ispLEVER 4.0

Process Flow User Guide

Line: 1-800-LATTICE or (408) 826-6002
Web Update: To view the most current version of this document, go to *www.latticesemi.com.*
LEVER-UG-FLOW 4.0.0

Copyright

Trademarks

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

April 2004

Limited Warranty

Lattice Semiconductor Corporation warrants the original purchaser that the Lattice Semiconductor software shall be free from defects in material and workmanship for a period of ninety days from the date of purchase. If a defect covered by this limited warranty occurs during this 90-day warranty period, Lattice Semiconductor will repair or replace the component part at its option free of charge.

This limited warranty does not apply if the defects have been caused by negligence, accident, unreasonable or unintended use, modification, or any causes not related to defective materials or workmanship.

To receive service during the 90-day warranty period, contact Lattice Semiconductor Corporation at:

Phone: 1-800-LATTICE or (408) 826-6002
E-mail: techsupport@latticesemi.com

If the Lattice Semiconductor support personnel are unable to solve your problem over the phone, we will provide you with instructions on returning your defective software to us. The cost of returning the software to the Lattice Semiconductor Service Center shall be paid by the purchaser.

Limitations on Warranty

Any applicable implied warranties, including warranties of merchantability and fitness for a particular purpose, are hereby limited to ninety days from the date of purchase and are subject to the conditions set forth herein. In no event shall Lattice Semiconductor Corporation be liable for consequential or incidental damages resulting from the breach of any expressed or implied warranties.

Purchaser's sole remedy for any cause whatsoever, regardless of the form of action, shall be limited to the price paid to Lattice Semiconductor for the Lattice Semiconductor software.

The provisions of this limited warranty are valid in the United States only. Some states do not allow limitations on how long an implied warranty lasts, or exclusion of consequential or incidental damages, so the above limitation or exclusion may not apply to you.

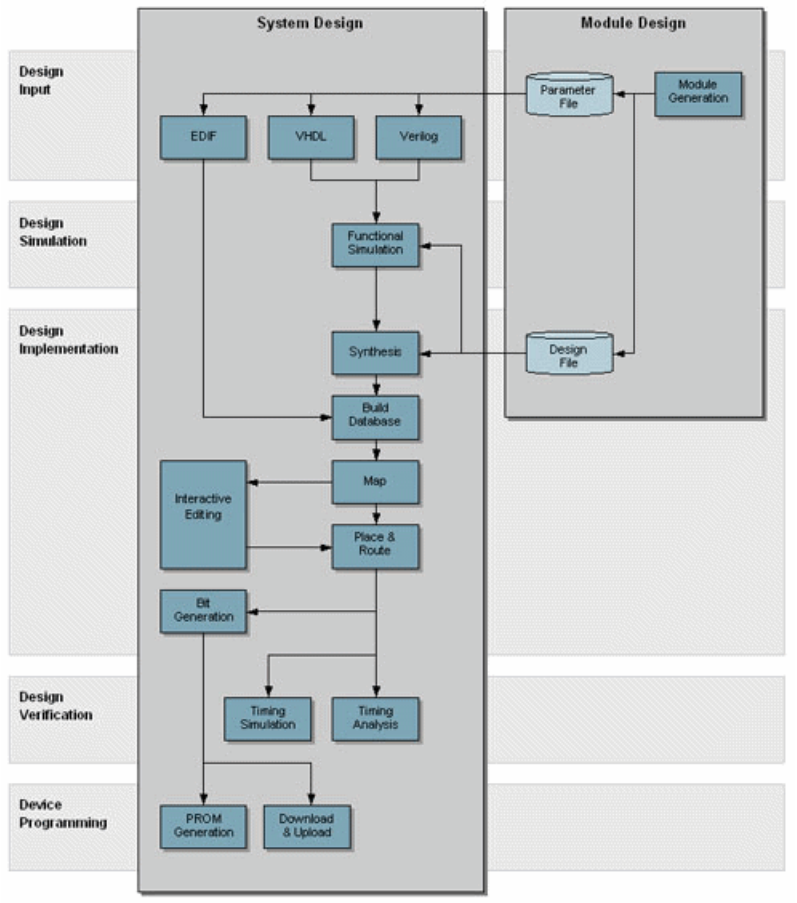
This warranty provides you with specific legal rights. You may have other rights which vary from state to state.

