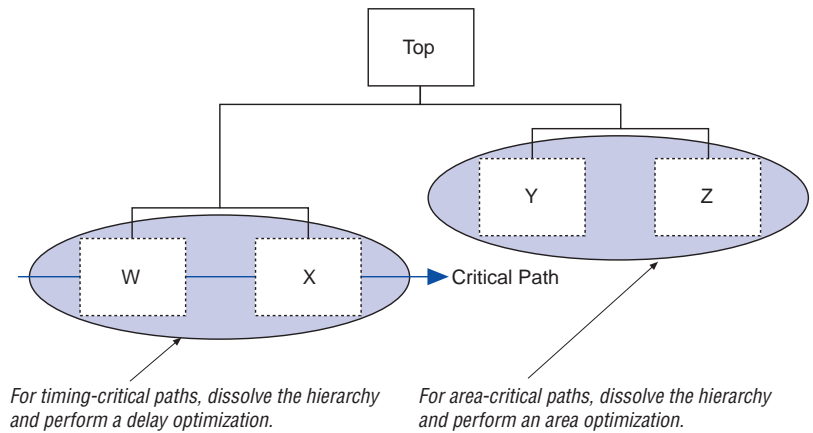
2. Generate area and timing reports. If both are acceptable, your design is complete; if your design requires further optimization, proceed to steps 3 and 4.
3. Dissolve the design by using the `ungroup -all -hier` command on blocks of up to 50,000 gates.

4. Re-optimize the design for area using the `optimize` command. Re-optimizing the design typically provides the best results, but can be time consuming for large designs.

Optimize Timing-Critical Blocks

The goal with timing-critical designs is to generate the smallest circuit possible that still meets the design's timing requirements. To achieve this goal, limit the use of delay and timing optimization to the critical blocks only. See [Figure 2](#).

Figure 2. Limiting Delay & Timing Optimization to Critical Blocks



Using a sample design file, the following steps explain how to limit the delay and timing optimization to the timing-critical blocks only.

1. After initial hierarchical optimization, identify critical paths from the timing report.
2. Use the `group` command to combine all timing-critical blocks into one hierarchical block and all non-timing critical blocks into a second hierarchical block.

```
LEONARDO{ }group w x -inst_name timing_critical ←
LEONARDO{ }group y z -inst_name area_critical ←
```

3. Use the `ungroup -all -hier` command to dissolve all hierarchy beneath the `timing_critical` block. This command dissolves the lower-level hierarchy, while preserving the top-level hierarchy.

```
LEONARDO{ }present_design work.timing_critical ←  
LEONARDO{ }ungroup -all -hier ←
```

4. Perform a delay optimization on the `timing_critical` sub-block. Use the macro switch to prevent the I/O buffers from being placed in the ports of each sub-block.

```
LEONARDO{ }optimize -target flex10 -delay -effort  
standard -macro ←
```

5. Repeat steps 3 and 4 with the `area_critical` block, except perform an area optimization.

```
LEONARDO{ }set present_design work.area_critical ←  
LEONARDO{ }ungroup -all -hier ←  
LEONARDO{ }optimize -hier -target flex10 -area -effort  
standard -macro ←  
LEONARDO{ }set present_design work.top ←
```

6. Save the netlist file.

```
LEONARDO{ }auto_write -format edif top.edf ←
```

Bottom-Up Design Methodology

Typically, bottom-up design methodologies are used when creating a team design or when working with extremely large designs. When using a bottom-up design methodology, the design begins with knowledge of the root and is then partitioned based on which primitives are available as leaf-nodes. Optimizations are performed on the individual sub-blocks and then the design is stitched together. For guidelines on how to partition designs efficiently, see [“Design Partitioning” on page 8](#).

The following sections provide guidelines for optimizing bottom-up designs for Leonardo Spectrum synthesis.

Register Placement Within Blocks

When creating a bottom-up design for synthesis in the Leonardo Spectrum software, you should place registers at the front or the back of a hierarchical boundary only; you should not place registers in both the front and back of a hierarchical boundary, because registers and hierarchical boundaries constrain optimization. If this design practice is followed, preserving hierarchy in a design will not impact optimization results and will allow much faster synthesis times.

Constraining Sub-Blocks for Timing

Ideally, only registers should be placed at hierarchical boundaries. However, because random logic is often placed at hierarchical boundaries, logic must be constrained appropriately. Unless more detailed information about the sub-block timing is known, you should apply constraints equal to half the clock period to the boundary. If both boundary sides meet the timing requirements, the combined blocks will meet the timing requirements.



You do not need to specify loading and drive constraints on sub-block pins for FLEX devices.

Saving Intermediate Netlist Files

In the Leonardo Spectrum software, you can use either of the following two commands to save design netlist files:

- `auto_write -format edif <filename>.edif`
- `write -format xdb <filename>.xdb`

The `auto_write -format edif` command invokes an architecture-specific netlist post-processor that modifies the design for seamless integration with the MAX+PLUS II software. Therefore, you should use the `write -format` command rather than the `auto_write -format edif` command to save intermediate netlist files. You should only use the `auto_write -format edif` command when creating a netlist file for the MAX+PLUS II place-and-route environment. Exemplar Logic recommends saving intermediate results as binary XDB files. This action saves both the netlist and timing data.

Design Stitching

Design stitching refers to the process of building up the entire design after performing bottom-up optimizations on individual sub-blocks. The Leonardo Spectrum software can connect sub-blocks with top-level structural netlists automatically, as long as all instance names, port names, and view names match.

Designs should be stitched bottom-up (i.e., all optimized lower-level blocks should be read into the Leonardo Spectrum software first). Then, synthesize the top-level structural VHDL or Verilog HDL file that connects the sub-blocks together using the following commands:

```
LEONARDO{ }read -format xdb A.xdb B.xdb C.xdb ←  
LEONARDO{ }read -format vhd1 top.vhd1 ←
```



View names between the sub-blocks and the instances contained within the top-level structural code must match exactly for bottom-up design stitching to be successful. Generally, problems are easiest to resolve by modifying the EDIF, VHDL, or Verilog HDL source code. Additionally, you can modify the view names by using the `add_rename_rule` command.

Final Optimization

Once a design has been stitched together, you should generate final area and timing reports. If the design meets your timing specification, run one final optimization to add the chip I/O buffers. To perform final optimizations, use the following command:

```
LEONARDO{ }optimize -target flex10 -chip -area  
-no_hierarchy ←
```

