FPGA Design Guide

Lattice Semiconductor Corporation
5555 NE Moore Court
Hillsboro, OR 97124
(503) 268-8000

September 16, 2008

Copyright

Trademarks

Disclaimers

to correct any errors contained herein or to advise any user of this document of any correction if such be made. LSC recommends its customers obtain the latest version of the relevant information to establish, before ordering, that the information being relied upon is current.

Type Conventions Used in This Document

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

Contents

1

Introduction

This manual discusses aspects of design with Lattice Semiconductor FPGAs. You can find more tips in the technical notes on Lattice Semiconductor's Web site at www.latticesemi.com.

This manual addresses the following topics.

Moving Designs from Altera or Xilinx FPGAs Moving a design from one brand of FPGA to another offers many challenges. "Moving Designs from Altera " on page 3 (for former users of Altera software) and "Moving Designs from Xilinx" on page 13 (for former users of Xiinx software) provide a variety of guidelines and tips for modifying existing designs to work with the ispLEVER software and Lattice Semiconductor FPGAs.

Incremental and Modular Design Small changes do not have to involve synthesizing and testing the entire design again. "Incremental and Modular Design Methods" on page 33 describes incremental and modular design methods. It begins with the benefits of this design approach, followed by instructions and guidelines for specific tasks, such as logic partitioning, device floorplanning, and simulation. A design example is provided to illustrate the strategies in practice.

Design Guidelines Whether you are new to FPGA design or just new to Lattice Semiconductor FPGAs, there are many considerations involved in producing an effective design. The next few chapters provide guidelines, with Verilog HDL and VHDL code examples, for getting the best performance and resource utilization from your design:

- "Logic Synthesis Guidelines" on page 51 provides a design flow for creating register-transfer-level (RTL) designs.
- "HDL Synthesis Coding Guidelines" on page 57 discusses useful Verilog HDL and VHDL coding styles for Lattice Semiconductor FPGAs. It includes design guidelines for both novice and experienced FPGA designers.

- “Attributes and Preferences for FPGA Designs” on page 85 describes the most common ispLEVER attributes used with RTL designs. This chapter also describes popular compiler directives, attributes, and library components for non-RTL (or non-algorithmic) code.
- “Synthesis Tips for Higher Performance” on page 111 provides tips on improving design performance by applying synthesis techniques for both Mentor Graphics® Precision® RTL Synthesis and Synplicity® Synplify® for Lattice Semiconductor synthesis software.

Strategies for Timing Closure “Strategies for Timing Closure” on page 135 describes placement and routing strategies that help achieve timing closure for the most aggressive design requirements. It begins with a brief description of the seven steps for successful placement and routing, followed by instructions for implementing each of these steps using the ispLEVER software. It discusses the following topics:

- Seven Steps to Successful Placement and Routing
- Constraining Designs
- Using the Place and Route Software (PAR)
- Analyzing Timing Reports
- Controlling Place and Route
- Floorplanning the Design

2

Moving Designs from Altera

The guidelines in this chapter provide practical advice for Altera users who want to migrate designs originally created for Altera FPGAs to Lattice Semiconductor devices. Given the relative FPGA capacity and feature set offered by both vendors, this chapter emphasizes replacement of Cyclone and Cyclone II devices with LatticeECP or LatticeEC devices. However, much of the advice is applicable to any Lattice Semiconductor FPGA family.

This chapter is based on Lattice Semiconductor ispLEVER software, version 5.1, and Altera Quartus II software, version 4.2.

For more information, see the ispLEVER Help and the Lattice Semiconductor Web site, www.latticesemi.com. The Help provides extensive information on process flows and on how to use the tools. It also provides tutorials, reference manuals, and user manuals for the Mentor Graphics and Synplicity simulation tools, which are included in the ispLEVER software. The Lattice Semiconductor Web site provides a large collection of white papers and application notes.

To gain some quick experience with the ispLEVER software and design flow, try the “FPGA Design with ispLEVER Tutorial” in the Help.

Conversion Guidelines

Converting a design originally targeted to an Altera Cyclone device to a LatticeECP/EC FPGA involves several steps:

- Replace Quartus II project-wide constraints and options with equivalent ispLEVER preferences and process properties.
- Replace Altera megafunctions with modules from ispLEVER IPexpress.

- Replace component- or signal-specific Quartus II timing and location constraints with corresponding ispLEVER constraints.
- Replace any Altera-specific library with the LatticeECP/EC library.
- Replace Altera-specific primitives, such as I/O and global clock buffers, by equivalent Lattice Semiconductor primitives or behavioral HDL code and preferences.
- Optimize HDL-inferred modules such as shift registers, counters, and multipliers.

Converting Design Constraints

Like the Quartus II design style, the ispLEVER software does not require you to add special components or attributes to your HDL design to establish the correct signal I/O buffer or interface standards. Instead, the Design Planner and preference language allow you to define these physical implementation details. The Lattice Semiconductor sysI/O buffer of the LatticeECP/EC device family supports a variety of standards, including SSTL, PCI, and LVDS.

If it is necessary to introduce device-specific elements, the *FPGA Libraries Guide* documents many atomic design elements. IPexpress produces medium- to large-scale components.

Table 1 lists the ispLEVER equivalents to Quartus II design constraints.

Table 1: Lattice Semiconductor Equivalents of Typical Quartus II Design Constraints

Constraint	Quartus II Software	ispLEVER Software
Device	Project Navigator: Assignments > Device	Project Navigator: Source > Set New Device
Synthesis options	Project Navigator: Assignments > Settings > Analysis & Synthesis Settings	Third-party synthesis tool Project Navigator: Build Database > Properties
Fitter options	Project Navigator: Assignments > Settings > Fitter Settings	Project Navigator: Map Design > Properties Project Navigator: Place & Route Design > Properties
I/O location and types	Project Navigator: Assignments > Pins	Project Navigator: Pre-Map Design Planner
Timing options	Project Navigator: Assignments > Timing Settings	Project Navigator: Pre-Map or Post-Map Design Planner
EDA tool settings	Project Navigator: Assignments > Settings > EDA Tool Settings	Project Navigator: Options > Select RTL Synthesis Project Navigator: Generate Timing Simulation Files

Table 2 compares Altera constraints and Lattice Semiconductor preferences.

Table 2: Lattice Semiconductor Equivalents of Altera Constraints

Altera Constraints	Lattice Semiconductor Preference
Maximum Delay assignment in Assignment Editor. Overrides any clock settings, if the assignment is applied to a path between two registers. However, an f_{MAX} constraint can be used. If the net is purely combinatorial, a t_{PD} assignment can be made.	MAXDELAY
Maximum Data Arrival Skew or Maximum Clock Arrival, depending on the net, in the Assignment Editor.	MAXSKEW
t_{SU} in the Assignment Editor.	INPUT_SETUP to specify the setup time at input port.
t_{CO} in the Assignment Editor.	OFFSET OUT
f_{MAX} can be specified in the Timing Settings dialog box. Make individual and global clock setting using the Timing Settings dialog box (Project menu).	PERIOD
Current Strength in the Option field in the Assignment Editor.	Edit ASIC I/O in Design Planner
Slow Slew Rate in the Option field in the Assignment Editor.	Edit ASIC I/O in Design Planner
Fast Input Register or Fast Output Register in the Option field in the Assignment Editor.	LOCATE
Adjust Input Delay to Input Register. Adjust the delay of the input pin to the input register. Turned to either ON or OFF in the Option field in the Assignment Editor	INDELAY/FIXEDDELAY attribute in Design Planner
I/O standards in the Assignment Editor.	Edit ASIC I/O in Design Planner
An LCELL between two nets prevents either net from being synthesized out.	LOCK

Table 3 compares Synopsys SDC constraints and Lattice Semiconductor preferences.

Table 3: Conversion of Synopsys SDC Constraints

Synopsys SDC Constraints	Description	Lattice Semiconductor Preferences
create_clock	Creates a base clock with the given name and waveform, and applies the clock to specified clock pin list.	FREQUENCY PORT/ NET PERIOD PORT/ NET
set_clock_latency	Inserts a source latency into an existing base clock.	NA

Table 3: Conversion of Synopsys SDC Constraints (Continued)

Synopsys SDC Constraints	Description	Lattice Semiconductor Preferences
set_false_path	Specifies that the timing paths that start from a designated start node and end at a designated destination node be false paths.	BLOCK
set_input_delay	Specifies the external input delay of a set of input or bidirectional pins with respect to the designated clock.	INPUT_SETUP INPUT_DELAY
remove_clock	Removes all clocks that are used in the current design if the all option is specified.	NA
create_generated_clock	Creates a derived or generated clock from the given clock source. A generated clock can be derived only from a base clock. The generated clock is always assumed to be propagated.	NA
get_clocks	Returns the list of clock pins as specified in the <*clock_pin_list*>. The input list is returned as the output. When <*no_port_list*> is specified, the command returns nothing.	Use Design Planner
remove_input_delay	Removes the specified input delay assignments from the current design.	Remove the specific preference from the Design Planner or preference file.
remove_output_delay	Removes the specified output delay assignments from the current design.	Remove the specific preference from the Design Planner or preference file.
reset_path	Removes the specified timing path assignments from the current design. If neither the setup or hold option is specified, both setup and hold paths are removed.	BLOCK RESETPATHS
set_max_delay	Specifies the maximum delay for the timing paths from the designated <*from_pin_list*> to <*to_pin_list*>.	MAXDELAY
set_min_delay	Specifies the minimum delay for the timing paths from the designated <*from_pin_list*> to <*to_pin_list*>.	Use hld in command line for MIN analysis.
set_multicycle_path	Specifies that the given timing paths have multicycle setup or hold delays with the number of cycles specified by <*path_multiplier*>.	MULTICYCLE
set_output_delay	Specifies t_{CO}, which is the external output delay of a set of output or bidirectional pins with respect to the designated clock. The delay applies to both the positive and negative edges of the clock.	CLOCK_TO_OUT or OFFSET OUT
set_propagated_clock	Specifies that a given clock be propagated using the actual clock network delays.	NA
get_ports	Returns the list of ports as specified in the <*port list*>.	Use the Design Planner.

Table 4 shows some examples of converting Synplify and Precision RTL Synthesis timing constraints to Lattice Semiconductor preferences. Always put: Block RESETPATHS; Block ASYNCPATHS.

Table 4: Converting Synplify and Precision RTL Synthesis Timing Constraints

Synplify (.sdc File Command)	Precision (.tcl File Command)	Lattice Semiconductor ispLEVER .prf File command
`define_clock -name {CLK_TX} -freq 400 -rise 1.0 -fall 2.5`	`create_clock -design rtl -name CLK_TX -freq 400 -waveform {1 2.5}`	`FREQUENCY 400.0 MHz;`
`define_input_delay 2.00 -clock CLK_TX {d0[7:0]}`	`set_input_delay -design rtl -clock CLK_TX 2 d0(7:0)`	`INPUT_SETUP ALLPORTS 2.0 ns CLKPORT "clk"`
`define_output_delay 2.00 -clock CLK_TX {Q[7:0]}`	`set_output_delay -design rtl -clock CLK_TX 2 Q(7:0)`	`CLOCK_TO_OUT ALLPORTS 2.0 ns CLKNET "clk_int"`
`define_path_delay -from {{p:RESET}} -to {{i:Q[7:0]}} -max 5.000`	`set_max_delay 11.0 -from {input_ A input_ B} -to Y_output`	`MAXDELAY FROM PORT "a" TO CELL "reg_Q" 5.0 ns;`
`define_multicycle_path 2 -from [get-pins reg_alu/Q] -to [get_pins reg_mult/D]`	`set_multicycle_ path 2 -from reg_alu* -to reg_mult`	`MULTICYCLE FROM CELL "reg_Q" CLKNET "clk_int" 5.0 ns;`
`define_false_path -from RESET`	`set_false_path -from RESET`	`BLOCK RESETPATHS;`

Converting Memory Blocks

Cyclone devices contain embedded RAM blocks organized into 4-kilobit structures. LatticeECP/EC devices contain sysMEM embedded block RAMs (EBRs) consisting of 9-kilobit RAMs with dedicated input and output registers. Use IPexpress to configure EBRs with the features, width, and depth desired, and produce Verilog HDL or VHDL output for your project. The sysMEM EBRs can be configured to support a variety of memory types, such as FIFO, ROM, and DPRAM.

Unlike Altera's Cyclone, LatticeECP/EC devices also support distributed memory based on PFUs.

Other factors to consider are the following:

- Generate an equivalent Lattice Semiconductor EBR module.
- Enable the output register, if required. Check that the clock latency from data input to data output is equivalent.

  ```
  defparam ram.REGMODE = "OUTREG";
  ```

- Cyclone memory outputs are cleared on power-up. For LatticeECP/EC devices, the output status is user-defined.
- Instead of the cyclone read-during-write mode, use the EBR read-before-write mode. Additional logic may be required.
- Always turn on the EBR pipelining register, for it improves the t_{CO}, especially when the EBR is in the critical path. The unregistered clock-to-

Q delay for Lattice Semiconductor's EBRs is approximately 3.5 ns, and their registered clock-to-Q delay is approximately 0.76 ns.

- Converting EBRs to distributed memory mapping when running out of EBRs:
  - SPRAM and DPRAM can only be mapped to distributed RAM.
  - True dual-port RAMs must be mapped to EBR.
- The Altera memory compiler produces a parameterizable altsyncram primitive. Write an RTL wrapper to connect an equivalent SCUBA-generated module.

The following M4K features are not supported by the sysMEM EBR and may require additional logic to implement:

- Parity bits
- Byte enable
- Embedded shift register

The following tables map the ports of various Altera RAM configurations to their LatticeECP/EC equivalents. (At Lattice Semiconductor, a simple dual-port RAM is called pseudo-dual-port RAM.)

Table 5: Single-Port RAM Port Equivalents

Port Description	Altera	Lattice Semiconductor
Data input	data	DI
Data output	q	Q
Address	address	Address
Write enable	wren	WE
Clock enable	inclocken	ClockEn
Clock input	inclock	Clock

Table 6: Simple Dual-Port RAM Port Equivalents

Port Description	Altera	Lattice Semiconductor
Data input	data	DI
Data output	q	Q
Write address	wraddress	ADW
Read address	rdaddress	ADR
Write enable	wren	WE
Read enable	rden	
In clock enable	inclocken	CEW
Out clock enable	outclocken	CER

Table 6: Simple Dual-Port RAM Port Equivalents

Port Description	Altera	Lattice Semiconductor
Set/reset	inaclr, outaclr	Reset
Read clock	outclock	CLKR
Clock input	inclock	CLKW

Table 7: True Dual-Port RAM Port Equivalents

Port Description	Altera	Lattice Semiconductor
Data input	dataA, dataB	DataInA, DataInB
Data output	qA, qB	QA, QB
Address	addressA, addressB	AddressA, AddressB
Write enable	wrenA, wrenB	WrA, WrB
Clock enable	inclockenA, inclockenB	ClockEnA, ClockEnB
Set/reset	aclrA, aclrB	ResetA, ResetB
Clock input	clockA, clockB	ClockA, ClockB

For more information on designing LatticeECP/EC memory, see the “How to Design with FPGA Memories” topic in the online Help.

Converting FIFO

Because synthesis does not infer FIFO, use IPexpress to generate and instantiate FIFO in RTL. Lattice Semiconductor does not have inherent hardware support for different read and write widths for FIFO in LatticeEC and LatticeXP devices. Implement the control logic in RTL. Be aware that the Lattice Semiconductor FIFO_DC has two clock latencies for the de-assertion of FIFO status flags.

MachXO 1K and 2K have built-in FIFO logic: primitive FIFO8KA.

Inferring Memory

Write in generic RTL to infer memory. See Figure 1 and Figure 2 for examples. Synthesis inferencing is not available for FIFO. Use the SCUBA FIFO memory compiler.

Converting PLL Blocks

Cyclone devices contain up to two analog phase-locked loops (PLLs) for clock management. LatticeECP/EC devices contain two to four analog PLLs called sysCLOCK PLLs. Use IPexpress to configure PLLs with operating frequency, phase controls, and duty cycle.

Table 8 maps the ports of the Altera altpll megafunction to the Lattice Semiconductor sysCLOCK PLL.

Figure 1: Inferring Single-Port RAM in Precision RTL Synthesis Verilog HDL

```
module sync_ram_singleport (clk, we, addr, data_in, data_out);
parameter addr_width = 10;
parameter data_width = 32;
input clk;
input we;
input [addr_width - 1:0] addr;
input [data_width - 1:0] data_in;
output[data_width - 1:0] data_out;
reg [addr_width - 1:0] addri;
reg [data_width - 1:0] mem [(32'b1 << addr_width):0];

always @ (posedge clk)
begin
  if (we)
    mem[addr] = data_in;
  addri = addr;
end

assign data_out = mem[addri];

endmodule
```

Figure 2: Inferring Pseudo-Dual-Port RAM in Precision RTL Synthesis Verilog HDL

```
module sync_ram_dualport (clk_in, clk_out, we, addr_in,
addr_out, data_in, data_out);
parameter data_width = 16;
parameter addr_width = 16;
input clk_in;
input clk_out;
input we;
input [addr_width - 1:0] addr_in;
input [addr_width - 1:0] addr_out;
input [data_width - 1:0] data_in;
output[data_width - 1:0] data_out;
reg [data_width - 1:0] data_out;
reg [data_width - 1:0] mem [(32'b1 << addr_width) - 1:0];

always @ (posedge clk_in) begin
  if (we)
    mem[addr_in] <= data_in;
end

always @ (posedge clk_out) begin
  data_out <= mem[addr_out];
end

endmodule
```

For more information on designing LatticeECP/EC PLLs, see the “How to Design with sysCLOCK PLLs and DLLs” topic in the online Help.

Table 8: PLL Port Equivalents

Port Description	Altera	Lattice Semiconductor
Clock input	inclk0	CLKI
Clock feedback input	None, feedback path is internal	CLKFB (PLL output, clock net, routing, ext)
Asynchronous reset	areset	RST (set to 1 to reset input clock divider)
Combined enable and reset, active high	pllena	
Clock outputs driving the internal global clock network	c[1..0]	CLKOS (phase/duty) CLKOP (no phase) CLKOK (second divider)
Clock output driving the single-ended or LVDS external clock outputs	e0	Any PLL clock outputs through normal routing
Enable for up/down output from the phase frequency detector (PFD)	pfdena	
PLL lock status	locked	LOCK (1 indicates locked to CLKI)

Converting DDR Interfaces

The key trick for the interface is shifting the DQS strobe-in pin by 90 degrees by the time it reaches the register. Like Cyclone II and Stratix, the LatticeEC and LatticeXP devices have a DQS shift circuit built in, so no changes to the design are needed.

3

Moving Designs from Xilinx

This chapter compares Xilinx FPGA software design tools and flows with Lattice Semiconductor software, provides device migration information for targeting Xilinx FPGA devices to comparable Lattice Semiconductor devices, and provides alternate reference sources for device and package selection. Although much of the advice is applicable to all device families, this chapter provides mostly specific information for Xilinx Spartan-3 device migration to LatticeEC devices. This chapter also includes a small section on Virtex II conversion guidelines.

This chapter assumes that you are familiar with the Spartan family and features. Familiarity with VHDL or Verilog HDL and third-party synthesis tools is also assumed. This chapter is based on Lattice Semiconductor ispLEVER software, version 5.1, and Xilinx ISE software, version 6.1i.

For more information, see the ispLEVER online Help and the Lattice Semiconductor Web site at www.latticesemi.com. The Help provides extensive information on process flows and on how to use the tools. It also provides tutorials, reference manuals, and user manuals for the Mentor Graphics and Synplicity synthesis tools, which are included in the ispLEVER software. The Lattice Semiconductor Web site provides a large collection of white papers and application notes.

Lattice Semiconductor provides information to assist you in deciding what device and package best fits your requirements. In the *Product Selector Guide*, you can view technical specifications for all of Lattice Semiconductor's product families, including available devices, packages, speed grades, design requirements, and feature support. In addition, you can use the *Package Selector Card* to find the information you need to determine what package is best for your device.

Migrating Xilinx Spartan Designs to LatticeECP/EC

This section compares Xilinx Spartan-3 hardware features and suggests specific steps and strategies for conversion to comparable LatticeEC and LatticeECP-DSP devices.

In general, you must:

- Comment out any Xilinx-specific library and add the LatticeECP/EC library, if required.
- Replace Xilinx-specific primitives, such as I/O buffers and global clock buffers, with Lattice Semiconductor primitives or behavioral HDL code and preferences.
- Replace Xilinx core modules, such as DCM, memory, and multipliers, with Lattice Semiconductor modules.
- Replace the Xilinx timing and device constraints (.ucf) file with a Lattice Semiconductor source constraints or preferences file (.prf).
- Optimize HDL-inferred modules such as shift registers, counters, and multipliers.

Replacing Commonly Used Xilinx Primitives

Table 9 shows commonly used Xilinx primitives and their Lattice Semiconductor counterparts.

Table 9: Xilinx Primitives and Lattice Semiconductor Equivalents

Xilinx Primitive	Description	Lattice Equivalent
BUF, 4, 8, 16	General purpose buffer	Primitive BUFBA or HDL attribute
BUFG	Global clock buffer	Location assignment, Preference USE PRIMARY
FD	D Flip Flop	Lattice primitives or behavioral HDL
IBUF_<*I/O_standard*>	Input buffer with selectable I/O standard	Primitives IB, IBM, and related; IO_TYPE attribute
IOBUF_<*I/O_standard*>	Bidirectional buffer with selectable I/O standard	Primitives BB, BBW and related; IO_TYPE attribute
OBUF_<*I/O_standard*>	Output buffer with selectable I/O standard	Primitives OB, OBZ, and related; IO_TYPE attribute
SRL16	LUT-based 16-bit shift register	HDL, use distributed RAM or EBR

Xilinx I/O Buffer Conversion

Input, output, and bidirectional buffers are automatically inserted by the Lattice Semiconductor ispLEVER compiler. As a result, all the I/O buffers in a Xilinx design can be removed. The output signal of the buffer can be replaced by the I/O signal or assigned to it. Figure 3 shows a Verilog HDL example of a buffer being removed.

Figure 3: Removing I/O Buffer

```
module top (a, b, c, clk);
  input a, b, clk;
  output c;
  reg c;
  wire clk_out;

  // BUFG inst1 (.I(clk), .O(clk_out)); No need for buffer.
  // Assign buffer's clk_out directly to module's input, clk.
  assign clk_out = clk;

  always @ (posedge clkout)
    begin
      c <= a & b;
    end
endmodule
```

You can convert Xilinx I/O standard primitives into Lattice Semiconductor primitives, or you can specify them in I/O Type attributes. Figure 4 and Figure 5 show how to specify I/O primitives using I/O Type (IO_TYPE) attributes. Table 10 shows the supported I/O types for the various Lattice I/O buffers.

Figure 4: Syntax for I/O Type Attributes in VHDL

```
ATTRIBUTE IO_TYPE : string;
ATTRIBUTE IO_TYPE OF [PinName]: SIGNAL IS "[Type]";
  => See Below I/O type table
ATTRIBUTE PULLMODE : string;
ATTRIBUTE PULLMODE OF [PinName]: SIGNAL IS "[Type]";
  => NONE, KEEPER, UP, DOWN
ATTRIBUTE DRIVE : string;
ATTRIBUTE DRIVE OF [PinName]: SIGNAL IS "[Type]";
  => 2,4,6,8,12,16,20
ATTRIBUTE SLEWRATE : string;
ATTRIBUTE SLEWRATE OF [PinName]: SIGNAL IS "[Type]";
  => FAST, SLOW
```

Figure 5: Syntax for I/O Type Attributes in Synplify Verilog HDL

```
PinType [PinName] /* synthesis IO_TYPE="[Type]" DRIVE="[Type]"
PULLMODE="[Type]" SLEWRATE="[Type]"*/;
// DRIVE [Type] = 2, 4, 6, 8, 12, 16, 20
// PULLMODE [Type] = NONE, KEEPER, UP, DOWN
// SLEWRATE [Type] = FAST, SLOW
```

Table 10: Supported I/O Types for LatticeECP/EC I/O Buffers

I/O Type	Input Buffer	Output Buffer	Bidirectional Buffer
LVTTL, LVCMOS33, LVCMOS25, LVCMOS18, LVCMOS15, LVCMOS12, PCI33	X	X	X
LVTTL_OD, LVCMOS33_OD, LVCMOS25_OD, LVCMOS18_OD, LVCMOS15_OD, LVCMOS12_OD		X	X
LVDS25E, LVDS25, BLVDS25, LVPECL33	X	X	X
HSTL18_I, HSTL18_II, HSTL18_III, HSTL15_I, HSTL15_III	X	X	X
HSTL18D_I, HSTL18D_II, HSTL18D_III, HSTL15D_I, HSTL15D_III	X	X	
SSTL33_I, SSTL33_II, SSTL25_I, SSTL25_II, SSTL18_I	X	X	X
SSTL33D_I, SSTL33D_II, SSTL25D_I, SSTL25D_II, SSTL18D_I	X	X	

You can quickly and easily generate the necessary syntax for your I/O type assignments by using the Design Planner. You can also do so manually in the preference (.prf) file.

MUXCY, XORCY, and MULT_AND Clusters

You can map MUXCY, XORCY, and MULT_AND clusters to combinations of FADD2, FSUB2, and MULT2 primitives, for example:

- MUXCY(DI,CI,S,O) mapped to MUX21(D0,D1,SD,Z)
- BUFGMUX(I0,I1,S,O) mapped to DCS(CLK0,CLK1,SEL,DCSOUT)

SRL16 Shift Register Conversion

You can configure each LUT as a 16-bit shift register without using a flip-flop in each slice. The SRL16 is slower than the flip-flop and is susceptible to soft-error upset. The SRL16 is automatically inferred by the software tool.

Use the following guidelines for converting a Xilinx SRL16 element to a viable Lattice Semiconductor function:

- For a small one-bit shift register, use a flip-flop-based shift register.
- For a large multi-bit shift register, use a circular buffer to emulate a shift register.
- Data is written at clock 0. After *n* clock cycles, the data is clocked out of the buffer while the new data is written into the same location.
- The dual-port RAM is set to Read_Before_Write mode.
- The in and out ports can be different widths.

Use the following guidelines for converting a shift register in a FIR filter design:

- In the FIR filter design, each tap of the shift register must be multiplied. Separate Write and Read address controls are required.
- Use a pseudo-dual-port RAM in an EBR or distributed RAM to emulate a multi-bit shift register in a FIR filter.

Replacing DCM/DLL Elements

DCM is a digital clock manager that implements a clock delay-locked loop (DLL), digital frequency synthesizer (DFS), and digital phase shift (DPS) functions. Table 11 shows Lattice Semiconductor equivalents of Xilinx DCM ports.

Table 11: Lattice Equivalents for Xilinx DCM Ports

Xilinx DCM Port	Description	Lattice PLL Port
CLKIN	Clock input to DCM	CLKI
CLKFB	Clock feedback input to DCM (from CLK, CLK2x)	CLKFB (PLL output, clocknet, routing, ext)
RST	Asynchronous Reset	RST (1 to reset input clock divider)
PSEN	Dynamic phase shift enable (1 = enable)	DDAMODE (1: pin control – dynamic)
PSINCDEC	Increment/decrement phase shift (0 = inc, 1 = dec)	DDAILAG (1 = lag, 0 = lead)
PSCLK	Clock input to dynamic phase shifter	CLKI
PSDONE	Dynamic phase shift complete (1 = done)	
CLK0, CLK90, CLK180, CLK270, CLK2X, CLK2X180	Same or double frequency as CLKIN, phase shift 0, 90, 180 and 270 degree	CLKOS: phase/duty CLKOP: no phase CLKOK: second divider
CLKDV	Divided clock output, CLKDV = CLKIN / CLKDV_DIVIDE	Any PLL clock outputs
CLKFX, CLKFX180	Synthesized clock output, CLKFX = CLKIN x CLKFX_MULTIPLY / CLKFX_DIVIDE	Any PLL clock outputs
STATUS[0:2]	[0]: Phase shift overflow [1]: CLKIN in not toggling [2]: CLKFX output stop	
LOCLKED	DCM locked to CLKIN, clock outputs are valid	LOCK (1 = lock to CLKI)

Note the following when converting this element:

- Digital spread spectrum (DSS) mode is not supported, so the DSSEN pin must be tied to GND.
- The CLK90, CLK270, CLK2X, and CLK2X180 outputs are *not* available if PLL_FREQUENCY_MODE is set to high.
- CLKFB must be sourced from CLK0 and CLK2X.
- Unlike Xilinx DCM, which requires a specific input buffer to feed into CLKIN, such as IBUFG or BUFGMUX, PLLs in Lattice Semiconductor devices do not require input buffers.
- Customize Lattice Semiconductor PLLs in the IPexpress.
- Do not replace DCM with DQSDLL, which is dedicated to the DDR interface.

Comparing Xilinx and Lattice Semiconductor Block Memory

This section compares Xilinx Spartan and LatticeECP/EC block memories. See Table 12 for block memory feature comparison. Table 13 shows a port mapping comparison between Xilinx Spartan-3 and LatticeECP/EC single-port RAM. Table 14 shows a port-mapping comparison between the Xilinx Spartan-3 and the LatticeECP/EC dual-port RAM.

Table 12: Block Memory Feature Comparison

Feature	Xilinx SelectRAM	Lattice sysMEM
Total RAM bits	18,432	9,216
Performance	~200 MHz	~300 MHz
Single-port	Yes	Yes
Pseudo-dual-port	Yes	Yes
True dual-port	Yes	Yes
FIFO, FIFO_DC	Yes	Yes
CAM	Yes	
ROM, initial RAM contents	Yes	Yes
Mixed data port width	Yes	Yes
Power-up condition	User data defined/default =zero	User data defined/default =zero

Table 13: Port Mapping for Single-Port RAM

Signal Description	Xilinx Port Name	Lattice Equivalent
Data input bus	DI	Data
Parity data input bus	DIP	Data

Table 13: Port Mapping for Single-Port RAM (Continued)

Signal Description	Xilinx Port Name	Lattice Equivalent
Data output bus	DO	Q
Parity data output bus	DOP	Q
Address bus	Addr	Address
Write enable	WE	WE
Clock enable	EN	ClockEn
Synchronous set/reset	SSR	Reset
Clock input	CLK	Clock

Table 14: Port Mapping for Dual-Port RAM

Signal Description	Xilinx Dual Port		Lattice Equivalent	
	Port A	Port B	Port A	Port B
Data input bus	DIA	DIB	DataInA	DataInB
Parity data input	DIPA	DIPB	DataInA	DataInB
Data output bus	DOA	DOB	QA	QB
Parity data output	DOPA	DOPB	QA	QB
Address bus	ADDRA	ADDRB	AddressA	AddressB
Write enable	WEA	WEB	WrA	WrB
Clock enable	ENA	ENB	ClockEnA	ClockEnB
Synchronous set/reset	SSRA	SSRB	ResetA	ResetB
Clock	CLKA	CLKB	ClockA	ClockB

When converting Xilinx Spartan-3 block memory to LatticeECP/EC memory, observe the following guidelines:

- Lattice EBR does not support a separate DIP bus. The parity data input and parity data output buses are integrated into the data input/output buses.
- The EN, WE, and SSR Xilinx block RAM is configurable and active high by default. Lattice Semiconductor ClockEn, WE, and reset are always active high, so you can change the polarity outside the EBR instantiation.
- Xilinx supports synchronous set/reset. When SSR is asserted, the DO/DOP outputs are synchronously set to initial values defined by the SSRVAL parameter. Lattice Semiconductor does not support this feature. Lattice Semiconductor supports synchronous or asynchronous resets.
- The Xilinx GSR is automatically connected. For Lattice Semiconductor, you can enable or disable the GSR in IPexpress.

- You must generate an equivalent Lattice Semiconductor EBR module using IPexpress or instantiate the EBR by utilizing the Parameterizable Module Inference (PMI).
- Enable output registers when required. Check that the clock latency from data input to data output is equivalent.

```
defparam ram.REGMODE = "OUTREG";
```

- Select the proper write mode (see Table 15).

Table 15: Xilinx and Lattice Semiconductor Write Modes

Xilinx Write Mode	Lattice Semiconductor Write Mode
Write_First (default)	Write Through
Read_First (recommended)	Read Before Write
No_Change	Normal

- Always turn on the EBR pipelining register, because it improves the t_{CO}, especially when the EBR is in the critical path. The Lattice Semiconductor EBR unregistered clock-to-Q delay is approximately 3.5 ns, and the registered clock-to-Q delay is approximately 0.76 ns.
- Converting EBR to distributed memory mapping when running out of EBRs:
  - SPRAM and DPRAM can only be mapped to distributed RAM.
  - True dual-port RAMs must be mapped to EBR.
- Mapping from larger block RAM (16 kb in Spartan-3) to 9 kb in LatticeEC and LatticeXP:
  - If the EBR uses less than 9 kb of the 16-kb EBR block, it can be mapped to a Lattice Semiconductor EBR.
  - If the EBR uses more than 9 kb of the 16-kb Xilinx EBR, you can:
    - Use the Module Manager to create it. The multiplexer is created automatically. Two Lattice EBRs are generated, with the 2:1 multiplexer using one address bit for control.
    - Code in generic RTL and allow synthesis to infer the memory.
- When converting distributed RAM:
  - Map RAM16X1S through RAM16X8S to multiple instances of the SPR16X2B primitive.
  - Map RAM16X1D to the DPR16X2B primitive.
  - Use the SCUBA memory compiler to generate equivalent cells for all other Xilinx distributed RAM primitives, such as RAM64X1S.

Converting FIFO

Because synthesis does not infer FIFO, use IPexpress to generate and instantiate in RTL. Lattice Semiconductor does not have inherent hardware support for different read and write widths for FIFO in LatticeEC and LatticeXP devices. Implement the control logic in RTL. Be aware that the Lattice Semiconductor FIFO_DC has two clock latencies for the de-assertion of FIFO status flags.

MachXO 1K and 2K have built-in FIFO logic: primitive FIFO8KA.

Inferring Memory

Write in generic RTL to infer memory. See Figure 6 and Figure 7 for examples. Synthesis inferencing is not available for FIFO. Use the SCUBA FIFO memory compiler.

Figure 6: Inferring Single-Port RAM in Precision RTL Synthesis Verilog HDL

```
module sync_ram_singleport (clk, we, addr, data_in, data_out);
parameter addr_width = 10;
parameter data_width = 32;
input clk;
input we;
input [addr_width - 1:0] addr;
input [data_width - 1:0] data_in;
output[data_width - 1:0] data_out;
reg [addr_width - 1:0] addri;
reg [data_width - 1:0] mem [(32'b1 << addr_width):0];

always @ (posedge clk)
begin
  if (we)
    mem[addr] = data_in;
  addri = addr;
end

assign data_out = mem[addri];

endmodule
```

Figure 7: Inferring Pseudo-Dual-Port RAM in Precision RTL Synthesis Verilog HDL

```
module sync_ram_dualport (clk_in, clk_out, we, addr_in,
addr_out, data_in, data_out);
parameter data_width = 16;
parameter addr_width = 16;
input clk_in;
input clk_out;
input we;
input [addr_width - 1:0] addr_in;
input [addr_width - 1:0] addr_out;
input [data_width - 1:0] data_in;
output[data_width - 1:0] data_out;
reg [data_width - 1:0] data_out;
reg [data_width - 1:0] mem [(32'b1 << addr_width) - 1:0];

always @ (posedge clk_in) begin
  if (we)
    mem[addr_in] <= data_in;
end

always @ (posedge clk_out) begin
  data_out <= mem[addr_out];
end

endmodule
```

Xilinx Multiplier Versus the Lattice Semiconductor DSP Block

This section compares the capabilities of Xilinx's multiplier to those of the LatticeECP block function. Note the following when comparing these device functions:

- Xilinx Spartan-3 devices are limited to offering only embedded multipliers to provide multiplication.
- You can configure the Lattice Semiconductor DSP block to perform any of the following functions:
  - Simple multiply
  - Multiply accumulate
  - Multiply add/subtract
  - Multiply add/subtract SUM
- The Lattice DSP block supports x9, x18, and x36 modes.
- The Lattice DSP block supports options for input, output, and pipeline registers, clock, clock enable, and reset.
- When converting to LatticeECP from Xilinx Spartan-3, you can convert Xilinx LUT-based shift registers by utilizing Lattice Semiconductor EBR or distributed RAM.

Converting DDR Interfaces

The key trick for the interface is shifting the DQS strobe-in pin by 90 degrees by the time it reaches the register. Unlike Spartan-3, the LatticeEC and LatticeXP devices have a DQS shift circuit built in, so remove the DQS LUT-based logic.

Implement the interface with IPexpress to avoid errors. The DQS is generated from DCM. You have to manually shift the clock because it does not dynamically adjust.

Figure 8 compares DDR interface code for Spartan-3 and LatticeEC devices.

Figure 8: Comparison of DDR Interface Code

Spartan-3

```
library ieee;
use ieee.std_logic_1164.all;
entity ddr_dqs_sp3 is
port(
  clk            : in std_logic;
  rd_clk         : in std_logic;
  DDR_DQS_reset  : in std_logic;
  DDR_DQS_enable : in std_logic;
  DDR_DQS        : inout std_logic;
  DQS            : out std_logic);
end ddr_dqs_sp3;

architecture V of ddr_dqs_sp3 is
begin
  VCC <= '1';
  GND <= '0';
  clk_b <= not clk;
  ...

  U1 : FD port map (D => DDR_DQS_enable,
    Q => DDR_DQS_enable1, C => clk);

  U2 : FDDRRSE port map (Q => DQS_q,
    C0 => clk, C1 => clk_b, CE => VCC,
    D0 => VCC, D1 => GND,
    R => DDR_DQS_reset, S => GND);

  U3 : OBUFT  port map (I => DQS_q,
    T => DDR_DQS_enable1 ,O => DDR_DQS);

  U4 : IBUF port map (I => DDR_DQS,
    O => DQS_in);

  U6 : FD port map (D => DQS_in, Q => DQS,
    C => rd_clk);

  U5 : keeper port map (o => DDR_DQS);

end V;
```

LatticeEC

```
library ieee;
use ieee.std_logic_1164.all;
entity ddr_dqs_ec is
port(
  clk            : in std_logic;
  rd_clk         : in std_logic;
  DDR_DQS_reset  : in std_logic;
  DDR_DQS_enable : in std_logic;
  DDR_DQS        : inout std_logic;
  ddrclkpol_sig  : out std_logic;
end ddr_dqs_ec;

architecture V of ddr_dqs_ec is
begin
  GND <= '0';
  clk_b <= not clk;

  process(clk)
  begin
    if rising_edge(clk) then
      DDR_DQS_enable1 <= DDR_DQS_enable;
    end if;
  end process;

  U2 : ODDRXB port map (DA => VCC,
    DB => GND, CLK => clkLSR =>
    DDR_DQS_reset, Q => DQS_q);
  U3 : BB port map (I => DQS_q,
    T => DDR_DQS_enable1, O => DQS_in,
    B => DDR_DQS);
  DQSBUF_inst : DQSBUFB port map (
    DQSI => DQS_in, CLK => clk,
    READ => rd_clk, DQSDEL => dqsdel,
    DQS => dqs, DDRCLKPOL => ddrclkpol_sig,
    DQSC => dqsc, PRMBDET => prmbdet);
  DQSDLL_inst : DQSDLL port map (
    CLK => clk, RST => DDR_DQS_reset,
    UDDCNTL => '1', LOCK => dll_lock,
    DQSDEL => dqsdel);
end V;
```

Replacing Constraints

Replace the Xilinx timing and device constraints (.ucf) file with a Lattice Semiconductor source constraints or preferences file (.prf). See Table 16 for equivalent Lattice Semiconductor preferences.

Table 16: Lattice Semiconductor Equivalents of Xilinx Constraints

Xilinx Constraint	Constraint Function	Lattice Preference
MAXDELAY	Specifies the maximum delay in a net.	MAXDELAY
MAXSKEW	Specifies the maximum skew in a net.	MAXSKEW
NODELAY	Reduces setup time at the cost of positive hold time.	INPUT_SETUP to specify the setup time at input port or input register.
OFFSET	Specifies correlation between a global clock and its associated data in and data out pin. Specifies t_{SU} and t_{CO} on data registers.	OFFSET
Period	Specifies the timing relationship of a global clock, such as an f_{MAX} requirement.	PERIOD
DRIVE	Controls the output pin current value.	Edit ASIC I/O in Design Planner
FAST	Turns on Fast Slew Rate Control.	Edit ASIC I/O in Design Planner
IOB	Specifies whether a register should be placed within the IOB of the device.	LOCATE
IOBDELAY	Specifies a delay before an input pad feeds the IOB, or an external element, from the IOB. The input pad can either feed the local IOB flip-flop or an external element from the IOB.	INDELAY/FIXEDDELAY attribute in Design Planner
IOSTANDARD	Specifies the I/O standard for an I/O pin.	Edit ASIC I/O in Design Planner
KEEP	Prevents a net from being absorbed by a block or synthesized out.	LOCK

Table 17 shows some examples of converting Synplify and Precision RTL Synthesis timing constraints to Lattice Semiconductor preferences. Always put: Block RESETPATHS; Block ASYNCPATHS.

Table 17: Converting Synplify and Precision RTL Synthesis Timing Constraints

Synplify (sdc File Command)	Precision (tcl Command)	Lattice ispLEVER .prf
define_clock -name {CLK_TX} -freq 400 -rise 1.0 -fall 2.5	create_clock -design rtl -name CLK_TX -freq 400 -waveform {1 2.5}	FREQUENCY 400.0 MHz;
define_input_delay 2.00 -clock CLK_TX {d0[7:0]}	set_input_delay -design rtl -clock CLK_TX 2 d0(7:0)	INPUT_SETUP ALLPORTS 2.0 ns CLKPORT "clk"
define_output_delay 2.00 -clock CLK_TX {Q[7:0]}	set_output_delay -design rtl -clock CLK_TX 2 Q(7:0)	CLOCK_TO_OUT ALLPORTS 2.0 ns CLKNET "clk_int"
define_path_delay -from {{p:RESET}} -to {{i:Q[7:0]}} -max 5.000	set_max_delay 11.0 -from {input_ A input_ B} -to Y_output	MAXDELAY FROM PORT "a" TO CELL "reg_Q" 5.0 ns;
define_multicycle_path 2 -from [get-pins reg_alu/Q] -to [get_pins reg_mult/D]	set_multicycle_ path 2 -from reg_alu* -to reg_mult	MULTICYCLE FROM CELL "reg_Q" CLKNET "clk_int" 5.0 ns;
define_false_path -from RESET	set_false_path -from RESET	BLOCK RESETPATHS;

Figure 9 shows a Perl script for converting a Xilinx .ucf into a Lattice Semiconductor .lpf.