Table of Contents

CHAPTER 1 *FPGA Process Flow*

Introduction

FPGA Design



The ispLEVER® 4.0 design tool allows you easy implementation of designs using Lattice FPGA devices. Synthesis library support is available for the major logic synthesis tools. The ispLEVER tool takes the output from these common synthesis packages and places and routes the design in the FPGA device. The tool allows floorplanning and the management of other constraints within the device. The tool also provides outputs to common timing analysis tools for timing analysis.

To increase designer productivity, Lattice provides a variety of pre-designed modules referred to as IP cores. These IP cores let you concentrate on the unique portions of your design while using pre-designed blocks to implement standard functions such as bus-interfaces, standard communication-interfaces, and memory-controllers.

Supported Device Families

- ORCA
- Lattice-XP
- Lattice-EC
- Lattice-ECP
- Lattice-SC

Overview of ispLEVER for FPGA

The ispLEVER program offers an integrated environment consisting of several tools necessary to implement Lattice FPGA devices. These tools are briefly described in the following paragraphs and covered in detail in their respective Help. They are listed in alphabetical order.

EPIC Device Editor

EPIC (Editor for Programmable ICs) is a graphical application for displaying and configuring FPGAs. EPIC can be used for placing and routing critical components before running automatic place and route tools on an entire design. It can also be used for manually finishing the placement and routing if the routing program was unable to route the design to completion. See the EPIC Device Editor Help for more information about this tool.

FPGA Floorplanner

The FPGA Floorplanner provides graphical user interface to facilitate the management of FPGA device real estate to conform to critical circuit performance requirements and to shorten design turn around time. See the FPGA Floorplanner Help for more information about this tool.

LeonardoSpectrum for Lattice

The LeonardoSpectrum™ synthesis environment from Mentor Graphics® lets you can create Lattice FPGA device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process. LeonardoSpectrum combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features. See the LeonardoSpectrum for Lattice documentation supplied by the manufacturer for more information about this tool.

ModelSim for Lattice

The ispLEVER software supports third-party HDL simulation and verification with ModelSim™ from Mentor Graphics®. Using this integrated package, you can simulate single Verilog or VHDL designs within one environment. See the ModelSim for Lattice documentation supplied by the manufacturer for more information about this tool.

Module/IP Manager

The Module/IP Manager enables you to generate cores (parameterized modules and IP cores) from templates supplied by Lattice Semiconductor and third-party vendors. After generating the module, the Module/IP Manager automatically stores the instantiation template and related files to your project folder. See the Module/IP Manager Help for more information about this tool.

Power Calculator

The Power Calculator is a stand-alone tool used to estimate device power consumption. The Power Calculator can be involved at any stage in the ispLEVER software place and route flow, but the power estimation is most accurate when the Power Calculator uses the post-routed NCD and simulation VCD result.

The Power Calculator graphical user interface (GUI) allows you to specify power parameters like design resource utilization, activity rate, and frequency. Parameters are set through a spreadsheet-type interface.

Device power consumption is calculated based on device characteristics and settings. See the Power Calculator Help for more information about this tool.

Preference Editor

The Preference Editor lets you specify or change placement, routing, and/or timing constraints produced by the user or by the technology mapper. The Preference Editor reads the preference file (`.prf`) and displays the preference settings. Modifications to the PRF file are made via the function dialog boxes. Most of the attributes can be modified directly in the sheet. Pin assignments can be set in the Package View with drag-and-drop functionality. See the Preference Editor Help for more information about this tool.