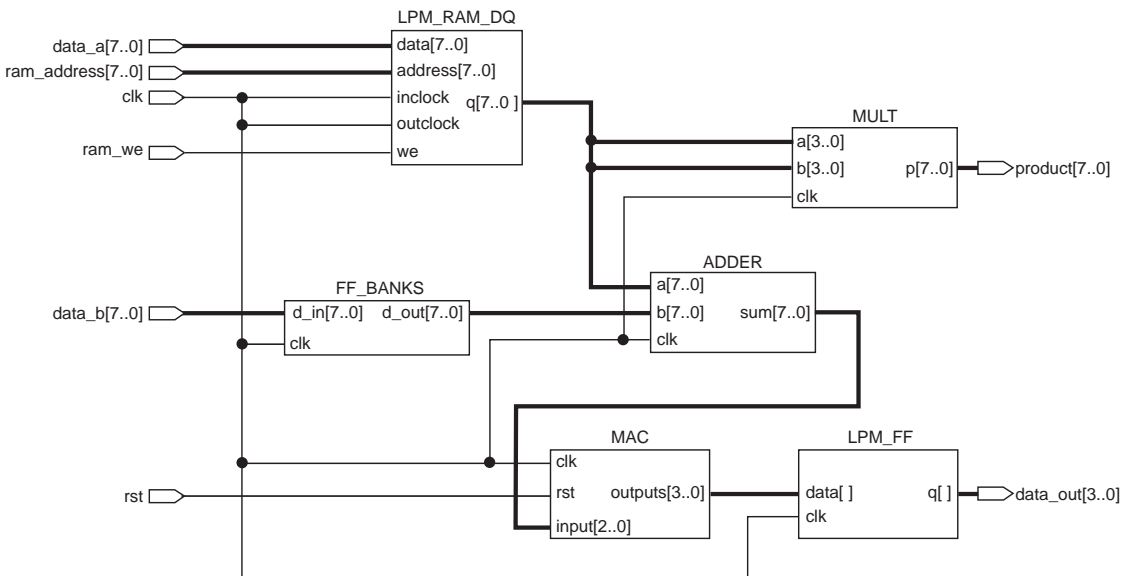
Design Partitioning

Many designs are too large to create in a single design file that incorporates all the functionality of the design. The Leonardo Spectrum software allows you to create multiple design files and then link the files together in a hierarchy. This structure allows you to simulate and optimize sub-designs rather than optimize the entire design. When partitioning your design, use the following guidelines:

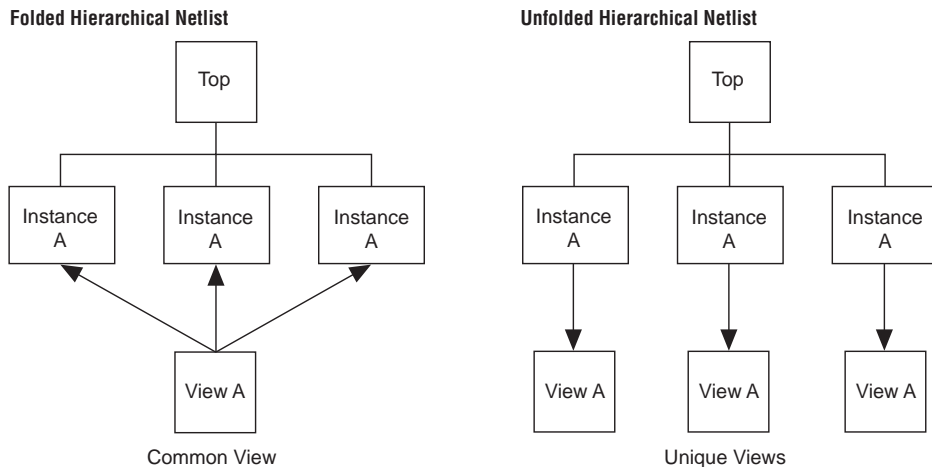
- Partition the design at functional boundaries. Block diagrams or high-level schematics help create natural boundaries, as shown in [Figure 3](#). For example, state machines, data paths, decoder logic, memory elements, and megafunctions all form natural boundaries.
- Do not use “glue logic” between hierarchical blocks. If you preserve hierarchy boundaries, glue logic is not merged with hierarchical blocks. The Leonardo Spectrum software optimizes glue logic separately, which can degrade synthesis results.
- Limit the size of sub-blocks to 10,000 to 50,000 gates for easier design debugging during functional simulation. You can flatten the sub-blocks to limit the block size.
- Limit clocks to one per block. The Leonardo Spectrum software does not support multiple asynchronous clocks per block.

- Place state machines in separate hierarchy blocks to speed optimization and provide greater control over encoding.
- Separate timing-critical blocks from non-timing critical blocks. The Leonardo Spectrum software performs timing and area optimizations separately. Pay close attention to blocks that may lend themselves to special area or delay optimizations.
- Limit the critical path to one hierarchical block. You can group the logic from several blocks to ensure the critical path resides in one block.
- Register all inputs and outputs of a block to simplify the synthesis process. Because outputs are registered, this step eliminates the need to specify output required times. In addition, because all logic is synchronous, glitches are avoided.

Figure 3. Diagram Detailing Hierarchy



By default, the Leonardo Spectrum software preserves hierarchy in a design. To ensure the fastest possible run times, the netlist file is “folded,” which means all common sub-blocks reference a single netlist and all blocks are optimized the same way for the worst-case conditions. The Leonardo Spectrum software only optimizes the netlist once. If you want to optimize two instances of a common sub-block differently, the netlist must be unfolded. For example, you can optimize one block for area and a second block for delay. Figure 4 shows the structure of a folded and unfolded hierarchical netlist.

Figure 4. *Folded & Unfolded Hierarchical Netlists*

Combinatorial Logic

Logic is described as combinatorial if outputs at a specified time are a function of the inputs at that time only, regardless of the previous state of the circuit. Examples of combinatorial logic functions are decoders, multiplexers, and adders.

You can optimize performance results obtained during synthesis by using registers in your designs. Because FLEX devices have registers rather than latches built into the silicon, designing with latches generates more logic and lower performance than designing with registers. For example, the MAX+PLUS II software uses two logic elements (LEs) to create a latch.

When designing combinatorial logic, you should avoid creating a latch unintentionally due to your HDL design style. For example, when Case or If Statements do not cover all possible input conditions, combinatorial feedback can generate latches. A latch is generated when the final Else Clause or When Others Clause is omitted from an If or Case Statement, respectively.

Sequential Logic

Logic is sequential if the outputs at a specified time are a function of the inputs at that time and at all preceding times. All sequential circuits must include one or more registers (i.e., flipflops). Each FLEX LE contains a D-type flipflop; thus, pipelining does not use any additional resources. If you want to create a break in the logic or store a value within your design, extra LEs or routing resources are not required.

You should avoid creating a feedback multiplexer unintentionally. A feedback multiplexer is created if all possible input conditions are not assigned when using If Statements. For example, if you omit the final Else Clause in Figure 5 (highlighted in blue), a feedback multiplexer is generated and the function requires four LEs. If you include the final Else Clause, the feedback multiplexer is removed and this function requires only three LEs.

Figure 5. VHDL Code that Prevents Feedback Multiplexer Generation

```

LIBRARY ieee;
USE IEEE.std_logic_1164.ALL;

ENTITY seq2 IS
    PORT ( a,b,c,d,clk,rst : IN STD_LOGIC;
          sel                : STD_LOGIC_VECTOR(3 DOWNTO 0);
          oput              : OUT STD_LOGIC);
END seq2;

ARCHITECTURE behave OF seq2 IS
BEGIN
    PROCESS(clk, rst)
    BEGIN
        IF rst = '1' THEN
            oput <= '1';
        ELSIF clk='1' AND clk'EVENT THEN
            IF sel(0) = '1' THEN
                oput <= a AND b;
            ELSIF sel(1) = '1' THEN
                oput <= b;
            ELSIF sel(2) = '1' THEN
                oput <= c;
            ELSIF sel(3) = '1' THEN
                oput <= d;
            ELSE
                oput<= 'x'; -- removes feedback
                           -- multiplexer
            END IF;
        END IF;
    END PROCESS;
END behave;

```

Internally Generated Gated Clocks

If possible, you should avoid using internally generated gated clocks. Internally generated gated clocks create logic delays and clock skew, using additional routing resources within FLEX devices. Internally generated clocks may also be subject to glitching, creating functional problems. In addition, a limited number of clocks are available per logic array block (LAB).