Figure 9: .ucf to .lpf Conversion Script

```
#ucf2lpf.pl
# Version 1.0, April 23, 2007, thscott
# Converts Xilinx UCF format to Lattice LPF format. I/O Placement Constraints.
#
#Potential enhancements?:
# -PERIOD TIME-SPEC
# -TIMING IGNORE
# -PATH EXCEPTIONS
#
#Input:
# Xilinx ucf format
#
#Output:
# Lattice lpf format
#
#Substitution rules:
# 1) Placement Constraints for I/O
# UCF: NET io_net_name LOC=P111; # PLCC/PQFP type
# LPF: LOCATE COMP "io_net_name" SITE "111";
# UCF: NET io_net_name LOC=A11; # PGA/BGA type
# LPF: LOCATE COMP "io_net_name" SITE "A11";
#
```

Figure 9: .ucf to .lpf Conversion Script (Continued)

```
#Header
if ($#ARGV == -1) {
  print "\nucf2lpf.pl: Version 1.0\n";
  print "Usage: ucf2lpf.pl [options] input > output\n";
  print " -plcc PLCC/PQFP type package (Default PGA/BGA type).\n";
  print " -lat Lattice style underscore __ bus delimiter.\n";
  print " -prtl Precision RTL style parens () bus delimiter.\n";
  print " Default Synplify style underscore _ bus delimiter.\n";
  print "Converts Xilinx UCF LOC to Lattice LPF LOCATE format.\n\n";
  die;
}

#Sort arguments
foreach $arg (@ARGV) {
  if ($arg eq "-plcc") { $plcc="TRUE";}
  if ($arg eq "-lat") { $lat="TRUE"; }
  if ($arg eq "-prtl") { $prtl="TRUE";}
}

#Access csv input file
$file = $ARGV[$#ARGV]; # Name the file
open(INFO, $file) or die "File $file not found"; # Open the file
@ucffile = ; # Read it into an array
close(INFO); # Close the file
#print "@ucffile"; # Print the array

foreach $ucfconstraint (@ucffile) {

  if ($ucfconstraint =~ /#/ ) { next; } # Skip ucf comment lines

  # Process I/O Placement Type Constraints
  if ($ucfconstraint =~ /NET.*LOC/) {
    $lpfconstraint = $ucfconstraint;
    $lpfconstraint =~ s/NET/LOCATE COMP/;

    if ($lat eq "TRUE") {
      $lpfconstraint =~ s/ $lpfconstraint =~ s/>/_/;
    } elsif ($prtl eq "TRUE") {
      $lpfconstraint =~ s/ $lpfconstraint =~ s/>/)/;
    } else {
      $lpfconstraint =~ s/ $lpfconstraint =~ s/>//;
    }

    if ($plcc eq "TRUE") {
      # Detect string "LOC" then zero or more whitepace characters then character =.
      # Replace with string "SITE".
      $lpfconstraint =~ s/LOC\s*=/SITE/;
      $lpfconstraint =~ s/SITE\s*"P/SITE "/;
      $lpfconstraint =~ s/SITE\s*P/SITE /;
    } else {
      $lpfconstraint =~ s/LOC\s*=/SITE /;
    };
    print "$lpfconstraint";
  };

}
```

Converting Xilinx Virtex II to LatticeECP/EC Devices

Take the following steps when converting Xilinx Virtex II to LatticeECP/EC devices:

- Convert DLL to PLL.
- Create MUXCY and MUXCY_L Verilog HDL modules.
- Create RAMB16_S36_S36 Verilog HDL modules.

Converting Xilinx DLL to Lattice Semiconductor PLL

This section illustrates the guidelines that you should follow when converting Xilinx DLL in Virtex II to Lattice Semiconductor PLL in LatticeECP/EC devices. Figure 10 and Figure 11 show how a DLL element is instantiated in Xilinx software and how the replacement would then look in Lattice Semiconductor software.

Figure 10: Xilinx Code – DLL Instantiation

```
//PCIXCLK input pad
IBUFG_LVCMOS33 PCIXCLK_IBUFG (
  .O(PCIXCLI_in),
  .I(PCIXCLK));

//PCIXCLK DLL
CLKDLL PCIXCLK_DLL (
  .CLKIN(PCIXCLK_in),
  .CLKFB(clock),
  .RST(1'b0),
  .CLK0(PCIXCLK_dll));

// PCIXCLK global clock buffer
BUFG PCIXCLK_BUFG (
  .O(clock),
  .I(PCIXCLK_dll));
PULLUP P1 (FFE_CRDY_N;
PULLUP P2 (NFL_CRDY_N;
```

Figure 11: Lattice Replacement Code – PLL Instantiation

```
/*Start
  input PCIXCLK; //PCI-X clock - 133 MHz */
  input PCIXCLK  /* synthesis IO_TYPE="LVCMOS33" */
/* End

PCIX_CLK_PLL PCIXCLK_PLL (
  .CLKI(PCIXCLK),
  .CLKFB(clock),
  .CLKOP(clock));
```

Creating MUXCY and MUXCY_L Verilog HDL Modules

The second step is to create MUXCY and MUXCY_L Verilog HDL modules in the example. Figure 12 shows how you would instantiate the MUXCY module for Virtex II.

Figure 12: Verilog HDL Code – MUXCY and MUXCY_L

```
module MUXCY ( // or MUXCY_L
  output reg O,
  input S, DI, SI);

  always @ (*)
    /* full_case, parallel_case */
    case ({S, DI, SI})
      3'b000: O = 1'b0;
      3'b001: O = 1'b0;
      3'b100: O = 1'b0;
      3'b110: O = 1'b0;
      default: O = 1'b1;
    endcase
endmodule
```

Wide Multiplexing

Map the Xilinx MUXF5 (I0, I1, S, O) to the Lattice Semiconductor PFUMUX (BLUT, ALUT, CO, Z).

Map the Xilinx MUXF6 (I0, I1, S, O) through MUXF8 (I0, I1, S, O) to the Lattice Semiconductor L6MUX21 (D0, D1, SD, Z).

Optimal Carry-Chain Handling

In LatticeECP/EC, LatticeXP, and MachXO devices, the non-registered carry-sum cannot bypass the transparent latch, so it incurs a TLATCH delay of ~0.9 ns. If this becomes the critical path, use a workaround, such as carry-save or other LUT-logic operation.

The alternative is to always modify RTL to use registered carry-sum.

Converting Xilinx RAMB16_S36_S36 to Verilog HDL

This section illustrates the conversion with a series of figures that show instantiations of modules in Verilog HDL. See Figure 13, Figure 14, and Figure 15.

Figure 13: Original Instantiation – RAMB16_S36_S36

```
// 512x36 block RAM
RAMB16_S36_S36 bram1  (
  .ADDRA(read_addr), .ADDRB(write_addr),
  .DIB(write_data[63:32]), .DIA(32'b0),
  .WEA(1'b0), .WEB(write_allow),
  .CLKA(read_clock), .CLKB(write_clock),
  .SSRA(1'b0), .DIPA(4'b0), .SSRB(1'b0), .DIPB(4'b0),
  .ENA(read_enable), .ENB(1'b1),
  .DOA(read_data[63:32]));

// 512x36 block RAM
RAMB16_S36_S36 bram0  (
  .ADDRA(read_addr), .ADDRB(write_addr),
  .DIB(write_data[31:0]), .DIA(32'b0),
  .WEA(1'b0), .WEB(write_allow),
  .CLKA(read_clock), .CLKB(write_clock),
  .SSRA(1'b0), .DIPA(4'b0), .SSRB(1'b0), .DIPB(4'b0),
  .ENA(read_enable), .ENB(1'b1),
  .DOA(read_data[31:0]));
```

Figure 14: Replacement Instantiation – RAMB16_S36_S36

```
// 512x36 block RAM
RAMB16_S36_S36 bram1  (
  .ADDRA(read_addr), .ADDRB(write_addr),
  .DIB(write_data[63:32]), .DIA(32'b0),
  .WEA(1'b0), .WEB(write_allow),
  .CLKA(read_clock), .CLKB(write_clock),
  .SSRA(1'b0), .SSRB(1'b0),
  .ENA(read_enable), .ENB(1'b1),
  .DOA(read_data[63:32]));

 // 512x36 block RAM
RAMB16_S36_S36 bram0  (
  .ADDRA(read_addr), .ADDRB(write_addr),
  .DIB(write_data[31:0]), .DIA(32'b0),
  .WEA(1'b0), .WEB(write_allow),
  .CLKA(read_clock), .CLKB(write_clock),
  .SSRA(1'b0), .SSRB(1'b0),
  .ENA(read_enable), .ENB(1'b1),
  .DOA(read_data[31:0]));
```

Figure 15: Replacement Code – RAMB16_S36_S36

```
module RAMB16_S36_S36 (
  input [8:0] ADDRA,
  input [8:0] ADDRB,
  input [31:0] DIB,
  input [31:0] DIA,
  input WEA,
  input WEB,
  input CLKA,
  input CLKB,
  input SSRA,
  input SSRB,
  input ENA,
  input ENB,
  output [31:0] DOA,
  output [31:0] DOB,);

  wire [31:0] DataInB, DataInA;
  assign DataInB[31:0] = DIB;
  assign DataInA[31:0] = DIA;
  wire [31:0] QA;
  wire [31:0] QB;
  assign DOA = QA;
  assign DOB = QB;

  // 512x16 EBR1
  RAMB16_S16_S16 Inst1_RAMB16_S16_S16 (
    .AddressA(ADDRA), .AddressB(ADDRB),
    .DataInB(DIB[15:0]), .DataInA(DIA[15:0]),
    .WrA(WEA), .WrB(WEB),
    .ClockA(CLKA), .ClockB(CLKB),
    .ResetA(SSRA), .ResetB(SSRB),
    .ClockEnA(ENA), .ClockEnB(ENB),
    .QA(DOA[15:0]), .QB(DOB[15:0]));

  // 512x16 EBR2
  RAMB16_S16_S16 Inst2_RAMB16_S16_S16 (
    .AddressA(ADDRA), .AddressB(ADDRB),
    .DataInB(DIB[31:6]), .DataInA(DIA[31:6]),
    .WrA(WEA), .WrB(WEB),
    .ClockA(CLKA), .ClockB(CLKB),
    .ResetA(SSRA), .ResetB(SSRB),
    .ClockEnA(ENA), .ClockEnB(ENB),
    .QA(DOA[31:6]), .QB(DOB[31:6]));
endmodule
```

Some considerations when converting Virtex II to LatticeECP/EC are that you must delete the Xilinx Verilog HDL source file from your project, insert tick marks instead of single quotation marks with include statements, timing considerations of MUXCY versus MUXCY_L, and overall timing of the design.

Converting DDR Interfaces

The key trick for the interface is shifting the DQS strobe-in pin by 90 degrees by the time it reaches the register. Unlike Virtex II, the LatticeEC and LatticeXP devices have a DQS shift circuit built in, so remove the DQS LUT-based logic.

Implement the interface with IPexpress to avoid errors. The DQS is generated from DCM. You must manually shift the clock because it does not dynamically adjust.

4

Incremental and Modular Design Methods

This chapter describes the strategies of incremental and modular design methods. It begins with the necessity for and benefits of this design approach, followed by instructions and guidelines for specific tasks and steps, such as logic partitioning, device floorplanning, and simulation. A design example is provided to illustrate the strategies in practice.

Necessity and Benefits

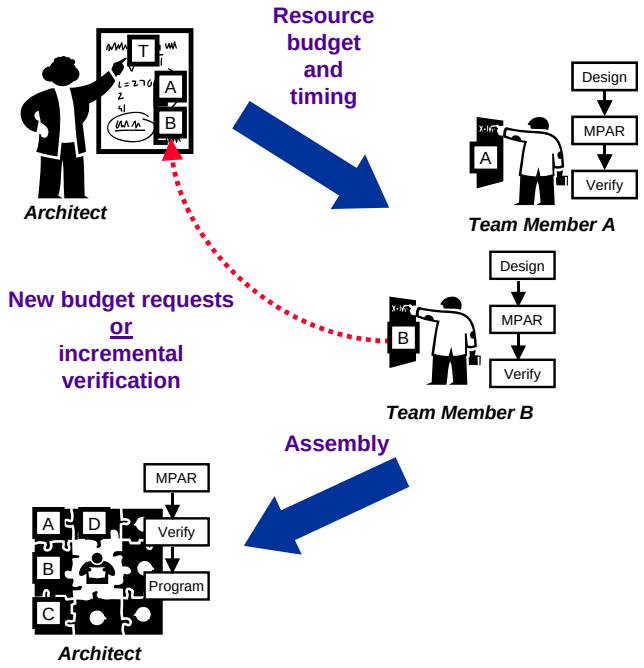
In conventional FPGA designs, a hierarchical design is flattened into a single netlist before logic synthesis and downloading, and the entire design must be recompiled for each small change. With incremental and modular design methods, you can keep part of your design unchanged as you make changes elsewhere in the design.

This approach works best for large designs that can be partitioned easily into self-contained modules on the chip. It requires good communication between design team members to ensure successful final assembly of the partitions. It also requires sound preliminary planning and iterative experimentation.

Figure 16 illustrates the team scenario that the modular design approach enables.The architect creates the top-level design. The rest of the design

team works on constituent designs that are to be merged into one cohesive design in the final assembly stage.

Figure 16: Team Scenario



Refinement feedback loops between team members and the architect are required to meet the budget. During those loops, incremental changes can be performed. The feedback loops should also accommodate incremental verification of the design at the RTL or gate levels.

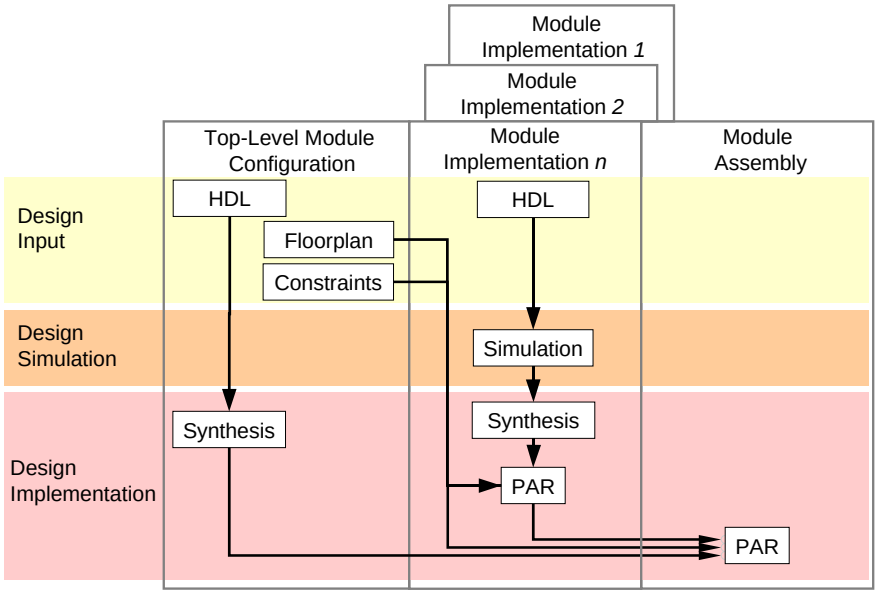
Typical Work Flow and Data Flow

Here is the typical work flow of the incremental and modular design approach:

1. Partitioning and budgeting
2. Independent implementation of each module
3. Assembly of the modules
4. Incremental change or expansion

Figure 17 shows a typical data flow of the incremental and modular design approach.

Figure 17: Typical Data Flow



First, the top-level RTL is partitioned to ease optimization, lower interconnect, and isolate critical paths. Then a top-level module budget is defined according to floorplanning and timing constraints. Usually this can be done concurrently with submodule synthesis, where each team member writes and simulates HDL sources. Ideally they can deliver area utilization (reported by synthesis tools) to the architect. The architect can then budget enough resources for each module. Finally all implemented modules are assembled with other external logic at the top level. The architect can control the degree of a submodule implementation for the final assembly.

Major Advantages

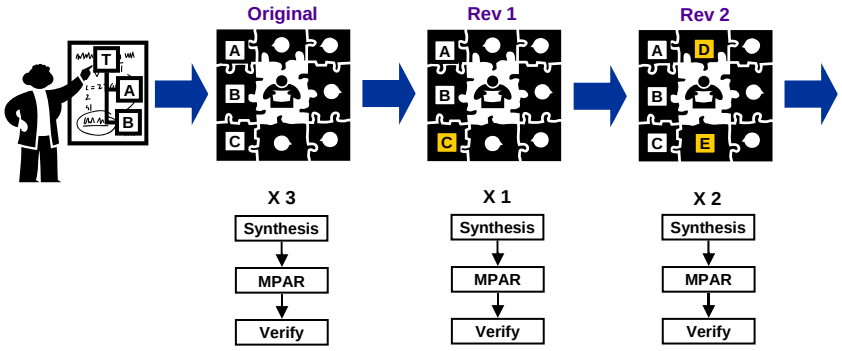
Here are the major advantages of the modular design methods:

- You can leverage the best people for the job, no matter where they are or in what area they are specialized.
- You can have multiple engineers work on a large-scale design to shorten the design period.
- Your incremental changes will have low impact to the entire design.
- Modular design complements your field upgrade strategy. For example, Lattice Semiconductor's MachXO and LatticeXP devices, along with the Lattice TransFR technology, enable you to make field upgrades with minimal down time.
- You can accommodate platform-oriented products that have different component combinations for certain markets. You can also support a family of products with different feature sets.

Incremental Changes

Modular design methods reduce the impact of incremental changes to earlier revisions of a design. Figure 18 illustrates some typical incremental change scenarios.

Figure 18: Incremental Change Scenarios



In the original revision, each module requires a synthesis, MPAR (map, place, and route), and verification flow, which is typically the most time-consuming flow.

In the first revision, only module C is to be corrected, optimized, or changed. This change should have a very low impact on other modules since only module C requires a new synthesis, MPAR, and verification pass.

In the second revision, two modules are to be added. Given a bus-oriented platform design, this again has a low impact on the entire design using the modular approach.

Identify Design Candidates

Here are some guidelines on how to identify design candidates for a modular design approach:

- The design should be large enough to warrant the extra effort of logic partitioning and floorplanning.
- The design should have clearly defined functional partitions.
- The design team must be well prepared for intense cooperation.
- The architect should be familiar with the device architecture and locality of certain resources like embedded blocks, specialized PIOs, and logic fabric.

You can run your block modular design project with the help of ispLEVER's Block Modular Design Wizard or entirely from the command line. Incremental changes can be easily realized at various stages of the ispLEVER design flow.

Block Modular Design Flow

The BMD (Block Modular Design) Wizard in the ispLEVER software assists distributed teams in collaborating on large FPGA designs. When used with incremental design strategy, it is especially effective in limiting changes to a design and minimizing impact to other modules in the design. The major process steps in the BMD design flow in ispLEVER include:

- Step 1. Top-Level Design Entry

  A top-level model is created in HDL with constituent design modules as black boxes, using good logic partitioning guidelines.

- Step 2. Block Module Synthesis

  The HDL design files for each block are synthesized. The utilization estimates reported by synthesis guide the top-level architect to budget enough resources for that module. Synthesis can be performed on blocks in any order.

- Step 3. Block Module Configuration

  In this step, the top-level architect budgets the resources and the timing target for each submodule. Each module is allocated a region with an anchor point and border. In the ispLEVER design flow, the top-level architect generates projects to archive and deploy to each team member.

- Step 4. Block Module Implementation

  This step implements each block and applies the top-level design constraints. This must be completed before final assembly. The top-level floorplan with region constraints must already be completed before this step.

  Successful implementation of blocks depends largely on the preferences assigned for area budgeting and reservation and I/O placement determined in the previous step. If incorrect, steps 3 and 4 must be repeated.

- Step 5. Assembly

  In this final step, all the blocks are merged into one cohesive design.

You can refer to the “Block Modular Design Step Guide” in the ispLEVER online Help for more detailed information on each step.

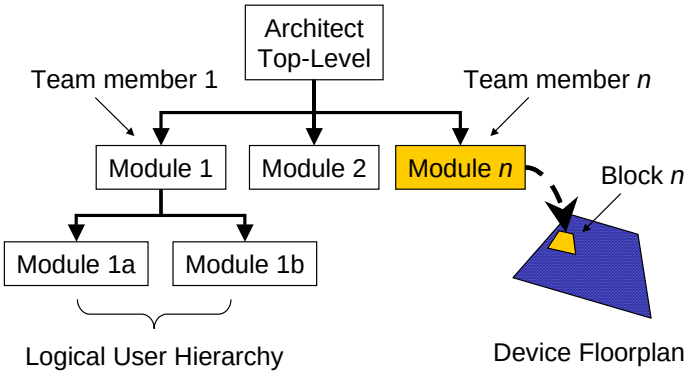
Logic Partitioning

Good results from the modular design approach begin with good logic partitioning of your RTL design to simplify the coding, synthesis, simulation, floorplanning, and optimization phases of the design.

Resource partitioning and budgeting is an iterative process. You should be aware of the required resources in terms of size and performance of each submodule. When assembling submodules that are already implemented, you can take advantage of the post-map results to guide your resource budgeting.

Figure 19 illustrates a partitioned FPGA design. The convention for most FPGA tools today is to allocate a branch of the design hierarchy to each module, along with a budget for timing and device resources. In a team environment, each team member can establish a logical user hierarchy to the degree appropriate for that design module.

Figure 19: Sample Partitioned FPGA Design



Partitioning Guidelines

Commonly recommended partitioning guidelines include the following:

- Submodules should be synchronous with registered I/Os. Registering the I/Os of a module isolates critical paths and helps the synthesis tool to implement the combinatorial logic and registers in the same module.
- Related combinatorial and arithmetic logic should be collected in the same module. Keeping related combinatorial terms and arithmetic in the same design module allows logic hardware resources to be shared. It also allows a synthesis tool to optimize the entire critical path in a single operation.
- Pieces of logic with different optimization goals should be separated. Separating critical paths from non-critical paths makes logic synthesis more efficient. If one portion of a design module must be optimized for size and a second portion must be optimized for speed, the two portions should be separated into two design modules.
- Interconnect between modules should be minimized to avoid routing congestion later when the design is assembled and routed.

- Use separate files as a housekeeping measure to avoid unnecessary recompiling of logic during incremental synthesis.

Directory Structure

Once logic partitioning is completed, you should create a proper directory structure for your top-module and submodule projects. If you use the Project Navigator and Block Modular Design Wizard tools to manage the design, the top-level design module is created in the default project directory, and each submodule is written to a subdirectory of the project directory. The Block Modular Design Wizard handles your directory structure automatically as you create the top-level and constituent submodules in the Wizard interface. In the command-line flow, you must define a root directory that contains a subdirectory of the root for submodule files.

Device Floorplanning

After the logic partitioning stage, you define a top-level module budget in terms of a floorplan and timing constraints. Usually you can define this budget when you synthesize the submodules. Each team member writes and simulates HDL and should be far enough along to deliver area utilization numbers reported by the synthesis tools to the architect, so they can budget enough resources for that module.

Device floorplanning is used in two contexts in the modular design flow: in the physical partitioning of modules and, optionally, in a module itself to achieve timing closure. The top-level floorplan should consider both the FPGA elements required per module and the relative data flow between modules.

Top-Level Floorplanning Procedures

Top-level floorplanning for modules is a critical task in the block module configuration step of the ispLEVER BMD flow. It typically includes the following procedures:

- Determine the best relative position of each module.

  The best way to visualize this is from the RTL schematic view commonly available from your synthesis tool, as well as a floorplan view where you can see PIO interconnect.

- Lock global resources like PIOs and PLL/DLLs.

  FPGAs may provide specialized I/O drivers for double data rate (DDR) or serializer/deserializer (SERDES) interfaces at specific locations of the device package, so you must allow for these locations in the floorplan.

- Allow for embedded FPGA blocks.

  The logic fabric of Lattice FPGAs is commonly split by rows of embedded blocks like memory or DSP functions. These also influence the position of modules.

- Allow for irregular shapes.

Apply prohibit or reserve regions within a module to create irregular shapes for FPGA systems with rectangular module shapes.

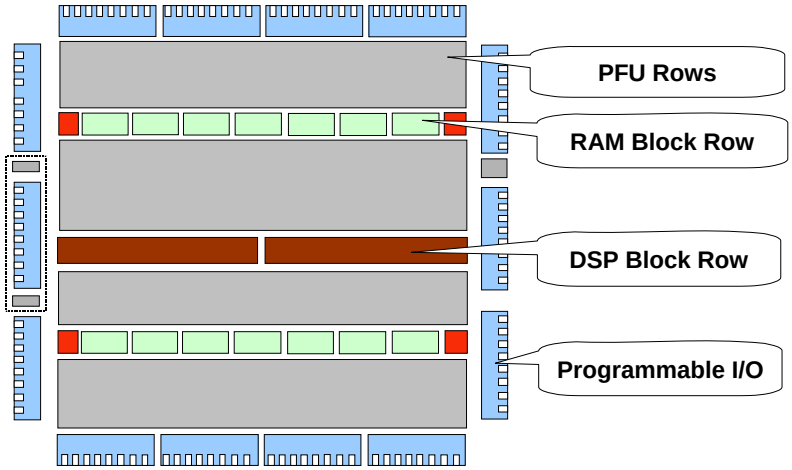
- Allow for future upgrades.

  Allow some regions for future upgrades. For products with long life cycles, such as aerospace or medical equipment, you may want to keep some unused resources for future field upgrades.

Taking Architectures into Account

The logic fabric of modern FPGAs is commonly split by rows of embedded blocks like memory or DSP functions, as shown in Figure 20.

Figure 20: Modern FPGA Device Architecture (LatticeECP)



Sometimes a segmented FPGA architecture makes it difficult to create a floorplan with a good data flow using rectangular module shapes. To address this problem, ispLEVER supports both resource sharing by module overlapping and the ability to mask off overlapped regions that are reserved for other modules.

Block-Level Implementation

Module implementation and block-level placement and routing is normally carried out in the block module implementation step of the BMD flow. The place-and-route tools use the floorplan established earlier to constrain the results to a particular area of the device. In an ideal team environment, this can be done largely in parallel. Synthesis and simulation tasks can be performed on individual modules or as part of an integrated build.

Since the success of the top-level budget is closely related to block-level size and timing reports generated from synthesis, these reports serve the architect as a guide for floorplanning and timing budgeting. This bottom-up approach is the fastest way to arrive at a good physical partition.

Top-Level Assembly

Top-level assembly stitches submodules together according to annotated data. Some FPGA vendors allow different degrees of annotation ability but usually placement is the most common. Often a global reroute on assembly is a necessary and beneficial technique to meet timing. After top-level assembly, you can perform the final timing analysis and simulation.

Here are some troubleshooting tips:

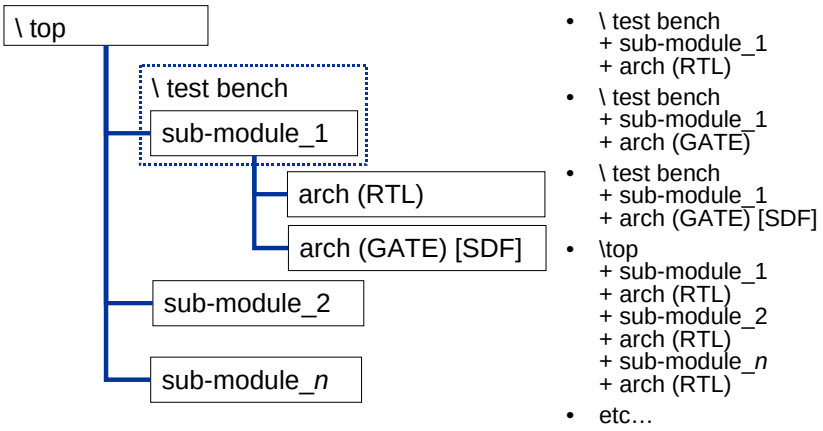
- At this stage, module overlaps designed earlier may cause resource conflicts. You may need to adjust modules to obtain a fit.
- The biggest side effect of a partitioned and floorplanned design often occurs after assembly when critical paths may cross module boundaries. This effect is usually the result of unsuccessful floorplanning of the module itself to account for the data flow.

Simulation Scenarios

You might employ the following simulation scenarios when you use a modular design approach.

Figure 21 shows a hierarchical view of a modular design with test bench and alternative implementations superimposed. You can see the \top top-level design with submodule_1 through _*n*. Each submodule can employ its own test bench to verify the functionality of the RTL and timing using the gate-level implementation. Another option is to leverage the RTL for surrounding modules to serve as drivers or loads.

Figure 21: Simulation Scenarios



Incremental Design Methods

The benefits of modular FPGA design become apparent when an incremental change or expansion of a design is required. Until modular methods were widely available from within the FPGA implementation tools, incremental changes were largely done on the flattened FPGA design that increased the run time for each change made. Now you can see how modular methods extend the options for incremental changes to a design.

The changes required determine the best place to start:

- If the model behavior must be changed, you must modify the HDL source and then trigger a re-synthesis.
- If you want to change an option after synthesis, like PAR optimization, timing, or location constraints, you can start from the PAR stage.
- Some device characteristics like memory initialization or PLL parameters can be changed after placement and routing. This is usually done with an ECO (engineering change order) post-processor or device editor utility. Changes taken at this point must be carefully documented because your physical implementation is out of synchronization with your original HDL source and preferences.

Some of the incremental design tools available in the FPGA design flow include the following:

- Incremental synthesis

  Incremental synthesis is available from most FPGA synthesis tools like Precision RTL Synthesis and Synplify. They allow you to create logical partitions and to compile and synthesize each partition independent of other partitions. You can benefit from isolated changes, and you can distribute the changes among other designers.

- Device editor

  You can use a device editor like ispLEVER's EPIC or a batch interface to update database parameters. These sorts of editors are best employed if you need to make a small, precise change very quickly and you are willing to have the physical implementation differ from the source files.

- ECO (engineering change order) post-processing

  Examples of ECO changes include changes to I/O buffer configuration, memory initialization, a PLL parameter, or additional routing to feed internal signals to a PIO for the sake of troubleshooting.

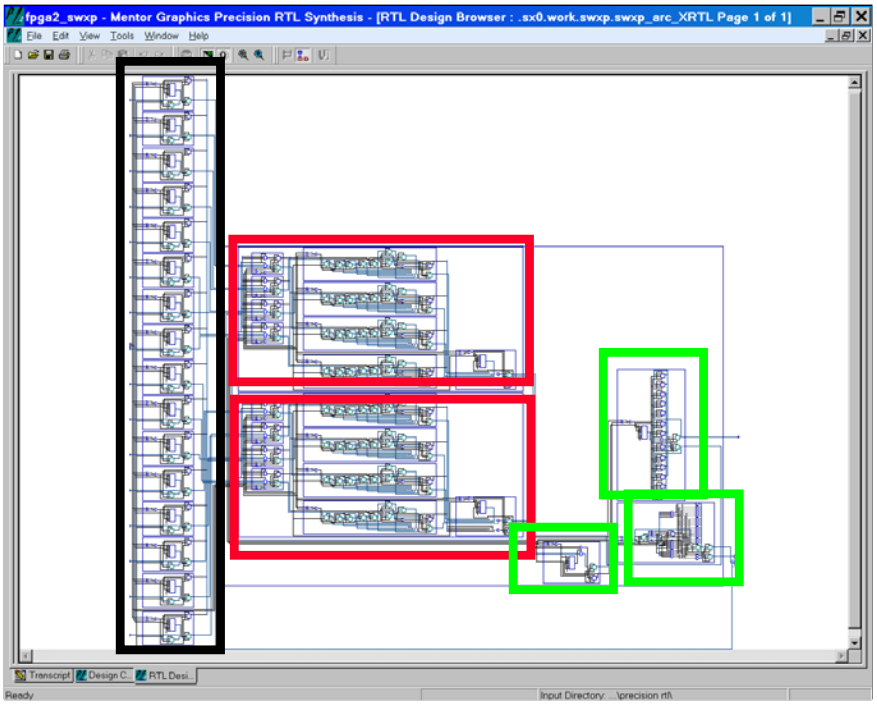
Design Example

This section provides a design example to better understand the ideas and tools involved in incremental and modular design methods.

Logic Partitions and Data Flow

Figure 22 shows the RTL view of the entire design example. It is a data-path intensive design with several pipelined data channels. These data channels are multiplexed and controlled by an internal timing circuit. The schematic of the blocks and interconnect generated from Precision RTL Synthesis is a good way to determine logic partitions. The partitioning is illustrated with black-, red-, and green-colored frames in the figure.

Figure 22: Logic Partitions of the Design Example

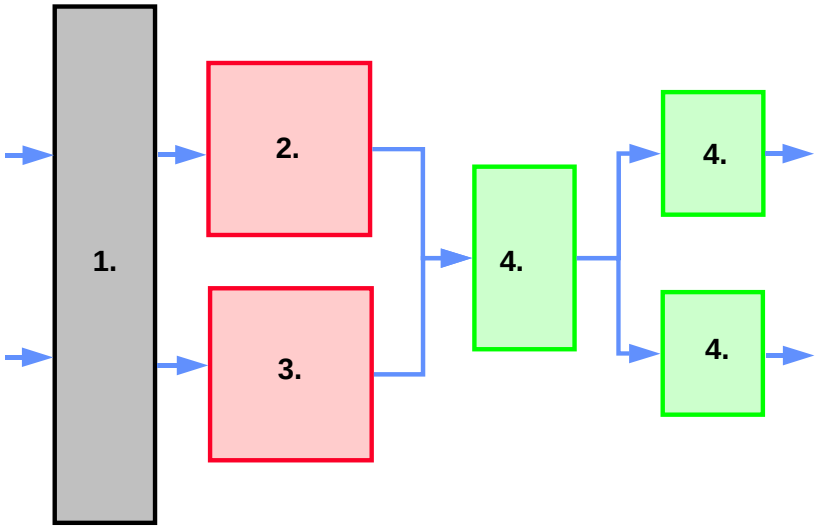


For a modular approach, it is recommended that you adjust logic synthesis to optimize module size and retain the hierarchy, which makes identification and grouping of logic easier for the implementation tools.

The example demonstrates a bottom-up approach in which you have all the RTL available as a guide to establish a module area budget. In practice, a design architect is likely to apply both bottom-up and top-down approaches, and budget modules with both accurate and inaccurate information.

Figure 23 shows a block diagram of the design data flow. The data flow influences the external PIO placement and the relative placement of the submodules.

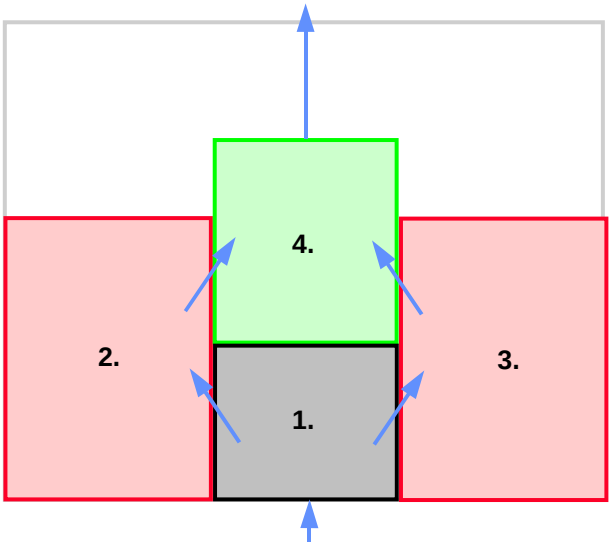
Figure 23: Data Flow Illustrated



Floorplan Sketch

Figure 24 shows the floorplan of the design example. In this design, most of the PIOs are input channels that feed the No.1 black block in the center bottom. The blue arrows illustrate the relative data flow among blocks. These connections influence any additional floorplanning within each module. The white empty region illustrates logic resources reserved for later expansion or changes of the design.

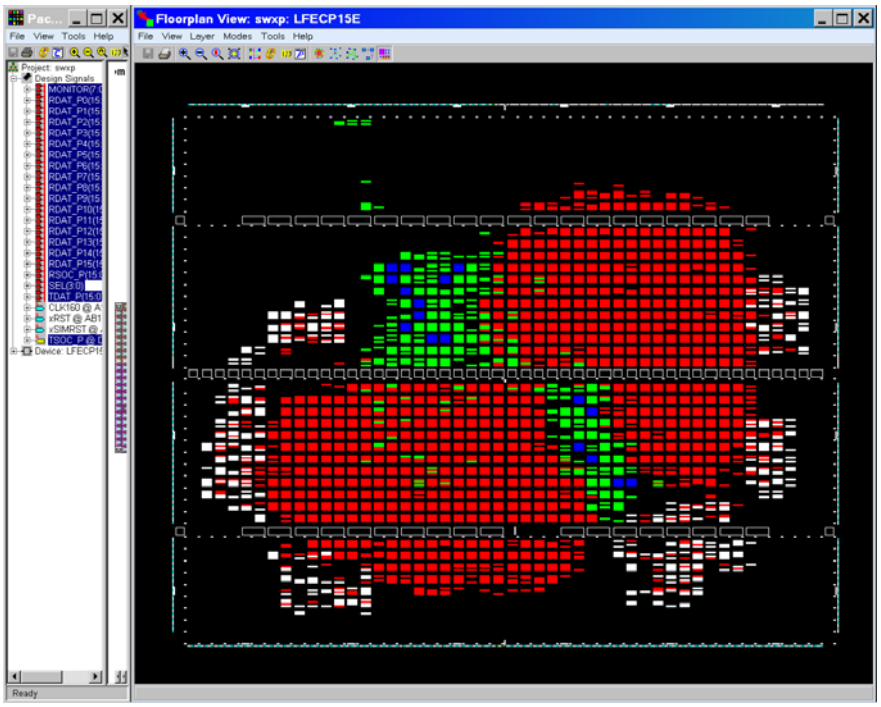
Figure 24: Floorplan Sketch of the Design Example



Flat Implementation

In this design example, all the RTL is available. You can perform a flat implementation to gain a preliminary understanding of the relative placement and area consumption of each logic partition. In Figure 25, the four blocks are highlighted in white, red, and green. Blocks are placed adjacent to certain physical resources. Their relative positions are based on connectivity and critical paths. In this way, you can obtain an early prototype of the device floorplan.

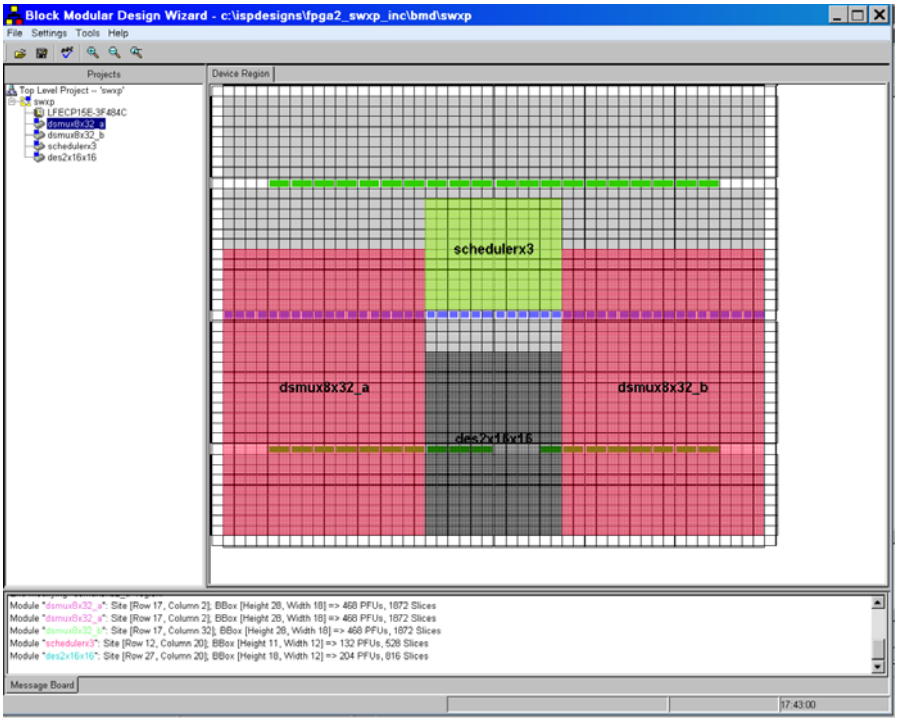
Figure 25: Flat Implementation of the Design Example



Submodule Floorplan

Figure 26 shows the design example in the ispLEVER Block Modular Design Wizard tool, where you can anchor and resize each module. The four module regions are allocated in black, red, and green. The Wizard reads the top-level design netlist, where each module is a black box.

Figure 26: Submodule Floorplan



The BMD Wizard in ispLEVER makes the design flow much easier. As the position of each module is anchored and the size of each module is decided, the resources of each module are reported in real time. Once a module is created, the design architect can archive the top- and submodule project for team members to perform placement and routing.

The BMD Wizard in ispLEVER also eases design migration. If the target device, package, or speed grade is updated, all the projects are updated with this information. And the device floorplan can be quickly validated, given a new resource set.

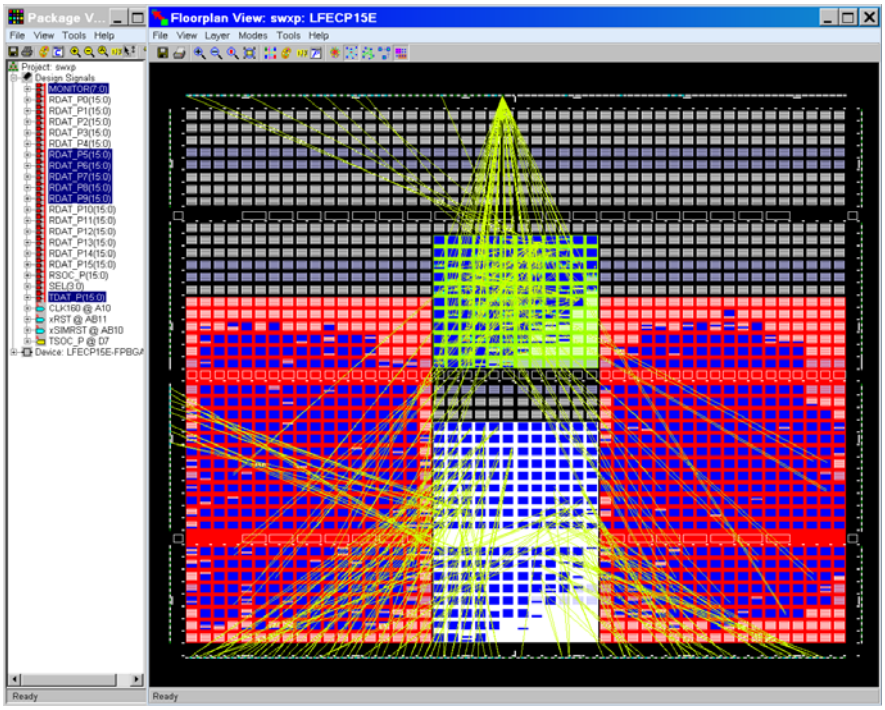
I/O Connectivity

At an early implementation stage, you can check the relative data flow between the modules and the PIO connections. In this example, the submodules are only placed, but not routed, for the sake of speed. So at the final assembly stage, you can inspect the utilization, placement, and connections.

Figure 27 shows the design example in the ispLEVER Design Planner with the Package View on the left and the Floorplan View on the right. You can

cross-probe between the external I/Os and the floorplan. The yellow flywires show logical connections.