Project Navigator

The Project Navigator is the primary interface for ispLEVER and provides an integrated environment for managing the project elements and processes, as well as accessing all ispLEVER tools. See the Project Navigator Help for more information about this tool.

Report Viewer

You can use the Report Viewer to view, but not edit, the various report files generated by ispLEVER. See the Report Viewer Help for more information about this tool.

Synplify for Lattice

Synplify® for Lattice is a logic synthesis tool for FPGA devices, developed by Synplicity®. Synplify starts with high-level designs written in Verilog or VHDL hardware description languages (HDLs). Synplify then converts the HDL into small, high-performance, design netlists that are optimized for Lattice devices. See the Synplify for Lattice documentation supplied by the manufacturer for more information about this tool.

Tcl Editor

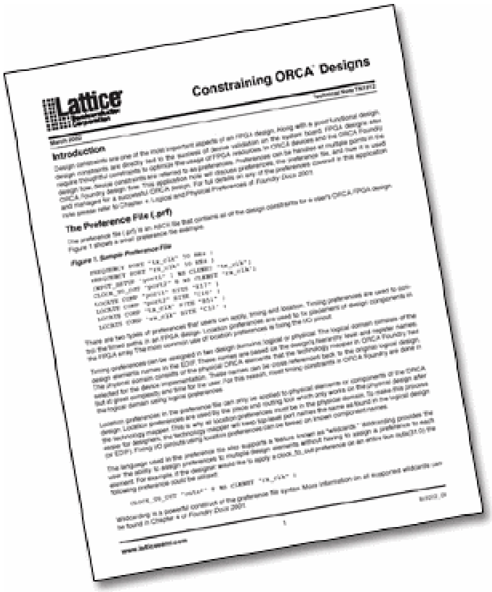
The Tcl Editor is a design tool that allows you to create, edit, and run a series of Tool Control Language (TCL) commands. These commands invoke individual ispLEVER processes for generating a complete programmable logic solution. The default text editor permits you to edit TCL code, and it highlights key TCL language elements in different colors to clarify the nature and use of each language element. You can generate a Tcl script for a project in Project Navigator, open it in the Tcl Editor, edit the script, and run it. The Tcl Editor, together with the robust features of the language, gives you more control over the design environment. See the Tcl Editor Help for more information about this tool.

Text Editor

The Text Editor is the ispLEVER text entry tool. You use this tool to create and edit text-based files, such as ABEL-HDL files, test stimulus files, and project documentation files. See the Text Editor Help for more information about this tool.

FPGA Application Notes

The Lattice Semiconductor web site lists several Application Notes for FPGA devices.



Design Entry

Verilog HDL Design Entry

The ispLEVER software supports Verilog HDL, a hardware description language used to design and document electronic systems. Verilog HDL allows designers to design at various levels of abstraction.

All Lattice devices support Verilog HDL design.

Adding a Verilog HDL Module to Your Design

To add a Verilog HDL module to a design, you can either import a `.v` file, or create a new Verilog HDL module file with the Text Editor.

Creating a New Verilog HDL Module

You can use the Text Editor to create a new Verilog HDL module.

To create a new Verilog HDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. Select **Verilog Module** and click **OK**. The Text Editor window appears together with the New Verilog Module dialog box.
3. In the dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new Verilog HDL file appears in the Text Editor window.
5. Use the commands on the Edit menu to Cut, Copy, Paste, Find, or Replace text.

Synthesizing Your Verilog HDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor Graphics LeonardoSpectrum. You can synthesize your Verilog HDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

VHDL Design Entry

VHDL is a language for describing the structure and function of integrated circuits. VHDL allows you to:

- Describe the hierarchical structure and interconnect of a design.
- Specify the function of designs using familiar programming language forms.
- Simulate the design before being manufactured, so that design alternatives can be quickly compared and tested.

All Lattice devices support VHDL design.