If you must implement an internally generated gated clock in your design, use the GLOBAL primitive to place the clock on one of the high-fan-out internal global signals. Figure 6 shows a sample VHDL design that implements an internally generated gated clock using the GLOBAL primitive (highlighted in blue).

Figure 6. Implementing an Internally Generated Gated Clock in VHDL (Part 1 of 2)

```

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY gate IS
    PORT
    (
        a,b      : IN    STD_LOGIC;
        c,d      : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
        outputn  : OUT   STD_LOGIC_VECTOR(3 DOWNTO 0)
    );
END GATE;

ARCHITECTURE a OF gate IS
    SIGNAL clock      : STD_LOGIC;
    SIGNAL gclk       : STD_LOGIC;
    SIGNAL count      : STD_LOGIC_VECTOR(3 DOWNTO 0);
    ATTRIBUTE noopt  : BOOLEAN;

    COMPONENT GLOBAL
        PORT ( \in\    : IN STD_LOGIC;
              \OUT\   : OUT STD_LOGIC );
    END COMPONENT;

    ATTRIBUTE noopt OF GLOBAL: COMPONENT IS TRUE;

```

Figure 6. Implementing an Internally Generated Gated Clock in VHDL (Part 2 of 2)

```
BEGIN
    clock <= a AND b;
    clk_buf: GLOBAL PORT MAP (clock, gclk);

PROCESS (gclk) BEGIN
    IF gclk='1' AND gclk'EVENT THEN
        count <= c + d;
    END IF;
    END PROCESS;
    outputn <= count;
END a;
```

State Machine Synthesis

The Leonardo Spectrum software encodes state machines during the synthesis process. Once 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 Leonardo Spectrum software to identify the state machine.



Although not required, Altera recommends isolating state machines into separate hierarchical blocks. This separation optimizes performance and allows easy modifications to state machine encoding.

The Leonardo Spectrum software supports the following state machine encoding styles:

- *Binary*—Generates state machines with the fewest possible flipflops. Binary state machines are useful for area-critical designs when timing is not a concern.
- *Gray*—Generates state machines where only one flipflop changes during each transition. Gray encoded state machines tend to be glitchless.
- *Random*—Generates state machines using random state machine encoding. Random state machine encoding should only be used when all other implementations are not achieving the desired results.
- *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.

You can instruct the Leonardo Spectrum software to use a particular state machine encoding style by setting VHDL attributes, using Verilog HDL pragmas, or using the `set` encoding variable.

Setting VHDL Attributes

To set the state machine encoding style using VHDL, insert the following statements into your code:

```
--Declare the type_encoding_style attributes
type_encoding_style is (BINARY, ONEHOT, GRAY, RANDOM);
ATTRIBUTE TYPE_ENCODING_STYLE: ONEHOT;

--Declare your state machine enumeration type
type_my_state_type is (s0,s1,s2,s3,s4);

--Set the type_encoding_style of the state type
ATTRIBUTE type_encoding_style of my_state_type is ONEHOT;
```

Using Verilog HDL Pragmas

To set the state machine encoding style using Verilog HDL, insert the following comment text into your Verilog HDL model, above the state machine model:

```
parameter[3:0]//pragma enum state_parameters onehot
idle=4'b0001,
halt=4'b0010
run=4'b0100
stop=4'b1000;
reg[3:0]/*pragma enum state_parameters*/state;
```



The first line of the sample code above specifies one-hot state machine encoding. However, you can set this optional specification to binary, gray, or random. The encoding default is one-hot, but can be changed via the set encoding variable.

Using the set encoding Variable

You can also set the state machine encoding style by using the set encoding variable prior to reading in the VHDL or Verilog HDL code. Once set, all subsequent state machines will use the specified encoding until another set encoding variable is encountered. [Table 1](#) shows the arguments used with the set encoding variable.

Table 1. set encoding Variable Arguments

Argument	Description
binary	Sets state machine encoding to binary
onehot	Sets state machine encoding to one-hot
gray	Sets state machine encoding to gray
random	Sets state machine encoding to random

VHDL:

```
LEONARDO{}set encoding onehot ←
LEONARDO{}read uart_control_sm.vhdl ←
```

Verilog HDL:

```
LEONARDO{}set encoding binary ←
LEONARDO{}read -format verilog control.v ←
```



VHDL attributes and Verilog HDL pragmas override the set encoding variable.

Module Generation

Traditionally, arithmetic and relational logic, commonly known as data path logic, has been difficult to synthesize with logic synthesis software. Exemplar Logic's **modgen** utility and the library of parameterized modules (LPMs) make it easier to perform architecture-specific data path synthesis.

modgen vs. Altera-Specific Operators

You can use either the LPM or Leonardo Spectrum's **modgen** feature to generate a function. The easiest method is to infer an operator in the Leonardo Spectrum software by using an arithmetic or logic operator symbol in VHDL or Verilog HDL. For example, in the code `sum <= a + b`, **modgen** recognizes the + symbol and builds an optimized circuit. Alternatively, you can instantiate a module cell directly into HDL code. The cell is passed with the EDIF netlist file to the MAX+PLUS II software.

modgen

The Leonardo Spectrum software supports various architecture-specific implementations of arithmetic and relational operators used in VHDL or Verilog HDL. Because these implementations are optimized for a target architecture, the synthesis results are usually smaller, faster, and take less time to compile. Table 2 lists the operators supported for Altera FLEX architectures.

Table 2. Supported Operators		
Operator Type	Symbol	Definition
Relational	=	Equal
	/=	Not equal
	<	Less than
	<=	Less than or equal to
	>	Greater than
	>=	Greater than or equal to
Arithmetic	+	Addition
	-	Subtraction
	*	Multiplication
Miscellaneous Functions	N/A	Counters (up/down, loadable) RAM Incrementer Decrementer Absolute value Unary minus

The module generator for each architecture uses dedicated hardware resources, such as carry and cascade chains, whenever possible. Therefore, the **modgen** implementation of operators, such as addition, subtraction, counters, and relational operators, is generally smaller in area and faster in delay.

The following list shows how **modgen** implements various operators in FLEX 10K devices:

- Adders are implemented in FLEX devices using dedicated carry chains, which leads to very fast carry propagation and results in excellent timing performance.
- Counters are implemented in FLEX 10K devices using counter modes, which results in faster and smaller designs.
- RAM is implemented in FLEX 10K devices using dedicated RAM blocks.

You can control whether **modgen** implements operators that are optimized for delay or area through a user-defined switch. This switch is set in Leonardo Spectrum's **Optimize** dialog box. The choices available in the **Optimize** dialog box depend on your architecture and format.

If the *Use Technology Specific Module Generation Library* option is turned on in the **Optimize** dialog box, you can control the type of optimization using the options in Table 3. If this option is turned off, **modgen** uses the default module generation library.

Table 3. Leonardo Spectrum Options

Option	Description	Setting (On/Off)	Interactive Shell Option Level 3	Batch Mode
Auto	Picks the smallest implementation if performing an area optimization; picks the fastest implementation for a delay optimization.	On (1)	set modgen_select auto	-select_modgen=auto
Smallest	Picks the best compact implementation available.	Off	set modgen_select smallest	-select_modgen=smallest
Small	Picks a compact implementation.	Off	set modgen_select small	-select_modgen=small
Fast	Picks a fast implementation.	Off	set modgen_select fast	-select_modgen=fast
Fastest	Picks the fastest implementation available.	Off	set modgen_select fastest	-select_modgen=fastest

Note:

(1) Graphical user interface (GUI) default option. You can also enter interactive shell and batch mode options.