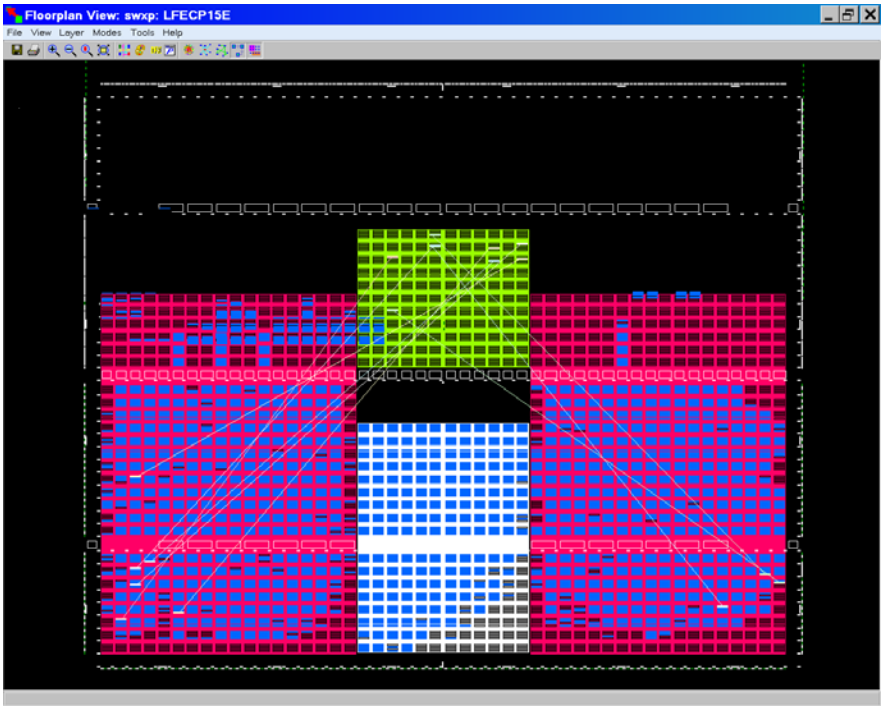
Figure 27: Design Example in ispLEVER Design Planner



Critical Paths

Figure 28 shows the results of the fully placed and routed design in the Design Planner. It illustrates a side effect of floorplanning: new critical paths now cross submodule boundaries. The critical paths are shown as flywires here again, running from the corners of the red modules up to the green module in the center. Even with these critical side effects, the design is already very close to meeting the timing constraints. With some more floorplanning to re-orient the output driver branch of the submodules on the left and right, you can easily exceed the goals.

Figure 28: Critical Paths



Sample Design Implementation

Figure 29 compares the utilization, performance, and run time of the conventional flat design methods with those of the modular design methods. The first column is for a basic flat and unfloorplanned implementation. Utilization is around 47 percent, with speed optimization selected with Precision RTL synthesis. Although you get quick results with about 6 minutes of PAR run time, the target frequency is not met. The multiplier with question marks indicates that there are likely multiple runs to close timing. Different placement seeds or routing delay reduction must be conducted to improve the design implementation results.

Figure 29: Comparison of Design Implementation Results

	Flat [1]	BMD [2]
Utilization		
SLICE	**47%**	**51%**
Performance [3]	**fMAX (Levels)**	**fMAX (Levels)**
CLK	**149.6 MHz (2)**	**160.7 MHz (2)**
PAR Runtime		
	5m 58s x ???	**1. 1m 20s** **2. 3m 58s** **3. 3m 58s** **4. 1m 40s** **Assembly: 16m**

Table Notes:
1. PAR: Placement effort=5, Routing iterations=6
2. Synthesis: Area optimized with “firm” hierarchy
3. LPF: Frequency=160 MHz

The second column shows the implementation results, using the block modular design methods. Area utilization is a bit larger than that of the flat implementation. But you have achieved better target frequency. What is more important is that the design is now in a much better state for expansion or updating in future revisions.

Conclusion

New modular FPGA design techniques provide major advantages to distributed design teams. Portions of an entire design can be processed independently, allowing multiple designers to work in parallel. The timing of each constituent functional module is preserved because each module can be assigned to a particular region on the device, and the tools are constrained to use resources from that region.

To get the best results from a modular approach, quality logic partitioning and quality floorplanning are needed to ease optimization, device usage, and timing closure.

Related Documentation

To supplement the information provided in this chapter, see the following documentation for related topics and guidelines:

- “Block Modular Design Step Guide” in the ispLEVER software online Help
- *FPGA Block Modular Design Tutorial*

5

Logic Synthesis Guidelines

This chapter provides general guidelines for creating register-transfer-level (RTL) designs. It also provides syntax examples for VHDL and Verilog HDL in Synplify and Precision RTL Synthesis.

Synthesis Design Flow and Guidelines

The largest influence you have over the performance and utilization of an FPGA is how your logic design is expressed and synthesized. Lattice Semiconductor recommends following the guidelines and flow described in this section. It also recommends that you study the synthesis style topics provided in the *Precision RTL Synthesis Style Guide* or the *Synplicity FPGA Synthesis Reference Manual* to write the best RTL source possible.

Note

You can view the P*recision RTL Synthesis Style Guide* and the *Synplicity FPGA Synthesis Reference Manual* by clicking the appropriate links in the "Third-Party Manuals" topic in the ispLEVER software online Help. You must install the synthesis tools to be able to view the manuals.

The following steps outline the general logic synthesis design flow recommended for Lattice Semiconductor FPGAs:

1. Create a design in Verilog HDL or in VHDL. The designs can be technology-independent or contain family-specific modules or IP cores; however, they cannot contain instances of library elements from other technology libraries. Refer to the Lattice Semiconductor *FPGA Libraries Manual* for module names that are reserved by Lattice Semiconductor for

each device family.

Note

You can view the *FPGA Libraries Manual* by clicking the appropriate link in the ispLEVER online Help “Software User Manuals” topic.

Consider using the following HDL coding styles to obtain the best results:

- Hierarchical coding style
- Design partitioning
- Design registering and pipelining
- Avoiding gated clocks
- Avoiding unintentional latches
- State machine encoding

2. When possible, use the IPexpress tool in the ispLEVER software to create a module. The output from IPexpress can be in EDIF, VHDL, or Verilog HDL. Since the modules generated are optimized for Lattice Semiconductor device architectures, they often provide speed and area benefits over netlists produced by a synthesis tool. The output from IPexpress (VHDL or Verilog HDL only) include an instantiation template that can be included directly in your design.
3. Verify that the design description is correct by simulating the HDL description with the ModelSim software or any HDL-compliant simulator.
4. Create a synthesis project by using the Precision RTL Synthesis software or Synplify software. Consider using the following synthesis mapping options to obtain the best results:
   - Fan-out limit: High-fan-out signals (greater than 100) can cause large delays and routability problems within an FPGA unless that signal can be assigned to specialized routing resources of the device. These “clock-spine”-type resources are ideal for clock, set/reset, or clock-enable signals. For other signals, most synthesis mappers try to keep the fan-out under a predefined fan-out limit.
     - Guidelines for Precision RTL Synthesis: To limit fan-out, Precision RTL Synthesis is guided by technology libraries that specify a global fan-out value. The default fan-out limit is 100. Specific cells like global buffers that are designed for high-fan-out situations carry a larger fan-out limit. Precision RTL Synthesis allows you to override the library default value on a per-net basis, using the max_fanout attribute. Precision RTL Synthesis addresses fan-out violations by splitting the net and replicating the driving cell. If replication is not possible, Precision RTL Synthesis will add buffers.
     - Guidelines for Synplify: The Synplify fan-out guide option uses the number specified as a guideline, and not as a hard limit. Synplify first reduces fan-out by replicating the driver of the high-fan-out net and splitting the net into segments. If replication is not possible, Synplify buffers the signals. Buffering is expensive both in terms of

intrinsic delay and consumption of resources, so it is not used unless a slightly higher fan-out limit is specified.

- Mapping to GSR resources: Lattice Semiconductor FPGA devices provide a dedicated prerouted resource for a global set/reset (GSR) that is connected to the set/reset of each flip-flop in the FPGA. The GSR is connected, regardless of any other set or reset defined by your design. Most synthesis mappers attempt to associate a common reset or set of your design with the GSR resource.
  - Guidelines for Precision RTL Synthesis: During the synthesis process, Precision RTL Synthesis analyzes the design to detect global set/reset signals and map to a GSR buffer. By default, if there is a single reset used in the design, Precision RTL Synthesis will connect that reset signal to a GSR instance, even if some flip-flops have no reset at all.
  - Guidelines for Synplify: Synplify creates a GSR block to access the GSR resource for Lattice Semiconductor FPGAs if it is appropriate for the design. By default, if there is a single reset used in the design, Synplify will connect that reset signal to a GSR instance, even if some flip-flops have no reset at all. Usually, flip-flops without set or reset can be safely initialized because the reset is only used when the device is turned on. If this is not the case, you must turn off the Force GSR Usage option. When this option is turned off, Synplify requires that all flip-flops have resets and that the resets be the same before it uses GSR.
- Disable I/O mapping: In some design scenarios, such as incremental or block modular design, you may wish to suppress the addition of I/O buffers to the EDIF output for your project. Synthesis mappers typically provide the option to override the I/O technology cell targeted.
  - Guidelines for Precision RTL Synthesis: The Add IO Pads option controls whether I/O buffers are added to the output EDIF netlist. The Pad Type port constraint directs what IO pad is applied.
  - Guidelines for Synplify: The Synplify Disable I/O Insertion option controls whether I/O buffers are added to the output EDIF netlist.

5. Perform logic synthesis of the design description, using Precision RTL Synthesis or Synplify to meet a desired area and timing target. If you are within 5 to 10 percent of your desired goal, you can map, place, and route the design. If not, return the design with additional constraints, recode, or both until you achieve the desired results. You should consider using the following synthesis optimizations to obtain the best results:
   - Timing constraints: These constraints should include a period or frequency target for all clocks, multi-cycle relationships, false paths, and I/O timing.
   - Area versus timing optimization: You can set this option globally or on a module-by-module basis, if this approach can help achieve timing closure by reducing signal congestion.
   - State machine encoding: You can set this option on a machine-by-machine basis to influence timing or area results.

- Guidelines for Precision RTL Synthesis: Precision RTL Synthesis provides area, timing, and timing violation reports based on your constraints. Schematic views are available to examine synthesis results. If you know the delays outside your chip for inputs and outputs, set them with input or output delay port constraints.
- Guidelines for Synplify: Basic area and timing analysis reports are produced by Synplify. For more complete review and analysis, use Synplify HDL Analyst to examine the RTL or technology schematics and critical paths. If you know the delays outside your chip for inputs and outputs, set them with SCOPE or the define_input_delay and define_output_delay timing constraints.

6. Complete an iteration of the map, placement, and routing design flow through ispLEVER. Examine the static timing analysis results. The output of the TRACE program reports delay through one or more critical delay paths.
   - Guidelines for Synplify: If you do not meet your timing goals, you can resynthesize your design with code changes or add or subtract route delay differences using the –route option with the define_input_delay and define_output_delay timing constraints defined earlier. To have Synplify restructure your design to speed up paths, add the –improve option to the timing constraints.
7. Using timing closure techniques, use ispLEVER to map, place, and route the design.
8. Verify that the post-route, gate-level design description is correct by simulating the HDL output from ispLEVER with the ModelSim software or any HDL-compliant simulator.

Reports Produced by Synthesis

Report files created by the synthesis phase provide essential information for you to understand the eventual timing and device utilization of the design.

Both Precision RTL Synthesis and Synplify create a resource usage report that lists the number of each type of cell used, including the number of look-up tables, registers, memories, and DSP blocks.

Timing reports created by Precision RTL Synthesis and Synplify typically show all instances and connections in the design that are near the critical path. “Near” means within one “level” of the most critical path. The reported instances are sorted by the arrival time at their pins, with the path ends at the top of the report and the path beginnings at the bottom. Each connection is listed with the delay to the connection and the length of the longest path passing through the connection.

Typically, path ends are inputs to flip-flops or primary output cells. Path beginnings are flip-flop outputs and primary input cells. You can trace paths through the report by matching net names on connections. Timing reports by synthesis are an estimate. The actual timing of the design depends heavily on placement and routing, the device, and speed grade.

To obtain a better estimate of the logic delays, run the Map TRACE Report process in the Project Navigator (trce) on the mapped design. After this step, go through one iteration of Place & Route Design process (PAR) and the Place & Route TRACE Report process (trce) to obtain an estimate of the routing delays and the critical paths in the design. The final frequency of the design depends on a number of factors: the number of logic levels in the design, the number of connections and number of nets in the design, the packing factor (also number of inputs to a PFU), utilization of tristate buffers, number of I/Os (and number of tristatable I/Os and the number of controls), the device size, and speed grade.

Related Documentation

To supplement the information provided in this chapter, see the following documentation for related topics and guidelines:

- The ispLEVER online Help
- TN1056 - *LatticeECP/EC and LatticeXP sysIO Usage Guide*
- *Precision RTL Synthesis Style Guide*
- *Synplicity FPGA Synthesis Reference Manual*
- *FPGA Libraries Manual*

6

HDL Synthesis Coding Guidelines

Coding style has a considerable impact on how an FPGA design is implemented and ultimately how it performs. Although many popular synthesis tools have significantly improved optimization algorithms for FPGAs, it is still the designer's responsibility to generate HDL code that guides the synthesis tools and achieves the best result for a given architecture. This chapter provides VHDL and Verilog HDL design guidelines for both novice and experienced designers.

The synthesis software itself has a significant effect on implementation. The style of the code that you employ in one synthesis tool for one outcome can vary greatly from that in another tool. Synthesis tools optimize HDL code for logic utilization and performance, but they do so in a way that might not be close to your intended design. Knowing the effects of these synthesis tools, as well as knowing the most efficient HDL code for your design, are both important.

This chapter also shows how to employ the "linting" technology of the HDL Explorer software to produce higher quality code. This analysis tool detects common design rule faults that can cause mismatches between pre-synthesis and post-synthesis behavior.

General HDL Practices

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

Hierarchical Coding

An HDL design can either be synthesized as a flat module or as many small hierarchical modules. Each methodology has its advantages and disadvantages. Since designs in smaller blocks are easier to keep track of, applying a hierarchical structure to large and complex FPGA designs is preferable. Hierarchical coding methodology allows a group of engineers to work on one design at the same time. It speeds up design compilation, makes changing the implementation of key blocks easier, and reduces the design period by allowing the re-use of design modules for current and future designs. In addition, it produces designs that are easier to understand.

However, if the design mapping into the FPGA is not optimal across hierarchical boundaries, it leads to lower device utilization and design performance. You can overcome this disadvantage with careful design consideration when choosing the design hierarchy.

Here are some tips for building hierarchical structures:

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

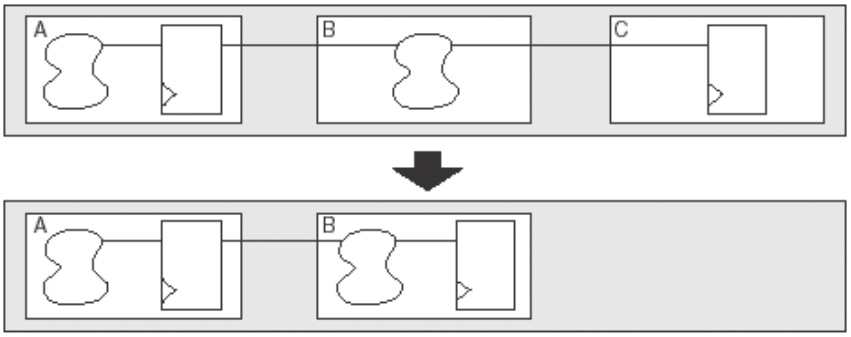
Design Partitioning

By effectively partitioning the design, you can reduce overall run time and improve synthesis results. Here are some recommendations for design partitioning. In the following descriptions, sub-blocks and blocks refer to either Verilog HDL modules or VHDL design units.

Maintain Synchronous Sub-Blocks by Registering All Outputs

Arrange the design boundary so that the outputs in each block are registered. Registering outputs helps the synthesis tool implement the combinatorial logic and registers in the same logic block. Registering outputs also makes the application of timing constraints easier since it eliminates possible problems with logic optimization across design boundaries. Using a single clock for each synchronous block significantly reduces the timing consideration in the block. It leaves the adjustment of the clock relationships of the whole design at the top level of the hierarchy. Figure 30 shows an example of synchronous blocks with registered outputs.

Figure 30: Synchronous Blocks with Registered Outputs

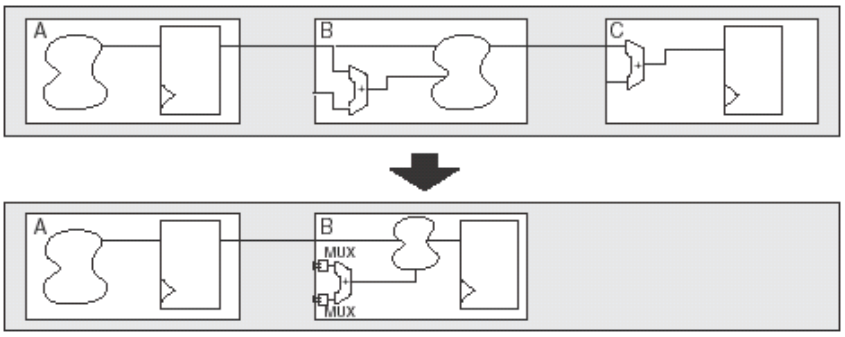


Keep Related Logic Together in the Same Block

Keeping related logic and sharable resources in the same block allows the sharing of common combinatorial terms and arithmetic functions within the block. It also allows the synthesis tools to optimize the entire critical path in a single operation. Since synthesis tools can only effectively handle optimization of certain amounts of logic, optimization of critical paths pending across the boundaries may not be optimal. The example in Figure 31 merges

sharable resource in the same block.

Figure 31: Merging Sharable Resource in the Same Block



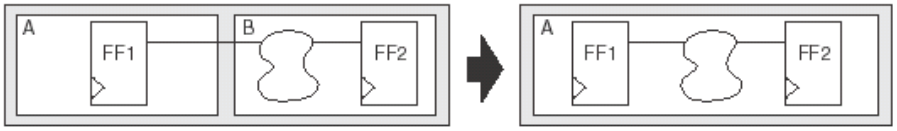
Separate Logic with Different Optimization Goals

Separating critical paths from non-critical paths may achieve efficient synthesis results. At the beginning of the project, you should consider the design in terms of performance requirements and resource requirements. If a block contains two portions, one that needs to be optimized for area and a second that needs to be optimized for speed, they should be separated into two blocks. By doing this, you can apply different optimization strategies for each module without the two modules being limited by one another.

Keep Logic with the Same Relaxation Constraints in the Same Block

When a portion of the design does not require high performance, you can apply this portion with relaxed timing constraints, such as Multicycle, to achieve high utilization of a device area. Relaxation constraints help to reduce overall run time. They can also help to efficiently save resources, which can be used on critical paths. Figure 32 shows an example of grouping logic with the same relaxation constraint in one block.

Figure 32: Logic with the Same Relaxation Constraint

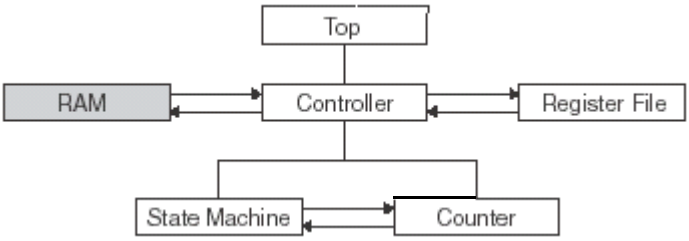


Keep Instantiated Code in Separate Blocks

Leave the RAM block in the hierarchy in a separate block, as shown in Figure 33, to enable easy swapping between the RAM behavioral code for simulation and the code for technology instantiation. In addition, this coding style facilitates the integration of the ispLEVER IPexpress tool into the

synthesis process.

Figure 33: Separate RAM Block



Keep the Number of FPGA Gates at 30 to 80 PFU Per Block

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

Design Registering

Pipelining can improve design performance by restructuring a long data path with several levels of logic and breaking it up over multiple clock cycles. This method allows a faster clock cycle by relaxing the clock-to-output and setup time requirements between the registers. It is usually an advantageous structure for creating faster data paths in register-rich FPGA devices. Knowledge of the FPGA’s architecture helps in planning pipelines at the beginning of the design cycle. When the pipelining technique is applied, special care must be taken for the rest of the design to account for the additional data path latency. The following illustrates the same data path before (Figure 34) and after pipelining (Figure 35).

Figure 34: Before Pipelining

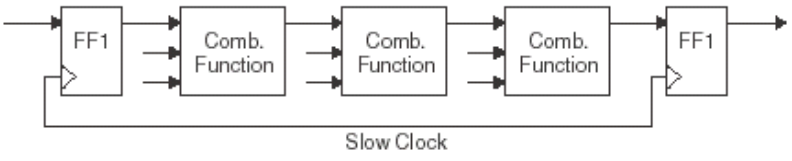
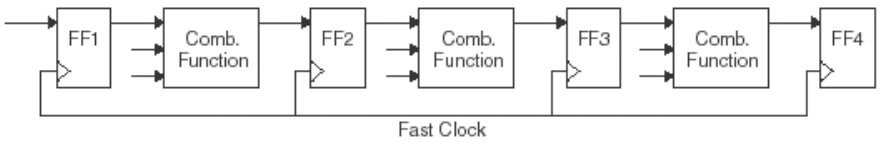


Figure 35: After Pipelining



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

Comparing If-Then-Else and Case Statements

Case and if-then-else statements are common for sequential logic in HDL designs. The if-then-else statement generally generates priority-encoded logic, whereas the case statement implements balanced logic. An if-then-else statement can contain a set of different expressions, but a case statement is evaluated against a common controlling expression. Both statements give the same functional implementation if the decode conditions are mutually exclusive, as shown in Figure 36.

Figure 36: Case and If-Then-Else Statements with Mutually Exclusive Conditions

Case Statement

```
process (s, x, y, z)
begin
  O1 <= '0';
  O2 <= '0';
  O3 <= '0';

  case (s) is
    when "00" => O1 <= x;
    when "01" => O2 <= y;
    when "10" => O3 <= z;
    when others => O1 <= '0'; O2 <= '0';
      O3 <= '0';
  end case;

end process;
```

If-Then-Else

```
process (s, x, y, z)
begin
  O1 <= '0';
  O2 <= '0';
  O3 <= '0';


  if s = "00" then O1 <= x;
  elsif s = "01" then O2 <= y;
  elsif s = "10" then O3 <= z;
  else O1 <= '0'; O2 <= '0'; O3 <= '0';
  end if;


end process;
```

However, the use of the if-then-else construct could make the design more complex than necessary, because extra logic is needed to build a priority tree. Consider the examples in Figure 37.

Figure 37: If-Then-Else Statement With Lower Logic Requirement

Complex O3 Equations

```
process (s1, s2, s3, x, y, z)
begin
  O1 <= '0';
  O2 <= '0';
  O3 <= '0';

  if s1 = '1' then O1 <= x;
  elsif s2 = '1' then O2 <= y;
  elsif s3 = '1' then O3 <= z;
  end if;

end process;
```

Simplified O3 Equations

```
process (s1, s2, s3, x, y, z)
begin
  O1 <= '0';
  O2 <= '0';
  O3 <= '0';

  if s1 = '1' then O1 <= x;
  end if;
  if s2 = '1' then O2 <= y;
  end if;
  if s3 = '1' then O3 <= z;
  end if;

end process;
```

If the decode conditions are not mutually exclusive, the if-then-else construct causes the last output to be dependent on all the control signals. The equation for O3 output in example A is:

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

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

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

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

Avoiding Unintentional Latches

Synthesis tools infer latches from incomplete conditional expressions, such as an if-then-else statement without an else clause. To avoid unintentional latches, specify all conditions explicitly or specify a default assignment. Otherwise, latches are inserted into the resulting RTL code, requiring additional resources in the device or introducing combinatorial feedback loops that create asynchronous timing problems. Unintentional latches can be avoided by using clocked registers or by employing any of the following coding techniques:

- Assign a default value at the beginning of a process.
- Assign outputs for all input conditions.
- Use else (when others) as the final clause.

Another way to avoid unintentional latches is to check the synthesis tool outputs. Most of the synthesis tools give warnings whenever there are latches

in the design. Checking the warning list after synthesis saves a tremendous amount of effort in trying to determine why a design is so large later in the place-and-route stage.

Register Control Signals

The general-purpose latches and flip-flops in the PFU are used in a variety of configurations, depending on the device family.

For example, in the LatticeEC family of devices, you can apply clock, clock-enable, and LSR control to the registers on a slice basis. Each slice contains two LUT4 lookup tables feeding two registers (programmed to be in flip-flop or latch mode) and some associated logic that allows the LUTs to be combined to perform functions, such as LUT5, LUT6, LUT7, and LUT8. Control logic performs set/reset functions (programmable as synchronous/asynchronous), clock-select, chip-select, and wider RAM/ROM functions.

When writing design codes in HDL, keep the architecture in mind to avoid wasting resources in the device. Here are several points for consideration:

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

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

Clock Enable

Figure 38 shows an example of gated clocking. Gating clocks is not encouraged in digital designs because it may cause timing issues, such as unexpected clock skews. The structure of the PFU makes the gating clock even more undesirable since it uses up all the clock resources in one PFU and sometimes wastes the flip-flop and latch resources in the PFU. By using the clock enable in the PFU, you can achieve the same functionality without worrying about timing issues, since only one signal is controlling the clock. Since only one clock is used in the PFU, all related logic can be implemented in one block to achieve better performance. Figure 39 shows the design using the clock enable signal.

Figure 38: Asynchronous: Gated Clocking

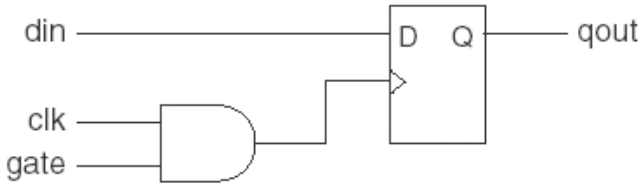
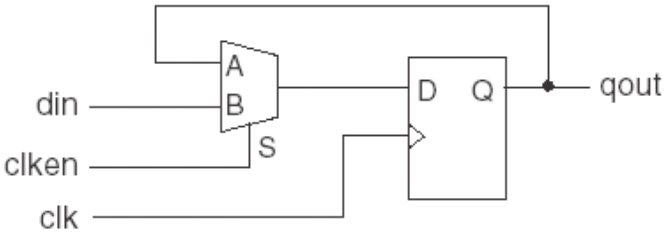


Figure 39: Synchronous: Clock Enabling



Samples of the VHDL and Verilog HDL code for clock enable are shown in Figure 40.

Figure 40: Clock Enable Coding

VHDL

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

Verilog HDL

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

The following are guidelines for coding the clock enable in Lattice Semiconductor FPGAs:

- Clock enable is only supported by flip-flops, not latches.
- Flip-flop pairs inside a slice block share the same clock enable.

- All flip-flops have a positive clock enable input.
- The clock-enable input has higher priority than the synchronous set/reset by default. However, you can program the synchronous LSR to have a higher priority than the clock enable by instantiating the library element in the source code. For example, the library element FD1P3IX is a flip-flop that allows the synchronous clear to override the clock enable. You can also specify the priority of generic coding by setting the priority of the control signals differently.

The examples in Figure 41 and Figure 42 demonstrate coding methodologies to help the synthesis tools set the priorities of the clock enable and the synchronous LSR.

Figure 41: Clock Enable over Synchronous LSR

VHDL

```
COUNT8: process (CLK, GRST)
begin
  if (GRST = '1') then
    cnt <= (others => '0');
  elsif (CLK'event and CLK = '1') then
    if (CKEN = '1') then
      cnt <= cnt + 1;
    elsif (LRST = '1') then
      cnt <= (others => '0');
    endif;
  endif;
end process COUNT8;
```

Verilog HDL

```
always @(posedge CLK or posedge GRST)
begin
  if (GRST)
    cnt = 4'b0;
  else if (CKEN)
    cnt = cnt + 1'b1;
  else if (LRST)
    cnt = 4'b0;
end
```

Figure 42: Synchronous LSR over Clock Enable

VHDL

```
COUNT8: process (CLK, GRST)
begin
  if (GRST = '1') then
    cnt <= (others => '0');
  elsif (CLK'event and CLK = '1') then
    if (LRST = '1') then
      cnt <= (others => '0');
    elsif (CKEN = '1') then
      cnt <= cnt + 1;
    endif;
  endif;
end process COUNT8;
```

Verilog HDL

```
always @(posedge CLK or posedge GRST)
begin
  if (GRST)
    cnt = 4'b0;
  else if (LRST)
    cnt = 4'b0;
  else if (CKEN)
    cnt = cnt + 1'b1;
end
```

Local Asynchronous and Synchronous Sets and Resets

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

When only one reset exists, always infer GSR. When more than one reset exists, pick the one that makes most sense to use as GSR. Disable the GSR for other resets.

Alternatively, you can manually instantiate GSRs. For multiple NGO flows, you must instantiate a GSR for every synthesis group. The ispLEVER mapping process removes redundant GSRs.

Figure 43 show the correct syntax for instantiating GSR in the VHDL and Verilog HDL codes.

Figure 43: Clock Enable Coding

VHDL

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

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

architecture behave of gsr_test is
signal cnt : std_logic_vector(1 downto 0);
begin
  u1: GSR port map (gsr => rst);

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

  cntout <= cnt;
end behave;
```

Verilog HDL

```
module gsr_test (clk, rst, cntout);
input clk, rst;
output[1:0] cntout;

reg[1:0] cnt;

GSR u1 (.GSR(rst));

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

assign cntout = cnt;
endmodule
```

Multiplexers

The flexible configurations of LUTs within slice blocks can realize any 4-, 5-, 6-, 7-, or 8-input logic function like 2-to-1, 3-to-1, 4-to-1, or 5-to-1 multiplexers.

You can efficiently create larger multiplexers by programming multiple 4-input LUTs. Synthesis tools can automatically infer Lattice Semiconductor FPGA optimized multiplexer library elements according to the behavioral description in the HDL source code. This provides the flexibility to the mapper and place-

and-route tools to configure the LUT mode and connections in an optimal fashion.

Figure 44: 16:1 Multiplexer

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

Finite State Machine Guidelines

A finite state machine is a hardware component that advances from the current state to the next state at the clock edge. This section discusses methods and strategies for state machine encoding.

State Encoding Methods for State Machines

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

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

tools. Most synthesis tools allow you to define encoding styles either through attributes in the source code or through the tool's user interface. Each synthesis tool has its own synthesis attributes and syntax for choosing the encoding styles. Refer to your synthesis tool's documentation for details about attributes syntax and values.

The following syntax defines an enumeration type in VHDL:

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

Here is a VHDL example of enumeration states:

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

The following is an example of Synplify VHDL synthesis attributes:

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

The following is an example of Precision RTL Synthesis VHDL synthesis attributes:

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

In Verilog HDL, you must provide explicit state values for states by using a bit pattern, such as 3'b001, or by defining a parameter and using it as the case item. The latter method is preferable. The following is an example using parameter for state values:

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

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

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

In Precision RTL Synthesis, it is also recommended that you define a Verilog HDL parameter and use it as the case item. The setup_design_encoding command in Precision RTL Synthesis is used to specify the encoding style.

In general, synthesis tools select the optimal encoding style that takes into account the target device architecture and size of the decode logic. You can

always apply synthesis attributes to override the default encoding style if necessary.

Coding Styles for State Machines

As mentioned earlier, the preferred scheme for FPGA architectures is one-hot encoding. This section discusses some common issues that you may encounter when constructing state machines, such as initialization and state coverage and special case statements in Verilog HDL.

General State Machine Description

Generally, there are two approaches to describing a state machine. One approach is to use one process or block to handle both state transitions and state outputs. The other is to separate the state transition and the state outputs into two different processes or blocks. The latter approach is more straightforward, because it separates the synchronous state registers from the decoding logic that is used in the computation of the next state and the outputs. This not only makes the code easier to read and modify but makes the documentation more efficient. If the outputs of the state machine are combinatorial signals, the second approach is almost always necessary because it prevents the accidental registering of the state machine outputs.

The examples in Figure 45 and Figure 46 describe a simple state machine in VHDL and Verilog HDL. In the VHDL example, a sequential process is

separated from the combinatorial process. In the Verilog HDL code, two always blocks are used to describe the state machine in a similar way.

Figure 45: VHDL Example for State Machine

```
architecture lattice_fpga of dram_refresh is
type state_typ is (s0, s1, s2, s3, s4);
signal present_state, next_state : state_typ;

begin
  -- process to update the present state
  registers: process (clk, reset)
  begin
    if (reset = '1') then
      present_state <= s0;
    elsif clk'event and clk='1' then
      present_state <= next_state;
    end if;
  end process registers;

  -- process to calculate the next state & outputs
  transitions: process (present_state, refresh, cs)
  begin
    ras <= 'X'; cas <= 'X'; ready <= 'X';
    case present_state is
      when s0 =>
        if (refresh = '1') then
          next_state <= s3;
          ras <= '1'; cas <= '0'; ready <= '0';
        elsif (cs = '1') then
          next_state <= s1;
          ras <= '0'; cas <= '1'; ready <= '0';
        else
          next_state <= s0;
          ras <= '0'; cas <= '1'; ready <= '1';
        end if;
      when s1 =>
        next_state <= s2;
        ras <= '0'; cas <= '0'; ready <= '0';
      when s2 =>
        if (cs = '0') then
          next_state <= s0;
          ras <= '1'; cas <= '1'; ready <= '1';
        else
          next_state <= s2;
          ras <= '0'; cas <= '0'; ready <= '0';
        end if;
      when s3 =>
        next_state <= s4;
        ras <= '1'; cas <= '0'; ready <= '0';
      when s4 =>
        next_state <= s0;
        ras <= '0'; cas <= '0'; ready <= '0';
      when others =>
        next_state <= s0;
        ras <= '0'; cas <= '0'; ready <= '0';
    end case;
  end process transitions;
```

Figure 46: Verilog HDL Example for State Machine

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

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

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

// always block to calculate the next state & outputs
always @ (present_state or refresh or cs)
begin
  next_state = s0;
  ras = 1'bX; cas = 1'bX; ready = 1'bX;
  case (present_state)
    s0 : if (refresh) begin
        next_state = s3;
        ras = 1'b1; cas = 1'b0; ready = 1'b0;
      end
      else if (cs) begin
        next_state = s1;
        ras = 1'b0; cas = 1'b1; ready = 1'b0;
      end
      else begin
        next_state = s0;
        ras = 1'b0; cas = 1'b1; ready = 1'b1;
      end
    s1 : begin
        next_state = s2;
        ras = 1'b0; cas = 1'b0; ready = 1'b0;
      end
    s2 : if (~cs) begin
        next_state = s0;
        ras = 1'b1; cas = 1'b1; ready = 1'b1;
      end
      else begin
        next_state = s2;
        ras = 1'b0; cas = 1'b0; ready = 1'b0;
      end
    s3 : begin
        next_state = s4;
        ras = 1'b1; cas = 1'b0; ready = 1'b0;
      end
    s4 : begin
        next_state = s0;
        ras = 1'b0; cas = 1'b0; ready = 1'b0;
      end
    default : begin
        next_state = s0;
        ras = 1'b0; cas = 1'b0; ready = 1'b0;
      end
  endcase
end
```

Initialization and Default State

A state machine must be initialized to a valid state after power-up. You can initialize it at the device level during power-up or by including a reset operation to bring it to a known state. For all Lattice Semiconductor FPGA devices, the global set/reset (GSR) is pulsed at power-up, regardless of the function defined in the design source code. In the examples in Figure 45 and Figure 46, an asynchronous reset can be used to bring the state machine to a valid initialization state.

In the same manner, a state machine should have a default state to ensure that the state machine does not go into an invalid state if not all the possible combinations are clearly defined in the design source code. VHDL and Verilog HDL have different syntax for default state declaration. In VHDL, if a case statement is used to construct a state machine, “when others” should be used as the last statement before the end of the statement. If an if-then-else statement is used, “else” should be the last assignment for the state machine. In Verilog HDL, use “default” as the last assignment for a case statement, and use “else” for the if-then-else statement. See the examples in Figure 47.

Figure 47: Initialization and Default State Example

When Others in VHDL

```
architecture lattice_fpga of FSM1 is
type state_typ is
  (deflt, idle, read, write);
signal next_state : state_typ;

begin
  process (clk, rst)
  begin
    if (rst = '1') then
      next_state <= idle; dout <= '0';
    elsif (clk'event and clk = '1') then
      case next_state is
        when idle =>
          next_state <= read;
          dout <= din(0);
        when read =>
          next_state <= write;
          dout <= din(1);
        when write =>
          next_state <= idle;
          dout <= din(2);
        when others =>
          next_state <= deflt;
          dout <= '0';
      end case;
    end if;
  end process;
```

Default Clause in Verilog HDL

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

reg[1:0] next_state;
reg dout;

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

Full Case and Parallel Case Specification in Verilog HDL

Verilog HDL has additional attributes for defining the default states without writing it specifically in the code. You can use “full_case” to achieve the same performance as “default.” Figure 48 shows the attribute usage for Precision RTL Synthesis and Synplify.

Figure 48: full_case versus default in Verilog HDL

Using full_case

```
case (current_state)
/* synthesis full_case */
// pragma full_case
  2'b00 : next_state = 2'b01;
  2'b01 : next_state = 2'b11;
  2'b11 : next_state = 2'b00;
endcase
```

Using default

```
case (current_state)
  2'b00 : next_state = 2'b01;
  2'b01 : next_state = 2'b11;
  2'b11 : next_state = 2'b00;
  default : next_state = 2'bxx;
endcase
```

The “parallel_case” attribute makes sure that all the statements in a case statement are mutually exclusive. It is used to inform the synthesis tools that only one case can be true at a time. Figure 49 shows the attribute usage when used in conjunction with the “full_case” attribute.

Figure 49: parallel_case in Verilog HDL

```
case (current_state)
/* synthesis full_case parallel_case */
// pragma full_case parallel_case
  2'b00 : next_state = 2'b01;
  2'b01 : next_state = 2'b11;
  2'b11 : next_state = 2'b00;
endcase
```

HDL Coding for Distributed and Block Memory

Although an RTL description of RAM is portable and the coding is straightforward, it is not recommended, because the structure of RAM blocks in every architecture is unique. Synthesis tools are not optimized to handle RAM implementation, and so they generate inefficient netlists for device fitting. For Lattice Semiconductor FPGA devices, generate RAM blocks through IPexpress in ispLEVER.

When implementing large memories, use the embedded block RAM (EBR) components found in every Lattice Semiconductor FPGA device. When implementing small memories, use the resources in the PFU. Using ispLEVER IPexpress, you can target a memory module to the PFU-based distributed memory or to the sysMEM EBR block.

Lattice Semiconductor FPGAs support many different memory types, including synchronous dual-port RAM, synchronous single-port RAM, synchronous FIFO, and synchronous ROM. For more information on

supported memory types per FPGA architecture, consult the Lattice Semiconductor FPGA data sheets.

Synthesis Control of High-Fan-Out Nets

Lattice Semiconductor FPGA device architectures are designed to handle high signal fan-outs. When you use clock resources, there are no hindrances on fan-outs. However, synthesis tools tend to replicate logic to reduce fan-out during logic synthesis. For example, if the code implies clock enable and is synthesized with speed constraints, the synthesis tool might replicate the clock-enable logic. This kind of logic replication occupies more resources in the devices and makes performance checking more difficult. Control the logic replication in the synthesis process by using attributes for a high-fan-out limit.

Bidirectional Buffers

You can instantiate bidirectional buffers in the same manner as regular I/O buffers or infer them from the HDL source, as shown in Figure 50 and Figure 51.

Figure 50: Verilog HDL RTL for Bidirectional Buffer

```
module bireg (datain, clk, en_o, Qo1, Qio);
   input  [7:0] datain;
   input  clk, en_o;
   output [7:0] Qo1;
   inout  [7:0] Qio;
   reg  [7:0] Q_reg;
   reg  [7:0] Qio_int;
   wire [7:0] Qo1;
   wire [7:0] Qio;
   always @(posedge clk)
   begin
      Q_reg = datain;
   end
   always @(en_o or Q_reg)
   begin
      if (en_o)
         Qio_int <= Q_reg;
      else
         Qio_int <= 8'hz;
   end
   assign Qio = Qio_int;
   assign Qo1 = Qio;
endmodule
```

Figure 51: VHDL RTL for Bidirectional Buffer

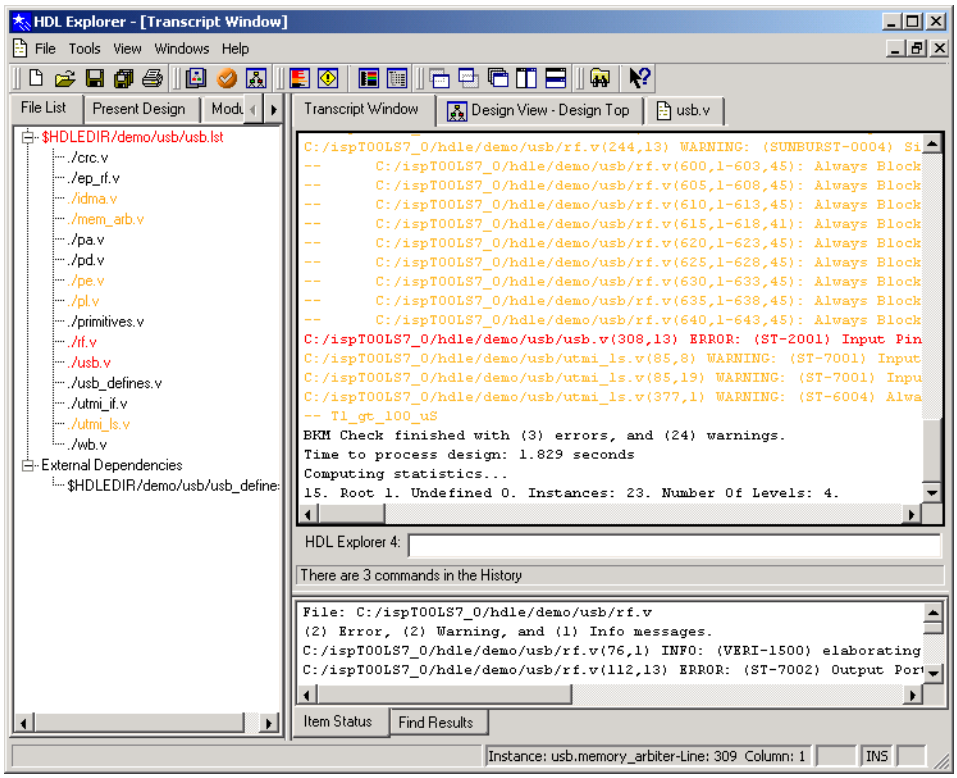
```
library ieee;
use ieee.std_logic_1164.all;

entity bireg is port (
   datain : in std_logic_vector (7 downto 0);
   clk,en_o : in std_logic;
   Qo1 : out std_logic_vector (7 downto 0);
   Qio : inout std_logic_vector (7 downto 0));
end bireg;
architecture beh of bireg is
   signal Q_reg   : std_logic_vector (7 downto 0);
   signal Qio_int : std_logic_vector (7 downto 0);
begin
   process(clk,datain) begin
      if clk'event and clk = '1' then
         Q_reg <= datain;
      end if;
   end process;
   process(Q_reg,en_o) begin
      if en_o = '1' then
         Qio_int <= Q_reg ;
      else
         Qio_int <= (others=>'Z');
      end if;
   end process;
   Qio <= Qio_int;
   Qo1 <= Qio;
end;
```

Coding to Avoid Simulation/Synthesis Mismatches

Certain coding styles can lead to pre-synthesis simulation that differs from post-synthesis gate level simulations. This problem is caused by HDL models that contain information that cannot be passed to the synthesis tool because of style or pragmas that are ignored by a simulator. Many error-prone coding styles will be detected by the HDL Explorer tool. This tool, which is included with the ispLEVER software, should be used to detect RTL design flaws as part of your verification strategy.

Figure 52: HDL Explorer Transcript Window



The examples in this section illustrate common mistakes to avoid. Where possible, examples of best-known-method (BKM) messages issued by HDL Explorer are also provided.