Adding a VHDL Module to Your Design

To add a VHDL module to a design, you can either import a `.vhd` file, or create a new VHDL module file with the Text Editor.

Creating a New VHDL Module

You can use the Text Editor to create a new VHDL module.

To create a new VHDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. In the dialog box, select VHDL Module and click **OK**. The Text Editor window appears together with the New VHDL Source dialog box.
3. In the New VHDL Source dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new VHDL file appears in the Text Editor window.
5. Use the commands in the Edit menu to Cut, Copy, Paste, or Replace text.

Synthesizing Your VHDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor graphics LeonardoSpectrum. You can synthesize your VHDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

EDIF Design Entry

The Electronic Design Interchange Format (EDIF) is a format used to exchange design data between different ECAD systems.

The EDIF format is designed to be written and read by computer programs that are constituent parts of EDA systems or tools. Its syntax has been designed for easy machine parsing and is similar to LISP.

The ispLEVER software supports EDIF Version 2 0 0.

All Lattice devices support EDIF design entry.

Importing an EDIF Netlist

You can import a design netlist description into the ispLEVER software from a third-party synthesis or schematic tool if the design file is formatted as EDIF 2 0 0.

***Note**: The project that you are importing the netlist into must be an EDIF project.*

To import an EDIF netlist into your project:

1. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
2. Change Files of type to **EDIF Netlist** (`*.ed*`), and then select the EDIF file that you want to import.
3. Click **Open** to open the Import EDIF dialog box.
4. The default setting for power and ground in the ispLEVER software are the VCC and GND symbols. If you know that the EDIF generated by other tools uses a different convention, you can change it in the window. Select **Custom**. Select either **Symbol** or **Net** representation. Then type the new names for VCC and GND.
5. If you are following the recommendation from Lattice for generating the EDIF file from the supported third party design kit, select **CAE Vendors**. From the list, choose the vendor that generated the EDIF file: Mentor, Synopsys, Synplicity, or Viewlogic.
6. Click **OK**. The software adds the selected EDIF file to the project sources.

Module and IP Design Entry

The Module/IP Manager helps you build large designs using Lattice Parameterized Modules (modules) and intellectual property cores (IPs). Including a module is as easy as:

- Selecting a module or IP core from the module tree
- Specifying parameters in the configuration window
- Instantiating the module or IP core with your text editor.

The Module/IP Manager is fully integrated with every level of the ispLEVER software, simplifying module and IP management within your design project.

Two Approaches to Modules and IP Cores

The ispLEVER software lets you choose the most convenient method for placing parameterized modules and IP cores into your design database. In both methods the software produces the instantiation template and its associated files and saves all of them in the project directory.