LPM Functions

Altera offers module generators for creating arithmetic, RAM, and counter logic through the LPM. These module generators are optimized for the Altera FLEX architecture.

During Leonardo Spectrum synthesis, LPM functions are compiled as black boxes. The LPM function's parameter values are passed to the EDIF file or Text Design File (.tdf) as Verilog HDL meta comments or as VHDL attributes.

Figure 7 shows a VHDL design that uses the `lpm_count` function to create a counter.

Figure 7. Instantiating an LPM Function in VHDL

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

ENTITY count4 IS
PORT (d          : IN  STD_LOGIC_VECTOR(3 DOWNTO 0);
      clk, clr   : IN  STD_LOGIC;
      result     : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END count4;

ARCHITECTURE lpm OF count4 IS

COMPONENT lpm_counter
  GENERIC (lpm_width  : POSITIVE;
          lpm_type    : STRING := "lpm_counter");
  PORT (data         : IN  STD_LOGIC_VECTOR(lpm_width-1 DOWNTO 0);
        clock        : IN  STD_LOGIC;
        aclr         : IN  STD_LOGIC;
        q            : OUT STD_LOGIC_VECTOR(lpm_width-1 DOWNTO 0));
END COMPONENT;

BEGIN
  u1: lpm_counter
    GENERIC MAP (lpm_width=>4);
    PORT MAP (data=>d, aclr=>clr, clock=>clk, q=>result);
END lpm;
```

Figure 8 shows a Verilog HDL design that uses the `lpm_add_sub` function to create an adder.

Figure 8. Instantiating an LPM Function in Verilog HDL

```

module accumulator (dataaa, datab, result);

    input [11:0] dataaa;
    input [11:0] datab;
    output [11:0] result;
    wire [11:0] result;

    lpm_add_sub #(
        12,                //LPM_WIDTH
        "SIGNED",         //LPM_REPRESENTATION
        "ADD",            //LPM_DIRECTION
        0,                //LPM_PIPELINE
        "LPM_ADD_SUB",   //LPM_TYPE
        "UNUSED",        //LPM_HINT
    ) lpm_add_sub_component (.dataaa(dataaa), .datab(datab), .result(result));

endmodule

module lpm_add_sub (clock, dataaa, datab, add_sub, result);

    parameter LPM_WIDTH = 32;
    parameter LPM_REPRESENTATION = "SIGNED";
    parameter LPM_DIRECTION = "ADD";
    parameter LPM_PIPELINE = 0;
    parameter LPM_TYPE = "LPM_ADD_SUB";
    parameter LPM_HINT = "UNUSED";

    input [31:0] dataaa, datab;
    input clock, add_sub;
    output [31:0] result;

endmodule // lpm_add_sub

```



See MAX+PLUS II Help for complete details on the LPM functions supported by the MAX+PLUS II software.

Inferring Memory Blocks

The Leonardo Spectrum software can infer RAM blocks from register transfer level (RTL) code. When RAM blocks are modeled as a two-dimensional array, the Leonardo Spectrum software recognizes the function and inserts a black box into the netlist with attached properties. The Altera MAX+PLUS II software then recognizes the properties and inserts the appropriate RAM block into the design.


You can create timing arcs by creating an auxiliary library and defining the RAM cell timing. For more information on creating timing arcs, go to Exemplar Logic's web site at <http://www.exemplar.com/support>.

Asynchronous Feedback Loops


When timing loops occur, timing analysis can be extremely slow because the Leonardo Spectrum software must evaluate each loop 5,000 times before the software concludes that a loop exists and moves on. Therefore, Altera recommends that you redefine the `delay_break_loops` variable to `TRUE` at the start of each session. This definition causes loops to break automatically and warning messages to be written to the timing report.

Setting constraints within the Leonardo Spectrum software is easy. Constraints can be as simple as specifying the target design frequency or as powerful as indicating multi-cycle paths between flipflops. Timing constraints indicate desired target arrival and required times used for setup and hold analysis.

Constraints should be applied after the design has been read into the Leonardo Spectrum software but before optimization. The Leonardo Spectrum software assumes intuitive defaults. At a minimum, you should define the clock, input port arrival times, and output port required times.

 You should not over-constrain the design. Doing so may have undesirable effects such as increasing the design size and optimization run times.

The following sections describe constraints and other options you can use in the Leonardo Spectrum software to achieve optimum performance.

 The following sections use specific Leonardo Spectrum shell commands to set design constraints. However, you can access all of these commands through the constraint editor in the Leonardo Spectrum software.

The easiest way to constrain a design is to specify a global timing constraint. For example, to set a design's maximum frequency to 20 MHz, you can set the clock period to 50 ns using the following command:

```
LEONARDO{ } set register2register 50 ←
```

Setting Leonardo Spectrum Constraints

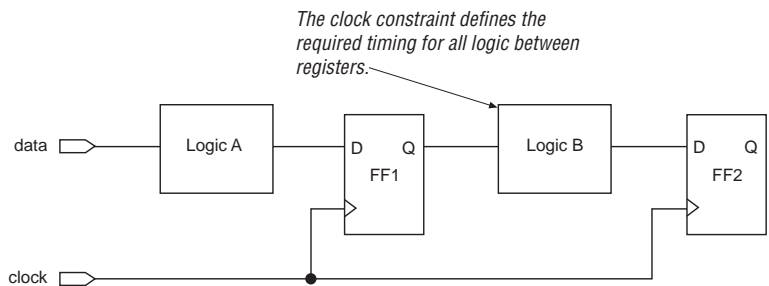
Similarly, you can use the following commands to specify other maximum delay values for the design.

```
LEONARDO{ } ←
  set input2register 50 ←
  set input2output 50 ←
  set register2output 50 ←
```

Clock Constraints

Clocks define the timing to and from registers (see [Figure 9](#)). Without defined clocks, all registers are assumed to be unconstrained. Therefore, all combinatorial logic between registers is ignored during timing optimization. When you define a clock, you effectively constrain the combinatorial logic between all registers to one clock period.

Figure 9. Design Showing a Clock Constraint

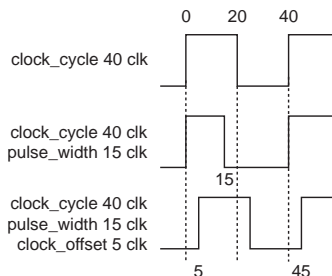


In [Figure 9](#), logic B is constrained to one clock period. Therefore, if the clock period is 50 ns, logic B has roughly a 50 ns setup for FF2 to meet timing.

You define clock constraints in the Leonardo Spectrum software by using three basic commands:

```
clock_cycle <clock period> <primary input port> ←
pulse_width <clock pulse width> <primary input port> ←
clock_offset <clock offset> <primary input port> ←
```

By default, the clock network is assumed to be ideal (i.e., without a delay). Thus, the clock arrives at all flipflops at the same time. To change the clock network to a propagated delay, set the `propagate_clock_delay` variable to `TRUE`. [Figure 10](#) shows the timing waveforms for a sample design.

Figure 10. Sample Timing Waveforms

In [Figure 10](#), the clock period for the first waveform is 40 ns and is attached to the `clk` port. Because the default duty cycle is 50%, the clock pulse width for the first waveform is 20 ns. The second waveform shows a pulse width of 15 ns. The third waveform demonstrates an offset clock, which is useful for specifying a clock skew relative to zero.

Input Arrival Time

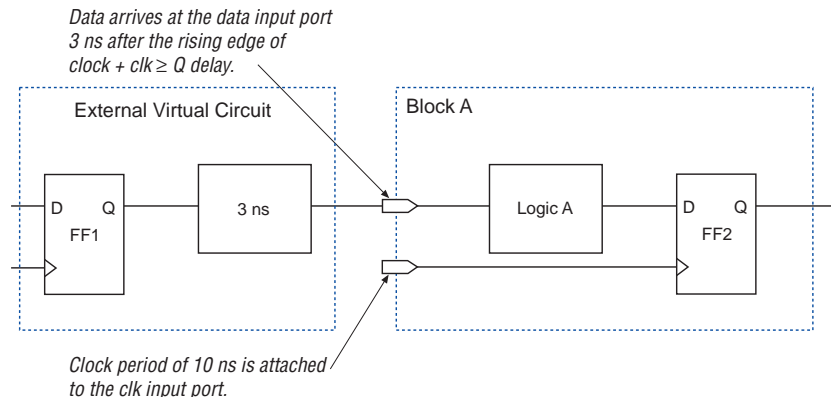
The input arrival time specifies the maximum delay to the synthesized design's input port through external logic. You can specify the input arrival time by using the `arrival_time` command. For example, to set the input arrival time for block A in [Figure 11](#) to 3 ns, you should use the following command:

```
LEONARDO{ }arrival_time 3 data_min ←
```

If the clock period in [Figure 11](#) is 10 ns, the setup of FF2 added to the combinatorial delay of logic A would need to be 7 ns to meet the timing requirement.

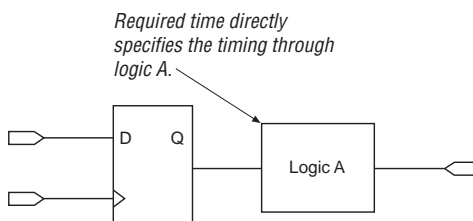


All input arrival times start at time zero and cannot be specified relative to a particular clock edge. To adjust the input arrival time to a particular clock edge, you must add the clock offset to the arrival time.

Figure 11. Design with Input Arrival Time Constraints

Output Required Time

The output required time specifies the data required time on output ports (see [Figure 12](#)). Time is always specified with respect to time zero (i.e., output required time cannot be specified relative to a particular clock edge).

Figure 12. Design Showing an Output Required Time Constraint

You can use the following command to specify the output required time:

```
LEONARDO{ }required_time <integer> portname ←
```

Multi-Cycle Path Constraints

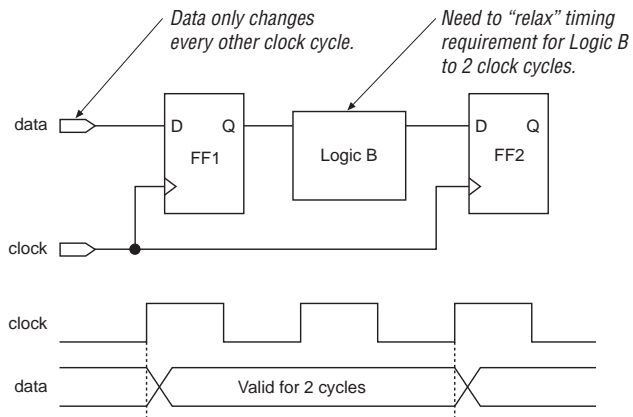
In Leonardo Spectrum software versions 4.2 and higher, you can constrain individual paths to more than one cycle by using the multi-cycle command. For example, to constrain the clock period of logic B in [Figure 13](#) to 3 ns, you should use the following command:

```
LEONARDO{ }set_multicycle_path -from{FF1} -to{FF2}
    -value 3 ←
```



Use multi-cycle constraints sparingly because they slow timing analysis. A few multi-cycle constraints have little effect; many slow timing optimization dramatically.

Figure 13. Design with Multi-Cycle Constraints



False Path Constraints

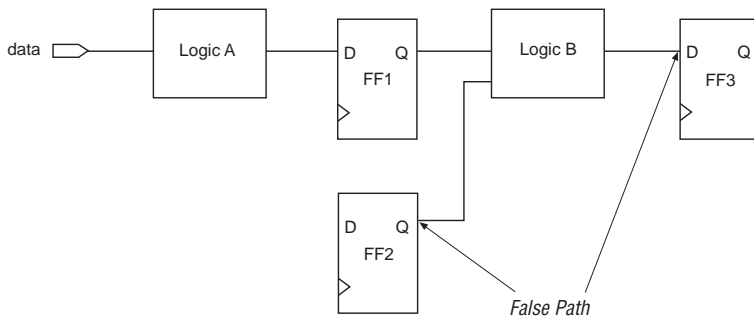
False paths are design paths ignored by the Leonardo Spectrum software during timing optimization. You can specify a path as false through the multi-cycle command. For example, to specify the path from FF2 to FF3 in [Figure 14](#) as false, you should use the following command:

```
LEONARDO{ }set_multicycle_path -value 1000 -from{FF2}
    -to {FF3} ←
```

Essentially, this command constrains the path between FF2 and FF3 to 1,000 clock cycles. Because it is unlikely that logic B would require more than 2,000 cycles, this path can be eliminated from timing optimization and timing analysis.



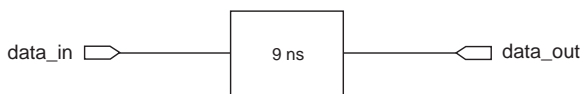
Do not specify too many false paths because it increases the timing analysis run times.

Figure 14. Design with a False Path

Purely Combinatorial Designs

Purely combinatorial designs do not have clocks. Thus, you can constrain these designs by simply using the `maxdly` constraint. For example, to set the maximum propagation delay in [Figure 15](#) to 9 ns, you should use the following command:

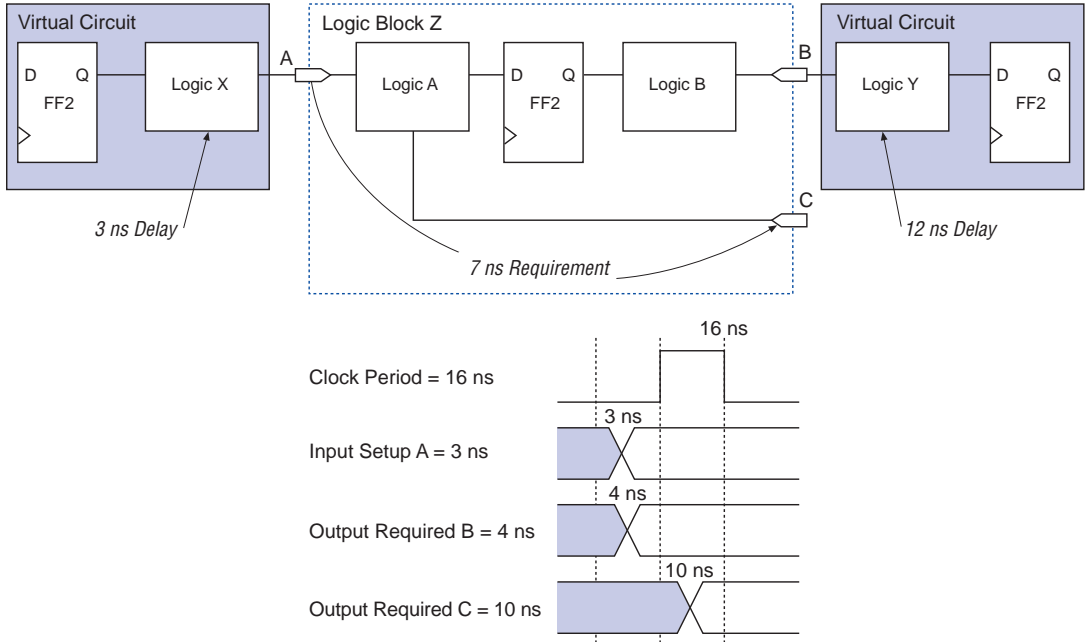
```
LEONARDO{}set input2output 9 ←
```

Figure 15. Design Constrained to a 9-ns Clock Period

Mixed Designs

Some blocks, such as Mealy state machines, have both synchronous and purely combinatorial paths through the circuit. To constrain these cases, you should apply synchronous constraints to the ports of the synchronous paths and asynchronous constraints to the ports of the asynchronous paths. [Figure 16](#) shows a sample mixed design with synchronous and asynchronous constraints.

Figure 16. Sample Constraints for a Mixed Design



The following steps explain how to set constraints on mixed designs, using the sample design shown in [Figure 16](#).

1. Define the clock constraints.