Sensitivity Lists

In Verilog and VHDL, combinational logic is typically modeled using a continuous assignment. Combinational logic can also be modeled when using a Verilog `always` statement or a VHDL `process` statement in which the event sensitivity list does not contain any edge events (`posedge/negedge` or `` `event ``). The event sensitivity list does not affect the synthesized netlist. Therefore, it might be necessary to include all the signals read in the event sensitivity list to avoid mismatches between simulation and synthesized logic.

In following Verilog example, `module code1b` uses a style that leads to a mismatch due to an incomplete sensitivity list. During pre-synthesis simulation, the `always` statement is only activated when an event occurs on variable a. However, the post-synthesis result will infer a 2-input `and` gate.

```
module code1b (o, a, b);
  output o;
  input a, b;
  reg o;

  always @(a)
    o = a & b;

endmodule
// Supported, but simulation mismatch may occur.
// To assure the simulation will match the synthesized logic,
// add variable b to the event list so the event list
// reads: always @(a or b).
```

The synthesis-related BKM check reported by HDL Explorer is:

```
WARNING: (ST-6003) Always Block 'code1b.@( a)' has the
following blocking assignment with driving signals that are not
in the sensitivity list. Possible Simulation/Synthesis
mismatch.
// o = (a & b) ;
```

Not all variables that appear in the right-hand side of an assignment are required to appear in the event sensitivity list. For example, Verilog variables that are assigned values inside the `always` statement body before being used by other expressions do not have to appear in the sensitivity list.

Blocking/Nonblocking Assignments in Verilog

A subtle Verilog coding style that can lead to unexpected results is the blocking/nonblocking style of variable assignment. The following guidelines are recommended:

- Use blocking assignments in `always` blocks that are written to generate combinational logic.
- Use nonblocking assignments in `always` blocks that are written to generate sequential logic.

- Use nonblocking assignments with register models to avoid race conditions.

Execution of blocking assignments can be viewed as a one-step process:

- Evaluate the RHS (right-hand side equation) and update the LHS (left-hand side expression) of the blocking assignment without interruption from any other Verilog statement.

  A blocking assignment "blocks" trailing assignments in the same `always` block, meaning that it prevents them from occurring until after the current assignment has been completed.

A problem with blocking assignments occurs when the RHS variable of one assignment in one procedural block is also the LHS variable of another assignment in another procedural block and both equations are scheduled to execute in the same simulation time step, such as on the same clock edge. If blocking assignments are not properly ordered, a race condition can occur. When blocking assignments are scheduled to execute in the same time step, the order execution is unknown.

According to the IEEE Verilog Standard for the language itself (not the synthesis standard), the two `always` blocks can be scheduled in any order.

In the following example, if the first always block executes first after a reset, both `y1` and `y2` will take on the value of 1. If the second `always` block executes first after a reset, both `y1` and `y2` will take on the value 0. This clearly represents a race condition.

```
module fbosc1 (y1, y2, clk, rst);
  output y1, y2;
  input clk, rst;

  reg y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 = 0; // reset
    else y1 = y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 = 1; // preset
    else y2 = y1;

endmodule
```

The HDL Explorer will also report potential problems given the combination of an edge-based sensitivity list with blocking assignments. For example:

```
// WARNING: (SUNBURST-0001) Always Block 'lfsrb1.@(posedge clk
or negedge pre_n)' has a edge based sensitivity list, but has
the following blocking assignments. Possible Simulation/
Synthesis mismatch.
// q3 = 1'b1 ;
// q2 = 1'b1 ;
// q1 = 1'b1 ;
// q3 = q2 ;
// q2 = n1 ;
// q1 = q3 ;
```

In Verilog, a variable assigned in an `always` statement cannot be assigned using both a blocking assignment (=) and a non-blocking assignment (<=) in the same `always` block.

```
always @ (IN1 or IN2 or SEL) begin
  OUT = IN1;
  if (SEL)
    OUT <= 2;
end
```

Synthesis Pragmas: full_case/parallel_case

The synthesis tool directive `full_case` gives more information about the design to the synthesis tool than is provided to the simulation tool. This particular directive is used to inform the synthesis tool that the case statement is fully defined and that the output assignments for all unused cases are “don’t cares.” The functionality between pre-synthesis and post-synthesis designs might remain the same when using this directive, or it might not.

Additionally, although this directive is telling the synthesis tool to use the unused states as “don’t cares,” it will sometimes make designs larger and slower than designs that omit it.

In the following module sample code4, a case statement is coded using the `full_case` synthesis directive. Without the `full_case` directive, the resultant design is a decoder built from 3-input `and` gates and inverters. The pre-synthesis and post-synthesis simulations will match. However, when the full_case directive is added, the `en` input is optimized away during synthesis and left as a dangling input. This is another case where pre-synthesis simulator results of modules will not match the post-synthesis simulation results.

```
module code4 (en, a, y);
  input        en;
  input  [1:0] a;
  output [3:0] y;
  reg    [3:0] y;
  always @(a or en) begin
    y = 4'h0;
    case ({en,a}) //pragma full_case
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule
```

The synthesis tool directive `parallel_case` also gives more information about the design to the synthesis tool than is provided to the simulation tool. This particular directive is used to inform the synthesis tool that all cases should be tested in parallel, even if there are overlapping cases, which would normally cause a priority encoder to be inferred.

Signal Fanout

Signal fanout refers to the number of inputs that can be connected to an output before the current required by the inputs exceeds the current that can be delivered by the output while maintaining correct logic levels or performance requirements. FPGA logic synthesis will automatically maintain reasonable fanout levels by replicating drivers or buffering a signal. Because of this behavior, the resulting FPGA route might be slower due to the additional intrinsic delays.

Signal fanout control is available with logic synthesis to maintain reasonable fanouts by controlling to what degree drivers are replicated. You should anticipate the availability of FPGA routing resources that are reserved for high fanout, low-skew networks like clocks, clock-enables, resets, and others. HDL Explorer can be configured to detect high fanout conditions as in the following example:

```
WARNING: (ST-5002) Net 'sc_dist_dpram.dec_wre1' violates Max
Fanout Rule with a load of '8' pins.
sc_dist_dpram.v(71,72-71,86): Input:mem_0_0.WRE
sc_dist_dpram.v(81,72-81,86): Input:mem_0_1.WRE
sc_dist_dpram.v(151,72-151,86): Input:mem_4_0.WRE
sc_dist_dpram.v(161,72-161,86): Input:mem_4_1.WRE
sc_dist_dpram.v(231,72-231,86): Input:mem_8_0.WRE
sc_dist_dpram.v(241,72-241,86): Input:mem_8_1.WRE
sc_dist_dpram.v(311,72-311,86): Input:mem_12_0.WRE
sc_dist_dpram.v(321,72-321,86): Input:mem_12_1.WRE
```

References

RTL Coding Styles that Yield Simulation and Synthesis Mismatches
Don Mills, LCDM Engineering
Clifford E. Cummings, Sunburst Design, Inc.

Nonblocking Asisgnments in Verilog Synthesis, Coding Styles That Kill!
Clifford E. Cummings, Sunburst Design, Inc.

www.sunburst-design.com/papers

7

Attributes and Preferences for FPGA Designs

This chapter describes the usage of the most common ispLEVER attributes used with register-transfer-level (RTL) designs.

In addition to syntax examples for VHDL and Verilog HDL (Synplify and Precision RTL Synthesis) code, this chapter gives examples of non-RTL or (non-algorithmic) code, such as compiler directives, attributes, and library components, that enable you to specify Lattice Semiconductor FPGA-specific constraints and design elements in your HDL source. There are dozens of ispLEVER HDL attributes that you can include in RTL source code, depending on the application. This chapter discusses a subset of the two most popular classes of constraints used by designers: sysIO buffer and floorplanning constraints.

This style of design has both advantages and disadvantages that you should consider before adding device-specific constraints to your HDL code. As an advantage, it helps unify the logical and physical design documentation and can be an easy way to infer many physical preferences from the concise logical description. For example, HGROUP/UGROUP attributes within RTL infer groups of many physical slices and embedded blocks.

As disadvantages, it makes the source code device-specific and less portable, and it may require a coding style specific to a synthesis vendor.

You may notice that some attributes have redundant functionality. For example, Precision RTL Synthesis' attributes for I/O locking include "loc" and "pin_number" as a result of synthesis tool support of both vendor-specific and cross-vendor attributes. You should consider the best choice for the sake of maintaining your source code over time. The samples shown in this chapter typically use the most concise form possible or the one that closely matches the naming conventions of ispLEVER attributes.

For more information on vendor-specific synthesis attributes and directives, refer to the *Precision RTL Synthesis Style Guide* or the *Synplicity FPGA Synthesis Reference Manual*.

About Attributes

An attribute is a value, constant, string, or so forth that can be associated with certain names in an HDL or EDIF description. In the Lattice Semiconductor FPGA design flow, there are two classes of attributes: those related to ispLEVER and those related to the synthesis vendor.

ispLEVER Attributes

ispLEVER attributes are typically used on library elements or signals in the EDIF netlist as EDIF properties and are interpreted by the mapping, placement, and routing tools. Most ispLEVER attributes are used in conjunction with the Lattice Semiconductor library elements documented in the *FPGA Library online Help* and are usually generated automatically by IPexpress in ispLEVER.

A subset of ispLEVER attributes is also helpful to write within RTL HDL. These attributes are most often used to direct the mapping, placement, and routing tools, but in some cases they can influence logic synthesis or targeting algorithms.

The main advantage of adding ispLEVER attributes to the RTL source code is to unify the design documentation and take advantage of the ability to infer many gate-level properties from the abstract RTL description. Examples of ispLEVER attributes include: “LOC,” “FREQUENCY,” and “UGROUP.”

The ispLEVER attributes are supported by both Precision RTL Synthesis and Synplify and are typically documented as “user-defined” attributes.

For more information on ispLEVER attributes, refer to the “HDL Attributes” topic in the ispLEVER online Help.

Vendor Attributes

Vendor attributes are typically used to control the optimization and targeting algorithms of logic synthesis and in some cases to infer ispLEVER attributes. For example, Precision RTL Synthesis’ “pin_number” attribute infers the ispLEVER LOC attribute for pin assignments. Common vendor attributes include “don't optimize/touch,” “fan-out limit,” and “use IO registers.”

For more information on vendor attributes, refer to the *Precision RTL Synthesis Style Guide* or the *Synplicity FPGA Synthesis Reference Manual*.

About Compiler Directives

A compiler directive controls the way a design is analyzed, optimized, and mapped by logic synthesis. In the Lattice Semiconductor FPGA design flow, compiler directives are an important tool to help you achieve device area and speed goals. Common compiler directives include “FSM encoding,” “translate on/off,” and “black box.”

Using Attributes and Compiler Directives in HDL

In Verilog HDL, attributes and compiler directives are attached to the appropriate objects by special comments. The syntax differs from vendor to vendor. In VHDL, they appear as VHDL attributes and are typically pre-defined within a VHDL package, for example, exemplar_1164 in Precision RTL Synthesis or synplify in Synplify. Both Precision RTL Synthesis and Synplify support user-defined attributes.

For more information on compiler directives, refer to the *Precision RTL Synthesis Style Guide* or the *Synplicity FPGA Synthesis Reference Manual*.

sysIO Buffer Constraints

The sysIO buffer feature of Lattice Semiconductor FPGAs and CPLDs is a programmable I/O cell organized into banks around the periphery of the device. SysIO buffers have several programmable options, including signal interface standard, drive strength, slew rate, and bus maintenance. It is a common practice to specify the programming of sysIO buffers as constraints within the RTL source. This section describes the most common attributes.

For large packages with many programming options, banks, and reference voltages, Lattice Semiconductor recommends using the I/O Assistant methodology to place and program I/Os. After you arrive at a legal placement, place the ispLEVER preferences back into the RTL source by using the guidelines in this section.

I/O Buffer Insertion

You can use two ways to insert I/O buffers or pads into the EDIF netlist produced by logic synthesis:

- Insert them by default during synthesis.
- Instantiate I/O buffers (automatic I/O insertion by synthesis must be disabled).

To minimize the amount of code required to design with I/O buffers, Lattice Semiconductor provides a Verilog HDL and a VHDL synthesis header library file for each major FPGA device family. Refer to the “Lattice Synthesis Header Libraries” topic in the ispLEVER online Help for details.

Because of Verilog HDL's case sensitivity, you must follow the conventions used by the synthesis header libraries when you describe module and port names. In general, names are in upper case.

The source code shown in Figure 53 in Verilog HDL and in Figure 54 in VHDL illustrates I/O buffer instantiation.

Figure 53: Instantiating I/O Buffers in Verilog HDL

```
module example(data, clock, out_put);
  input  [1:0] data;
  input        clock;
  output [1:0] out_put;

  wire   [1:0] data_in, data_out;
  wire clk;

  // LatticeEC I/O buffers
  IB u0(.I(data[1]),.O(data_in[1]));
  IB u1(.I(data[0]),.O(data_in[0]));
  IB u2(.I(clock),.O(clk));
  OB u3(.I(data_out[1]),.O(out_put[1]));
  OB u4(.I(data_out[0]),.O(out_put[0]));

  // logical description goes here...
endmodule
```

Figure 54: Instantiating I/O Buffers in VHDL

```
library IEEE, ec;
use ec.components.all;  -- Component package for LatticeEC
use IEEE.std_logic_1164.all;

entity example is port(
   data    : in  std_logic_vector(1 DOWNTO 0);
   clock   : in  std_logic;
   out_put : out std_logic_vector(1 DOWNTO 0) );
end example;

architecture io_buf of example is
   signal data_in, data_out:std_logic_vector(1 DOWNTO 0);
   signal clk : std_logic;

begin

   -- LatticeEC I/O buffers
   u0 : IB port map(I=>data(1),O=>data_in(1));
   u1 : IB port map(I=>data(0),O=>data_in(0));
   u2 : IB port map(I=>clock,O=>clk);
   u3 : OB port map(I=>data_out(1),O=>out_put(1));
   u4 : OB port map(I=>data_out(0),O=>out_put(0));

   -- logical description goes here...

end;
```

I/O Buffer Configuration

The programmable sysIO buffer provides a variety of configurations controlled by preferences or HDL-based constraints. For a complete description of sysIO buffer usage, see the related Lattice Semiconductor application notes for your target device family.

Constraints can be used along with buffer instantiation, as shown in “I/O Buffer Insertion” on page 87, or by automatic insertion by logic synthesis.

The source code shown in Figure 55 in Verilog HDL and in Figure 56 in VHDL illustrates the usage of some common attributes: IO_TYPE, DRIVE, PULLMODE, and SLEWRATE. These attributes are technology-dependent.

Figure 55: I/O Constraints in Verilog HDL

```
module example(data, clock, out_put);
   input  [1:0] data;
   input        clock;
   output [1:0] out_put
   /* synthesis
   IO_TYPE ="LVTTL33"
   DRIVE   ="16"
   PULLMODE="UP"
   SLEWRATE="FAST" */ ;
   //pragma attribute out_put IO_TYPE  LVTTL33
   //pragma attribute out_put DRIVE    16
   //pragma attribute out_put PULLMODE UP
   //pragma attribute out_put SLEWRATE FAST

   wire   [1:0] data_in, data_out;
   wire clk;

   // logical description goes here...

endmodule
```

Figure 56: I/O Constraints in VHDL

```
library IEEE, ec;
use ec.components.all;  -- Component package for LatticeEC
use IEEE.std_logic_1164.all;

entity example is port(
   data    : in  std_logic_vector(1 DOWNTO 0);
   out_put : out std_logic_vector(1 DOWNTO 0) );
end example;

architecture io_buf of example is
   signal data_in, data_out:std_logic_vector(1 DOWNTO 0);
   -- LatticeEC I/O buffer constraints
   attribute IO_TYPE : string;
   attribute DRIVE : string;
   attribute PULLMODE : string;
   attribute SLEWRATE : string;
   attribute IO_TYPE  OF out_put: SIGNAL IS "LVTTL33";
   attribute DRIVE    OF out_put: SIGNAL IS "16";
   attribute PULLMODE OF out_put: SIGNAL IS "UP";
   attribute SLEWRATE OF out_put: SIGNAL IS "FAST";

begin

   -- logical description goes here...

end;
```

Overriding Default I/O Buffer Type

Logic synthesis automatically inserts input/output (I/O) buffers or pads into the synthesized design. The default input and output pads are “IB” and “OB,” respectively (“BB” for bidirectional), which are generic buffers that optionally carry attributes to specify such things as I/O type, drive, and pull mode. The pad type mapped by Precision RTL Synthesis or Synplify logic synthesis can be overridden for a particular I/O by using synthesis attributes within HDL, constraint files, or GUI controls.

- In Precision RTL Synthesis, the “pad” attribute is a string attribute that must be attached to a top-level port to override the I/O cell used by Precision RTL Synthesis when it synthesizes the I/O cell.
- In Synplify, the “orca_padtype” attribute is a string attribute that must be attached to a top-level port to override the I/O cell used by Synplify when it synthesizes the I/O cell.

The source code shown in Figure 57 in Verilog HDL and in Figure 58 in VHDL illustrates how to override a pad type.

Figure 57: Override of Buffer Type in Verilog HDL

```
module example(data, clock, out_put);
   // Choose padtype IBPD to select an input buffer with pull-
down
   input data /* synthesis orca_padtype="IBPD" */;
   //pragma attribute data pad IBPD
   // logical description goes here...
endmodule
```

Figure 58: Overriding of Buffer Type in VHDL

```
-- declare the Precision RTL pad attribute
   attribute pad: string;
   -- declare the Synplify orca_padtype attribute
   attribute orca_padtype: string;

   -- Choose padtype IBPD to select an input buffer with pull-
down
   attribute pad of data: signal is "IBPD";
   attribute orca_padtype of data: signal is "IBPD";
```

Locking I/O Pins

There are two methods for locking I/O pins in the ispLEVER design flow for FPGAs:

- HDL-based attributes
- The ispLEVER preference file

Both Precision RTL Synthesis and Synplify support HDL-based attributes to lock I/O pins based on a port name from within source code. The synthesis mapper then converts the port to a specific I/O buffer and adds the pin number as an attribute to the cell instance in the EDIF netlist. When ispLEVER maps, places, and routes the design, it uses the attribute to lock the PIC/IOB site in the FPGA.

- In Precision RTL Synthesis, the “array_pin_number” (VHDL only), “pin_number,” or “loc” attribute is a string attribute that must be attached to a top-level port to assign pin numbers.
- In Synplify, the “loc” attribute is a string attribute that must be attached to a top-level port to assign pin numbers.

The source code in Figure 59 in Verilog HDL and in Figure 60 in VHDL illustrates how to lock I/O pins.

Figure 59: Pin Locking in Verilog HDL

```
// Lock I/O assignment for port: datain
   input  [7:0] datain /* synthesis
loc="43,36,83,52,91,45,84,78" */;
   //pragma attribute datain pin_number
"43,36,83,52,91,45,84,78"
```

Figure 60: Pin Locking in VHDL

```
-- Lock clock port to pad p10
   attribute loc : string;
   attribute loc of clock : signal is "p10";
   attribute pin_number : string;
   attribute pin_number of clk : signal is "90";
   type mentor_string_array is array (natural range <>,
      natural range <>) of character ;
   attribute array_pin_number : mentor_string_array ;
   attribute array_pin_number of datain: signal is
      ("43","36","83","52","91","45","84","78");
```

Optimization Constraints

Logic optimization by logic synthesis can dramatically influence the gate-level implementation of your design. This section describes common optimization controls used within the RTL source. For complete information on controls available for Precision RTL Synthesis or Synplify, refer to the respective user and style reference guides.

Black-Box Module Instances

By default, logic synthesis elaborates the design hierarchy until all leaf nodes are represented by a Lattice Semiconductor library macro or expression that infers one or more macros. However, in some cases you may want to treat some module instances as black boxes that cause the logic optimizer to stop elaboration at that point and pass the module as is into the EDIF 2 0 0 netlist. The most common application for black-box modules is when you use an incremental or block modular design technique.

- In Precision RTL Synthesis, the “dont_touch” attribute is a Boolean attribute that must be declared as a Verilog HDL comment near the module instance or as a VHDL attribute of a component declaration.
- In Synplify, the “syn_black_box” directive is a Boolean attribute that must be attached to a Verilog HDL module declaration or as a VHDL attribute of a component declaration.

The source code in Figure 61 in Verilog HDL and in Figure 62 in VHDL illustrates the usage of the black-box concept.

Figure 61: Black-Box Module Instances in Verilog HDL

```
// ------------------------------------------------------------
// Controls to switch between RTL, "black box", and mixed RTL and
// gate-level versions of the top-level design module.
// ------------------------------------------------------------
   `define BBox_mode; // Comment out for RTL_mode simulation.
//   `define RTL_mode;// Comment out to exclude module definitions.

   `ifdef BBox_mode // then bind to empty modules for synthesis,

      module multreg16(q, dataa, datab, datac, sel, clk, rst)
    /* synthesis syn_black_box */;
        output [15:0] q;
        input   [7:0] dataa, datab, datac;
        input         clk /* synthesis syn_isclock = 1 */;
        input         sel, rst;
        reg    [15:0] q;
      endmodule

      module rotate(q, data, clk, r_l, rst)
    /* synthesis syn_black_box */;
        output [15:0] q;
        input  [15:0] data;
        input         clk /* synthesis syn_isclock = 1 */;
        input         r_l, rst;
      endmodule

   `else

      `ifdef RTL_mode

        `include "multreg16/multreg16.v"
        `include "rotate/rotate.v"

      `else

// Do not provide module definitions - instead rely on
       //gate-level models created by ispLEVER.

      `endif

   `endif

// ------------------------------------------------------------
// Top-level design.
// ------------------------------------------------------------

   module verilog_hierarchical_design(q, a, b, c, sel, r_l, pllclk,
rst);
     output [15:0] q;
     input   [7:0] a, b, c;
     input         sel, r_l, pllclk, rst;
     wire   [15:0] reg_out;
     wire          clk, rst_l;
     //pragma attribute clk preserve_signal true
```

Figure 61: Black-Box Module Instances in Verilog HDL (Continued)

```
assign rst_l = !rst;
     // Global set/reset and power up reset signal drivers
     GSR GSR_INST
     (.GSR (rst_l));
     PUR PUR_INST
     (.PUR (rst_l));

     // LatticeEC sysCLOCK PLL
     /* Verilog module instantiation template generated by SCUBA
ispLever_v50_SP1_Build (12) */
     /* Wed May 25 11:53:58 2005 */
     /* parameterized module instance */
     LatticeEC_66MHz_PLL PLL_1 (.CLK(pllclk), .RESET(rst),
.CLKOP(clk), .LOCK( ));
   //pragma attribute PLL_1 dont_touch

     // multiplexer/multiply/register
     multreg16 multreg16_1
     (.q(reg_out),
      .dataa(a),
      .datab(b),
      .datac(c),
      .sel(sel),
      .clk(clk),
      .rst(rst));
     //pragma attribute multreg16_1 dont_touch

     // register or rotate
     rotate rotate_1
     (.q(q),
      .data(reg_out),
      .clk(clk),
      .r_l(r_l),
      .rst(rst));
     // pragma attribute rotate_1 dont_touch

   endmodule
```

Figure 62: Black-Box Component Instances in VHDL

```
------------------------------------------------------------
--      Top level--
------------------------------------------------------------
library ieee,ec;
use ec.components.all;
use ieee.std_logic_1164.all;

entity vhdl_hierarchical_design is
  port (
  q                   : out std_logic_vector (15 downto 0);
  a, b, c             : in  std_logic_vector (7 downto 0);
  sel, r_l, pllclk, rst: in  std_logic
  );
end vhdl_hierarchical_design;

architecture arch of vhdl_hierarchical_design is
  -- parameterized module component declaration
  component LatticeEC_66MHz_PLL
    port (CLK: in std_logic; RESET: in std_logic; CLKOP: out
std_logic;
        LOCK: out std_logic);
  end component;

component multreg16  -- component declaration for multreg16
  port (
    q     : out std_logic_vector (15 downto 0);
    dataa : in  std_logic_vector (7 downto 0);
    datab : in  std_logic_vector (7 downto 0);
    datac : in  std_logic_vector (7 downto 0);
    clk   : in  std_logic;
    sel   : in  std_logic;

    rst   : in  std_logic);
  end component;

  component rotate  -- component declaration for rotate
  port (
    q    : out std_logic_vector (15 downto 0);
    data : in  std_logic_vector (15 downto 0);
    clk  : in  std_logic;

    r_l  : in  std_logic;

    rst  : in  std_logic);
  end component;

  -- declare the internal signals here
  signal reg_out: std_logic_vector (15 downto 0);

  signal clk, rst_l: std_logic;
```

Figure 62: Black-Box Component Instances in VHDL (Continued)

```
-- Precision RTL compiler directives
  attribute preserve_signal : boolean;
  attribute preserve_signal of reg_out : signal is true;
  attribute dont_touch : boolean;
  attribute dont_touch of multreg16_1: label is true;
  attribute dont_touch of rotate_1   : label is true;

  -- Synplify compiler directives
  attribute syn_black_box : boolean;
  attribute syn_black_box of multreg16: component is true;
  attribute syn_black_box of rotate   : component is true;
  attribute syn_noprune : boolean;
  attribute syn_noprune of GSR_INST: label is true;
  attribute syn_noprune of PUR_INST: label is true;

begin

  rst_l <= not(rst);
  -- Global set/reset and power up reset signal drivers
  GSR_INST: GSR port map(GSR => rst_l);
  PUR_INST: PUR port map(PUR => rst_l);

  -- VHDL module instantiation generated by SCUBA
ispLever_v50_SP1_Build (12)
  -- Wed May 25 11:53:41 2005
  -- parameterized module component instance
  PLL_1 : LatticeEC_66MHz_PLL
    port map (CLK=>pllclk, RESET=>rst, CLKOP=>clk, LOCK=>open);

  multreg16_1:  multreg16 port map (
    q    => reg_out,
    dataa=> a,
    datab=> b,
    datac=> c,
    sel  => sel,
  clk  => clk,
  rst  => rst);

  rotate_1: rotate port map (
    q    => q,
    data => reg_out,
    clk  => clk,
    r_l  => r_l,
    rst  => rst);

end arch;
```

Preserving Signals

To ensure that an internal design signal is preserved by logic optimization, both Precision RTL Synthesis and Synplify provide a “preserve”-type constraint.

- In Precision RTL Synthesis, the “preserve_signal” attribute is a Boolean attribute that is declared as a Verilog HDL comment near a wire or reg declaration or as a VHDL attribute of a signal.

- In Synplify, the “syn_keep” directive is a Boolean attribute that is declared as a Verilog HDL comment within a wire or reg declaration or as a VHDL attribute of a signal declaration.

The source code in Figure 63 in Verilog HDL and in Figure 64 in VHDL illustrates the usage of the preserve signal concept.

Figure 63: VHDL for Preserve Signal

```
// Preserve signal load
  wire load /* synthesis syn_keep=1 */;
  //pragma attribute load preserve_signal true
```

Figure 64: Verilog HDL for Preserve Signal

```
Verilog HDL for Preserve Signal

   -- Preserve signal load (Precision RTL)
   attribute preserve_signal:boolean;
   attribute preserve_signal of load: signal is true;

   -- Preserve signal load (Synplify)
   attribute syn_keep:boolean;
   attribute syn_keep of load: signal is true;
```

Floorplanning Constraints

Device floorplanning constraints in the HDL source are a powerful means of directing placement of design logic from a logical abstract level. It is a common practice in a timing closure methodology to iterate between the Design Planner application and the place-and-route program, PAR, to arrive at a superior implementation, then use the guidelines in this section to place the physical floorplanning constraints into the RTL code as logical constraints.

The floorplanning strategy is usually part of a timing closure or block modular design style. Before you attempt floorplanning techniques, Lattice Semiconductor recommends that you review “Floorplanning the Design” on page 173 and the “Block Modular Design Step Guide” section of the ispLEVER FPGA Flow Help in the online Help to understand whether your design could benefit from these methods.

Logic synthesis passes constraints into the target EDIF 2 0 0 netlist, where they appear as EDIF properties. The design mapper program, MAP, then converts the logical references, such as signals and module instances, into physical references, such as slices, EBR blocks, and DSP blocks, and writes the FPGA preferences into the ispLEVER preference (.prf) file. Table 18

shows the conversion from constraints based on the logical HDL to constraints based on the physical preference file.

Table 18: Logical and Physical Floorplanning Constraints

HDL Constraint	Post-Map Preference	Purpose
UGROUP	PGROUP	Group logic
HGROUP	PGROUP	Group logic (grouped elements retain hierarchical path)
PBBOX	PGROUP BBOX	Bounding box of a PGROUP
PLOC	LOCATE PGROUP	Anchor point of a PGROUP
PREGION	REGION	User-defined REGION name
PRLOC	REGION “RnCm”	Anchor point of REGION
PRBBOX	REGION Y X	Bounding box of REGION
COMP	Used with PGROUP or LOCATE	User-defined name for slice-based logic
LOC	LOCATE COMP	Device site for a block

Floorplanning constraints are written as VHDL attributes or as Verilog HDL embedded comments. A common VHDL style can be used between Precision RTL Synthesis and Synplify; however, Verilog HDL requires that vendor-specific keywords precede each comment. The source code examples in this section show both styles.

This section describes the most common attributes used by designers.

Locating a Block to a Device Site

The simplest floorplanning technique from within HDL is to anchor a logic block to a particular device site by using the LOC HDL attribute. Blocks can be anchored independently of a group or region floorplan. The most common type of “block” to locate is PIOs, as shown in “sysIO Buffer Constraints” on page 87.

This section illustrates how to specify the anchor points of a slice- or embedded-block-type logic in the FPGA array.

If you intend to floorplan design elements that will be mapped to slice device sites, you must add the COMP=*comp_name* HDL attribute to each module instance in the HDL source, as in the following Verilog HDL sample:

```
REG2 REG2inst (<port_list>) /* synthesis COMP=regpair
LOC=R10C20D */;
```

In this sample, the design mapper (MAP) applies the “regpair” COMP name to all elements that can be covered by a single slice and assigns it to the R10C20D device site. If the logic overflows a single slice, MAP appends a “.*number*” to the name for the post-map netlist, and the placer automatically chooses the site locations for the remaining slices.

Note

COMP/LOC attributes for slice-based logic can be included as part of a logic grouping strategy described in the floorplanning preferences section by specifying a device site relative to the anchor point of a PGROUP. This approach is not supported for embedded blocks.

To floorplan design elements that will be mapped to embedded blocks, such as PLL/DLL, EBR, or DSP sites, the LOC=<*device_site*> HDL attribute is added to each module instance in the HDL source, as in the following Verilog HDL samples:

```
ebr1 ram_dq_16 (<port_list>) /* synthesis LOC=EBR_R6C6  */;
pll1 pll66MHz  (<port_list>) /* synthesis LOC=PLL3_R6C1 */;
```

The graphical floorplan views of the Design Planner or EPIC provide device site addresses for slices or embedded blocks.

Note

Design elements such as PIO, EBR, DSP, PLL/DLL, and MACO blocks do not require the COMP attribute because MAP retains the original name used in the native generic (.ngd) database.

Grouping Logic

The HGROUP (Hierarchical Group) or UGROUP (Universal Group) can be used as an attribute in VHDL and Verilog HDL source code to bound and locate sections of a design for grouping in the FPGA array. For details on this method and the application of HGROUPs and UGROUPs, refer to “Floorplanning the Design” on page 173 and “Design Performance Enhancement Strategies” on page 193.

The source code in Figure 65 in Verilog HDL and in Figure 66 in VHDL illustrates the usage of the UGROUP constraint. The design mimics the hierarchy shown in the example in “Floorplanning the Design” on page 173 and illustrates the scenario in which the critical path is between the REGISTER_FILE and the STATE_MACHINE modules. Both modules are grouped into a UGROUP named “CRITICAL_GROUP.”

Note

In some cases, such as when using Precision RTL Synthesis, you may need to use “preserve hierarchy” compiler directives on module instances that you want to group to prevent logic optimization from flattening the design.

Note

By default, groups are placed into the smallest square available on the device floorplan. See “Design Performance Enhancement Strategies” on page 193 for information about group bounding boxes and anchors and for details on how to specify an alternative area shape and position.

Note

Groups that are composed of both slice-based and embedded block logic, such as EBR- and DSP-type blocks, must be anchored. Groups composed solely of slice-based logic, such as LUTs and registers, can float. For more information, see “Floorplanning the Design” on page 173.

Figure 65: Grouping Constraints in Verilog HDL

```
module STATE_MACHINE (clk, reset, cs, refresh, ras, cas, ready)
/* synthesis UGROUP="CRITICAL_GROUP" */;
   input  clk;
   input  reset;
   input  cs;
   input  refresh;
   output ras;
   output cas;
   output ready;

   parameter /* exemplar enum gray */ s0 = 0, s1 = 1, s2 = 2,
s3 = 3, s4 = 4;

   reg [2:0] /* exemplar enum gray */ present_state, next_state
;
   reg                                ras, cas, ready;

  // logical description goes here…

endmodule //

module CONTROLLER (clk, reset, cs, regdata, ras, cas, ready,
load);
   input        clk;
   input        reset;
   input        cs;
   input [15:0] regdata;
   output       ras;
   output       cas;
   output       ready;
   input        load;

   wire [15:0]  count;
   wire         refresh;

   COUNTER       b1 (clk, reset, cs, regdata, refresh, load);
   STATE_MACHINE b2 (clk, reset, cs, refresh, ras, cas, ready);
   //pragma attribute b2 UGROUP CRITICAL_GROUP
   //pragma attribute b2 hierarchy preserve

endmodule //
```

Figure 65: Grouping Constraints in Verilog HDL (Continued)

```
module REGISTER_FILE (clk, reset, cs, data, regdata)
/* synthesis UGROUP="CRITICAL_GROUP" */;
   input         clk;
   input         reset;
   input         cs;
   input  [15:0] data;
   output [15:0] regdata;

   reg    [15:0] regdata;

  // logical description goes here…

end

endmodule //

module TOP (clk, reset, cs, ras, cas, ready, data, load);
   input        clk;
   input        reset;
   input        cs;
   output       ras;
   output       cas;
   output       ready;
   input [15:0] data;
   input        load;

   wire [15:0] regdata;

CONTROLLER    b1 (clk, reset, cs, regdata, ras, cas, ready,
load);
REGISTER_FILE b2 (clk, reset, cs, data, regdata);
//pragma attribute b2 UGROUP CRITICAL_GROUP
//pragma attribute b2 hierarchy preserve

endmodule //
```

Figure 66: Grouping Constraints in VHDL

```
-- Precision RTL package
-- library exemplar;
-- use exemplar.exemplar_1164.all;

-- Synplify package
library synplify;
use synplify.attributes.all;

library ieee;
use ieee.std_logic_1164.all;

entity STATE_MACHINE is
  port (
  clk          : in std_logic;
  reset        : in std_logic;
  cs           : in std_logic;
  refresh      : in std_logic;
  ras          : out std_logic;
  cas          : out std_logic;
  ready        : out std_logic);
end STATE_MACHINE;

architecture ARCH of STATE_MACHINE is
   -- Precision RTL attributes for encoding style
   -- attribute TYPE_ENCODING_STYLE : encoding_style;
   -- Declare the state machine enumeration type
   type state_typ is (s0, s1, s2, s3, s4);
   -- Set the type_encoding_style of the state type
   -- attribute TYPE_ENCODING_STYLE of state_typ : type is
GRAY;

   signal present_state, next_state : state_typ;

   -- Synplify attributes for encoding style
   -- Set the type_encoding_style of the state signal
   attribute syn_encoding of present_state, next_state : signal
is "gray";
   attribute syn_keep     of present_state, next_state : signal
is true;

begin
   -- logical description goes here…

end ARCH;

library ieee;
use ieee.std_logic_1164.all;

entity CONTROLLER is
  port (
  clk      : in  std_logic;
  reset    : in  std_logic;
  cs       : in  std_logic;
  regdata  : in  std_logic_vector(15 downto 0);
  ras      : out std_logic;
```

Figure 66: Grouping Constraints in VHDL (Continued)

```
  cas      : out std_logic;
  ready    : out std_logic;
  load     : in  std_logic);
end CONTROLLER;

architecture ARCH of CONTROLLER is
  component COUNTER
  port (
  clk                : in  std_logic;
  reset              : in  std_logic;
  cs                 : in  std_logic;
  regdata            : in  std_logic_vector(15 downto 0);
  refresh            : out std_logic;
  load               : in  std_logic);
  end component;

component STATE_MACHINE
  port (
  clk          : in std_logic;
  reset        : in std_logic;
  cs           : in std_logic;
  refresh      : in std_logic;
  ras          : out std_logic;
  cas          : out std_logic;
  ready        : out std_logic);
  end component;

signal count   : std_logic_vector(15 downto 0);
   signal refresh : std_logic;

   attribute hierarchy : string ;
   attribute hierarchy of b2: label is "preserve";

   -- LatticeEC Floorplan Constraints
   attribute ugroup    : string ;
   attribute ugroup    of b2: label is "CRITICAL_GROUP";

begin
   b1 : COUNTER       port map (clk, reset, cs, regdata,
refresh, load);
   b2 : STATE_MACHINE port map (clk, reset, cs, refresh, ras,
cas, ready);

end ARCH;

library ieee;
use ieee.std_logic_1164.all;

entity REGISTER_FILE is
  port (
  clk      : in  std_logic;
  reset    : in  std_logic;
  cs       : in  std_logic;
  data     : in  std_logic_vector(15 downto 0);
  regdata  : out std_logic_vector(15 downto 0));
end REGISTER_FILE;
```

Figure 66: Grouping Constraints in VHDL (Continued)

```
architecture ARCH of REGISTER_FILE is
   signal regdata_n: std_logic_vector(15 downto 0);

begin
   -- logical description goes here…

end ARCH;

library ieee;
use ieee.std_logic_1164.all;

entity TOP is
  port (
  clk      : in  std_logic;
  reset    : in  std_logic;
  cs       : in  std_logic;
  ras      : out std_logic;
  cas      : out std_logic;
  ready    : out std_logic;
  data     : in  std_logic_vector(15 downto 0);
  load     : in  std_logic
  );
end TOP;

architecture ARCH of TOP is
   component CONTROLLER
  port (
  clk      : in  std_logic;
  reset    : in  std_logic;
  cs       : in  std_logic;
  regdata  : in  std_logic_vector(15 downto 0);
  ras      : out std_logic;
  cas      : out std_logic;
  ready    : out std_logic;
  load     : in  std_logic);
   end component;

component REGISTER_FILE
  port (
  clk      : in  std_logic;
  reset    : in  std_logic;
  cs       : in  std_logic;
  data     : in  std_logic_vector(15 downto 0);
  regdata  : out std_logic_vector(15 downto 0));
   end component;

signal regdata : std_logic_vector(15 downto 0);

   attribute hierarchy : string ;
   attribute hierarchy of b2: label is "preserve";

   -- ispLEVER attributes for flooplanning
   attribute UGROUP    : string ;
   attribute UGROUP    of b2: label is "CRITICAL_GROUP";
```

Figure 66: Grouping Constraints in VHDL (Continued)

```
begin
   b1: CONTROLLER    port map (clk, reset, cs, regdata, ras,
cas, ready, load);
   b2: REGISTER_FILE port map (clk, reset, cs, data, regdata);

end ARCH;
```

Group Bounding Boxes and Anchors

You can further control the HGROUP (Hierarchical Group) or UGROUP (Universal Group) by adding an optional bounding box to specify the shape and size of the floorplan area (PBBOX) allocated to the logic and an optional anchor point (PLOC) that provides a specific row and column address that indicates the anchor point of the upper left corner of the bounding box. For details on this method, refer to “Floorplanning the Design” on page 173 and “Design Performance Enhancement Strategies” on page 193.

The source code in Figure 67 in Verilog HDL and in Figure 68 in VHDL illustrates the usage of the PBBOX and PLOC constraints in conjunction with the UGROUP constraint shown in “Grouping Logic” on page 99.

Figure 67: Group Bounding Boxes and Anchors in Verilog HDL

```
// Comment for Synplify-style Verilog HDL
module REGISTER_FILE (clk, reset, cs, data, regdata)
/* synthesis ugroup="CRITICAL_GROUP" PBBOX="5,5" PLOC="R7C7D"
*/;
// Comment for Precision-style Verilog HDL
REGISTER_FILE b2 (clk, reset, cs, data, regdata);
//pragma attribute b2 ugroup CRITICAL_GROUP PBBOX 5,5 PLOC
R7C7D
```

Figure 68: Group Bounding Boxes and Anchors in VHDL

```
 attribute UGROUP      : string ;
   attribute PBBOX     : string;
   attribute PLOC      : string;
   attribute UGROUP    of b2: label is "CRITICAL_GROUP";
   attribute PBBOX     of b2: label is "5,5";
   attribute PLOC      of b2: label is "R7C7D";begin

begin
   b1: CONTROLLER    port map (clk, reset, cs, regdata, ras,
cas, ready, load);
   b2: REGISTER_FILE port map (clk, reset, cs, data, regdata);
```

Regional Groups

You can optionally assign the HGROUP (Hierarchical Group) or UGROUP (Universal Group) to a user-defined, named area of the device floorplan by using region constraints. PREGION is a user label for the region, PRBBOX is the bounding box definition, and PRLOC is the row and column address that indicates the anchor point for the upper left corner of the bounding box. For details on this method, refer to “Floorplanning the Design” on page 173 and “Design Performance Enhancement Strategies” on page 193.

The source code in Figure 69 in Verilog HDL and in Figure 70 in VHDL illustrate the usage of the PREGION, PRBBOX, and PRLOC constraints in conjunction with a UGROUP constraint shown in “Grouping Logic” on page 99.

Note

Regional groups can contain any number and combination of floating (unanchored), bounded or unbounded (PBBOX) HGROUP/UGROUPs. Regions themselves must be anchored and bounded.

Figure 69: Regional Groups in Verilog HDL

```
// Comment for Synplify-style Verilog HDL
module REGISTER_FILE (clk, reset, cs, data, regdata)
/* synthesis ugroup="CRITICAL_GROUP" PREGION="CENTER_REGION"
PRLOC="R5C5D" PRBBOX="7,10" */;

// Comment for Precision-style Verilog HDL
REGISTER_FILE b2 (clk, reset, cs, data, regdata);
//pragma attribute b2 ugroup CRITICAL_GROUP PREGION
CENTER_REGION PRLOC R5C5D PRBBOX 7,10
```

Figure 70: Regional Groups in VHDL

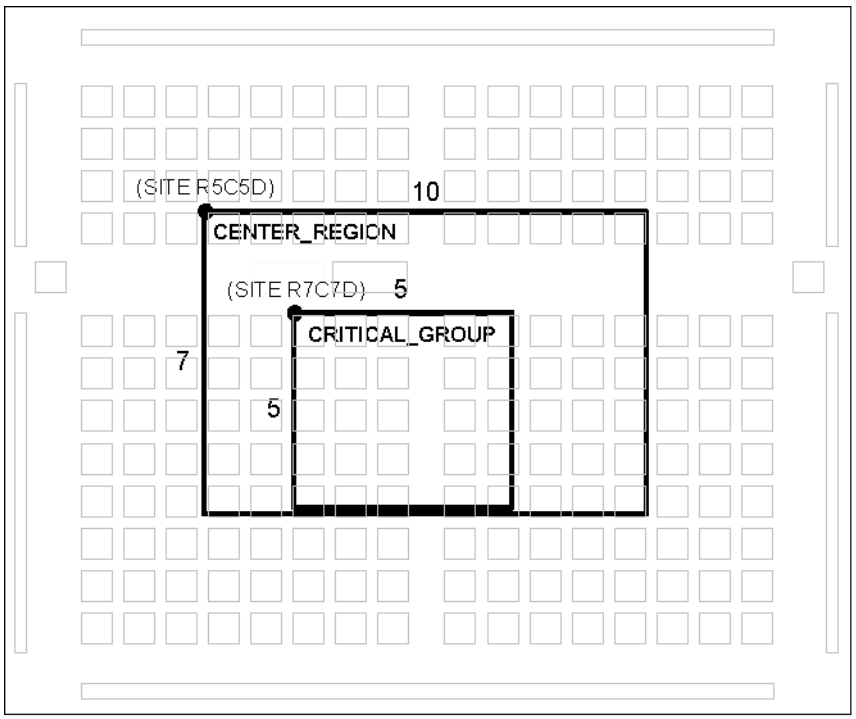
```
   attribute ugroup    : string ;
   attribute PBBOX     : string;
   attribute PLOC      : string;
   attribute PREGION   : string;
   attribute PRLOC     : string;
   attribute PRBBOX    : string;

   attribute ugroup    of b2: label is "CRITICAL_GROUP";
   attribute PBBOX     of b2: label is "5,5";
   attribute PLOC      of b2: label is "R7C7D";
   attribute PREGION   of b2: label is "CENTER_REGION";
   attribute PRLOC     of b2: label is "R5C5D";
   attribute PRBBOX    of b2: label is "7,10";

begin
   b1: CONTROLLER    port map (clk, reset, cs, regdata, ras,
cas, ready, load);
   b2: REGISTER_FILE port map (clk, reset, cs, data, regdata);
```

Figure 71 illustrates the floorplan that results in the context of an LFEC1E.

Figure 71: Anchored PGROUP Within a Region



LFEC1E

Register-Oriented Groups (Synplify Only)

In Synplify, you can specify that the registers driving a particular signal be grouped. This style can be beneficial to establish groups along pipeline stages of a data-path-style design.

You can use the floorplanning constraints described in this section on registered signals of Verilog HDL “reg” or VHDL signal declarations. On the basis of the signal attributes, Synplify adds PGROUP properties to all registers inferred by the RTL.

The source code in Figure 72 in Verilog HDL and in Figure 73 in VHDL illustrates the use of the register-oriented UGROUP constraint.

Note

In some cases, you may need to use “preserve signal” compiler directives on signal or register elements that you want to group to prevent logic optimization from eliminating them.

Figure 72: Register-Oriented Groups in Verilog HDL

```
module COUNTER (clk, reset, cs, regdata, refresh, load);
input clk;
input reset;
input cs;
input [15:0] regdata;
output refresh;
input load;

reg [15:0] count /* synthesis syn_keep=1 ugroup="COUNTER_GROUP"
PBBOX="3,3" PREGION="COUNTER_REGION" PRLOC="R5C10D"
PRBBOX="5,5" */;

  always @ (posedge clk or posedge reset)
    if (reset)
      count <= 1'b0;
    else if (cs) begin
      if (load)
        count <= regdata;
      else begin
        count[15:1] <= count[14:0];
        count[0] <= ~count[15];
      end
    end

  assign refresh = count[0];

endmodule //
```

Figure 73: Register-Oriented Groups in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
entity COUNTER is
port (
  clk : in std_logic;
  reset : in std_logic;
  cs : in std_logic;
  regdata : in std_logic_vector(15 downto 0);
  refresh : out std_logic;
  load : in std_logic
  );
end COUNTER;

architecture ARCH of COUNTER is
  signal count: std_logic_vector(15 downto 0);
  -- ispLEVER attributes for flooplanning
  attribute UGROUP : string;
  attribute PBBOX : string;
  attribute PREGION : string;
  attribute PRLOC : string;
  attribute PRBBOX : string;
  attribute UGROUP of count : signal is "counter_group";
  attribute PBBOX of count : signal is "3,3";
  attribute PREGION of count : signal is "counter_region";
  attribute PRLOC of count : signal is "R5C10D";
  attribute PRBBOX of count : signal is "5,5";

  begin
    process (clk, reset, regdata)
    begin
      if (reset='1') then
        count <= (others=>'0');
      elsif clk'event and clk='1' then
        if (load='1') then
          count <= regdata;
        else
          count(15 downto 1) <= count(14 downto 0);
          count(0) <= not(count(15));
        end if;
      end if;
    end process;

refresh <= count(0);

end ARCH;
```

Related Documentation

To supplement the information provided in this chapter, see the following documentation for related topics and guidelines:

- The ispLEVER software online Help
- FPGA Libraries Online Help

- TN1056 - LatticeECP/EC and LatticeXP sysIO Usage Guide
- TN1102 - LatticeECP2 sysIO Usage Guide
- TN1088 - LatticeSC PURESPEED I/O Usage Guide
- TN1091 - MachXO sysIO Usage Guide
- Precision RTL Synthesis Style Guide
- Synplicity FPGA Synthesis Reference Manual

8

Synthesis Tips for Higher Performance

This chapter provides tips on applying synthesis techniques for both Mentor Graphics Precision RTL Synthesis and Synplicity Synplify to improve design performance when you target LatticeECP/EC, LatticeXP, and MachXO devices.

Depending on your design, one tool could provide slightly better results than the other, so you may want to try both tools to see which one yields the best results.

Register Balancing and Pipelining

You can use several techniques for register balancing and pipelining to improve the maximum frequency (f_{MAX}).

Retiming

Retiming logic optimization can be used to balance the logic levels among register pairs to maximize clock rate. In fully synchronized designs, the logic between the registers determines the f_{MAX}. Keep this in mind during design synthesis. Although synthesis tools offer retiming as an option, you should attempt to balance register pairs between critical paths whenever possible.

Pipelining

Another design techniques is pipelining, which is the insertion of an additional pipeline register to achieve better clock frequency. Clock latency is introduced by pipelining. If the logic levels in-between registers cannot be reduced or balanced, try adding allowable pipeline registers. Another way to improve f_{MAX} is to turn on the retiming feature in the synthesis tool.

Always use the pipeline register available in the module generator for modules such as RAM and DSP blocks. DSP blocks contain input, pipeline, and output registers. The recommendation is to use all these registers for best system performance, as long as system latency is allowed.

As shown in Figure 74, adding allowable registers can significantly increase f_{MAX} by creating shorter t_{SU} and t_{CO} delays. The pipeline to output will be the f_{MAX}. This f_{MAX} for the DSP is not in the report since it is implied. A warning note reports that pipeline to output is the critical path. If there is another register after the output, this f_{MAX} will be reported if it is the critical path.

Figure 74: Fully Pipelining the DSP Block for Best Performance

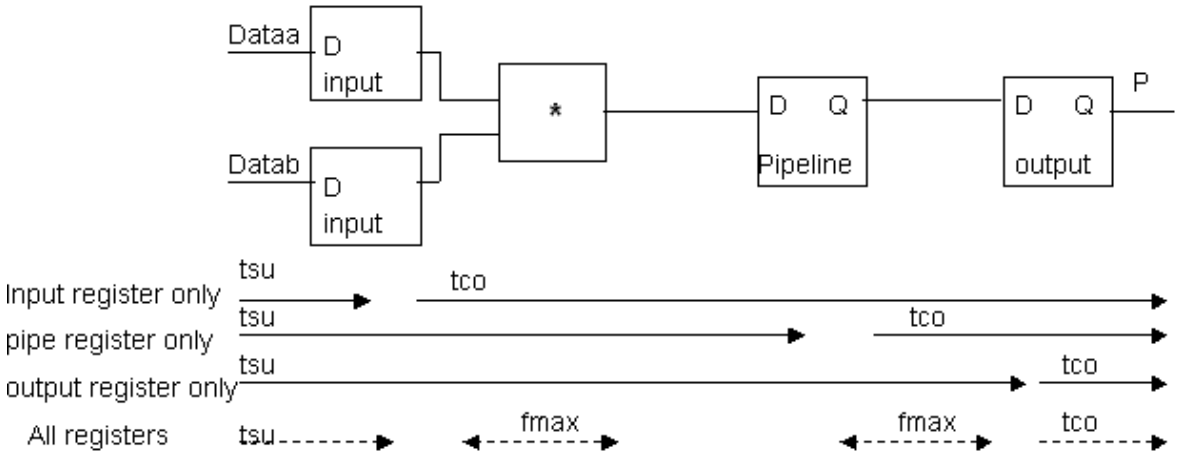


Figure 75 shows the HDL code used to achieve this result.

Figure 75: HDL Code for Fully Pipelining the DSP Block for Best Performance

```
always @(posedge clock or posedge reset)
    begin
begin
                dataa_reg = dataa; //input register
                datab_reg = datab; //input register
                qout_p = dataa_reg * datab_reg; //pipeline register
                out = qout_p; //output register
            end
    end
```

The following examples show how to reduce or eliminate TLATCH delays. Figure 76 shows a block diagram of a sample design that causes TLATCH delay. From this design, the data path goes through two adders, LUT logic, and pipeline registers. The critical path is from the adders to the LUTs.

Figure 76: Sample Design That Causes TLATCH Delay

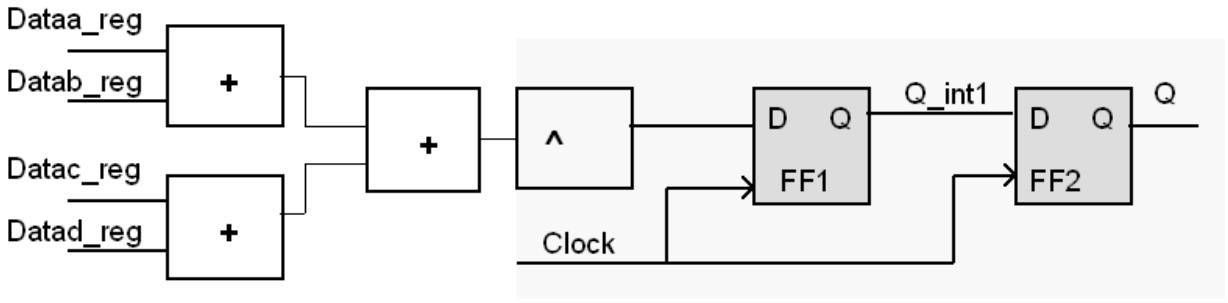


Figure 77 shows the timing report of the design shown in Figure 76.

Figure 77: Timing Report showing TLATCH Delay

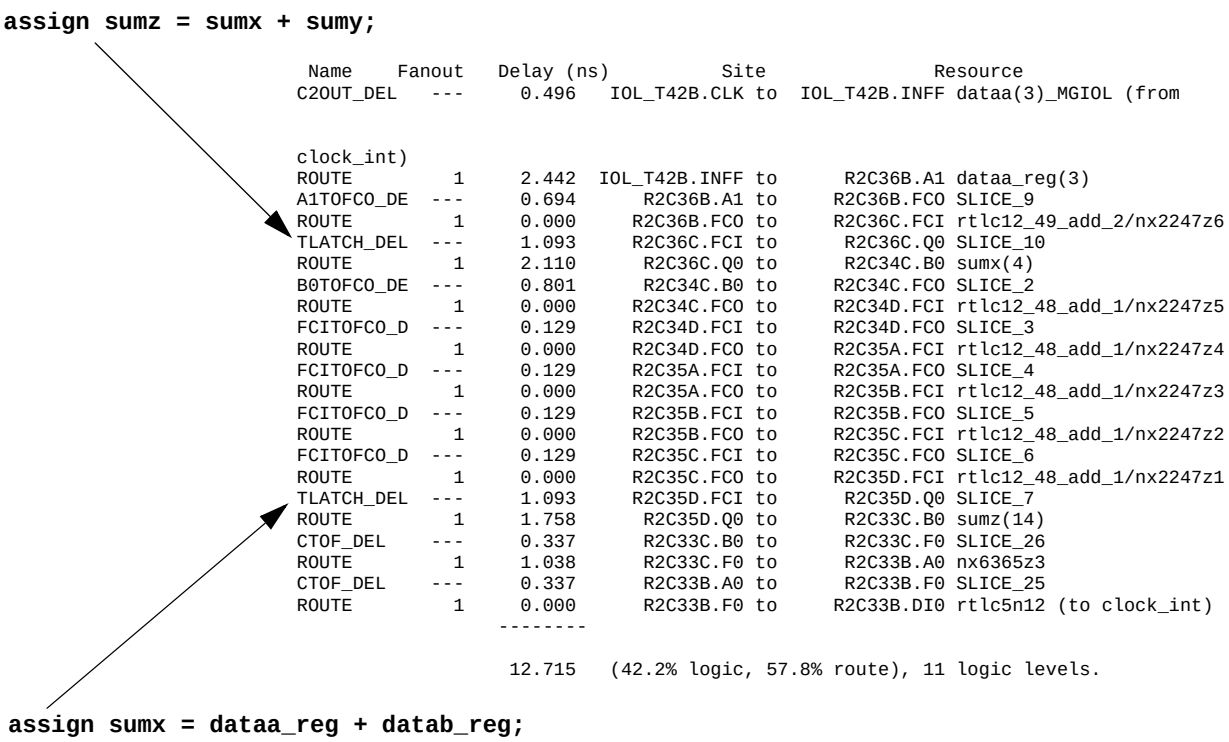


Figure 78 shows the implementation in sample code.

Figure 78: HDL Code used For Redistributing the Register to Reduce the "TLATCH_DEL" Delay

```
assign sumx = dataa_reg + datab_reg;
  assign sumy = datac_reg + datad_reg;

  @always (clock,reset)
              ……..
        q_int1 <= sumz
        q <= q_int1

              ……..
```

Because of hardware limitations, two TLATCH delays of about 1 ns each have been introduced. If arithmetic outputs are not registered, the carryout goes to a transparent latch.

Adding registers to the output of the adders eliminates these delays. There is now registered balancing with no more TLATCH delays, as shown in Figure 77. By adding registers in RTL, the design that previously ran at 80 MHz now runs at 200 MHz.

Figure 79 shows a block diagram that eliminates the TLATCH delay by redistributing the registers.

Figure 79: Redistribute the Register to Reduce the TLATCH_DEL Delay

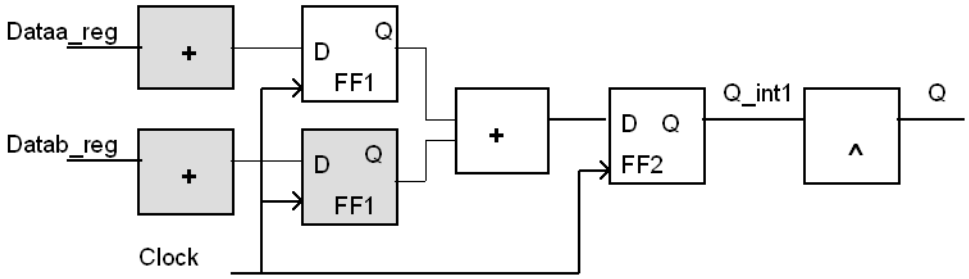


Figure 80 shows the HDL code used to achieve these results.

Figure 80: HDL Code Used to Redistribute the Register to Reduce the "TLATCH_DEL" Delay

```
@always (clock,reset)
              sumz <= sumx + sumy;
              sumx <= dataa_reg + datab_reg;
              sumy <= datac_reg + datad_reg;

Assign q <= ^Q_int1;
```

Figure 81 shows the result of the register redistribution.

Figure 81: Result of Register Redistribution

```
  Name    Fan-out   Delay (ns) Site               Resource
REG_DEL     ---     0.508    R48C31B.CLK to     R48C31B.Q1 SLICE_1 (from clock_int)
ROUTE         1     2.708     R48C31B.Q1 to     R34C31B.A1 sumx(3)
A1TOFCO_DE  ---     0.694     R34C31B.A1 to    R34C31B.FCO SLICE_18
ROUTE         1     0.000    R34C31B.FCO to    R34C31C.FCI rtlc8_16_add_2/nx2247z6
FCITOFCO_D  ---     0.129    R34C31C.FCI to    R34C31C.FCO SLICE_19
ROUTE         1     0.000    R34C31C.FCO to    R34C31D.FCI rtlc8_16_add_2/nx2247z5
FCITOFCO_D  ---     0.129    R34C31D.FCI to    R34C31D.FCO SLICE_20
ROUTE         1     0.000    R34C31D.FCO to    R34C32A.FCI rtlc8_16_add_2/nx2247z4
FCITOFCO_D  ---     0.129    R34C32A.FCI to    R34C32A.FCO SLICE_21
ROUTE         1     0.000    R34C32A.FCO to    R34C32B.FCI rtlc8_16_add_2/nx2247z3
FCITOFCO_D  ---     0.129    R34C32B.FCI to    R34C32B.FCO SLICE_22
ROUTE         1     0.000    R34C32B.FCO to    R34C32C.FCI rtlc8_16_add_2/nx2247z2
FCITOFCO_D  ---     0.129    R34C32C.FCI to    R34C32C.FCO SLICE_23
ROUTE         1     0.000    R34C32C.FCO to    R34C32D.FCI rtlc8_16_add_2/nx2247z1 (to


clock_int)
                  --------
                    4.555   (40.5% logic, 59.5% route), 7 logic levels.
```

Always register the output of the arithmetic functions to avoid the extra "TLATCH_DEL" delay.

Using Dedicated Resource GSR for f_{MAX} Improvement

If your design contains set/reset high-fan-out nets, it is recommended that you use the dedicated hardwired GSR resource. This will result in less routing congestion and could improve routability and performance.

If no GSR is used, the design will use the resources of the local set/reset that can be used for other purposes. Synthesis can automatically infer GSR whenever possible.

- When only one reset exists, always infer GSR.
- When more than one reset exists, pick the one that makes the most sense, especially the one with the biggest fan-out. Then disable the GSR for others.
- Do not infer more than one GSR cell from the RTL.
- A mapping error is caused if the GSR comes from different sources. If the GSR comes from the same source, they can be merged.

GSR can be instantiated in the RTL code to ensure usage. GSR can be assigned to any source, whether or not it has a small or large fan-out, depending on whether you want to use the GSR on a low- or high-fan-out signal. This should be considered during simulation.

Instantiating Dedicated Resource GSR in RTL Code

The following is a Verilog HDL example that instantiates dedicated resource in RTL code:

```
GSR GSR_INST( reset_sig)
```

Next is a VHDL example that instantiates dedicated resource in RTL code:

```
GSR_INST : GSR
 port map (GSR=>reset_sig)
```

Note

You must name the instance GSR_INST in order for the simulator to recognize the global implied connections to all sequential components in the design. The GSR cell is active low (when GSR = ‘0’ reset). In the software, GSR is active low by default. If you choose active high, you must tie an inverter before it goes to the GSR for correct simulation.

Improving Timing Through the I/O Register

You can improve the t_{SU} and t_{CO} timing by turning the I/O register on or off. Turning on the input register can improve the setup. Turning on the output register can improve the clock-to-out time.

There are different levels of control in synthesis. You can control synthesis:

- In the RTL code
- In the synthesis tool
- Globally or locally

After turning on the input/output register, ensure that the timing can still meet setup time, as well as f_{MAX} requirements.

Example Coded in Precision RTL Synthesis

Figure 82 shows a Verilog HDL example that turns the I/O register on and off, using the Precision RTL Synthesis tool. Note the code in bold.

Figure 82: Precision RTL Synthesis Verilog HDL Example Turning the I/O Register On and Off

```
module io_flops(q, dataa, datab, clk, rst);
    output [15:0] q;            //pragma attribute q outff true (or false)
    input   [7:0] dataa;       //pragma attribute dataa inff true (or false)
    input   [7:0] datab;       //pragma attribute datab inff true (or false)
    input   clk, rst;
    reg    [15:0] q;
    reg    [15:0] dataa_reg, datab_reg;
    wire   [7:0] mux_out;

   always @(posedge clk or posedge rst)
    begin
        if (rst)
             begin
                dataa_reg = 0;
                datab_reg = 0;
             end
        else
             begin
                dataa_reg = dataa;
                datab_reg = datab;
             end
    end
    assign mux_out = dataa_reg + datab_reg;
    always @(posedge clk or posedge rst)
    begin
        if (rst)
                q = 0;
        else
                q = mux_out;
    end

  endmodule
```

Figure 83 shows a VHDL example that turns the I/O register on and off, using the Precision RTL Synthesis tool. Note the code in bold.

Figure 83: Precision RTL Synthesis VHDL Example Turning the I/O Register On and Off

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity io_flops is
port (        q : out std_logic_vector (7 downto 0);
          dataa : in std_logic_vector (7 downto 0);
          datab : in std_logic_vector (7 downto 0);
            clk : in std_logic);
-- default is turn on the io flops.
      attribute inff  : boolean;
      attribute outff : boolean;
      attribute outff of q : signal is true;
      attribute inff of dataa : signal is true;
      attribute inff of datab : signal is true;
end io_flops;
architecture rtl of io_flops is
signal dataa_reg : std_logic_vector(7 downto 0 ) ;
signal datab_reg : std_logic_vector(7 downto 0 ) ;
signal q_int     : std_logic_vector(7 downto 0 ) ;
begin
reg_input : process (clk)
begin
   if (clk'event and clk = '1') then
               dataa_reg <= dataa ;
               datab_reg <= datab ;
   end if ;
end process reg_input ;
q_int <= dataa_reg + datab_reg;
reg_output : process (clk)
begin
   if (clk'event and clk = '1') then
       q <= q_int;
   end if ;
end process reg_output;

end rtl;
```

Examples Coded in Synplify

Figure 84 shows a Verilog HDL example that turns the I/O register on and off, using the Synplify synthesis tool. Its effect is similar to that of the DIN and DOUT attribute. But “syn_useioff” can be used for both inputs and outputs. Note the code in bold.

Figure 84: Synplify Verilog HDL Example Turning the I/O Register On and Off

```
module io_flops(q, dataa, datab, clk, rst);
    output [15:0] q;  // synthesis syn_useioff = 1  (or 0)
    input   [7:0] dataa; //synthesis syn_useioff = 1 (or 0)
    input   [7:0] datab; //synthesis syn_useIoff = 1 (or 0)
    input   clk, rst;
    reg    [15:0] q;
    reg    [15:0] dataa_reg, datab_reg;
    wire   [7:0] mux_out;

   always @(posedge clk or posedge rst)
    begin
        if (rst)
             begin
                dataa_reg = 0;
                datab_reg = 0;
             end
        else
             begin
                dataa_reg = dataa;
                datab_reg = datab;
             end
    end
    assign mux_out = dataa_reg + datab_reg;
    always @(posedge clk or posedge rst)
    begin
        if (rst)
                q = 0;
        else
                q = mux_out;
    end
  endmodule
```

Figure 85 shows a VHDL example that turns the I/O register on and off, using

the Synplify synthesis tool. Note the code in bold.

Figure 85: Synplicity VHDL Example Turning the I/O Register On and Off

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity io_flops is
port (        q : out std_logic_vector (7 downto 0);
          dataa : in std_logic_vector (7 downto 0);
          datab : in std_logic_vector (7 downto 0);
            clk : in std_logic);
    attribute syn_useioff : boolean;
    attribute syn_useioff of q : signal is true;
    attribute syn_useioff of dataa : signal is true;
    attribute syn_useioff of datab : signal is true;
end io_flops;
architecture rtl of io_flops is
signal dataa_reg : std_logic_vector(7 downto 0 ) ;
signal datab_reg : std_logic_vector(7 downto 0 ) ;
signal q_int     : std_logic_vector(7 downto 0 ) ;
begin
reg_input : process (clk)
begin
   if (clk'event and clk = '1') then
              dataa_reg <= dataa ;
              datab_reg <= datab ;
   end if ;
end process reg_input ;
   q_int <= dataa_reg + datab_reg;
reg_output : process (clk)
begin
   if (clk'event and clk = '1') then
       q <= q_int;
   end if ;
end process reg_output;
end rtl;
```

Adding Delays to Input Registers

Designs that have registered inputs can incur hold-time violations if the clock path is too fast. Therefore, a feature was added in silicon to give this fixed delay on the input register. Input registered must be inferred before the fixed delay can be turned on.

Examples Coded in Precision RTL Synthesis

Figure 86 shows a Verilog HDL example that adds delays to the input register, using the Precision RTL Synthesis tool. Note the code in bold.

Figure 86: Precision RTL Synthesis Verilog HDL Example of Adding Delays to Input Register

```
module io_flops(q, dataa, datab, clk, rst);
    output [15:0] q;
    input   [7:0] dataa;       //pragma attribute dataa inff true
  //pragma attribute dataa fixeddelay
    input   [7:0] datab;
    input   clk, rst;
    reg    [15:0] q;
    reg    [15:0] dataa_reg, datab_reg;
    wire   [7:0] mux_out;

   always @(posedge clk or posedge rst)
    begin
        if (rst)
             begin
                dataa_reg = 0;
                datab_reg = 0;
             end
        else
             begin
                dataa_reg = dataa;
                datab_reg = datab;
             end
    end
    assign mux_out = dataa_reg + datab_reg;
    always @(posedge clk or posedge rst)
    begin
        if (rst)
                q = 0;
        else
                q = mux_out;
    end
  endmodule
```

Figure 87 shows a VHDL example that adds delays to the input register, using the Precision RTL Synthesis tool. Note the code in bold.

Figure 87: Precision RTL Synthesis VHDL Example Adding Delays to Input Register

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity io_flops is
port (        q : out std_logic_vector (7 downto 0);
          dataa : in std_logic_vector (7 downto 0);
          datab : in std_logic_vector (7 downto 0);
            clk : in std_logic);
       attribute fixeddelay  : string;
       attribute fixeddelay of dataa : signal is “true”;
end io_flops;
architecture rtl of io_flops is
signal dataa_reg : std_logic_vector(7 downto 0 ) ;
signal datab_reg : std_logic_vector(7 downto 0 ) ;
signal q_int     : std_logic_vector(7 downto 0 ) ;
begin
reg_input : process (clk)
begin
   if (clk'event and clk = '1') then
               dataa_reg <= dataa ;
               datab_reg <= datab ;
   end if ;
end process reg_input ;
q_int <= dataa_reg + datab_reg;
reg_output : process (clk)
begin
   if (clk'event and clk = '1') then
       q <= q_int;
   end if ;
end process reg_output;
end rtl;
```

Examples Coded in Synplify

Figure 88 shows a Verilog HDL example that adds delays to the input register, using the Synplify synthesis tool. Note the code in bold.

Figure 88: Synplify Verilog HDL Example of Adding Delays to Input Register

```
module io_flops(q, dataa, datab, clk, rst);
    output [15:0] q;
    input   [7:0] dataa; //synthesis syn_useioff = 1 FIXEDDELAY=TRUE
    input   [7:0] datab;
   input   clk, rst;
    reg    [15:0] q;
    reg    [15:0] dataa_reg, datab_reg;
    wire   [7:0] mux_out;

   always @(posedge clk or posedge rst)
    begin
        if (rst)
             begin
                dataa_reg = 0;
                datab_reg = 0;
             end
        else
             begin
                dataa_reg = dataa;
                datab_reg = datab;
             end
    end
    assign mux_out = dataa_reg + datab_reg;
    always @(posedge clk or posedge rst)
    begin
        if (rst)
                q = 0;
        else
                q = mux_out;
    end

  endmodule
```

Figure 89 shows a VHDL example that adds delays to the input register, using the Synplify synthesis tool. Note the code in bold.

Figure 89: Synplify VHDL Example of Adding Delays to Input Register

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity io_flops is
port (        q : out std_logic_vector (7 downto 0);
          dataa : in std_logic_vector (7 downto 0);
          datab : in std_logic_vector (7 downto 0);
            clk : in std_logic);
    attribute syn_useioff : boolean;
    attribute FIXEDDELAY: string;
    attribute syn_useioff of dataa : signal is true;
    attribute FIXEDDELAY of dataa : signal is “TRUE”;

end io_flops;
architecture rtl of io_flops is
signal dataa_reg : std_logic_vector(7 downto 0 ) ;
signal datab_reg : std_logic_vector(7 downto 0 ) ;
signal q_int     : std_logic_vector(7 downto 0 ) ;
begin
reg_input : process (clk)
begin
   if (clk'event and clk = '1') then
               dataa_reg <= dataa ;
               datab_reg <= datab ;
   end if ;
end process reg_input ;
   q_int <= dataa_reg + datab_reg;
reg_output : process (clk)
begin
   if (clk'event and clk = '1') then
       q <= q_int;
   end if ;
end process reg_output;
end rtl;
```

Maximum Fan-Out Control for f_{MAX} Improvement

Maximum fan-out can be selectively applied to the critical path to reduce fan-out. This attribute in the design will override the global maximum fan-out control.

In most cases, if registers are duplicated to reduce the maximum fan-out, it will increase the register count in the design.

Examples Coded in Synplify

Figure 90 shows a Verilog HDL example of maximum fan-out control coded with the Synplify synthesis tool. Note the code in bold.

Figure 90: Synplify Verilog HDL Example of MAX Fan-Out Control

```
module test (registered_ data_ out, clock, data_ in);
output [31: 0] registered_ data_ out;
input clock;
input [31: 0] data_ in /* synthesis syn_ maxfan= 1000 */;
reg [31: 0] registered_ data_ out /* synthesis syn_ maxfan= 1000 */;
```

Figure 91 shows a VHDL example of maximum fan-out control coded with the Synplify synthesis tool. Note the code in bold.

Figure 91: Synplify VHDL Example of Maximum Fan-Out Control

```
entity test is
port( clock : in bit;
         data_in : in bit_vector( 31 downto 0);
          egistered_data_out: out bit_ vector( 31 downto 0) )

          attribute syn_maxfan : integer;
          attribute syn_maxfan of data_in : signal is 1000;
```

Examples Coded in Precision RTL Synthesis

Figure 92 shows a Verilog HDL example of maximum fan-out control coded in the Precision RTL Synthesis tool. Note the code in bold.

Figure 92: Precision RTL Synthesis Verilog HDL Example of Maximum Fan-Out Control

```
module test (registered_data_out, clock, data_ in);
output [31: 0] registered_data_out;
input clock;
input [31: 0] data_in
reg [31: 0] registered_ data_out //pragma attribute max_fanout 100;
```

Figure 93 shows a VHDL example of maximum fan-out control coded with the Precision RTL Synthesis tool. Note the code in bold.

Figure 93: Precision RTL Synthesis VHDL Example of Maximum Fan-Out Control

```
entity test is
port( clock : in bit;
         data_in : in bit_vector( 31 downto 0);
          egistered_data_out: out bit_ vector( 31 downto 0) )

          attribute max_fanout : integer;
          attribute max_fanout of data_in : signal is 10;
```

Clock-Enable Control for f_{MAX} Improvement

The clock-enable net is typically a high-fan-out net driving several D flip-flops.

The placement and routing process uses the fan-out to decide whether to implement the clock enable by using a secondary clock resource, which sometimes incurs a larger delay (approximately 3 ns). You can specify a constraint to avoid using the secondary clock.

If some clock enables are in the critical path, you can identify them in the source code, and you can set the clock enable off to avoid a delay.

Examples Coded in Synplify

Figure 94 shows a Verilog HDL example of clock-enable control coded with the Synplify synthesis tool. Note the code in bold.

Figure 94: Synplify Verilog HDL Example of Clock_Enable Control

```
reg [3: 0] q /* synthesis syn_useenables = 0 */;
always @( posedge clk)
if (enable)
q <=d;
```

Figure 95 shows a VHDL example of clock-enable control coded with the Synplify synthesis tool. Note the code in bold.

Figure 95: Synplify VHDL Example of Clock_Enable Control

```
signal q_int : std_logic_vector( 3 downto 0);
Attribute syn_useenables : boolean;
attribute syn_useenables of q_int : signal is false;
process( clk)
begin
if (clk'event and clk = '1') then
           if (enable = '1') then
                 q_int <= d;
           end if;
end if;
end process;
```

Examples Coded in Precision RTL Synthesis

Figure 96 shows a Verilog HDL example of clock-enable control coded with the Precision RTL Synthesis tool. Note the code in bold.

Figure 96: Precision RTL Synthesis Verilog HDL Example of Clock-Enable Control

```
reg [3: 0] q // pragma attribute q use_dffenables false;
always @( posedge clk)
if (enable)
q <=d;
```

Figure 97 shows a VHDL example of clock-enable control coded with the Precision RTL Synthesis tool. Note the code in bold.

Figure 97: Precision RTL Synthesis VHDL Example of Clock-Enable Control

```
signal q_int : std_logic_vector( 3 downto 0);
Attribute use_dffenables : boolean;
attribute use_dffenables of q_int : signal is false;
 ...
begin
...
process( clk)
begin
if (clk'event and clk = '1') then
           if (enable = '1') then
                 q_int <= d;
end if;
end if;
end process;
```

General Constraint Considerations

General constraint considerations include the following:

- Synthesis constraints are sometimes different from those in placement and routing.
- ispLEVER tools cannot “merge” the timing constraints from synthesis.
- For Synplify, over-constraining generally produce better results.
- Do not over-constrain the placement and routing.
- Under-constrain the design the first time. Try to estimate the design performance, then try to optimize it.
- Precision retiming is very sensitive to timing constraints.
- For multiple clock designs, put adequate constraint on each clock, and do not put same constraints on all clocks. For example, in a design with two clocks, clock 1 can operate at 100 MHz, and the requirement is 40 MHz; clock 2 can operate at 80 MHz, and the requirement is 120 MHz. It is

better to set clock 1 to 50 MHz and clock 2 to 140, instead of setting both to 140 MHz.

Block-Level Synthesis Methods in Synplify

As designs become bigger, such as designs with multiple interfaces between blocks, it is recommended that you modularize them. In Synplify, block-level synthesis methods are needed to keep the modules intact.

I/Os are not inserted on the top level. They are treated as a macro and not optimized at compilation.

You can control I/O insertion globally or on a port-by-port basis. You can use the appropriate attribute before synthesizing an entire design to check the area requirements. If you disable automatic I/O insertion, the design will not have any I/O pads, unless you instantiate them manually.

Compile the block with no I/O buffer insertion by:

- Setting syn_force_pad to 0
- Attaching the syn_hier = macro property or setting syn_black_box to true

You can compile this as part of your larger design.

Block-Level Synthesis Methods in Precision RTL Synthesis

In Precision RTL Synthesis, block-level synthesis methods are needed to keep the modules intact.

Compile the block with no I/O buffer insertion:

- Set nopad to true on each port.
- Attach the dont_touch property to the module.

You can compile this as part of your bigger design.

Note

For some cores with pre-inserted IO pads, such as PCI, you must apply black_box_pad_pin to avoid aditional I/O insertion.

Turning Off Mapping DSP Multipliers in RTL

The following examples show how to turn off mapping DSP multipliers in the Synplify and Precision RTL Synthesis tools.

Examples Coded with Synplify

Figure 98 shows a Verilog HDL example, coded in the Synplify synthesis tool, that turns off mapping DSP multipliers. Note the code in bold.

Figure 98: Synplify Verilog HDL Example Turning Off Mapping DSP Multiplier

```
module multi_resource(qout, dataa, datab, clock, reset);

    input [17:0] dataa, datab;
    input clock, reset;
    output [17:0] qout;

    reg    [35:0] qout;
    reg    [35:0] qout_p;
    reg    [35:0] dataa_reg;
    reg    [35:0] datab_reg;
   wire [35:0] qout_mult /*  Synthesis syn_multstyle="logic" */;
    assign qout_mult = dataa_reg * datab_reg ;
    always @(posedge clock or posedge reset)
    begin
  if (reset)
                                  begin
                                         dataa_reg = 0;
                                         datab_reg = 0;
                                         qout = 0;
                                  end
  else
                                     begin
                                          dataa_reg = dataa;
                                          datab_reg = datab;
                                          qout = qout_mult;
          end
    end
  endmodule
```

Figure 99 shows a VHDL example, coded with the Synplify synthesis tool, that turns off mapping DSP multipliers. Note the code in bold.

Figure 99: Synplify VHDL Example Turning Off Mapping DSP Multiplier

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity multi_resource is
port (        q : out std_logic_vector (35 downto 0);
         dataa : in std_logic_vector (17 downto 0);
         datab : in std_logic_vector (17 downto 0);
        rst : in std_logic;
            clk : in std_logic);
end multi_resource;
architecture rtl of multi_resource is
signal dataa_reg : std_logic_vector(17 downto 0 ) ;
signal datab_reg : std_logic_vector(17 downto 0 ) ;
signal q_int     : std_logic_vector(35 downto 0 ) ;
attribute syn_multstyle : string;
attribute syn_multstyle of q_int : signal is "logic";

begin
reg_input : process (clk)
begin
   if (clk'event and clk = '1') then
               dataa_reg <= dataa ;
               datab_reg <= datab ;
   end if ;
end process reg_input ;
   q_int <= dataa_reg * datab_reg;
reg_output : process (clk)
begin
   if (clk'event and clk = '1') then
       q <= q_int;
   end if ;
end process reg_output;
end rtl;
```

Examples Coded with Precision RTL Synthesis

Figure 100 shows a Verilog HDL example, coded with the Precision RTL Synthesis tool, that turns off mapping DSP multipliers. Note the code in bold.

Figure 100: Precision RTL Synthesis Verilog HDL ExampleTurning Off Mapping DSP Multiplier

```
module multi_resource(qout, dataa, datab, clock, reset);
    input [17:0] dataa, datab;
    input clock, reset;
    output [35:0] qout;
    reg    [35:0] qout;
    reg    [35:0] dataa_reg;
    reg    [35:0] datab_reg;
    wire [35:0] qout_mult;
   assign qout_mult = dataa_reg * datab_reg;

//pragma attribute qout_mult dedicated_mult OFF
// Currently does not work, need to work around in setting the operator in the GUI

    always @(posedge clock or posedge reset)
    begin
        if (reset)
             begin
                    dataa_reg = 0;
                    datab_reg = 0
                     qout = 0;
          end
        else
                begin
                     dataa_reg = dataa;
                     datab_reg = datab
                     qout = qout_mult;
          end
    end
  endmodule
```

Figure 101 shows a VHDL example, coded with the Precision RTL Synthesis tool, that turns off mapping DSP multipliers. Note the code in bold.