```
LEONARDO{ }clock_cycle 16 clk ←
```

2. Apply an input arrival constraint, assuming the design is purely sequential.

```
LEONARDO{ }arrival_time 3 A ←
```

3. Apply an output required time to the sequential output ports only. Set the constraints for a sequential circuit, ignoring the combinatorial paths for now.

```
LEONARDO{ }required_time 4 B ←
```

- Apply an output arrival time to the combinatorial output paths. The maximum delay constraint applied to these paths is the difference between the input arrival time and the output arrival time. In this example, the input arrival time is set to 3. To have a 7 ns maximum delay through the combinatorial path, you must set the output required time to 10 ns (i.e., $10 \text{ ns} - 3 \text{ ns} = 7 \text{ ns}$).

```
LEONARDO{ }required_time 10 C←
```

Optimization Strategies

The Leonardo Spectrum software can perform two types of design optimizations: area and delay. Due to the routing-dominated nature of PLDs, the smallest design is often the fastest. For this reason, Altera recommends that you perform area optimizations first (i.e., use the MAX+PLUS II software to perform place-and-route and then focus on timing-critical areas).

In the Leonardo Spectrum software, you specify the type of optimization to perform using the `optimize` command. Table 4 shows the arguments used with the `optimize` command.

Argument	Description
<code>-target</code>	Specify the target architecture for the design.
<code>-single_level</code>	Perform optimization on the top level of the hierarchy only.
<code>-effort</code>	Optimization effort: <code>remap</code> , <code>quick</code> , or <code>standard</code> .
<code>-nopass <list></code>	Explicitly avoid an optimization pass.
<code>-chip</code> <code>-macro</code>	<code>-chip</code> inserts I/O buffers, for top-level blocks; <code>-macro</code> does not insert I/O buffers, for sub-blocks.
<code>-pass <list></code>	Explicitly run a pass.
<code>-area</code> <code>-delay</code>	<code>-area</code> optimizes to obtain minimum area (default); <code>-delay</code> optimizes to obtain minimum delay.
<code>-flatten</code>	Dissolve all hierarchy.

The following sections describe how to perform area, timing, and Altera-specific optimizations using the Leonardo Spectrum software.



The optimization strategies outlined in this section use specific Leonardo Spectrum shell commands. However, you can access all of these commands from the toolbar or pull-down menus in the Leonardo Spectrum software.

Area Optimization Strategy

If a design comfortably meets timing requirements and you want the smallest possible circuit, follow the procedure below.

1. Flatten hierarchical blocks that contain up to 50,000 gates using the `present_design` and `ungroup -all -hier` commands.
2. Set the `area_weight` variable to 1 and the `delay_weight` variable to 0. These settings redefine the optimization cost function to favor area over delay.

```
LEONARDO{ }set area_weight 1 ←
LEONARDO{ }set delay_weight 0 ←
```

3. Perform a standard-effort area optimization.

```
LEONARDO{ }optimize -target flex10 -area -effort
standard ←
```

Timing Optimization Strategy

Generally, designs have a combination of timing-critical blocks and non-timing critical blocks. The goal of timing optimization is to create the smallest design possible that still meets your timing specification. The Leonardo Spectrum software has two timing optimization commands. The `optimize -delay` command creates fast structures during the mapping process and runs algorithms designed to reduce logic levels; the `optimize_timing` command performs full constraint-based timing optimization. [Table 5](#) shows the arguments used with the `optimize_timing` command.

Argument	Description
<code>-through <list></code>	Specify an explicit list of end points to optimize.
<code>-single_level</code>	Perform optimization on the top level of hierarchy only.
<code>-force</code>	Force negative slack then optimize.

The following steps describe how to perform a timing optimization:

1. Perform an area optimization with quick effort and hierarchy preserved. The results of this optimization will be used to identify the timing-critical blocks.

```
LEONARDO{ }optimize -target flex10 -area -effort
quick ←
```

2. Generate a timing report to identify the timing-critical blocks.
3. Use the `group` command to combine all critical blocks into one block. Once combined, use the `present_design` and `ungroup -all -hier` commands to dissolve hierarchy within that block.

```
LEONARDO{ }group a b -inst_name ab_instance ←
LEONARDO{ }present_design work.ab_instance ←
LEONARDO{ }ungroup -all -hier ←
```

4. Perform a delay optimization with standard effort on the timing-critical sub-block and generate a timing report.
5. Perform a second standard effort delay optimization if timing is still not met. Continue this procedure until the results no longer change.
6. Set timing constraints and perform an `optimize_timing` command if timing is still not met. This command invokes a second optimization engine that performs constraint-based timing optimization.
7. Set the `top_level` instance as the present design once timing is met in the timing-critical sub-block. Verify that the top-level design meets your timing specification. If not, repeat steps 4 through 6 for other timing-critical blocks. Continue to step 8 if all blocks meet timing.
8. Group all non-critical blocks into a single level of hierarchy and follow the procedure outlined in [“Area Optimization Strategy” on page 28](#).

Altera-Specific Optimization

This section describes how to optimize your design using commands in the Leonardo Spectrum software that are designed specifically for the Altera FLEX 10K architecture.

Making Pin Assignments

You can define I/O signal pins in the Leonardo Spectrum software using commands from the interactive shell or the VHDL design source.

Interactive Shell:

```
pin_number 10 <pathname>
```

VHDL:

```
ATTRIBUTE PIN_NUMBER OF <signal name>: SIGNAL is <value>
```

Figure 17 shows how to make pin assignments using a sample VHDL file.

Figure 17. Making Pin Assignments in VHDL

```
LIBRARY ieee; USE ieee.std_logic_1164.all;
LIBRARY exemplar; USE work.exemplar_1164.all;