Figure 101: Precision RTL Synthesis VHDL Example Turning Off Mapping DSP Multiplier

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity multi_resource is
port (        q : out std_logic_vector (35 downto 0);
          dataa : in std_logic_vector (17 downto 0);
          datab : in std_logic_vector (17 downto 0);
        rst : in std_logic;
            clk : in std_logic);
end multi_resource;
architecture rtl of multi_resource is
signal dataa_reg : std_logic_vector(17 downto 0 ) ;
signal datab_reg : std_logic_vector(17 downto 0 ) ;
signal q_int     : std_logic_vector(35 downto 0 ) ;
attribute dedicated_mult : string;
attribute dedicated_mult of q_int : signal is "OFF";
begin
reg_input : process (clk)
begin
   if (clk'event and clk = '1') then
              dataa_reg <= dataa ;
              datab_reg <= datab ;
   end if ;
end process reg_input ;
   q_int <= dataa_reg * datab_reg;
reg_output : process (clk)
begin
   if (clk'event and clk = '1') then
       q <= q_int;
   end if ;
end process reg_output;
end rtl;
```

Achieving Improved Synthesis Results by Assigning Black-Box Timing to Large Embedded Blocks

If you instantiate a large embedded block like DSP or EBR, synthesis will treat the large block as a “black-box.” The timing information will be ignored in the synthesis tool, and sometimes a warning message will be displayed during synthesis.

If the large block is part of the critical path, you can assign timing delay properties to the black-box so that the synthesis tool can apply the correct timing for the synthesis and mapping processes.

You can make changes to these timing delays on the black-box to over-compensate or under-compensate timing in the remaining portion of the critical path.

Table 19 shows supported Synplicity syntax that allows you to apply black-box timing to instantiated blocks.

Table 19: Synplicity Syntax for Black-Box Timing

Syntax	Description
syn_isclock	Specifies a clock port on a black-box.
syn_tpd<n>	Timing propagation for combinational delay through the black box.
syn_tsu<n>	Timing setup delay required for input pins relative to the clock.
syn_tco<n>	Timing clock to output delay through the black-box.

Figure 102 shows an example of applying black-box timing in VHDL using Synplify.

Figure 102: VHDL Black-Box Timing Example Using Synplify

```
COMPONENT spr16x4a
PORT(
        di0  : IN std_logic;
        di1  : IN std_logic;
        di2  : IN std_logic;
        di3  : IN std_logic;
        ck  : IN std_logic;
        wre  : IN std_logic;
        ad0 : IN std_logic;
        ad1 : IN std_logic;
        ad2 : IN std_logic;
        ad3 : IN std_logic;
        do0 : OUT std_logic;
        do1 : OUT std_logic;
        do2 : OUT std_logic;
        do3 : OUT std_logic);
END COMPONENT;

attribute syn_tpd1 of rcf16x4z : component is
   "ado,ad1,ad2,ad3 -> do0,do1,do2,do3 = 1.1";
attribute syn_tsu1 of rcf16x4z : component is
   "ado,ad1,ad2,ad3 -> ck = 0.5";
attribute syn_tsu2 of rcf16x4z : component is
   "wre -> ck = 0.5";
```

Figure 103 shows an example of applying black-box timing in Verilog using Synplify.

Figure 103: Verilog Black-Box Timing Example Using Synplify

```
module SPR16X4A (DI0, DI1, DI2, DI3, AD0, AD1, AD2, AD3, WRE, CK,
                 DO0, DO1, DO2, DO3)
/* synthesis black_box
syn_tpd1="AD0,AD1,AD2,AD3->DO0,DO1,DO1,DO3 =1.4"
syn_tsu1="AD0,AD1,AD2,AD3->CK = 0.5"
syn_tsu2="WRE->CK = 0.5" */;

  input AD0,AD1,AD2,AD3,DI0, DI1, DI2, DI3, CK, WRE;
  output DO0, DO1, DO2, DO3;
```

9

Strategies for Timing Closure

This chapter describes strategies that will help achieve timing closure for the most aggressive design requirements. It begins with a brief description of the seven steps for timing closure, followed by instructions for implementing each of these steps using the ispLEVER software.

Seven Steps to Timing Closure

A timing closure strategy always begins with the creation of meaningful and efficient HDL code. For information about coding techniques for FPGA designs, see “HDL Synthesis Coding Guidelines” on page 57.

After writing FPGA-friendly code, use the following seven-step strategy to help achieve timing closure:

1. Set FPGA preferences to achieve timing goals.

   Along with a good functional design, a good set of FPGA timing preferences are crucial for meeting timing goals.

   See “Constraining Designs” on page 136.

2. Run an initial Place & Route (PAR) Design process.

   Select timing-driven placement and specify a low placement effort level for this first PAR process.

   See “Using the Place and Route Software (PAR)” on page 151.

3. Analyze timing.

   Run the Timing Reporter and Circuit Evaluator (TRACE) after you run the initial Place & Route Design process. Examine the timing information in the TRACE report, Map report, Place and Route report, and PAD report.

   See “Performing Static Timing Analysis” on page 152.

4. Modify timing preferences.

   Assign primary and secondary clocks, tune I/O timing with PLLs, and group components along critical paths.

   See “Adding and Modifying Timing Preferences” on page 145

5. Run a second Place & Route Design process.

   Use timing-driven placement, and experiment with increased placement effort and multiple routing passes.

   See “Controlling Placement and Routing” on page 165.

6. Analyze timing.

   Identify high-fan-out nets, critical path nets, and long delay paths.

7. Floorplan to direct the physical layout of the circuit.

   For designs that do not meet performance goals, use groups and regions to place components closer together and shorten routing distances. Use reiterative floorplanning, repeating steps 5 through 7 until performance goals are achieved.

   See “Floorplanning the Design” on page 173.

Constraining Designs

FPGA designs require effective constraints in order to optimize the usage of resources. For Lattice FPGA devices, such design constraints are referred to as preferences.

You can set and edit design preferences at multiple points in the FPGA design flow. New and modified preferences are saved to the logical preference file (.lpf).The logical equivalents of physical preferences, such as groups, regions, and pin assignments, are also saved to the logical preference file with the Save command in the Design Planner. During the Map Design process, these physical preferences are written to the physical preference file (.prf).

Logical Preference File (.lpf)

The logical preference file (.lpf) contains all the design constraints that you specify for an FPGA design after the Build Database process. All preferences that are created or modified are written to the .lpf. You can add or modify preferences before mapping, after mapping, or after placement and routing. Post-map or post-PAR changes are implemented by rerunning the Map Design process.

HDL-based attributes are a potential source for preferences. HDL attributes are converted to EDIF properties by logic synthesis, and in some cases they are converted into preferences by the design mapper. After the Build Database process, you can view some attributes as preferences in the Design Planner. After they are modified, they are also listed in the .lpf, and these modified preferences take precedence over the HDL.

For more information about using HDL attributes, refer to the “HDL Attributes” section of the ispLEVER FPGA Design online Help.

Figure 104 shows a small example from an ASCII logical preference file.

Figure 104: Sample Logical Preference File

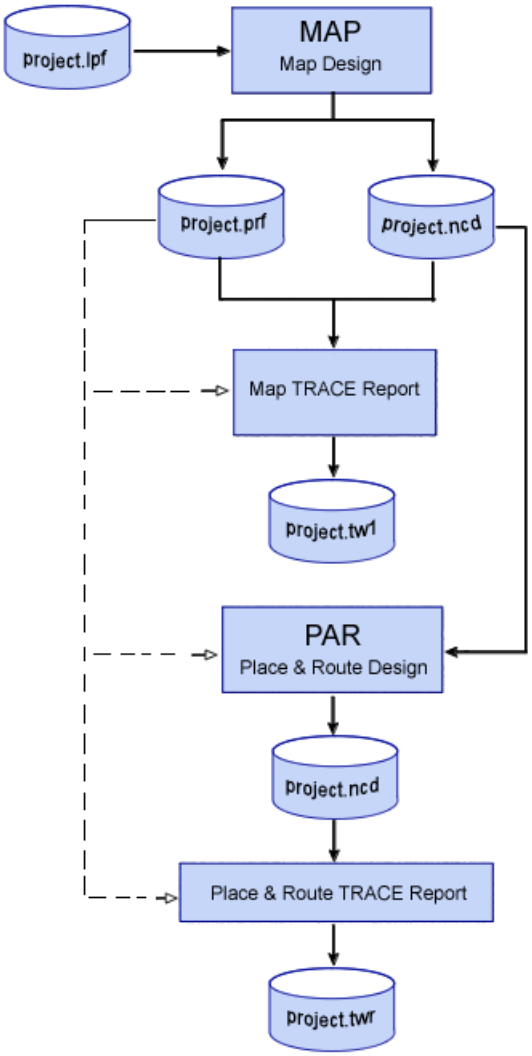
```
FREQUENCY PORT "tx_clk" 50 MHz ;
FREQUENCY PORT "rx_clk" 50 MHz ;
INPUT_SETUP "port1" 1 NS CLKNET "tx_clk";
CLOCK_TO_OUT "port2" 8 NS CLKNET "rx_clk";
LOCATE COMP "port1" SITE "B17" ;
LOCATE COMP "port2" SITE "C16" ;
LOCATE COMP "tx_clk" SITE "B11" ;
LOCATE COMP "rx_clk" SITE "C13" ;
```

You can apply two types of preferences: timing and location. Timing preferences control the timed paths in an FPGA design. Location preferences affix the placement of design components in the FPGA array and are most commonly used for setting the I/O pinout.

How ispLEVER Uses the Logical Preference File

The ispLEVER software uses the logical preference file (.lpf) to guide the tools while implementing the functional design from the EDIF file. The ispLEVER Map Design process (MAP) reads the logical preference file to map the logical elements and generates a physical preference file (.prf) that is used by PAR and TRACE. This flow is illustrated in Figure 105.

Figure 105: Project Preferences Applied to MAP, PAR, and TRACE



MAP

The ispLEVER Map Design process (MAP) takes the logical design from the Build Database process and maps the logical elements to specific elements, such as slices and PIOs. The logical preference file (.lpf) is used as an input to MAP. The mapper filters the preference file to remove any syntax errors or invalid preferences. Invalid preferences are preferences that do not correspond to any logical or physical elements that are in the design. Invalid preferences are most often caused by a typographical error in the element

name.

MAP generates a physical preference file (.prf), which contains all of the valid preferences contained in the logical preference file, as well as all of the attributes that were included in the EDIF file through the HDL code. This file is used for placement, routing, and static timing analysis.

TRACE

You have the option, both after mapping and after placement and routing, of performing static timing analysis on the physical design (.ncd) file, using the Timing Reporter and Circuit Evaluator (TRACE). TRACE takes the physical preference file as input and uses the timing constraints contained in it to produce a report file.

To run TRACE for a pre-routed design:

- In the Project Navigator, select the target device, and double-click **Map TRACE Report** in the Processes pane.

  The Map TRACE report (.tw1 file) appears in the output pane on the bottom right.

You should examine the results of the Map TRACE report before continuing on to placement and routing. Considerations include warnings, errors, and potential design issues; for example, a high number of logic levels might severely restrict design performance, and performance might benefit from a different partitioning or pipelining. Since no routing exists yet between logical connections, the Map TRACE delay report reflects an ideal situation—usually about twice the performance that will be shown in the PAR TRACE report (.twr file). Therefore, you should anticipate that routing delays will represent 40-50 percent of the delay along combinatorial paths.

To run TRACE for a post-routed design:

- In the Project Navigator, select the target device, and double-click **Place & Route TRACE Report** in the Processes pane.

  The PAR TRACE report appears in the output pane.

Timing checkpoints are a feature of the Project Navigator where TRACE is automatically run before and after PAR and a report is output to the Automake.log file. To access checkpoint options, choose **Tools > Timing Checkpoint Options** and use the Timing Checkpoint Options dialog box to specify whether the forward progress should stop or continue when the checkpoint fails.

For more information about TRACE, see “Performing Static Timing Analysis” on page 152.

PAR

The Place and Route (PAR) process uses both location and timing preferences to drive the placement and routing of the design in the FPGA device. The output of PAR is a placed and routed design, as well as a report and I/O pinout file.

Creating and Editing Preferences

Creating design preferences is a process that continuously evolves throughout the design process. Certain preferences are used during the map phase, and others are applied during placement and routing. Finally, preferences are interpreted by reporting tools such as TRACE for static timing analysis to provide important information on the final design. Preferences can be applied at many points in the design flow process.

Note

Location preferences can be assigned as attributes in the HDL code for device floorplanning. Such preferences are included in the EDIF file after synthesis, and they are carried into the physical preference file created by MAP. See “Floorplanning the Design” on page 173.

Place-and-Route Preference Format

Remember the following points about the preference file format:

- The preference file can contain any number of preferences and any number of comments in any order.
- Comments must be preceded by the pound sign (#) or double slashes (//).
- The ispLEVER programs automatically comment or ignore illegal preferences.

Rules for Preferences

Observe the following precedence rules when setting preferences:

- Preferences saved in the .lpf take precedence over HDL preferences.
- When more than one preference applies to a net or path, more specific preferences are honored before less specific ones. For example, individual net or path preferences supersede group (bus) preferences, and group preferences supersede global preferences.
- If there is more than one preference at the same level of specificity for a net or path object, the last such preference in the preference file takes precedence.

To illustrate, suppose that the preference file contains the following preferences:

```
MAXDELAY NET W 10 NS;
MAXDELAY ALLNETS 30 NS;
DEFINE BUS B NET Y NET Z;
DEFINE BUS A NET Y NET X NET W;
MAXDELAY BUS B 20 NS;
MAXDELAY BUS A 25 NS;
MAXDELAY NET W 15 NS;
```

Net W gets 15 nanoseconds because this preference is more specific than BUS or ALLNETS, and it comes after the 10-nanosecond preference.

Net X gets 25 nanoseconds because the BUS A preference is more specific than ALLNETS.

Net Y gets 25 nanoseconds because the BUS A preference comes after the Bus B preference.

Net Z gets 20 nanoseconds because the BUS preference is more specific than ALLNETS.

All other nets get 30 nanoseconds.

Logical and Physical Preferences

Timing preferences can be assigned in two design domains: logical and physical. The logical domain consists of the design element names of the EDIF netlist. These names are based on the hierarchy level and register names in the design. The physical domain consists of the physical elements that the mapper has selected for the device implementation.

Location preferences in the preference file can only be applied to physical elements or components of the design. Location preferences are used by the place-and-route tool, which works on the mapped physical design. To make this process easier for you, the mapper keeps top-level port names the same as those found in the logical design (or EDIF netlist). Fixing I/O pinouts using location preferences can be based on known component names.

For logical preferences, the preference file also supports a feature known as “wildcards.” Wildcards enable you to assign preferences to multiple design elements without having to assign a preference to each element. For example, to apply a clock_to_out preference on an entire bus, outa(31:0), you could use the following preference:

```
CLOCK_TO_OUT "outa*" 8 NS CLKNET
```

In the preference file, it is possible to create conflicts between preferences. These conflicts typically occur when a global preference covers a path that is covered specifically by another preference. In this situation, the more specific preference is used for the specified path. If the situation arises where both preferences are at the same level, the ispLEVER software uses the

preference that is last in the file. Table 20 summarizes the differences between logical and physical preferences.

Table 20: Logical vs. Physical Preferences

Logical	Physical
Based on your design (registers, I/O ports)	Based on mapped design (PFUs, SLICES, and PIOs)
Can write in terms of register names, port names	Need to know component (PFU, SLICE, or PIO) names defined by the mapper
Wildcards allowed	No wildcards allowed
Front-end oriented (EDIF)	Back-end oriented (.ncd): nets, components

Preference Editing Tools

The ispLEVER software provides two tools for editing preferences: the ASCII logical preference file and the Design Planner.

ASCII Logical Preference File (.lpf)

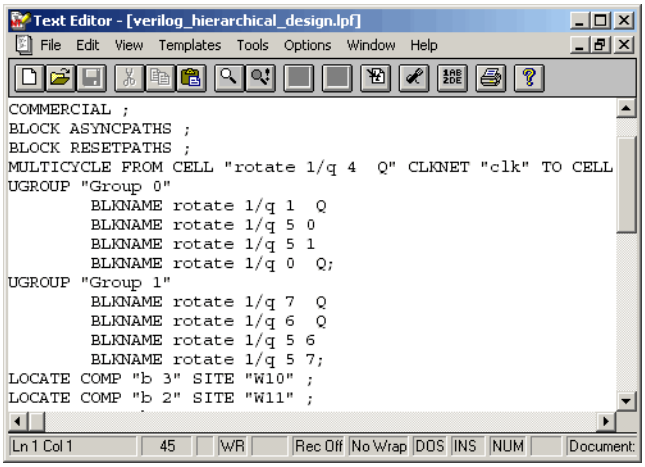
The ASCII logical preference file enables you to edit preferences in a text editor after you build the database.

To edit preferences in the ASCII logical preference file:

- In the Project Navigator, double-click **Edit Preferences (ASCII)**.

  The file opens in the default ispLEVER Text Editor or in the text editor that you have selected in the ispLEVER Environment Options dialog box.

Figure 106: ASCII Logical Preference File

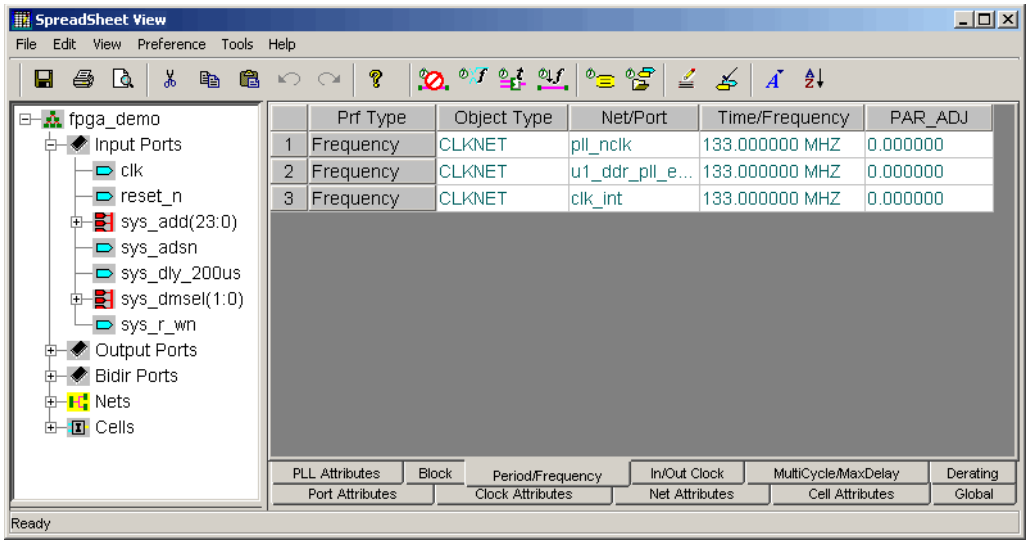


Design Planner

The ispLEVER Design Planner enables you to create and edit preferences in the pre-map stage, as well as the post-map and post-PAR stages.

Pre-Map Design Planner In the pre-map stage, the Design Planner opens the Spreadsheet View and the Package View for setting timing and constraints and making pin assignments. Many of the FPGA preferences can be edited within the Spreadsheet View of the Design Planner, including global and component-specific preferences. The Design Planner provides a spreadsheet format for each preference and organizes them in individual tabs across the bottom of the right pane, as shown in Figure 107.

Figure 107: Pre-Map Design Planner Spreadsheet View



The Spreadsheet View toolbar launches dialog boxes for setting timing preferences, including Period and Frequency, Block, Input Setup/Clock-to-Out, Multicycle/Maxdelay. New and modified preferences that are saved in the Design Planner are written to the logical preference file.

To run the pre-map Design Planner, select the targeted device in the Project Navigator, and then do the following:

- Double-click **Design Planner (Pre-Map)**.

  The Pre-Map Design Planner loads the logical design database (.ngd) and displays the Spreadsheet View and the Package View.

Post-Map Design Planner The Post-Map Design Planner enables you to create and edit location, group, and region preferences and view the assignments in a floorplan layout. All design changes, including the logical equivalents of physical groups and regions, are saved to the logical preference file.

To run the post-map Design Planner, select the targeted device in the Project Navigator, and then do the following:

- Double-click **Design Planner (Post-Map)** or **Design Planner (Post-PAR)**.

See “Using the Design Planner Interface” on page 192.

Preference Flow

The ispLEVER preference flow is designed to allow you to work with the FPGA design in terms of high-level logical elements, such as ports, nets, registers, and special Lattice Semiconductor blocks derived from the RTL source through logic synthesis. The ASCII logical preference file (.lpf) is the primary interface for defining timing and location constraints in terms of logical elements. This preference flow is shown in Figure 138 on page 192.

The .lpf is interpreted by the design mapper (MAP) and converts, when necessary, preferences that are written in terms of logical elements into physical preferences, such as PIOs, slices, and ASIC blocks. These new preferences are written to an ASCII physical preference file (.prf), which is used by the placement and routing (PAR) and static timing analysis (TRACE) tools.

In some implementation scenarios, you may need to interact with the ASCII physical preference file (.prf) directly, using a text editor, Design Planner, or the EPIC Device Editor. The .prf file always stores preferences that refer to physical elements.

For each new project, the ispLEVER software creates a logical preference file (.lpf) for the design that includes some default preferences. However, most FPGA designs require additional timing and location constraints for placement and routing or static timing analysis.

Preferences can originate from one of the preference editing tools just described, the input EDIF netlist, or as a direct output of your logic synthesis tool.

The preferences that appear in the .lpf file depend on the process that was last executed and any modifications that you have made using a preference editing tool such as the Design Planner.

Preference Flow Example

The following scenario illustrates a typical ispLEVER preference flow:

1. HDL-based attributes are produced by IPexpress for a sysCLOCK PLL as part of the module source code. Logic synthesis writes these attributes as EDIF properties into the netlist.
2. You run the Build Database process and then launch the Pre-Map Design Planner to lock PIO locations, define signal standards for sysIO buffers, and establish timing constraints. The editors write the new preferences to the .lpf.
3. You run the Map Design process. The .lpf is converted into a .prf. You run the Place & Route Design process and the Place & Route TRACE Report process to inspect the timing and utilization results.
4. On the basis of the analysis results, you decide to group physical elements along a critical path by using the Post-PAR Design Planner. You

modify the .prf and rerun the Place & Route Design process and Place & Route TRACE Report process.

Note

In many cases, HDL-based attributes that refer to logical elements do not appear in the .lpf unless they are modified by the Pre-Map Design Planner. For example, UGROUP/HGROUP attributes are included in the .lpf only after they are modified.

General Strategy Guidelines

Observe the following general recommendations for setting preferences, analyzing timing, and running PAR:

- Analyze timing results carefully in the Timing Reporter and Circuit Evaluator (TRACE) Map report (.tw1 file) and PAR report (.twr file).
- Before you place and route a design, look at the mapped frequency in the preference file and check for errors and warnings. Also, check for logic depth, which is reported in the .tw1 files as logic levels (components).
- Determine if design changes are required. A typical example design change is pipelining, or registering, the data path. This technique might be the only way to achieve high internal frequencies if the design's logic levels are too deep.
- Perform placement and routing early in the design phase, using a preliminary preference file, to gather information about the design.
- Tune up your preference file to include all I/O and internal timing paths, as appropriate. See "Translating Board Requirements into FPGA Preferences" on page 148 for an appropriate preference file example.
- Establish the pinout in the preference file. You can also locate I/O in the HDL, as well as in synthesis constraint files.
- Push PAR when necessary by running multiple routing iterations and multiple placement iterations.
- Revise the preference file as appropriate; use multi-cycle opportunities when possible.
- Floorplan the design if necessary. See "Floorplanning the Design" on page 173.

Adding and Modifying Timing Preferences

Use the following guidelines for adding and modifying timing preferences.

Assign Primary or Secondary Clocks

For designs with many clocks, assign the clocks manually.

- Primary clock resources on a device are limited. Therefore, during PAR, the clock nets with the most loads are assigned the primary clock resources. You can override this default by using PRIMARY and SECONDARY preferences.

```
# Primary/secondary preference example
USE PRIMARY
USE SECONDARY
```

- To get an accurate 90-degree phase shift, use two primary clock nets: one for the feedback path and one for the shifted clock.

  This strategy limits uncertainty to the insertion delay of sysCLOCK PLL (pad to input). The uncertainty can then be reconciled with FDEL settings in 250-picosecond increments.

- Place the source of internally generated clocks (divider) as close to the center of the device as possible to reduce injection time.

  This step is especially important for secondary clocks, since they do not have feed lines.

Tune I/O Timing with PLLs

Tuning the I/O timing with PLLs reconciles internal timing to an external specification.

Group Components Along Critical Paths

UGROUP and PGROUP preferences direct the PAR software to place components close together, which shortens routing distances.

Typical Design Preferences

The full preference language includes many different design constraints, from global preferences to very specific preferences. Listed here are the recommended preferences that you can apply to all designs.

- Block Asynchronous Paths

  This preference prevents the timing tools from analyzing any paths from input pads to registers or from input pads to output pads.

- Block RAM Reads During Write

  If you are using PFU-based RAM, this preference will prevent timing analysis on a RAM read during a write on the same address in a single clock period.

- Frequency/Period <*net*>

  Each clock net in the design should contain a frequency or period preference.

- Input Setup

  Each synchronous input should have an Input Setup preference.

- Clock-to-Out

  Each synchronous output should have a Clock-to-Out preference.

- Block <*net*>

  All asynchronous reset nets in the design should be blocked.

- Multicycle

  The Multicycle preference allows you to relax a frequency or period constraint on selected paths.

For more information about individual preferences, refer to the “Setting Preferences” section of the ispLEVER FPGA and Crossover Design online Help.

Proper Preferences

Providing proper preferences is key to a successful design. Use the following recommendations to avoid a design that is either overconstrained or underconstrained.

Overconstraining

If the constraints of a preference file are tighter than the system requirements, the design will become overconstrained, so PAR run times will be considerably longer. In addition, overconstraining non-critical paths forces PAR to waste unnecessary processing power in trying to meet these constraints, creating possible conflicts with real critical paths that ought to be optimized first.

Common causes of overconstrained timing preferences include the following:

- Unspecified multi-cycle paths
- Multi-cycle paths to or from I/Os with different specifications

Overconstrained designs also need a significantly larger amount of processing power and computing resources. As a result, it might be necessary to increase some of the allocated system resources, such as the PC virtual memory paging size.

Underconstraining

If a preference file is underconstrained compared to real system requirements, real timing issues not previously seen during dynamic timing simulations and static timing analysis could appear. These potential problems can be observed on a test board or during production.

Common causes of underconstrained timing preferences include:

- Undefined I/O specifications
- Asynchronous logic without MAXDELAY preferences
- Internally generated or unintentional clocks not specified in the preference file
- Critical paths blocked

To make sure that no critical paths were left out because of underconstraining, you should check for path coverage at the end of a TRACE report (.twr) file.

An example of such an output is shown in Figure 108.

Figure 108: TRACE Report (.twr) Timing Summary Example

```
Timing summary:
---------------
Timing errors: 4906    Score: 25326584
Constraints cover 36575 paths, 6 nets, and 8635 connections
(99.0% coverage)
```

This particular example shows a 99.0 percent coverage. The way to find unconstrained paths is to run TRACE with the “Check Unconstrained Paths” option selected. This option gives a list of all of the signals that are not covered under timing analysis. In some designs, many of these signals are a common ground net that indeed does not need to be constrained. You should understand this and use TRACE to check unconstrained paths and ensure that no timing-critical design paths are being missed.

Also, note the timing score shown in Figure 108. The timing score shows the total amount of error, in picoseconds, for all timing preferences constraining the design. PAR attempts to minimize the timing score; PAR does not attempt to maximize frequency.

The above explanation can be summarized by the following:

Quality of Preference File = Quality of PAR Results

For more helpful information about timing analysis and TRACE, see “Performing Static Timing Analysis” on page 152.

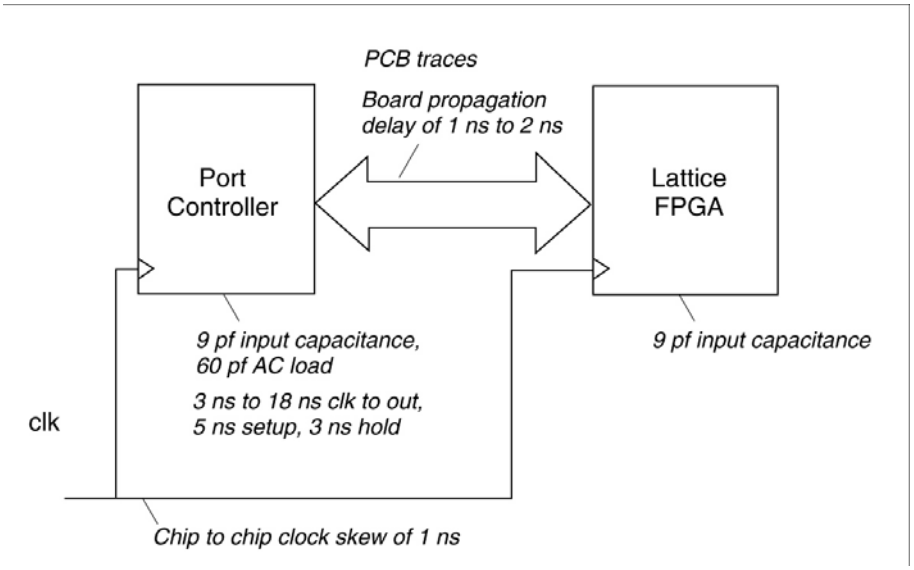
Translating Board Requirements into FPGA Preferences

Understanding the system-board-level timing and design constraints is the primary requirement for producing a complete preference file. As a result, major requirements, such as clock frequency, I/O timing, and loads, can be translated into the appropriate preference statements in a constraint file.

Following is an example showing how to extract preferences from system conditions.

Figure 109 shows an example system involving the interface between a port controller and a Lattice FPGA.

Figure 109: Interface Timing Example



In this system, several parameters have already been provided:

- System clock frequency: period (P): 30 ns
- Port controller maximum output propagation delay (PDMAXp): 18 ns
- Port controller minimum output propagation delay (PDMINp): 3 ns
- Port controller input setup specification (TSp): 5 ns
- Port controller input hold specification (THp): 3 ns
- Maximum board propagation delay (PDMAXb): 6 ns
- Minimum board propagation delay (PDMINb): 1 ns
- Clock skew of the port controller to the FPGA device and vice versa (Tskew): 1 ns
- Board trace AC loading (Cbac): 60 pF
- Board trace parasitic capacitance (Cb): 5 pF
- Port controller input capacitance (Cp): 9 pF
- FPGA device input capacitance (Co): 9 pF

The information just given was specified under the following environmental conditions:

- Maximum ambient temperature (Ta): 70 (C)
- Estimated power consumption (Q): 2 W
- 680 PBGAM package thermal resistance (Φ j) at 0 feet per minute (fpm) airflow: 13.4 °C/W

The goal of this exercise is to compute the following device I/O constraints:

- Input setup specification
- Input hold specification
- Maximum output propagation delay
- Minimum output propagation delay
- Output loading
- Temperature

The only parameter that can be obtained from this information is the device junction temperature:

Tj = Φj * Q - Ta
= 13.4 * 2 + 70
= 96.8 ºC

The required constraints can be computed as follows:

Input setup specification = P - PDMAXp - PDMAXb - Tskew
= 30 - 18 - 2 - 1
= 9 ns

Input hold specification = PDMINp + PDMINb - Tskew
= 3 + 1 - 1
= 3 ns

Output maximum propagation delay requirement
= P - TSp - PDMAXb - Tskew
= 30 - 5 - 6 - 1
= 18 ns

Output minimum propagation delay requirement
= Thp - PDMINb + Tskew
= 3 - 1 + 1
= 3 ns

Output loading = Cbac + Cb + Cp
= 60 + 5 + 9
= 74 pf

The preference file to use for this example is shown in Figure 110.

Figure 110: Interface Timing Preference File Example

```
PERIOD PORT "clk" 30 NS ;
INPUT_SETUP "port_controller*" 9 NS HOLD 3 NS CLKNET "clk";
CLOCK_TO_OUT "port_controller*" 18 NS MIN 3 NS CLKNET "clk";
OUTPUT PORT "port_controller*" LOAD 74 PF ;
TEMPERATURE 96.8 C ;
```

For more preference language syntax and examples, refer to the “Setting Preferences” section of the ispLEVER FPGA and Crossover Design online Help.

Using the Place and Route Software (PAR)

After a design has undergone the necessary translation to bring it into the mapped physical design (.ncd file) format, it is ready for placement and routing. This phase is handled by ispLEVER’s timing-driven PAR software program. You can invoke PAR from the Project Navigator or from the command line.

PAR performs the following tasks:

- Takes a mapped physical design (.ncd file) and a preference file (.prf) as input files.
- Places and routes the design, attempting to meet the location and timing preferences in the input .prf file.
- Creates a file that can be processed by the ispLEVER design implementation tools.

Placement

The PAR process places the mapped physical design (.ncd file) in two stages: a constructive placement and an optimizing placement. PAR writes the physical design after each of these stages is complete.

During constructive placement, PAR places components into sites on the basisd of factors such as the following:

- Constraints specified in the input file. For example, certain components must be in certain locations.
- The length of connections
- The available routing resources
- Cost tables that assign random weighted values to each of the relevant factors. There are 100 possible cost tables.

Constructive placement continues until all components are placed.

Optimizing placement is a fine-tuning of the results of the constructive placement.

Routing

Routing is also done in two stages: iterative routing and delay reduction routing (also called cleanup). PAR writes the physical design (.ncd file) only after iterations where the routing score has improved.

During iterative routing, the router attempts to converge on a solution that routes the design to completion or minimizes the number of unrouted nets.

During cleanup routing (also called delay reduction), the router takes the results of iterative routing and reroutes some connections to minimize the signal delays within the device. There are two types of cleanup routing that you can perform:

- A faster cost-based cleanup routing, which makes routing decisions by assigning weighted values to the factors (such as the type of routing resources used) that affect delay times between sources and loads.
- A more CPU-intensive, delay-based cleanup routing, which makes routing decisions on the basis of computed delay times between sources and loads on the routed nets.

If PAR finds timing preferences in the preference file, timing-driven placement and routing is automatically invoked.

Timing-Driven PAR Process

The ispLEVER software offers timing-driven placement and routing through the integrated static timing analysis utility (that is, it does not depend on input stimulus to the circuit). Placement and routing is executed according to the timing preferences that you specify up front in the design process. PAR attempts to meet these timing constraints without exceeding the timing constraints.

To use timing-driven PAR, you simply write timing preferences into the logical preference (.lpf) file and map the design. The mapping process then writes these preferences to the physical preference file (.prf), which serves as input to the integrated static timing analysis utility.

See the “Setting Preferences” section of the ispLEVER FPGA and Crossover Design online Help for more information about the PAR software and ispLEVER design flow.

Performing Static Timing Analysis

Static timing analysis (STA) is a fast and powerful verification technique that you can rely on to validate design performance. It is one of the most important steps in the design flow and should be considered as important as the functional verification performed with a logic simulator. Static timing analysis tools verify circuit timing by totaling the propagation delays along paths between clocked or combinational elements in a circuit. The analysis can determine and report timing data, such as the critical path, setup- and hold-time requirements, and the maximum frequency.

Static timing analysis tools enable you to:

- Confirm that the timing constraints supplied to timing-driven place and route will be met
- Examine the timing characteristics of any part of the design
- Perform what-if scenarios with different device speed grades or timing objectives

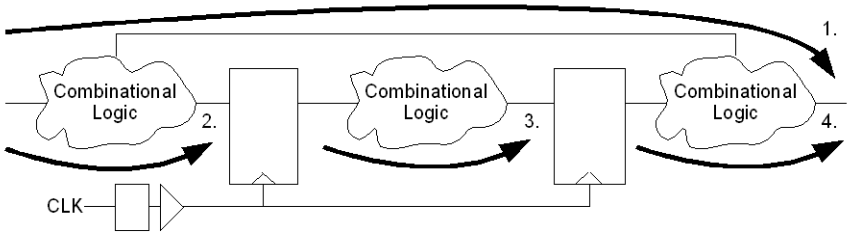
Why Perform Static Timing Analysis?

With a traditional dynamic logic simulator, timing violations must be detected and reported by sensitizing a simulation model, using test vectors and assertions that you must write. Depending on the size of the design and the number of states that it represents, the simulation run time can be very long and require a very sophisticated test fixture to detect all potential problems. The static timing analysis approach is far faster compared to dynamic simulation and verifies all parts of the gate-level design for timing.

Static timing analysis provides the following types of analysis:

- From primary input to primary output (t_{PD})
- From input to register
- From register to register
- From register to output (t_{CO})

Figure 111: Types of Static Timing Analysis



The following samples illustrate each analysis, using the ispLEVER preference language:

```
# STA analysis samples:
1. MAXDELAY FROM PORT "comb_in*" TO PORT "comb_out*" 16 ns;
2. INPUT_SETUP "comb_in*" 8 ns CLKPORT="clk";
3. PERIOD PORT "clk" 100 ns ;
4. CLOCK_TO_OUT ALLPORTS 8 ns CLKPORT "clk" ;
```

Note

For details on the syntax and semantics of the preference language, see “Creating and Editing Preferences” on page 140.

Timing-driven placement and routing (PAR) interprets these constraints as timing objectives, so you can selectively use preferences for PAR or static timing analysis, depending on your verification approach. For example, it is common for additional preferences to be added post-PAR exclusively for the sake of static timing analysis.

Analyzing Timing Reports Produced by TRACE

You can run the ispLEVER Timing Reporter and Circuit Evaluator (TRACE) on mapped designs, on completely placed and routed designs, or on designs that are placed, routed, or both to any degree of completion. The report issued by TRACE depends on the completeness of the placement and routing of the input design.

You can run TRACE automatically from the ispLEVER Project Navigator GUI or from the command line using the trce program.

To run TRACE from the Project Navigator:

1. In the Project Navigator, select the targeted device.
2. Double-click either the **Map TRACE Report (.tw1)** process or the **Place & Route TRACE Report (.twr)** process.

   The software runs TRACE and generates a report based on the mapped or placed and routed design.

To modify TRACE options from the Project Navigator:

- From the Project Navigator, choose **Tools > TRACE Options** to open the dialog box.

  The Before Route options apply to the Map TRACE Report (.tw1) process. The After Route options apply to the Place & Route TRACE Report (.twr) process.

Note

By default, TRACE performs analysis for setup-time violations, using worst-case operating conditions and the speed grade specified for the target device. To perform hold-time analysis using best-case operating conditions, enable **Check Hold Times** and select **m** (minimum) from the Overrride Speed Grade list.

To run TRACE from the command line:

- Type **`trce`** on the command line with, at minimum, the names of your input .ncd and .prf files, for example:

  ```
  trce design.ncd design.prf
  ```

  For more information about TRACE command-line options, see “Running TRACE from the Command Line” in the ispLEVER FPGA and Crossover Design online Help.

Timing Exceptions

Timing exceptions are exceptions to preferences that are used to describe the special behavior of certain design paths. Most designs contain paths that require these additional preferences. Without timing exceptions, the static timing analysis performed by TRACE will likely assume worst-case timing scenarios and report lower design performance, and the place-and-route program, PAR, will spend an undue amount of effort optimizing the path.

There are two common path types that require timing exceptions: multi-cycle paths and false paths.

Multi-Cycle Paths

In most synchronous circuits, the receiving register captures data launched by a driving register within one clock cycle. A multi-cycle path refers to cases where this relationship is different. Since single-cycle behavior is assumed by PAR and static timing analysis, a multi-cycle type of preference is used to express the relationship. The amount of time taken by the data to reach a destination register is indicated by a multiplier value, as shown in this example:

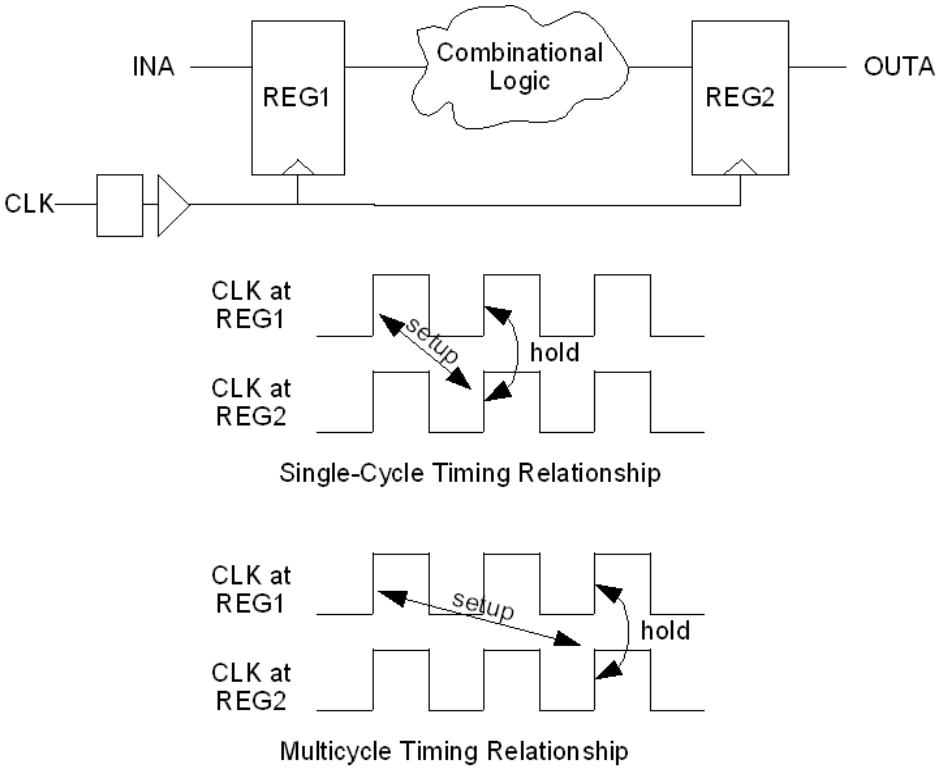
```
# 2 X multicycle sample:
FREQUENCY NET "CLK" 66 MHZ ;
MULTICYCLE FROM CELL "REG1" TO CELL "REG2" 2 X ;
```

Note

The ispLEVER preference language allows you to portray most clock relationships that can be practically analyzed using STA algorithms. However, if there are asynchronous clock domain crossings in your design, STA can only report on those occurrences where the clock edges are coincident. In these scenarios, your verification strategy might need to rely on dynamic verification, using a traditional simulator or formal verification methods.

Figure 112 illustrates a single-cycle versus a multi-cycle relationship. In the multi-cycle definition, a multiplier value of “2 X” is used to inform TRACE (and PAR) that the data latching occurs at REG2 after an additional clock pulse.

Figure 112: Single Versus Multi-Cycle Clock Relationship



Example 1 – Multi-Cycle Between Two Different Clocks

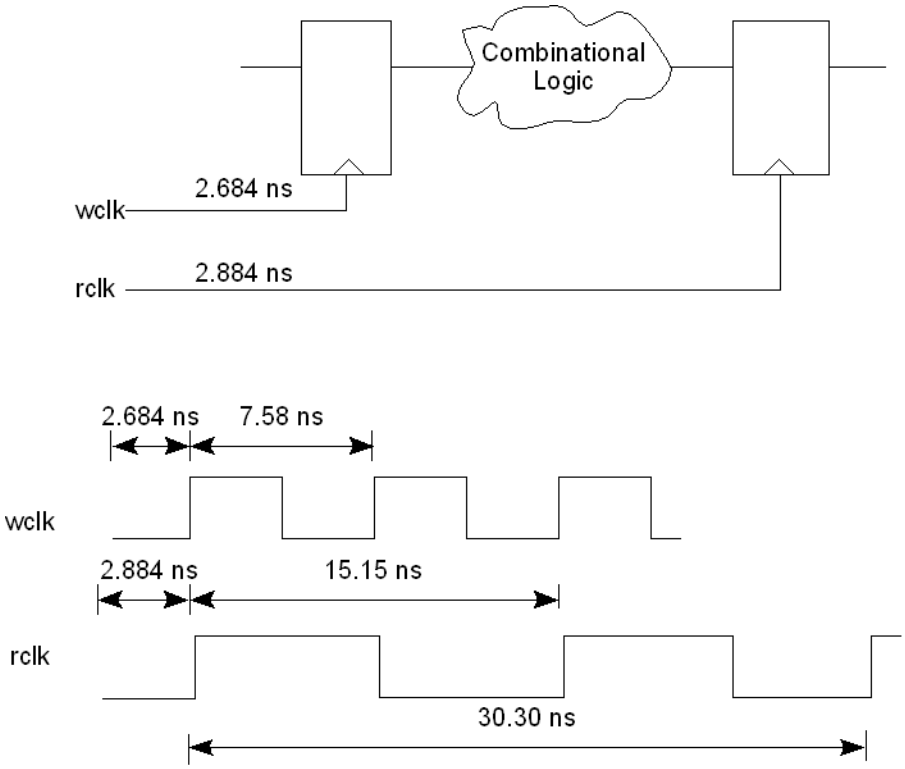
You can also use the Multicycle preference to describe a design that uses separate clocks with different frequencies. This relationship is often referred to as frequency skew. In this design example for a LatticeECP/EC FPGA, wclk and rclk were assigned 132 MHz and 66 MHz frequencies, respectively.

For this example, a variation of the Multicycle constraint is used to describe the relationship between the clocks in terms of a period instead of a frequency multiplier; you can choose either form for the sake of clarity and documentation:

```
FREQUENCY NET "wclk" 132 MHZ ;
FREQUENCY NET "rclk" 66 MHZ ;
MULTICYCLE FROM CLKNET "wclk" TO CLKNET "rclk" 30.30 NS ;
```

The block diagram and waveform for this example is shown in Figure 113.

Figure 113: Multi-Cycle Clock Domains Block Diagram and Waveform



The resulting TRACE report for this example is shown in Figure 114. Notice how the path is described in terms of “Logical Details.”

Figure 114: TRACE Report for Multi-Cycle Clock Domains Example

```
 ================================================================================

Preference: MULTICYCLE FROM CLKNET "wclk" TO CLKNET "rclk" 30.300000 ns ;
            13 items scored, 0 timing errors detected.
--------------------------------------------------------------------------------
Passed:  The following path meets requirements by 23.540ns

 Logical Details:  Cell type  Pin type       Cell name  (clock net +/-)

   Source:         PDP8KA     Port           fifomem_RAM_DP_16X8_RAM_DP_16X8_0_0_0
(from wclk +)
   Destination:    FF         Unknown        ioregout_reg_outdata(4)  (to rclk +)

   Delay:               7.202ns  (63.2% logic, 36.8% route), 1 logic levels.

 Constraint Details:

       7.202ns physical path delay fifomem_RAM_DP_16X8_RAM_DP_16X8_0_0_0 to
rdata_p(4)_MGIOL meets
      30.300ns delay constraint less
      -0.200ns skew and
       0.000ns feedback compensation and
      -0.242ns ONEG0_SET requirement (totaling 30.742ns) by 23.540ns

 Physical Path Details:

   Name    Fanout   Delay (ns)          Site               Resource
C2Q_DEL     ---     4.554  EBR_R6C8.CLKR to   EBR_R6C8.DO4
fifomem_RAM_DP_16X8_RAM_DP_16X8_0_0_0 (from wclk)
ROUTE         1     2.648   EBR_R6C8.DO4 to  IOL_T6A.ONEG0 rdata(4) (to rclk)
                  --------
                    7.202   (63.2% logic, 36.8% route), 1 logic levels.
Clock Skew Details:

 Source Clock Path:

   Name    Fanout   Delay (ns)          Site               Resource
PADI_DEL    ---     0.667         19.PAD to       19.PADDI clk_p
ROUTE         1     0.000       19.PADDI to PLL3_R6C1.CLKI clk_p_int
CLK2OUT_DE  ---    -0.500 PLL3_R6C1.CLKI to LL3_R6C1.CLKOP fifo1_pll_fifo1_pll_0_0
ROUTE        24     2.517 LL3_R6C1.CLKOP to  EBR_R6C8.CLKR wclk
                  --------
                    2.684   (6.2% logic, 93.8% route), 2 logic levels.

PLL3_R6C1.CLKOP attributes: FDEL = -2

 Destination Clock Path:

   Name    Fanout   Delay (ns)          Site               Resource
PADI_DEL    ---     0.667         19.PAD to       19.PADDI clk_p
ROUTE         1     0.000       19.PADDI to PLL3_R6C1.CLKI clk_p_int
CLK2SEC_DE  ---     0.066 PLL3_R6C1.CLKI to LL3_R6C1.CLKOK fifo1_pll_fifo1_pll_0_0
ROUTE        19     2.151 LL3_R6C1.CLKOK to    IOL_T6A.CLK rclk
                  --------
                    2.884   (25.4% logic, 74.6% route), 2 logic levels.
```

```
Source Clock f/b:

   Name    Fanout   Delay (ns)          Site               Resource
CLKOP_DEL   ---     0.000 LL3_R6C1.CLKFB to LL3_R6C1.CLKOP fifo1_pll_fifo1_pll_0_0
ROUTE        24     2.151 LL3_R6C1.CLKOP to LL3_R6C1.CLKFB wclk
                   --------
                     2.151   (0.0% logic, 100.0% route), 1 logic levels.

PLL3_R6C1.CLKOP attributes: FDEL = -2

 Destination Clock f/b:

   Name    Fanout   Delay (ns)          Site               Resource
CLKOP_DEL   ---     0.000 LL3_R6C1.CLKFB to LL3_R6C1.CLKOP fifo1_pll_fifo1_pll_0_0
ROUTE        24     2.151 LL3_R6C1.CLKOP to LL3_R6C1.CLKFB wclk
                   --------
                     2.151   (0.0% logic, 100.0% route), 1 logic levels.

PLL3_R6C1.CLKOP attributes: FDEL = -2
```

This section shows both the source and destination registers using unmapped names from the EDIF file. This is a feature that allows you to recognize the type of logic being analyzed.

On the basis of the declared frequencies for both clocks, you already know the following:

- wclk period = 9.6 ns.
- rclk period = 15.15. ns.
- No relative phase information exists between both clocks. As a result, TRACE does not factor in the skews on either clock. To add relative timing between two clocks, use the CLKSKEWDIFF preference:

  ```
  CLKSKEWDIFF CLKPORT "rclk" CLKPORT "wclk" 0.2 NS;
  ```

  This preference would mean that the clock arrives at rclk with a 0.2 ns delay after wrclk.

As a consequence, ignoring everything else (clock skews, register library setups, and so forth), you know that a single-cycle positive-edge-to-positive-edge setup available from wrclk to rclk is 15.15 ns (refer to the waveforms in Figure 113). Therefore, with 2X multi-cycle, the resulting setup would be twice that number, or:

Ts = 30.30 ns

This is shown as a delay constraint in the Constraint Details section of the TRACE report. The notation used in the site details refers to the slice row/column location in the device floorplan and the slice signal names.

The available setup margin (23.542 ns) is now calculated as follows:

M = (Ts - Td) – Tclkskew - Ds

where:

- Td = path delay from clock pin of source embedded block ram (EBR) to D pin of destination = 7.202 ns. This is shown in the Physical Path Details section of the TRACE report.
- Tclkskew = TSB – TSA, where TSA is the delay on the source clock, and TSB is the delay on the destination clock.
- Ds = destination cell library setup requirement = 0.242 ns. This matches ONEGO_SET (Output, Negative 0, Input Setup) under the Constraint Details section of the TRACE report.

The clock skews are:

- TSA = delay or skew on source clock wclk = 2.684 ns. It is shown in the Clock Skew Details section of the TRACE report.
- TSB = delay or skew on destination clock rclk = 2.884 ns. It is shown in the Clock Skew Details section of the TRACE report.

Therefore:

M = (30.30 – 7.202) – (-0.200) - (-0.242) = 23.542 ns. M matches the number in the PASSED section at the top of the TRACE report.

Example 2 – Clock-to-Output with PLL Feedback

In this example for a LatticeECP/EC FPGA, clk_p is assigned to 66 MHZ, and the clock-to-out propagation delays are constrained in the preference file:

```
CLOCK_TO_OUT ALLPORTS 7.000000 ns CLKPORT "clk_p" ;
```

The block diagram for this example is shown in Figure 115. The resulting TRACE report is shown in Figure 116.

Figure 115: CLOCK_TO_OUT with PLL

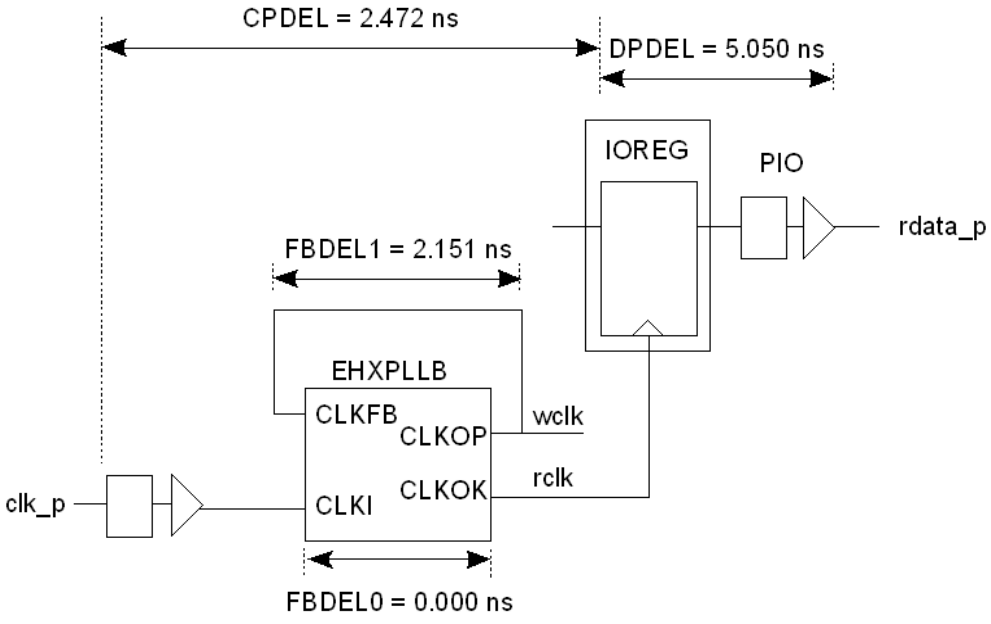


Figure 116: TRACE Report for CLOCK_TO_OUT with PLL

```
================================================================================
Preference: CLOCK_TO_OUT ALLPORTS 7.000000 ns CLKPORT "clk_p" ;
            10 items scored, 0 timing errors detected.
--------------------------------------------------------------------------------


Passed:  The following path meets requirements by 1.629ns

 Logical Details:  Cell type  Pin type       Cell name  (clock net +/-)

   Source:         FF         Q              wptr_full_reg_wfull  (from wclk +)
   Destination:    Port       Pad            wfull

   Data Path Delay:     5.050ns  (54.4% logic, 45.6% route), 2 logic levels.

   Clock Path Delay:    2.472ns  (6.8% logic, 93.2% route), 2 logic levels.

 Constraint Details:

      2.472ns delay clk_p to SLICE_21 less
      2.151ns feedback compensation
      5.050ns delay SLICE_21 to wfull (totaling 5.371ns) meets
      7.000ns offset clk_p to wfull by 1.629ns

 Physical Path Details:

      Clock path clk_p to SLICE_21:

   Name    Fanout   Delay (ns)          Site               Resource
PADI_DEL    ---     0.667         19.PAD to       19.PADDI clk_p
ROUTE         1     0.000       19.PADDI to PLL3_R6C1.CLKI clk_p_int
CLK2OUT_DE  ---    -0.500 PLL3_R6C1.CLKI to LL3_R6C1.CLKOP fifo1_pll_fifo1_pll_0_0
ROUTE        24     2.305 LL3_R6C1.CLKOP to     R5C10B.CLK wclk
                  --------
                    2.472   (6.8% logic, 93.2% route), 2 logic levels.

PLL3_R6C1.CLKOP attributes: FDEL = -2

      Data path SLICE_21 to wfull:

   Name    Fanout   Delay (ns)          Site               Resource
REG_DEL     ---     0.436     R5C10B.CLK to      R5C10B.Q0 SLICE_21 (from wclk)
ROUTE         6     2.304      R5C10B.Q0 to       77.PADDO wfull_dup0
DOPAD_DEL   ---     2.310       77.PADDO to         77.PAD wfull
                  --------
                    5.050   (54.4% logic, 45.6% route), 2 logic levels.

      Feedback path:

   Name    Fanout   Delay (ns)          Site               Resource
CLKOP_DEL   ---     0.000 LL3_R6C1.CLKFB to LL3_R6C1.CLKOP fifo1_pll_fifo1_pll_0_0
ROUTE        24     2.151 LL3_R6C1.CLKOP to LL3_R6C1.CLKFB wclk
                  --------
                    2.151   (0.0% logic, 100.0% route), 1 logic levels.

PLL3_R6C1.CLKOP attributes: FDEL = -2

Report:    5.371ns is the minimum offset for this preference.
```

The path measurements were obtained from the TRACE report as follows:

- CPDEL = Clock Path Delay = 2.472 ns. It is shown under Physical Path Details -> Clock path in the timing report.
- DPDEL = Data Path Delay = 5.050 ns. It is shown under Physical Path Details-> Data path in the timing report.
- FBDEL0 = Feedback cell delay across PLL = 0.000 ns, which is the first entry value under Feedback Path.
- FBDEL1 = Feedback routing delay from PLL output to PLL FB pin = 2.151 ns, which is the second entry value under Feedback Path.

Notice the -0.500 ns CLK2OUT_DE delay under “Physical Path Details” of the report file. This delay (or in this case compensation) within the clock path is produced by a sysCLOCK PLL fine delay adjust step value (FDEL) of –2. Such a delay is a programmable attribute (FDEL) of the sysCLOCK PLL. This value is can be set to any step value from –8 to +8 to advance or retard the output clock in 250-picosecond steps. FDEL is assigned in the Design Planner’s Spreadsheet View or in the EPIC Device Editor.

To verify the available margin on the CLOCK_TO_OUT preference, the margin is reported as follows:

M = CLOCK_TO_OUT - (CPDEL - FBDEL + DPDEL)

= 7.000 - (2.472 – 2.151 + 5.050) = 1.629 ns

This value matches the one at the top of the report file (“Passed” section). It also matches the final value under “Constraint Details.”

Example 3 – Hold-Time Analysis

In this example for a LatticeECP/EC FPGA, the clock-to-output constraint is examined in terms of hold time. The TRACE report for this example is shown in Figure 117.

Figure 117: CLOCK_TO_OUT Hold-Time Check

```
================================================================================
Preference: CLOCK_TO_OUT ALLPORTS 7.000000 ns CLKPORT "clk_p" ;
            10 items scored, 0 timing errors detected.
--------------------------------------------------------------------------------
Passed:  The following path meets requirements by 1.412ns

 Logical Details:  Cell type  Pin type       Cell name  (clock net +/-)

   Source:         FF         Q              ioregout_reg_outdata(1)  (from rclk +)
   Destination:    Port       Pad            rdata_p(1)

   Data Path Delay:     1.404ns  (100.0% logic, 0.0% route), 2 logic levels.

   Clock Path Delay:    0.904ns  (24.8% logic, 75.2% route), 2 logic levels.

 Constraint Details:

      0.904ns delay clk_p to rdata_p(1)_MGIOL less
      0.896ns feedback compensation
      1.404ns delay rdata_p(1)_MGIOL to rdata_p(1) (totaling 1.412ns) meets
      0.000ns hold offset clk_p to rdata_p(1) by 1.412ns

 Physical Path Details:

      Clock path clk_p to rdata_p(1)_MGIOL:

   Name    Fanout   Delay (ns)          Site               Resource
PADI_DEL    ---     0.200         19.PAD to       19.PADDI clk_p
ROUTE         1     0.000       19.PADDI to PLL3_R6C1.CLKI clk_p_int
CLK2SEC_DE  ---     0.024 PLL3_R6C1.CLKI to LL3_R6C1.CLKOK fifo1_pll_fifo1_pll_0_0
ROUTE        19     0.680 LL3_R6C1.CLKOK to    IOL_T2B.CLK rclk
                  --------
                    0.904   (24.8% logic, 75.2% route), 2 logic levels.

      Data path rdata_p(1)_MGIOL to rdata_p(1):

   Name    Fanout   Delay (ns)          Site               Resource
C2OUT_DEL   ---     0.644    IOL_T2B.CLK to  IOL_T2B.IOLDO rdata_p(1)_MGIOL (from
rclk)
ROUTE         1     0.000  IOL_T2B.IOLDO to       98.IOLDO nx27852z1
DOPAD_DEL   ---     0.760       98.IOLDO to         98.PAD rdata_p(1)
                  --------
                    1.404   (100.0% logic, 0.0% route), 2 logic levels.

      Feedback path:

   Name    Fanout   Delay (ns)          Site               Resource
CLKOP_DEL   ---     0.000 LL3_R6C1.CLKFB to LL3_R6C1.CLKOP fifo1_pll_fifo1_pll_0_0
ROUTE        24     0.896 LL3_R6C1.CLKOP to LL3_R6C1.CLKFB wclk
                  --------
                    0.896   (0.0% logic, 100.0% route), 1 logic levels.

PLL3_R6C1.CLKOP attributes: FDEL = -2

Report:    1.412ns is the maximum offset for this preference.
```

TRACE produces a hold-time check based on your timing preferences by using the –hld command-line option or by enabling the Check Hold Times option in the TRACE Options dialog box of the Project Navigator.

By default, TRACE analyzes designs for setup-time violations, using the worst-case operating conditions for the target speed grade. Speed grade (-sp) is a TRACE command-line option that you can adjust in the TRACE Options dialog box. In contrast to setup-time analysis, hold-time analysis should be done under best-case operating conditions. This approach of analyzing at both corners of the operating conditions establishes a well-defined range in which the device will operate successfully.

To specify the best-case operating condition for hold-time checks, use the –sp m (minimum) option from the TRACE command line or use the TRACE Options dialog box. In most cases, but not all, the minimum option represents the worst-case scenario for hold-time analysis. The most rigorous STA methodology would have you run TRACE against all speed grades.

The path related to rdata_p(1) is reported by TRACE as worst-case when a hold-time analysis is performed. The path measurements were obtained from the TRACE report as follows:

- DPDEL = Data Path Delay = 1.404 ns. It is shown under Physical Path Details-> Data path in the timing report.
- CPDEL = Clock Path Delay = 0.904 ns. It is shown under Physical Path Details-> Clock path in the timing report.
- FBDEL0 = Feedback cell delay across PLL = 0.000 ns, which is the first entry value under Feedback Path.
- FBDEL1 = Feedback routing delay from PLL output to PLL FB pin = 0.896 ns, which is the second entry value under Feedback Path.

Now verify the available hold-time margin on the CLOCK_TO_OUT preference. The margin is reported as:

M = CPDEL – FBDEL + DPDEL

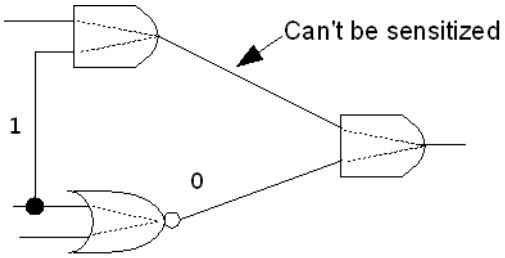
= 0.904 – 0.896 + 1.404 = 1.412 ns

This value matches the one at the top of the report file (“Passed” section). It also matches the final value under “Constraint Details.”

False Paths

Many designs include paths that are asynchronous relative to the clocks of the design or connections that never propagate a signal state because of logic encoding. A false path illustration is shown in Figure 118. By default, STA is performed on all paths of the design so that timing reports include all path segments, including false ones. This condition can “mask” the violations of real timing paths and make the performance results overly pessimistic.

Figure 118: False Path Example



False paths are treated as unconstrained by TRACE and timing-driven PAR. If you can accurately describe false paths, design performance will usually be improved since a false path is treated by PAR as unconstrained. With “relaxed” timing objectives, PAR optimizes the true critical paths instead. In a similar manner, unconstrained paths are ignored by STA and true critical paths reported instead.

The ispLEVER Block preference allows you to identify false paths to the system and provides a variety of ways to isolate them.

Controlling Placement and Routing

Extensive benchmark experiments have been performed to determine the optimum per-device default settings for all PAR options. At times, you can obtain improved timing results on a design-by-design basis by trying different variations of the PAR options. This section describes the techniques that you can use within the ispLEVER software to improve timing results from TRACE on placed and routed designs.

Running Multiple Routing Passes

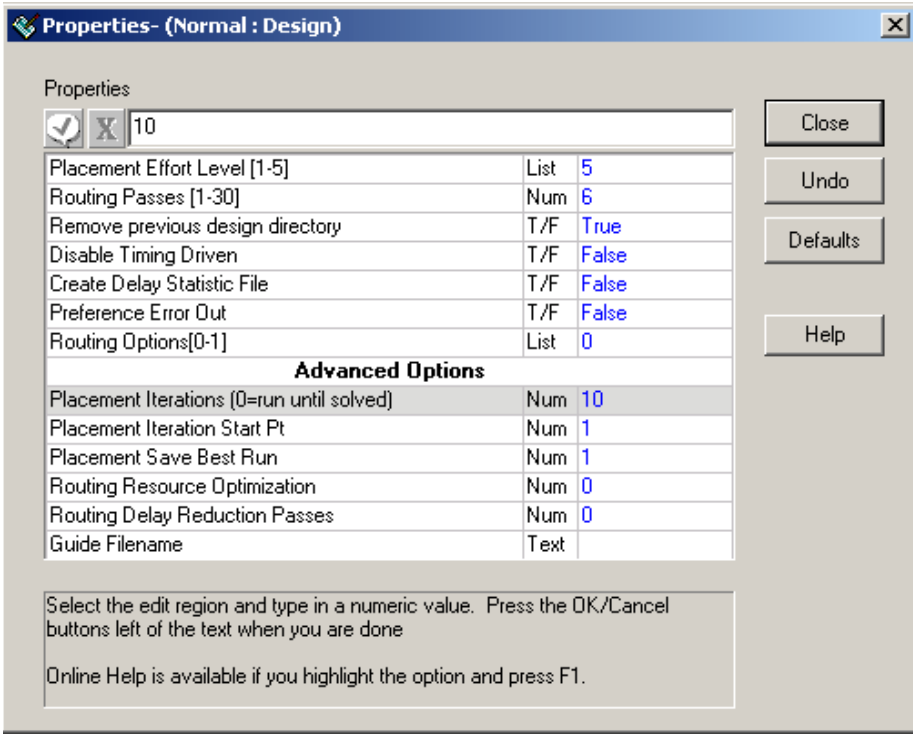
You can obtain improved timing results by increasing the number of routing passes during the routing phase of PAR.

To open the PAR options dialog box:

1. In the Project Navigator Source window, select the targeted FPGA device.
2. In the Processes window, right-click the **Place & Route Design** process and select **Properties** to open the dialog box.

As seen in Figure 119, the router routes the design for six routing iterations or until all the timing preferences are met, whichever comes first. For example, PAR stops after the second routing iteration if it hits a timing score of zero on the second routing iteration.

Figure 119: PAR Properties Dialog Box



The highest selection in placement effort level results in longer PAR run times but may give better design timing results. A lower placement effort results in shorter PAR run times but will likely give less than optimal design timing results.

The place and route (.par) report file contains execution information about the PAR command run, as shown in Figure 120.

Figure 120: Example PAR Report (.par) File, Routing Section

```
0 connections routed; 26590 unrouted.
Starting router resource preassignment
Completed router resource preassignment. Real time: 11 mins 31 secs
Starting iterative routing.
End of iteration 1
26590 successful; 0 unrouted; (151840) real time: 14 mins 29 secs
Dumping design to file
d:\ip\design.ncd.
End of iteration 2
26590 successful; 0 unrouted; (577) real time: 16 mins 23 secs
Dumping design to file
d:\ip\design.ncd.
End of iteration 3
26590 successful; 0 unrouted; (0) real time: 17 mins 39 secs
Dumping design to file
```

The PAR report also shows the steps taken as the program converges on a placement and routing solution. In the routing convergence example text in Figure 120, the number in parenthesis is the timing score after each iteration. In this example, timing was met after three routing iterations, as you can see from the (0) timing score.

Using Multiple Placement Iterations (Cost Tables)

You can specify multiple placement iterations in the Advanced Options of the PAR Properties dialog box.

As shown in Figure 119, the number of iterations is set to 10 and the placement start point is set to iteration 1 (cost table 1). Only the best .ncd file is to be saved, as seen in the following line. After PAR runs, the tool loops back through the PAR flow until the number of iterations has reached 10. In this example, the .ncd file with the best timing score is saved.

The tool keeps track of the timing and routing performance for every iteration in a file called the multiple par report (.par). Such a file is shown in Figure 121.

Figure 121: Multiple PAR Report (.par)

```
Level/       Number     Timing  Run      NCD
Cost [ncd]   Unrouted   Score   Time     Status
----------   --------   ------- -----    --------

5_4          0          0       01:58    Complete
5_6          0          25      02:01    Complete
5_2          0          102     01:45    Complete
5_7          0          158     02:15    Complete
5_3          0          186     01:54    Complete
5_10         0          318     02:39    Complete
5_1          0          470     01:51    Complete
5_8          0          562     02:25    Complete
5_5          0          732     02:00    Complete
5_9          0          844     02:27    Complete

*  :  Design saved.
```

Figure 121 indicates that:

- The “5_” under the Level/Cost column means that the placement effort level was set to 5. The placement effort level can range from 1 (lowest) to 5 (highest).
- 10 different iterations ran (10 cost tables).
- Timing scores are expressed in the total number of picoseconds (ps) by which the design is missing constraints on all preferences. This number is additive for all paths in the design.
- Iteration number 4 (cost table 4) achieved a 0 timing score, so it is the design that was saved. More than one .ncd file can be saved. You can control this by the “Placement Save Best Runs” value box shown in Figure 119.
- Each iteration routed completely.

Note

You should save more than one .ncd file from a multi-PAR run and use TRACE on each example. Since the timing score is a composite of all timing constraints, a low score might not be ideal for your application.

If “Placement Iterations (0=run until solved)” in Figure 119 is set to 0, the tool will run indefinitely through multiple iterations until a 0 timing score is reached. In a design that is known to have large timing violations, a 0 timing score is never reached. As a consequence, you must intervene and stop the flow at a given point in time.

In general, multiple placement iterations can help placement, but they can also use many CPU cycles. Multiple placement iterations should be used carefully because of system limitations and the uncertainty of results. It is better to fix the root cause of timing problems in the design stage.

Re-Entrant Routing

In the Project Navigator, a Reentrant Route Design process is provided that allows you to run more routing iterations to try to achieve timing goals. The Reentrant Route Design process saves time by using the current routed design file (.ncd) as the starting point.

The Reentrant Route Design process also enables you to override the current .ncd file with a routed .ncd file from one of your project's Revision Control directories:

1. In the Project Navigator, right-click **Reentrant Route Design** and select **Properties**.
2. Select **NCD File Name** in the Properties dialog box.
3. Type the complete path of the .ncd file that you want to use and click **Close**.

For further information on re-entrant flow, refer to the ispLEVER FPGA and Crossover Design online Help.

Clock Boosting

Clock boosting is the deliberate introduction of clock skew on a target flip-flop to increase the setup margin. Every programmable flip-flop in the device has programmable delay elements before clock inputs for this purpose. The automated clock-boosting tool attempts to meet setup constraints by introducing delays to as many target registers as needed to meet timing; in effect, it borrows register hold margins to meet register setup timing. Clock boosting is accomplished through the following features:

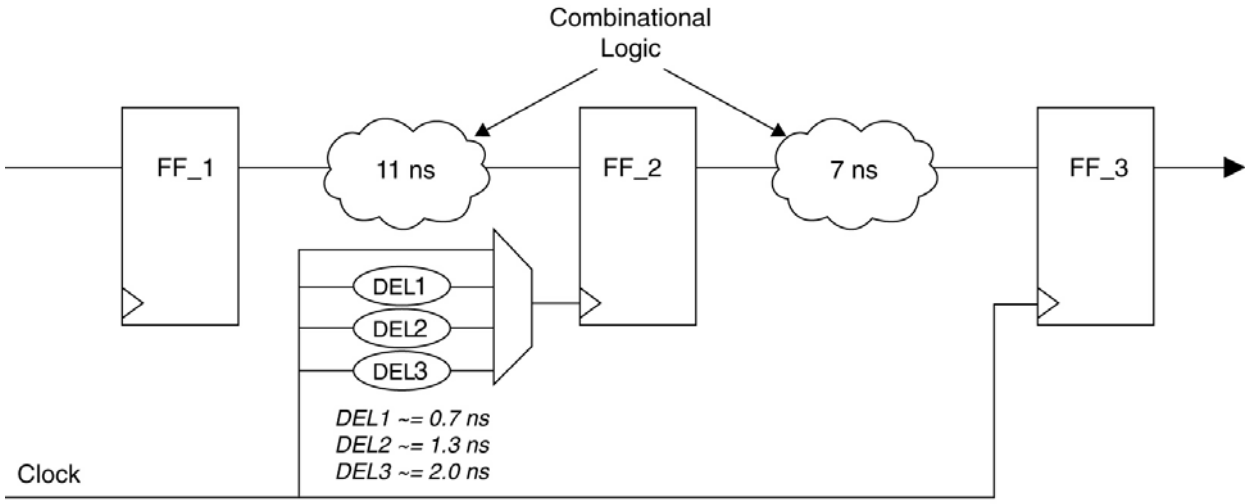
- A 4-tap delay cell structure in front of the clock port of every flip-flop in the device (includes I/O flip-flops)
- The ability to borrow clock cycle time from one easily met path and give this time to a difficult-to-meet path

Clock boosting is typically most useful in designs that are only missing timing on a few paths for one or two preferences. If the design is missing timing by over a few nanoseconds on any given path, clock boosting cannot schedule skew in a way that eliminates enough timing to make the critical preference. Clock boosting run times can be shortened by using a preference file that contains only the failing preferences.

The example illustrated in Figure 122 shows two register-to-register transfers that both need to meet the 10-ns period constraint. By using the DEL2 delay cell to delay the clock input on flip-flop FF_2, the first register transfer makes its period constraint with a new minimum period of approximately 9.7 ns, and the second register transfer makes its period constraint by approximately 8.3

ns. The D1, D2, and D3 delays shown vary, depending on the speed grade and FPGA device family.

Figure 122: Clock Boosting Example



Target Performance: 10 ns period (100 MHz)

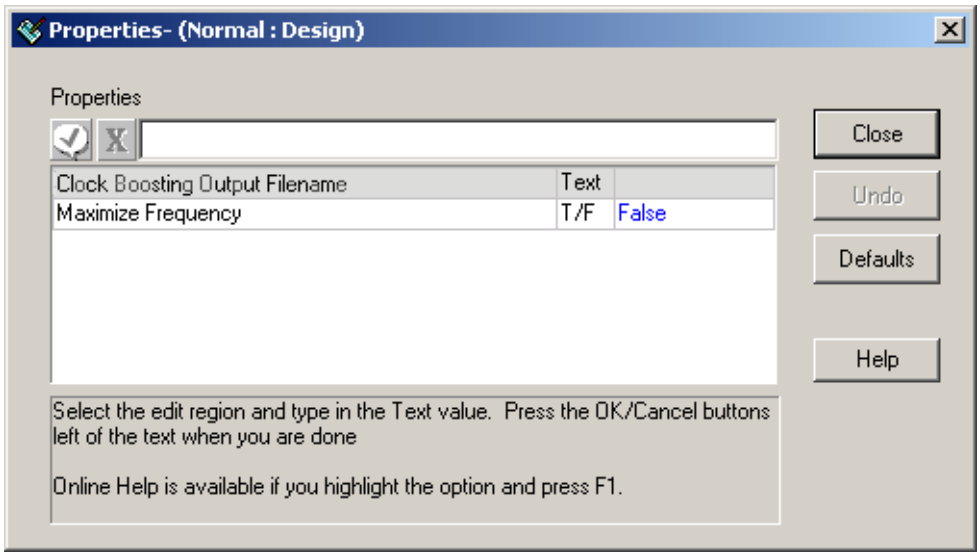
For complete timing information, refer to the timing data sheet included with ispLEVER for the desired Lattice Semiconductor FPGA device family

To perform clock boosting in the Project Navigator:

1. In the Project Navigator Sources window, select the targeted device.
2. In the Processes window, right-click **Clock Boosting** and select **Properties** from the pop-up menu.

3. In the dialog box, select the **Clock Boosting Output Filename** property from the property list, as shown in Figure 123, and type the name of the output file name in the edit region (<*file_name*>.ncd).

Figure 123: Clock Boosting Property Dialog Box



4. Click **Close** to exit the dialog box.

Selecting **Maximize Frequency** pushes the tool to improve the frequency beyond the input preference requirement. This feature is generally only useful for bench-marking.

Other important considerations on the practicality of using clock boosting:

- Some circuits show much improvement, but others show no gain. Clock boosting results are very design-dependent.
- Clock boosting uses minimum delay values that have not yet been validated at the system level.
- Automatic clock boosting identifies skew and hold-time issues. However, after clock boosting is performed, it is recommended that you run TRACE twice: once with regular maximum delay analysis and again with minimum delays. Afterwards, read over both resultant .twr timing reports to make sure that there are no timing errors. The minimum delay analysis is done by checking the “Check Hold Times” check box in the TRACE Options dialog box.

Map Register Retiming

Map register retiming is an optimization technique that moves registers across combinatorial logic to balance the timing according to the t_{SU} (INPUT_SETUP), t_{CO} (CLOCK_TO_OUT), and f_{MAX} (FREQUENCY) constraints.

To enable map register retiming in your design:

1. In the Project Navigator, select the targeted device.
2. In the Processes window, right-click **Map Design** and select **Properties** from the pop-up menu.
3. In the dialog box, under Advanced Options, select **Register Retiming** and change the setting to **True**.
4. Close the dialog box.

There is no guarantee that map register retiming will achieve a better f_{MAX}, since the f_{MAX} constraint activates retiming around all registers. The t_{SU} and t_{CO} constraints might deactivate retiming on I/O registers, depending on the balancing of t_{SU} vs. f_{MAX} and t_{CO} vs. f_{MAX}. However, register retiming can be very useful for optimization because it allows for more delay shifting.

Map Register Retiming vs. Clock Boosting

Map register retiming has the same goal as clock boosting, which adjusts the timing by introducing predefined clock delays. The following considerations should be taken into account when using either of these features for optimizing timing.

Optimizing with Map Register Retiming

Map register retiming can be either forward or backward. Forward retiming moves a set of registers that are the inputs of logic to a single register at its output. Backward retiming moves a register that is at the output of a logic to a set of registers at its input. Retiming works on a data path and has variable delay shift and variable area cost from design to design. A drawback to register retiming is that it changes your netlist, making debugging more difficult. It also has a minimum delay shift of one logic level (for example, one LUT).

Optimizing with Clock Boosting

Clock boosting works on clock paths and has a fixed delay, such as 0 ns, 1 ns, 2 ns, or 3 ns, and it has a fixed area cost (on silicon). The delay shift is accurate after placement and routing and can be as fine as less than or equal to 1 ns. However, clock boosting requires the use of extra silicon area even if it is not used, and delay shift is limited to a few choices up to about 3 ns or more.

For more information about map register retiming, refer to the “Mapping” section of the ispLEVER FPGA and Crossover Design online Help.

Floorplanning the Design

If performance goals cannot be met with FPGA timing preferences and additional effort levels of the Place & Route Design process, you can improve performance by directing the physical layout of the circuit in the FPGA. This step, often referred to as floorplanning, is done by specifying FPGA location preferences.

Floorplanning Definition

Floorplanning is the physical or logical partitioning of design elements, which results in a change in the design's physical placement or implementation.

With Lattice Semiconductor FPGAs, floorplanning is an optional methodology to help you improve the performance and density of a fully and automatically placed and routed design. Floorplanning is particularly useful in structured designs and data path logic. Design floorplanning is very powerful and provides a combination of automation and user control for design reuse and modular, hierarchical, and incremental design flows.

Complex FPGA Design Management

Lattice Semiconductor FPGAs can implement large system designs that contain millions of gates, hundreds of thousands of embedded memory bits, and intellectual property (IP) components. Design teams often work on large designs. The design complexity requires electronic design automation (EDA) tools to manage and optimize the design. Large design management is difficult, but performance optimization is even more difficult. Optimization requires many design iterations when adding or modifying components. Complex, large system designs require the following:

- The use of modular, hierarchical, or incremental design methods
- Software that makes management and optimization easier
- The use of IP blocks
- The reuse of previously optimized design elements

By controlling the placement of specified logic elements, design floorplanning methodologies help meet the requirements of large system design.

Floorplanning Design Flow

In both traditional and floorplanning FPGA design flows, you divide the system into modules. The modules can be individual circuits, parts of circuits, or parts of the design hierarchy. After module design and optimization, you integrate the modules into the system. Finally, you test and optimize the system.

In the traditional flow, the system might not meet performance requirements even if each module meets the requirements before integration. Even when timing requirements have been satisfied, changes to one module can affect the performance of others. Re-optimizing modules to meet system performance results in many design iterations.

Floorplanning methodologies assist in the design, testing, and optimization of each individual module while retaining the optimized characteristics of the individual modules. Module integration into the system requires only system optimization between modules. The floorplanning methodologies provide additional flexibility by allowing the ispLEVER software to automatically place defined modules, or by allowing you to control the placement of specific modules.

When to Floorplan

Floorplanning methodologies are intended for users who require some degree of handcrafting of their designs. You must understand both the details of the device architectures and the ways floorplanning can be used to refine a design. Successful floorplanning is very much an iterative process, and it can take time to develop a floorplan that outperforms an automatic software-processed design. Because of the nature of floorplanning and its interaction with the automatic MAP and PAR software tools, several prerequisites are necessary in order to floorplan a design successfully:

- Detailed knowledge of the specific characteristics of the target architecture and device
- Detailed knowledge of the specific characteristics of the design being implemented
- A design that lends itself to floorplanning
- A willingness to iterate a floorplan to achieve the desired results
- Realistic performance and density goals

For Lattice Semiconductor FPGAs, the general rule of thumb is that floorplanning should be considered when the desired performance cannot be met and when routing delays account for over 60 percent of the critical path delays. This can be a problem with large designs in high-density FPGAs because of the possibilities of long-distance routes. As programmable logic design densities continue to escalate beyond 100,000 gates, traditional design flow—design entry to synthesis to placement and routing—sometimes does not yield predictable, timely, and optimized results.

The guidelines previously discussed only apply to designs that have been routed by the software for several routing iterations. The default number of routing iterations through the ispLEVER Project Navigator vary, depending on the selected Lattice Semiconductor FPGA device family.

Note

Path delays in programmable devices are made up of two parts: logical delays and routing delays. Logical delays in this context are delays through components, such as a programmable function unit (PFU), a programmable input/output (PIO), a slice, or an embedded function, such as a block RAM, PLL, or FPSC ASIC. The routing delay is the delay of the interconnect between components. Figure 124 and Figure 125 show delay examples from timing wizard report files (.twr).

Properly applied, design floorplanning not only preserves but improves design performance. You can use floorplanning methodologies to place modules, entities, or any group of logic into regions in a device's floorplan. Because

floorplanning assignments can be hierarchical, you can have more control over the placement and performance of modules and groups of modules.

Floorplanning may be able to help bring the registers shown in Figure 124 closer.

Figure 124: Inefficient Routing

```
Logical Details:   Cell type    Pin type     Cell name (clock net +/-)

Source:            FF           Q            ibuf/reg_init_start (from clk_ib+)
Destination:       FF           Data in      ibuf/sd/reg_new_state (to clk_ib +)

Delay:                 8.062ns (18.2% logic, 81.8% route), 2 logic levels.
```

Floorplanning is not needed in Figure 125 because the routing is efficient.

Figure 125: Efficient Routing

```
Logical Details:   Cell type    Pin type     Cell name (clock net +/-)

Source:            FF           Q            mem_if_tx_address_8 (from clk_c +)
Destination:       FF           Data in      mem_if_tx_address_17 (to clk_c +)

Delay:                 7.358ns (61.2% logic, 38.8% route), 4 logic levels.
```

In addition to hierarchical blocks, such as a group consisting of an entire VHDL entity or Verilog HDL module, you can floorplan individual nodes. For example, you can instantiate a library element for a function in the critical path and then group the library element. This technique is useful if the critical path spans multiple design blocks.

Note

Although floorplanning can increase performance, it may also degrade performance if it is not applied correctly within software limitations.

Floorplan to Preserve Module Performance

Floorplanning with design preferences maintains design performance by grouping the placement of nodes in a device, which ensures that the relative placement of logic within a grouped region remains constant. The ispLEVER software then places the grouped region into the top-level design with these preferences. When placing logic in a region, the ispLEVER software does not preserve the routing information. This approach provides more flexibility when the software imports the region into the top-level design, and it helps fitting.

Floorplan for Design Reuse

Floorplanning facilitates design reuse by its ability to reproduce the performance of a module designed in a different project. For frequently used modules, you can create a library of verified designs that can be incorporated into other larger designs. The library must only contain the VHDL or Verilog HDL source code, along with grouping attributes and some comments detailing information useful to you, such as performance and size. With a parameterized module, in-code assignments can specify the module's size and grouping assignments.

Targeting the same device used in the original design usually achieves the best results, although other devices in the same family will likely work well. When using a different device in the same family, the exact placement of the region might not be possible. Similar performance, however, might be achieved by moving or floating regions. A floating region groups the logic together and guides the ispLEVER software toward achieving a placement that meets the performance requirements of the module. A similar approach can also be taken if exact placement of a module is not applicable because of multiple instantiations of a module in a top-level design.

Floorplanning Preferences

Floorplanning preferences, such as logic groups and regions, can be set in the Verilog HDL or VHDL source through the use of HDL attributes; they can be set in the ASCII logical preference file (.lpf) through the Design Planner user interface; or they can be set through a combination of both methods.

The Design Planner, with its graphical design views and its facility for querying timing paths, can be extremely useful for establishing floorplan preferences, such as logical groups, regions, and device site assignments. It is a common practice, in a timing closure methodology, to iterate between the Design Planner application and the place-and-route program (PAR) to arrive at a superior implementation, and afterwards, to migrate the physical constraints into the RTL code as logical constraints.

The ispLEVER software supports a logic grouping mechanism that enables you to direct the placement algorithm of the PAR program to pack logic elements in proximity to each other and, optionally, to place them within a particular region of the FPGA array.

Two main floorplanning preferences are available as group attributes in HDL:

- HGROUP – hierarchical physical attribute. An HGROUP's logical identifier is prepended with text that describes the identifier's hierarchy. During the mapping process, the HGROUP is expanded into individual placement groups (PGROUPs).
- UGROUP – universal logical attribute. Prepending the hierarchy on the block instance identifier does not change a UGROUP's logical identifier. In other words, an HGROUP enforces strict hierarchical control, but a UGROUP allows for a grouping of blocks in different hierarchies or a grouping of blocks with no hierarchy at all. During the mapping process, the UGROUP is expanded into individual placement groups (PGROUPs).

The HGROUP attribute can be placed on multiple instantiations of modules—for example, VHDL generate statements—and each instantiation has its own HGROUP. A UGROUP does not work in this case.

The Design Planner produces UGROUP preferences and saves the result in the logical preference file (.lpf).

The following FPGA device elements can be grouped using a PGROUP:

- Slice/PFU
- sysMEM memory
- sysDSP blocks

In Figure 126 and Figure 127, the arrows represent control and data paths where there is interaction between different levels of hierarchy. The thick-lined arrow represents the critical path where the design fails to make performance.

Figure 126: PGROUP Same Hierarchy Example, PGROUP CONTROLLER

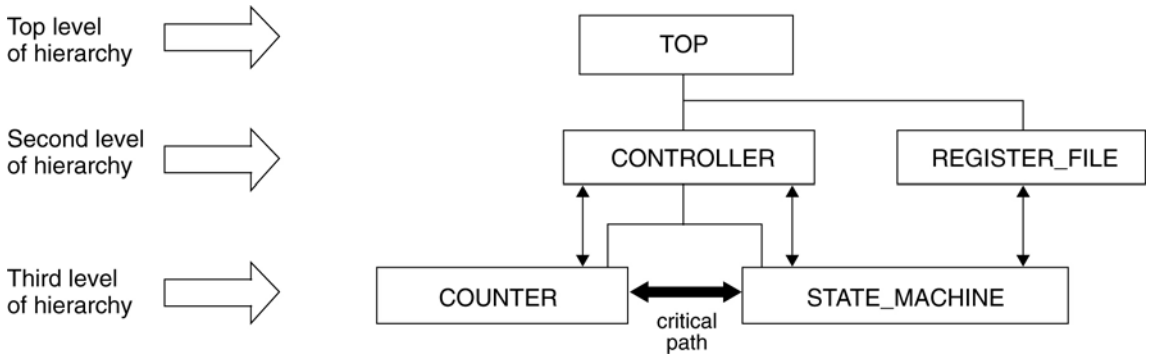


Figure 126 illustrates a design hierarchy where the failing paths are the connections between COUNTER and STATE_MACHINE design blocks. The easiest implementation for this example is to HGROUP the CONTROLLER, which is the module in which the COUNTER and STATE_MACHINE are instantiated.

For example, if the following Synplify and RTL Precision Synthesis attribute is in the Verilog HDL file:

```
module CONTROLLER (<port_list>)
/* synthesis hgroup=”CONTROL_GROUP” */;
//pragma attribute CONTROLLER hgroup CONTROL_GROUP
```

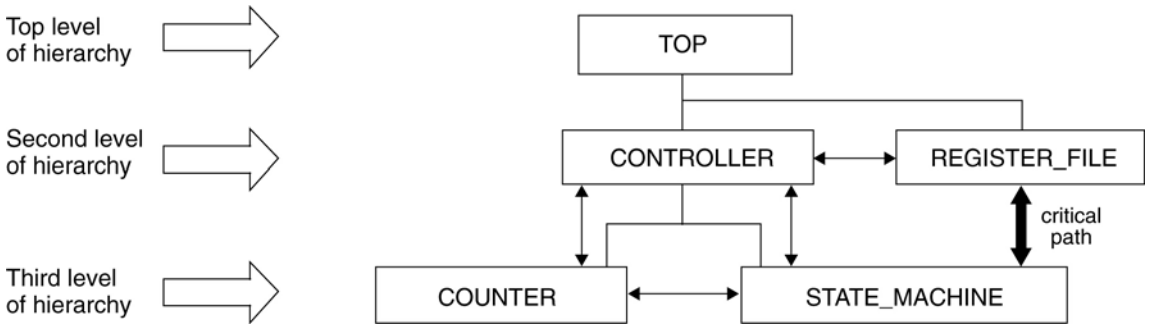
the COUNTER and STATE_MACHINE will be grouped in the FPGA inside a boundary box. Now assume that the COUNTER is mapped into PFU_0 and PFU_1 and that the STATE_MACHINE is mapped into PFU_2. The resulting group generated by MAP and written to the physical preference file (.prf) will be:

```
PGROUP “TOP/CONTROLLER/CONTROL_GROUP”
COMP “PFU_0”
COMP “PFU_1”
COMP “PFU_2”;
```

Notice that the TOP/ hierarchy is prepended to the CONTROLLER PGROUP.

Figure 127 shows an example design hierarchy where the failing paths are the connections between REGISTER_FILE and STATE_MACHINE modules. The simplest thing to do is UGROUP the REGISTER_FILE and STATE_MACHINE together.

Figure 127: UGROUP Different Hierarchy Example with UGROUP REGISTER_FILE and STATE MACHINE



For example, if the following Synplify and Precision attributes are in the Verilog HDL file:

```
module REGISTER_FILE (<port_list>) /*synthesis
ugroup=”CRITICAL_GROUP” */;
//pragma attribute REGISTER_FILE ugroup CRITICAL_GROUP
```

and