ENTITY example IS
  PORT(
    clk : bit;
    din : in std_logic_vector (4 DOWNT0 0)
    q   : out std_logic_vector (4 DOWNT0 0)
  );
  ATTRIBUTE PIN_NUMBER OF clk : SIGNAL IS "1"
  ATTRIBUTE ARRAY_PIN_NUMBER OF din : SIGNAL IS ("2", "3",
    "4", "5", "6");
END example;

ARCHITECTURE exemplar OF example IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF (clk = '1' and clk'event) then
      q <= din;
    END IF
  END PROCESS
END exemplar
```

Look-Up Table Mapping

Look-up table (LUT) mapping places logic in 4-input LUTs and in cascade gates to minimize the total number of LUTs or the delay. In the output EDIF netlist file, the LUT boundaries are marked with LCELL buffers.

Figure 18 shows sample code for a 4-to-1 multiplexer.

Figure 18. Verilog HDL Code for a 4-to-1 Multiplexer

```
module mux4 (out, in, sel);  
    output out;  
    input[3..0] in;  
    input[1..0] sel;  
    assign out = in[sel];  
end module
```

LUT mapping maps the multiplexer in Figure 18 to two 4-input LUTs and one cascade gate. The Leonardo Spectrum software then decomposes the LUTs to AND-OR gates for output to the MAX+PLUS II software.

Implement in Embedded Array Blocks

The Leonardo Spectrum software allows you to implement wide functions, such as multiplier and arithmetic logic unit (ALU) functions, in FLEX 10K EABs. You can access EABs with the Leonardo Spectrum software by turning on the `implement_in_eab` attribute for the desired instance. You can set this attribute in either VHDL or Verilog HDL designs. Figure 19 shows VHDL code for implementing a sample function in an EAB.

Figure 19. VHDL Code to Implement a Function in an EAB

```

ENTITY mult IS
    PORT (A,B: INTEGER RANGE 0 TO 255);
        Q: OUT INTEGER RANGE 0 TO 255);
END mult;

ARCHITECTURE BEHAVIOR OF mult IS
BEGIN
    Q <= A * B
END BEHAVIOR

ENTITY eab_test IS
    PORT (CLK,MAC,RST:bit; A,B: INTEGER RANGE 0 TO 15;
        Q: BUFFER INTEGER RANGE 0 TO 255);
END eab_test;

ARCHITECTURE BEHAVIOR OF eab_test IS
    SIGNAL P: integer range 0 to 255;

COMPONENT mult
    PORT (A,B: IN INTEGER RANGE 0 TO 15;
        Q: OUT INTEGER RANGE 0 TO 255);
END COMPONENT;

ATTRIBUTE logic_option:STRING;
ATTRIBUTE noopt:BOOLEAN;
ATTRIBUTE logic_option OF u1:LABEL IS
    "implement_in_eab=on";
ATTRIBUTE NOOPT of u1:LABEL IS TRUE;

BEGIN
    u1:mult PORT MAP (A,B,P); --Product of A and B
    PROCESS (RST,CLK)
    BEGIN
        IF (RST='1') THEN --Reset
            Q <= 0;
        ELSE
            IF (CLK='1' and CLK'event) THEN --Clock (edge
                --triggered)

                IF(MAC='1') THEN
                    Q <= P + Q;
                ELSE
                    Q <= P;
                END IF;
            END IF;
        END PROCESS;
    END BEHAVIOR;

```

Generate ACFs

The Leonardo Spectrum software can generate Assignment & Configuration Files (.acf) for the MAX+PLUS II software automatically. This file contains complete information for performing place-and-route, such as timing constraints, device types, pin locations, library mapping file (LMF) locations, and global logic synthesis settings. To ensure accurate results when generating ACFs, you must constrain your design properly in the Leonardo Spectrum software.

To create ACFs automatically, execute the `place_and_route` command. A sample `place_and_route` command is shown below.