```
module STATE_MACHINE (<port_list>) /*synthesis
ugroup=”CRITICAL_GROUP” */;
//pragma attribute STATE_MACHINE ugroup CRITICAL_GROUP
```

the REGISTER_FILE and STATE_MACHINE will be grouped in the FPGA inside a default boundary box.

Now assume that the REGISTER_FILE is mapped into PFU_4 and PFU_5 and that the STATE_MACHINE is mapped into PFU_3. The resulting group generated by MAP and written to the .prf will be:

```
PGROUP “CRITICAL_GROUP”
  COMP “PFU_3”
  COMP “PFU_4”
  COMP “PFU_5”;
```

The TOP/ hierarchy is not appended to the PGROUP identifier CRITICAL_GROUP. The UGROUP results in a PGROUP.

If HGROUP attributes instead of UGROUP attributes had been used for Figure 127:

```
module REGISTER_FILE (<port_list>) /*synthesis
hgroup=”CRITICAL_GROUP” */;
//pragma attribute REGISTER_FILE hgroup CRITICAL_GROUP
```

and

```
module STATE_MACHINE (<port_list>) /*synthesis
hgroup=”CRITICAL_GROUP” */;
//pragma attribute STATE_MACHINE hgroup CRITICAL_GROUP
```

the resulting groups generated by MAP and written to the .prf would be:

```
PGROUP “TOP/CONTROLLER/STATE_MACHINE/CRITICAL_GROUP”
  COMP “PFU_3”
PGROUP “TOP/REGISTER_FILE/CRITICAL_GROUP”
  COMP “PFU_4”
  COMP “PFU_5”;
```

So with PGROUP attributes, the STATE_MACHINE module would be grouped together in one bounding box, and the REGISTER_FILE module would be grouped together separately in another bounding box. The critical path shown in Figure 127 would not be optimized.

These examples do not utilize all the possible tools available for floorplanning. For more small syntax examples, refer to the “HDL Attributes” section of the ispLEVER FPGA and Crossover Design online Help.

Implementation of Floorplan Preferences

Floorplan preferences set from within the HDL or the Design Planner GUI are validated, then they are translated by the design mapper (MAP) into physical preferences in terms of post-map physical components. Both hierarchical groups (HGROUPs) and universal groups (UGROUPs) refer to logical block references that you can easily recognize from the HDL source. The design mapper produces the native physical database (.ncd) and converts HGROUPs and UGROUPs into to placement groups (PGROUPs), which refer to post-map components of the .ncd file.

This section describes several examples of PGROUP and LOCATE preferences that are implemented by the place and route program into a LatticeEC LFEC1E FPGA.

Locating a Block to a Device Site

The simplest floorplan technique in ispLEVER is to anchor a logic block to a particular device site, using the LOCATE preference. Blocks can be anchored independently of a group/region floorplan. The most common type of block to locate is a PIO.

The following procedure describes how to locate a block to a device site.

1. This step is optional. If you intend to floorplan design elements that will be mapped to slice or PFU device sites, you must add the COMP <*comp_name*> HDL attribute to each module instance in the HDL source, as shown in the following Verilog HDL sample. This sample illustrates attributes that are compatible with Precision RTL Synthesis and Synplify.

   ```
   REG2 REG2inst (<port_list>) /* synthesis COMP=regpair */;
   //pragma attribute REG2inst COMP regpair
   ```

   In this sample, the COMP name regpair is applied by the design mapper (MAP) to all elements that can be covered by a single slice. If the logic

overflows a single slice, MAP appends a *.number* to the name for the post-map netlist.

During the floorplanning steps that follow, you reference the *comp_name* of the Design Planner Post-Map View to assign it to a specific device site.

Note

Design elements such as PIO, EBR, DSP, PLL/DLL, and MACO blocks do not require the COMP attribute, because MAP retains the original name used in the native generic database (.ngd).

2. Run the **Design Planner (Post-Map)** process to view the Post-Map Physical Netlist View and Floorplan View of the Design Planner tool.

   You can use the graphical views of the Design Planner to assign one or more instances (PIO, PLL/DLL, slice, EBR, DSP, and so forth) to device sites. PIOs are typically assigned in the Package View or Spreadsheet View. Embedded blocks such as EBR and DSP blocks are typically assigned in the Floorplan View. Each post-map device element can be selected and assigned to a specific device site through a drag-and-drop action. The result of the action is a LOCATE COMP preference in the logical preference file (.lpf).

   ```
   LOCATE COMP "FIFOinst/FIFOeab/syn_dpram_512x8" SITE
   "EBR_R23C5";
   ```

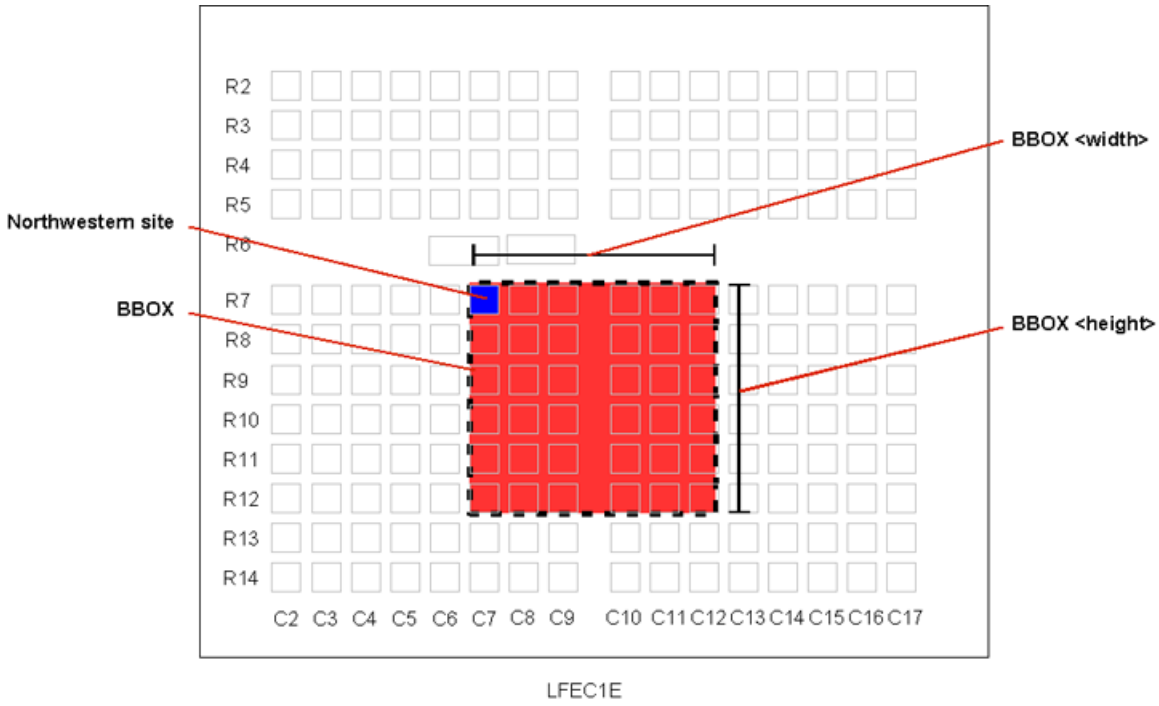
   Note

   Slice-type device elements must be named with the COMP attribute, as described in step 1. Otherwise, the physical references that result are not recognized by the design mapper in future runs.

Anchored PGROUP

An anchored PGROUP refers to a placement group that is affixed to a certain row and column site or anchor point. This fixed site or anchor point is defined at a specific slice site or PFU device site by a hard LOCATE preference. Placement grouping of the PGROUP elements are restricted to within the dimension of a bounding box (BBOX), as shown in Figure 128.

Figure 128: Anchored PGROUP



An anchored PGROUP has the following characteristics:

- It is hard-located on the device with the LOCATE preference.
- It is located at a slice site by means of an anchor point at the northwestern corner.
- A slice anchor point is made to the D slice of the 4-slice PFU.
- A PGROUP's bounding box (BBOX) defines a fixed area in row and column dimensions.
- PGROUP components are located relatively in the BBOX definition, as shown in the following example.

```
PGROUP "my_pgroup" BBOX 6 6 DEVSIZE
COMP "SLICE_1" "R1C1A"
COMP "SLICE_2";
LOCATE PGROUP "my_pgroup" SITE "R7C7D";
```

- A PGROUP commonly contains a mix of slice-based and embedded block logic.

Note

- Anchors must be at a slice or PFU-type device site. Embedded block type device sites cannot serve as group anchor points.

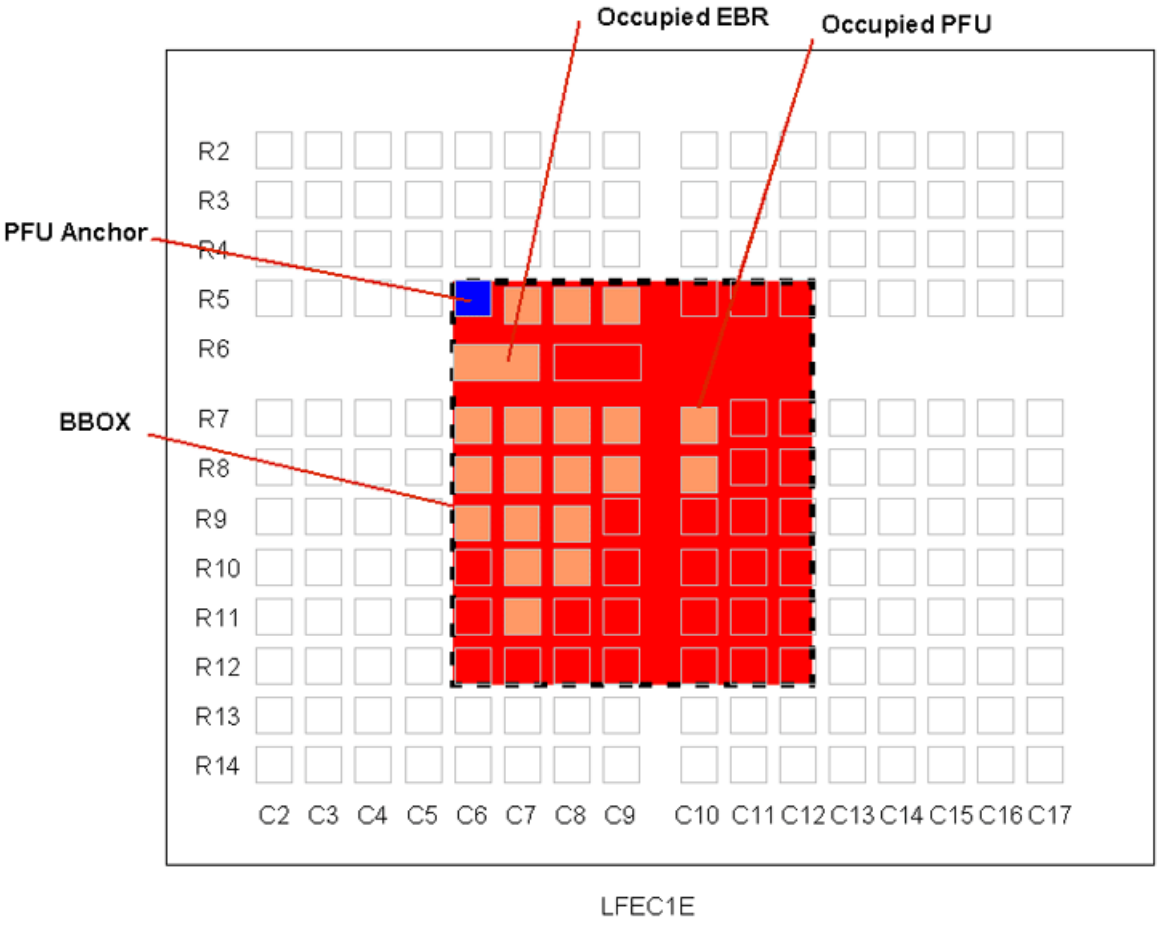
The *my_pgroup* PGROUP defines the user-defined PGROUP. The bounding box has the dimensions of 6 rows by 6 columns of contiguous device rows and columns (DEVSIZE), as defined by BBOX 6 6. The COMP name defines the slice members of the group after the BBOX definition. In the first line, the row and column dimension R1C1A is the relative anchor site or northwestern site of the BBOX with the SLICE_1 component occupying that corner. The LOCATE preference on the last line anchors the PGROUP to the R7C7D device site.

Note

The Design Planner does not support assignment of individual slices to device sites through the UGROUP preference, as shown in the examples.

Placement grouping of a PGROUP that contains both slice and embedded block elements is also restricted to the dimension of a BBOX. The PGROUP anchor point must be at a slice-type device site, and the bounding box should encompass enough resources to accommodate all of the group elements. Figure 129 shows an example of both PFU- and EBR-based logic.

Figure 129: Anchored PGROUP with Slice and Embedded Blocks



Regional PGROUPs

To specify regional PGROUPs, you use the REGION preference. REGION defines a rectangular area within which a PGROUP can float; that is, the PGROUP can be optionally placed anywhere within that specified region.

Note

Groups that are composed of both slice-based and embedded block logic, such as EBR and DSP-type blocks, must be anchored. Groups composed solely of slice-based logic, such as LUTs and registers, can float.

Regional PGROUPs have the following characteristics:

- The region area is defined with a REGION preference.
- The region's northwestern site defines its anchor point.
- The PGROUP is located to float within the defined region area.
- No anchor point is defined for the PGROUP.

- A bounding box (BBOX) defines the size of the PGROUP.
- Components are located relatively in the BBOX definition.

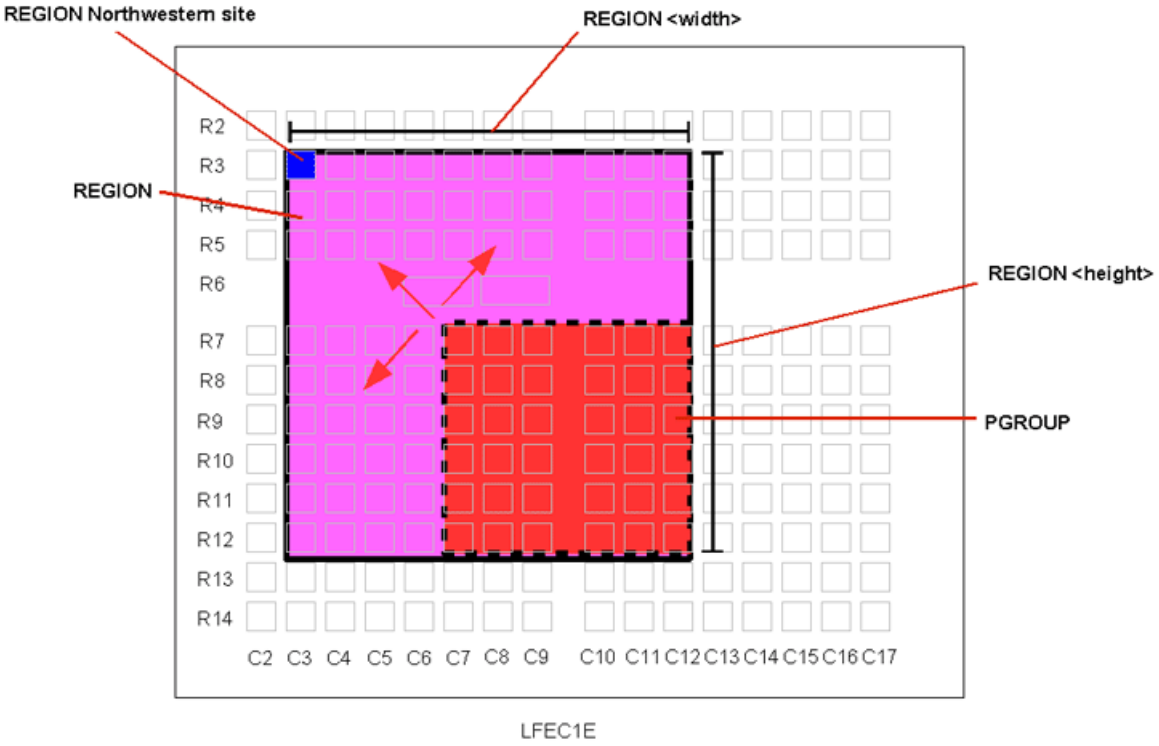
Note

Anchor points must be at a slice- or PFU-type device site. Embedded block type device sites cannot serve as region anchor points.

The following example places the PGROUP within the *my_region* REGION, which is a fixed rectangular area between R3C3 and R12C12, as illustrated in Figure 130.

```
REGION "my_region" "R2C2D" 10 10 DEVSIZE;
PGROUP "my_pgroup" BBOX 6 6 DEVSIZE
COMP "SLICE_1" "R1C1D"
COMP "SLICE_2";
LOCATE PGROUP "my_pgroup" REGION "my_region";
```

Figure 130: PGROUP Floating within a REGION



Completely Floating PGROUPs

A PGROUP can also completely float; that is, the PGROUP can be placed anywhere on the device. You can do this by precluding the LOCATE PGROUP.

The following example creates a completely floating PGROUP, as shown in Figure 131:

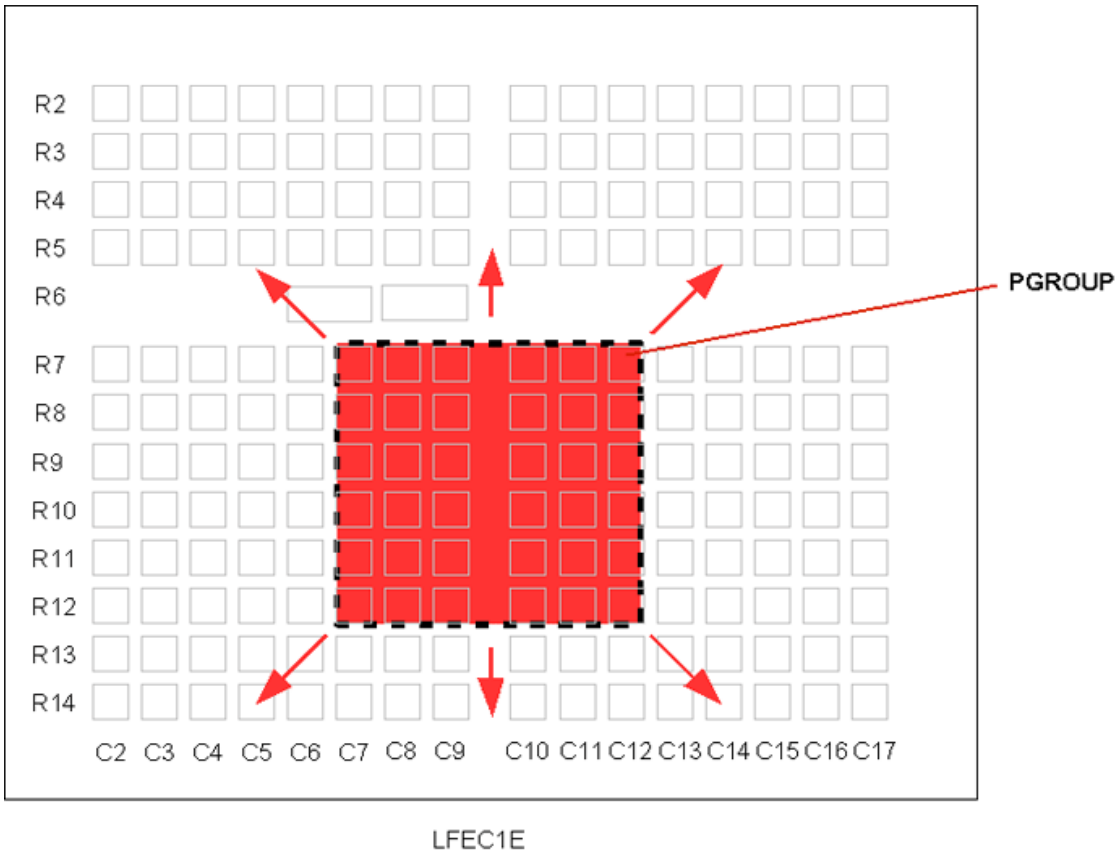
```
PGROUP "my_pgroup" BBOX 6 6 DEVSIZE
```

```
COMP "SLICE_1" "R1C1D"
COMP "SLICE_2";
```

Note

Groups that are composed of both slice-based and embedded block logic, such as EBR and DSP-type blocks, must be anchored. Groups composed solely of slice-based logic, such as LUTs and registers, can float.

Figure 131: PGROUP Completely Floating on a Device



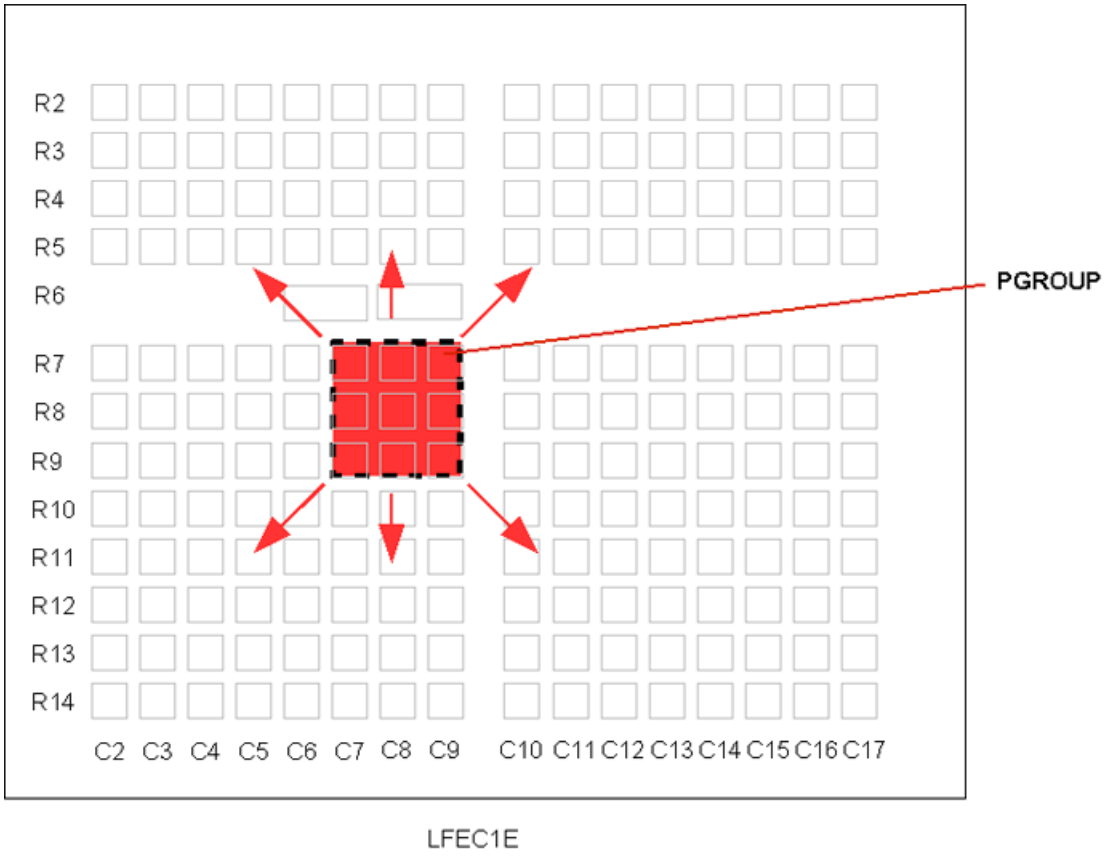
Completely Floating PGROUP with Minimum BBOX

The previous examples all specify BBOX in the PGROUP. Since the BBOX parameter is optional, some PGROUPs' definitions might not specify a BBOX. In such cases, a minimally sized BBOX is generated for the PGROUP internally by default. See Figure 132.

Note

Groups that are composed of both slice-based and embedded block logic, such as EBR and DSP-type blocks, must be anchored. Groups composed solely of slice-based logic, such as LUTs and registers, can float.

Figure 132: PGROUP Completely Floating with Minimal BBOX



The following example defines a completely floating PGROUP with a 3 x 3 (3 rows in height and 3 columns wide) or minimal BBOX, as shown in Figure 132.

```
PGROUP “my_pgroup"
COMP "SLICE_1" "R1C1A"
COMP "SLICE_2";
```

The R1C1A parameter in the COMP definition refers to the relative (not absolute) northwestern site, or origin, of the PGROUP's bounding box and should not be confused with an anchor point. Anchor points use the LOCATE preference.

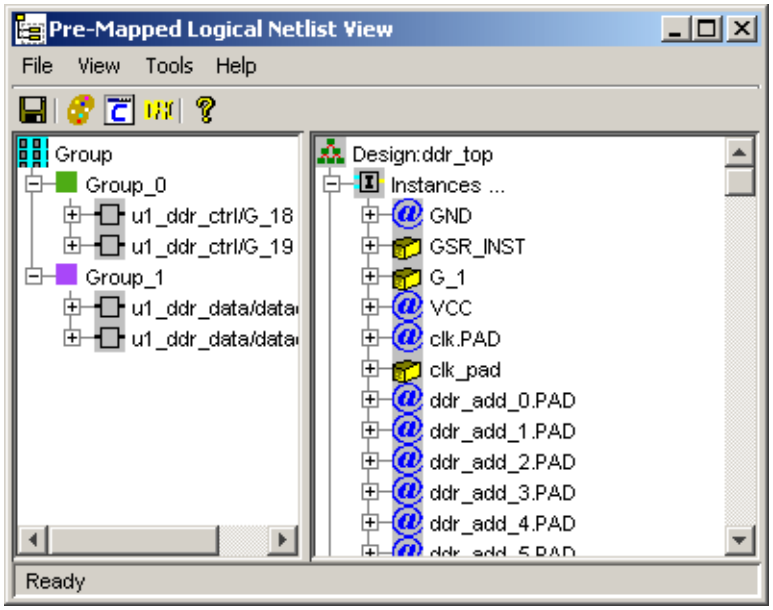
Setting Group Preferences in the Design Planner

The Design Planner user interface allows you to graphically view and modify any group preferences that were set in the HDL and to create additional groups to improve design performance.

Pre-Mapped View for UGROUPs

In the Pre-Map Logical Netlist View, shown in Figure 133, logical components from the pre-mapped netlist can be combined into UGROUPs. The Pre-Map Logical Netlist View also displays individual UGROUPs that were expanded from HGROUPs in the HDL during the Build Database process. Modified or newly created UGROUPs are written to the logical preference file (.lpf) with the Save command.

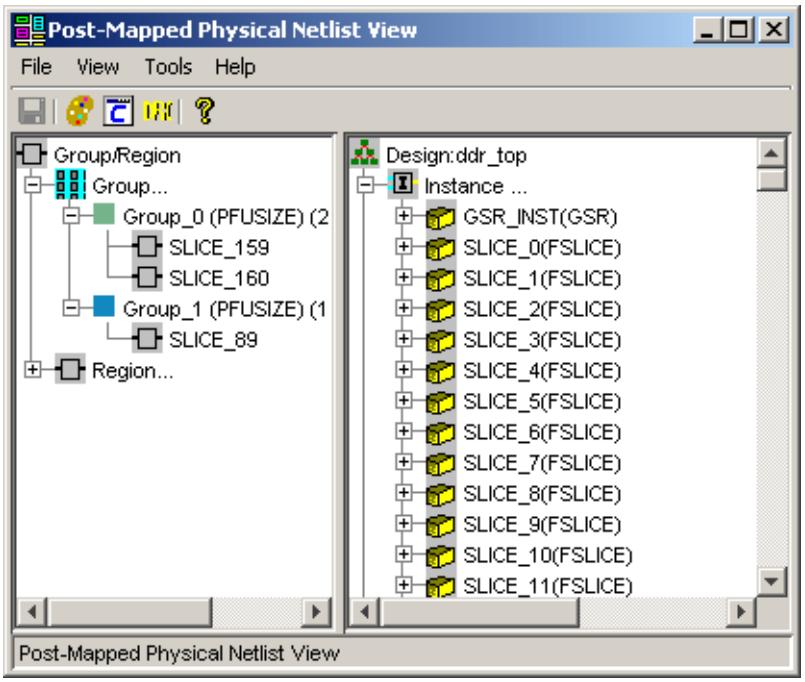
Figure 133: Pre-Mapped View



Post-Mapped View for PGROUPs

Physical components are displayed in the Post-Mapped Physical Netlist View, shown in Figure 134, and they can be combined into PGROUPs and placed into regions. Since the mapping process translates UGROUPs to physical groups, any pre-map logical groups become PGROUPs in the post-mapped netlist.

Figure 134: Post-Mapped View



Floorplan and Physical Views

After placement and routing (PAR), you can view and edit PGROUPs and regions in the Floorplan View (Figure 135) and in the Physical View (Figure 136).

Figure 135: Floorplan View

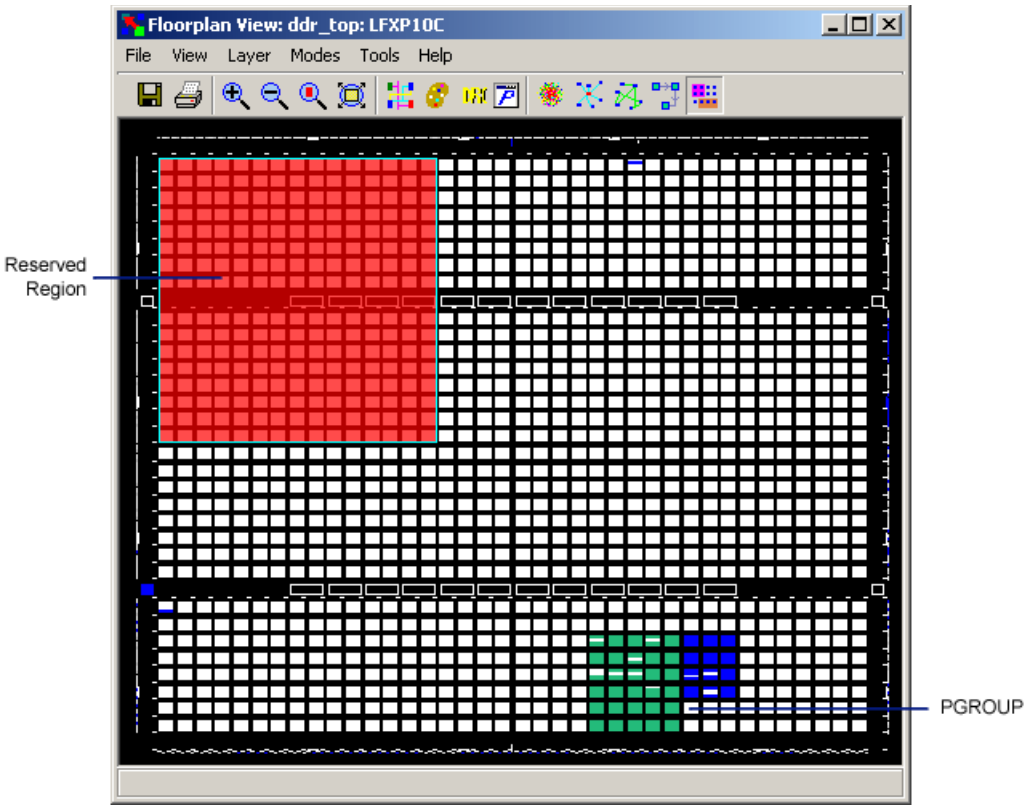
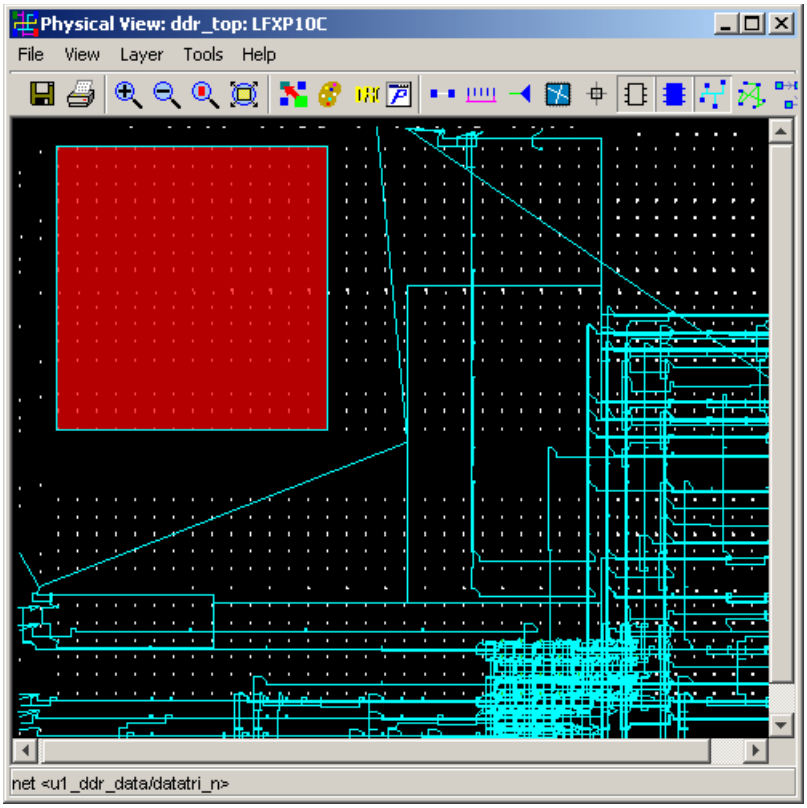


Figure 136: Physical View

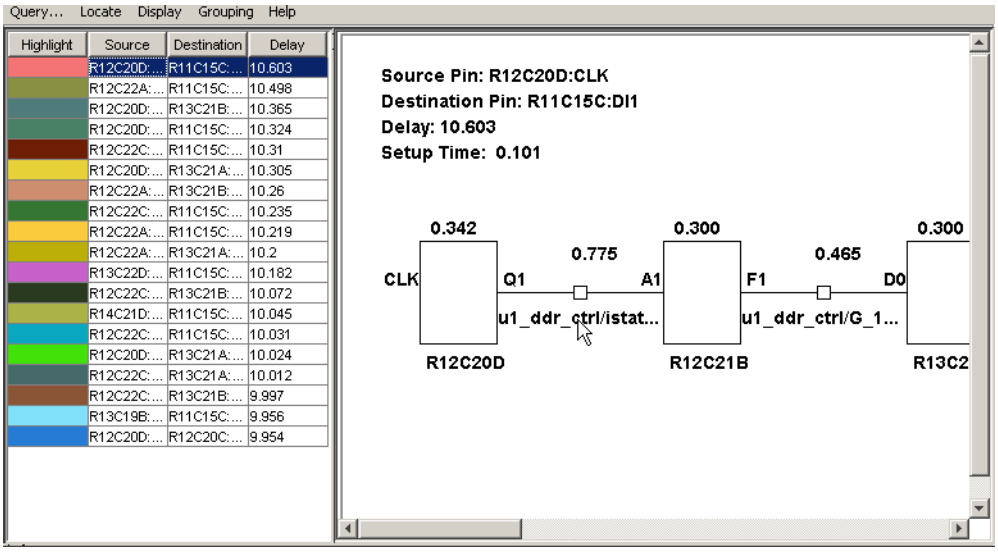


You can draw regions in the Floorplan View and the Physical View layouts. This feature is especially useful for reserving areas of the floorplan for other modules.

Path Tracer

The Design Planner's Path Tracer, shown in Figure 137, enables you to query a timing path, identify the longest delay, and then group those components that are scattered around the fabric in order to reduce the delay.

Figure 137: Path Tracer



Saving Preferences

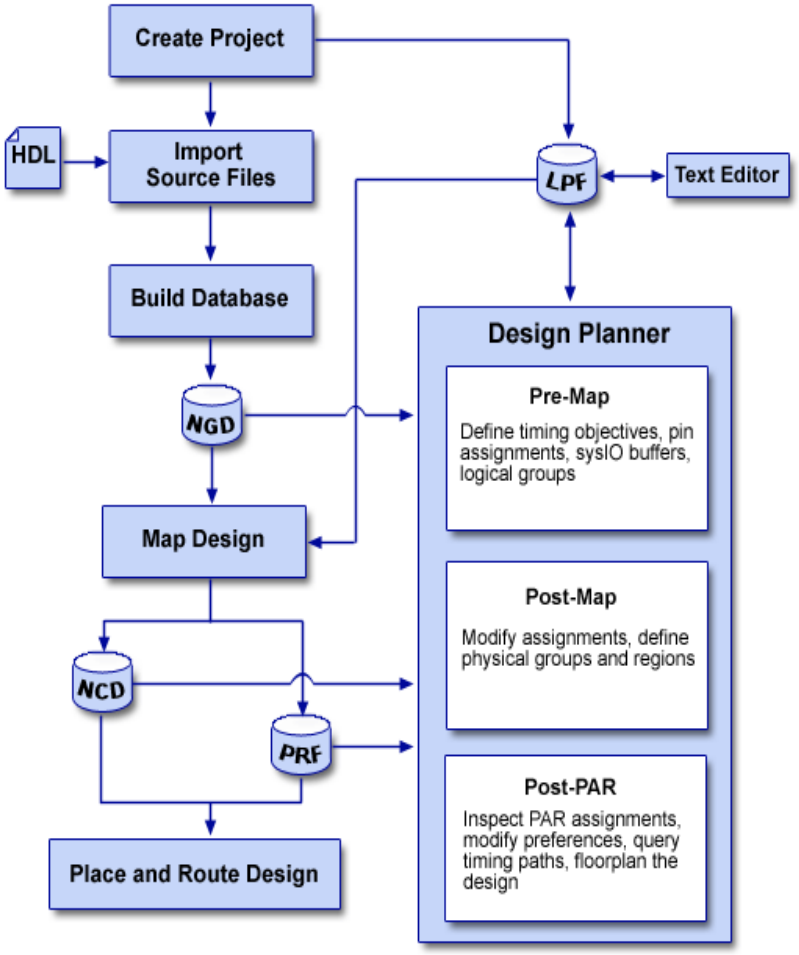
Design changes made in the Design Planner are written to the logical preference file (.lpf) when the design is saved. When new or modified PGROUPs and REGIONs are saved in the Design Planner, their logical equivalents are immediately written to the .lpf, and they will be written to the physical preference file (.prf) when the design is remapped.

HGROUP and UGROUP attributes from the HDL are not written to the .lpf until they are modified, after which the .lpf preferences take precedence.

Persistence of Preferences

Because the Design Planner saves all logical preferences in the .lpf, including the logical equivalents of physical preferences such as PGROUPs, the preferences persist through repeated modifications and design iterations. This makes the Design Planner a highly useful tool for setting preferences, since there is less need to go back and modify the HDL. The flow of preferences from the HDL, through the Design Planner and mapping, are shown in Figure 138.

Figure 138: Preference Flow



Using the Design Planner Interface

To run the Design Planner, select the targeted device in the Project Navigator, and then select one of the following options:

- Double-click **Design Planner (Pre-Map)** to open the Spreadsheet View and Package View. Define timing constraints and make pin assignments in the pre-mapped stage.

- Double-click **Design Planner (Post-Map)** to open the Post-Map Physical Netlist View, Pre-Map Logical Netlist View, and Floorplan View. Create or modify physical groups and regions.
- Double-click **Design Planner (Post-PAR)** to open the Post-Map Physical Netlist View, Pre-Map Logical Netlist View, and Floorplan View. Modify placement, query timing paths, and make further adjustments
- From a command prompt window, type the **flmainapp** command, then open the desired pre-map (.ngd) or post-map (.ncd) database file from the Design Planner Control.

Note

For more information about running the Design Planner from the command line, see the “Command Line” section of the Design Planner online Help.

Design Performance Enhancement Strategies

The design strategies for performance enhancement depend on the structure of a particular circuit. Strategies include the following:

- Defining regions based on design hierarchy, if the hierarchy closely resembles the structure of the circuit. Such designs typically consist of tightly integrated modules, where the logic for each module is self-contained and the modules communicate through well-defined interfaces.

  Use the Design Planner’s Pre-Map Logical Netlist View, shown in Figure 133, to create logical groups based on the design hierarchy.
- Defining regions based on the critical path, if the critical path is long and spans multiple modules.

  Use the Design Planner’s Path Tracer, shown in Figure 137, to identify the critical path and keep the nodes in the critical path together by grouping the logical components. This can lead to improved performance.
- Defining regions based on connections by grouping nodes together that contain high fan-outs and high fan-ins.

  Use the Floorplan View, shown in Figure 135, and Physical View, shown in Figure 136, to view them on the physical design layout, and then group nodes together to reduce delays in connections and wiring congestion.

It might be necessary to change the existing design hierarchy and structure to make the design more amenable to floorplanning, especially if modular hierarchy and structure were not considered at the beginning of design conception.

You can elect to optimize modules individually and can exercise varying amounts of control over the placement by using different types of regions. When bounding boxes and location anchors are used selectively, the ispLEVER software can automatically determine the best size and location for a region.

Another approach is to optimize the top-level design without first optimizing the individual modules. This approach allows the ispLEVER software to place nodes within regions and move regions across the device. You assign

modules to regions, then compile the entire design. With this approach, you can place elements from different modules in a region.

Note

For more information about using the Design Planner user interface, see the ispLEVER Design Planner online Help.

Special Floorplanning Considerations

The following sections describe the use of elements such as embedded block RAM and certain types of groupings that require special consideration.

Embedded Block RAM Placement

Block RAM placement can be done with simple LOCATE preferences. It is not always necessary to locate block RAMs. Do not use the PGROUPs, UGROUPs, or the Design Planner GUI to group Block RAMs.

I/O Grouping

There is a complete set of physical constraints for grouping I/O components. Refer to the “HDL Attributes” section of the ispLEVER FPGA and Crossover Design online Help.

Large Module Grouping

Larger PGROUPs and UGROUPs (with many logical elements) should be anchored and bounded by LOCATE and BBOX keywords.

The BBOX should be strategically shaped and sized according to the module to be placed inside the BBOX. If the BBOX shape and size are not specified, the default BBOX size will be a square that is as small as possible. This is not the optimal BBOX for typical modules.

You should shape the design with the data path in mind and size the BBOX to be larger than needed so that the ispLEVER placer can have more flexibility in placing logic elements inside the BBOX. You can determine the BBOX size by counting the number of slices from a grouped module that has already been mapped.

Carry Chains and Bus Grouping

Carry chains (used by ripple arithmetic functions like adders, counters, and multipliers) and logic modules connected by buses can easily be floorplanned inappropriately if you are not aware of the internal routing resources available for optimizing these carry chain and bus routes. Certain groupings can reduce the performance of a design compared to no floorplanning at all.

An example of a broken carry chain is a 9-bit adder that is PGROUPed with no relative placement on the adder.

Logic elements such as PFUs might give worse performance because the adder carry-chain is broken.

SLICs in Groups

Supplemental Logic and Interconnect Cells (SLICs), which are contained in some Lattice Semiconductor FPGA device families, are automatically removed from PGROUPs and UGROUPs by the ispLEVER software unless they are relatively placed. This is because SLICs are used by the tools for interconnects that you cannot foresee. If SLIC placement must be controlled for a design, you must instantiate and locate the SLICS in the preference or HDL files. It is recommended that you allow the ispLEVER software to place SLICs automatically.

Conclusion

For a placement and routing strategy that will meet timing objectives, start with a good set of FPGA timing preferences. For a design's first placement and routing, run PAR at the low placer effort level and with a low number of routing iterations. There is no point in running 100 cost tables if the design's logic depth is too high. Use TRACE to analyze timing, then modify preferences to help improve it. Experiment with different strategies for controlling placement and routing; use TRACE to analyze timing for each iteration. If performance goals still are not met, use floorplanning to group components along critical paths and shorten routing distances.

For hands-on training, see the “Achieving Timing Closure in FPGA Designs Tutorial.”

Index