```
place_and_route design.edf -target flex10 -max_acf_only ←
```

Table 6 shows the arguments you can use with the `place_and_route` command in the Leonardo Spectrum software.

Argument	Description
<code>-target</code>	Target device family.
<code>-part</code>	Target device.
<code>-speed_grade</code>	Target speed grade.
<code>-max_acf_only</code> <code>-gui</code>	<code>-max_acf_only</code> generates an ACF only. <code>-gui</code> runs the MAX+PLUS II Compiler in GUI mode.
<code>-max_no_acf</code>	Suppress generation of ACFs.
<code>-max_ta_setup</code>	Perform timing analysis and report setup and hold time.
<code>-max_ta_reg</code>	Perform timing analysis and report maximum register frequency.
<code>-max_area</code> <code>-max_delay</code>	<code>-max_area</code> optimizes place-and-route for minimum area; <code>-max_delay</code> optimizes place-and-route for minimum delay.
<code>-max_auto_fast_io</code>	Turn on the <i>Fast I/O</i> option in the MAX+PLUS II software.
<code>-max_auto_implement_packing</code>	Turn on the <i>Implement Packing</i> option in the MAX+PLUS II software.
<code>-exe_path <pathname></code>	Explicitly specify the path to the MAX+PLUS II executable.

MAX+PLUS II Options for High Performance

Back-Annotation Flow

After the MAX+PLUS II software performs place-and-route, you can import the Standard Delay Format or netlist files back into the Leonardo Spectrum software. You can then use the Leonardo Spectrum software to perform further static timing analysis on the design.

After synthesizing your design in the Leonardo Spectrum software, you can import the resulting EDIF netlist file into the MAX+PLUS II software. The following sections describe the recommended synthesis style and other options you can use to achieve optimum performance.

Synthesis Style

When the *Map Logic to LCELLs* option is turned on in the Leonardo Spectrum software, the software automatically optimizes the design for the FLEX 10K architecture. Therefore, Altera recommends using the *WYSIWYG* logic synthesis style in the MAX+PLUS II software (with the *NOT-Gate Push-Back* option turned off). If this style does not give the required performance, try the *FAST* logic synthesis style. Generally, the *WYSIWYG* logic synthesis style gives the best results, but in some cases the *FAST* logic synthesis style produces better results.

Perform the following steps to set your MAX+PLUS II compilation options:

1. Start the MAX+PLUS II software.
2. Choose **Project Name** (File menu). In the **Project Name** dialog box, select the appropriate Leonardo Spectrum software-generated EDIF file `<working directory>/<project name>.edf` and click **OK**.



Altera recommends that you store the Leonardo Spectrum EDIF file in a separate directory from the HDL source files.

3. Choose **Compiler** (MAX+PLUS II menu).
4. Select *Leonardo Spectrum* as the vendor in the **EDIF Netlist Reader Settings** dialog box (Interfaces menu). Click **OK**.
5. Select the appropriate device in the **Device** dialog box (Assign menu). Click **OK**.
6. Turn on the appropriate netlist writer command (Interfaces menu). For example, if you want to create a VHDL Output File (`.vho`), turn on the **VHDL Netlist Writer** command.

7. Choose the appropriate netlist writer settings (Interfaces menu). For example, if you want to create a VHDL Output File, choose the **VHDL Netlist Writer Settings**. In the resulting dialog box, turn on the appropriate netlist writer settings. For example, if you want to create a VHDL Output File, turn on the *VHDL Output File [.vho]* option. Click **OK**.
8. Choose **Global Project Logic Synthesis** (Assign menu). In the **Global Project Logic Synthesis** dialog box, set the synthesis style to *WYSIWYG* for FLEX designs that map LEs in the Leonardo Spectrum file. Click **Define Synthesis Style**. In the **Define Synthesis Style** dialog box, click **Advanced Options**. In the **Advanced Options** dialog box, turn off the *NOT-Gate Push-Back* option. Click **OK** three times to close the dialog boxes.
9. Click the **Start** button to run the MAX+PLUS II Compiler.



For more information on creating designs and compiling them within the MAX+PLUS II software, go to the MAX+PLUS II ACCESS Key Guidelines for the Leonardo Spectrum software.

Using the Fast I/O Logic Option

FLEX 10K devices have built-in registers close to the I/O pins. These input/output element (IOE) registers have faster clock-to-output delays (t_{CO}) than buried registers.

Fast I/O is a logic option that can be applied to registers. This option instructs the MAX+PLUS II Compiler to implement the register in an LE or IOE having a fast, direct connection to an input or I/O pin. Turning on the *Fast I/O* option helps maximize timing performance (e.g., by permitting fast setup times).

You can also apply the *Fast I/O* logic option to pins with the following results:

- On input pins, the MAX+PLUS II Compiler moves the assignment to the IOE or LE fed by the input.
- On output pins, it moves the assignment to the I/O cell or logic cell feeding the output.

Perform the following steps to allow the MAX+PLUS II Compiler to make project-wide *Fast I/O* assignments:

1. Choose **Global Project Logic Synthesis** (Assign menu).
2. Turn on the *Automatic Fast I/O* option in the **Global Project Logic Synthesis** dialog box. Click **OK**.

Perform the following steps to create *Fast I/O* assignments on individual registers:

1. Choose **Logic Options** (Assign menu). In the **Logic Options** dialog box, enter the node name of the register.
2. Click **Individual Logic Options**. In the **Individual Logic Options** dialog box, turn on the *Fast I/O* option. Click **OK** twice in the appropriate dialog boxes to save your changes.
3. Click **Start** in the MAX+PLUS II Compiler to compile.

Timing-Driven Compilation

On average, timing-driven compilation improves device performance by 15% to 30%, depending on resource utilization.



Generally, devices with higher resource utilization take longer to compile. When devices are fully utilized, the MAX+PLUS II software works harder to fit the design, while ensuring that all timing requirements are met. This full utilization can increase the compile time by as much as 10 times.

You can specify the four timing constraints shown in [Table 7](#) when performing timing-driven compilation.

Parameter	Definition	Implementation in the MAX+PLUS II Software
t_{PD}	t_{PD} (input to non-registered delay) is the time required for a signal from an input pin to propagate through combinatorial logic and appear at an external output pin.	You can specify a required t_{PD} for an entire project and/or for any input pin, output pin, or TRI buffer that feeds an output pin.
t_{CO}	t_{CO} (clock-to-output delay) specifies the maximum acceptable clock-to-output delay. The output pin is fed by a register (after a clock signal transition) on an input pin that clocks the register. This time always represents an external pin-to-pin delay.	You can specify a required t_{CO} for an entire project and/or for an input pin, output pin, or TRI buffer that feeds an output pin.
t_{SU}	t_{SU} (clock setup time) is the length of time for which data that feeds a register via its data or enable input(s) must be present at an input pin before the clock signal that clocks the register is asserted at the clock pin.	You can specify a required t_{SU} for an entire project and/or for any input pin or bidirectional pin.
f_{MAX}	f_{MAX} (maximum clock frequency) is the maximum clock frequency that can be achieved without violating internal setup and hold time requirements.	You can specify a required f_{MAX} for an entire project and/or for any input pin, bidirectional pin, or a register.



To assign timing requirements globally, choose **Global Project Timing Requirements** (Assign menu). To assign timing requirements to individual critical paths, choose **Timing Requirements** (Assign menu).

Pin Locking

Within the MAX+PLUS II software, you can enter assignments for the current project by choosing Assign menu commands, moving node and pin names in the Floorplan Editor, or manually editing the ACF. All of these assignment methods edit the ACF. However, only one application can edit the ACF at a time. If you manually edit the ACF in a Text Editor window, you must save your edits before choosing an Assign menu command or switching to the Floorplan Editor. Because manually editing the ACF is more likely to generate errors, Altera recommends entering assignments with Assign menu commands or the Floorplan Editor.

Additionally, Altera recommends compiling the project for the first time without any assignments. However, if you want to make particular assignments before an initial compilation, follow these steps:

1. Choose **Pin/Location/Chip** (Assign menu). In the **Pin/Location/Chip** dialog box, enter the node name or pin for which you wish to enter assignments.
2. Select a pin, logic cell, or other resource under *Chip Resources*.
3. Click **Add**, and then click **OK**.
4. Click **Start** in the MAX+PLUS II Compiler window to compile.

The Leonardo Spectrum software provides three tool levels: Level 1, Level 2, and Level 3.

Leonardo Spectrum Levels

Leonardo Spectrum Level 1

Leonardo Spectrum level 1 is an easy-to-use synthesis tool that uses a powerful logic synthesis engine. To produce high-quality netlist files, the designer simply selects the input design and target device and then clicks the **Run** button.

Leonardo Spectrum Level 2

Leonardo Spectrum level 2 is an easy-to-use synthesis, timing analysis, and back-annotation tool. To produce a high-quality netlist, you select the input design and target device and then click the **Run** button. Level 2 provides the following distinct features:

- Architecture-independent specifications
- Architecture-specific operator generation
- Architecture-specific optimization
- Accurate architecture-specific timing analysis
- Certified PLD design flows
- RTL and gate-level post-synthesis verification
- Timing back-annotation
- Platform independence

You can invoke Leonardo Spectrum Level 2 from the GUI and in command-line mode.

Leonardo Spectrum Level 3

Leonardo Spectrum level 3 is a versatile and interactive logic synthesis, optimization, and analysis tool developed to allow the use of architecture-independent design methods for PLDs. You can consolidate multiple designs efficiently and economically into one design and preserve and manipulate the design hierarchy. Leonardo Spectrum level 3 offers the following distinct features:

- Certified PLD design flows: Netlist files and directives generated by Leonardo Spectrum level 3 successfully pass through the MAX+PLUS II software; post place-and-route timing information can be read and back-annotated for timing and logic verification purposes.
- Architecture-specific module generation
- Architecture-specific optimization
- Accurate architecture-specific timing analysis
- RTL and gate-level post-synthesis verification
- Timing back-annotation

Conclusion

Design time and performance are valuable commodities in the programmable logic industry. This application note demonstrates various techniques to help you achieve performance goals and save design time by streamlining your design. By using VHDL and Verilog HDL coding techniques, Leonardo Spectrum software constraints, and MAX+PLUS II software options, you can improve the performance in FLEX 10K devices and ultimately improve your overall design.

Revision History

The information contained in *Application Note 102 (Improving Performance in FLEX 10K Devices with Leonardo Spectrum Software)* version 1.01 supersedes information published in previous versions. *Application Note 102 (Improving Performance in FLEX 10K Devices with Leonardo Spectrum Software)* version 1.01 contains the following changes:

- [Figure 6](#) was updated.
- The example in [Figure 8](#) was updated.

Copyright © 1995, 1996, 1997, 1998, 1999 Altera Corporation, 101 Innovation Drive, San Jose, CA 95134, USA, all rights reserved.

By accessing this information, you agree to be bound by the terms of Altera's Legal Notice.