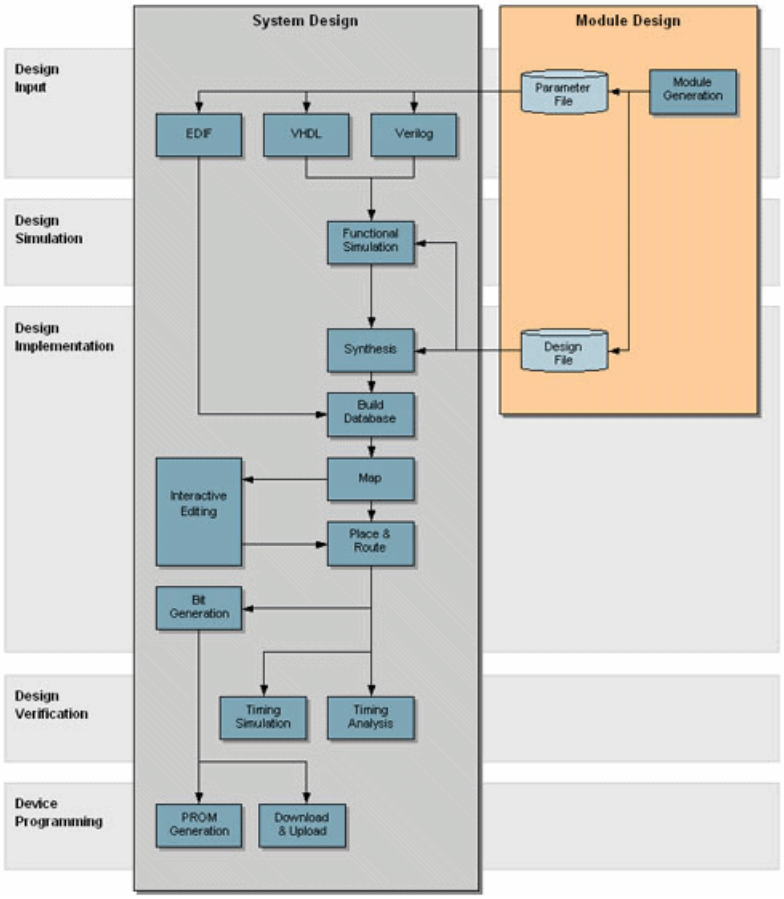
GUI Approach

The Module/IP Manager graphic user interface (GUI) simplifies the process of selecting and configuring a module or IP core. The interface lists all available modules by type and displays the parameters in an easy-to-use dialog box. The GUI approach requires more time for editing a design than the PMI approach.

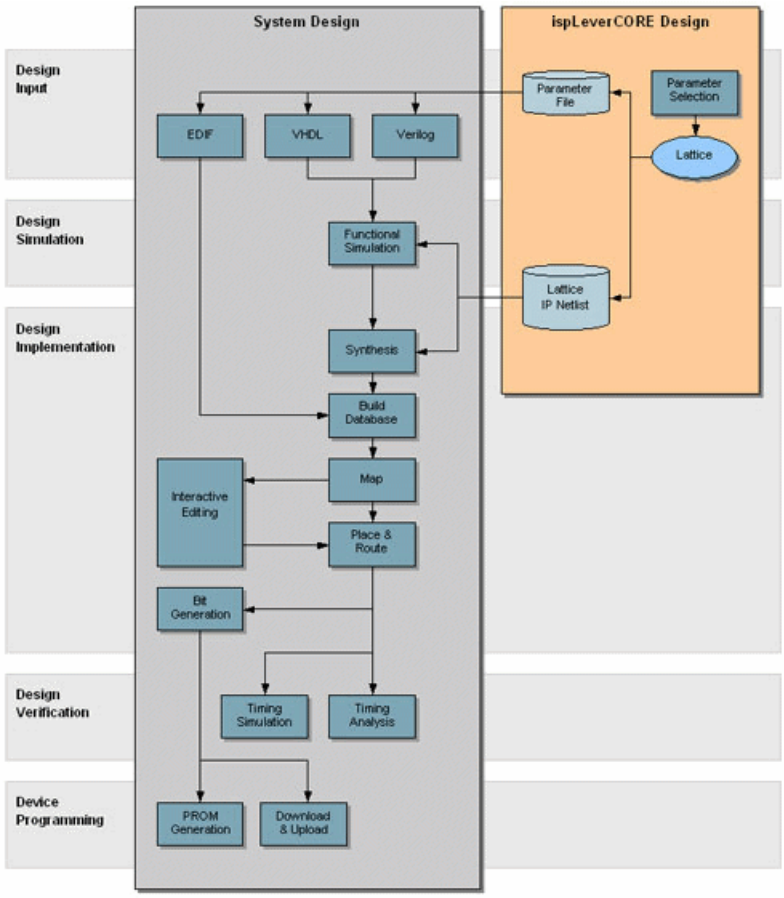
PMI Approach

The Parameterized Module Instantiation template (PMI), though more difficult to use than the GUI, gives you the flexibility of editing a module at any time without having to recreate it. Using the PMI template, you embed the core description of your module into your HDL source files.

FPGA Module Design Process Flow



FPGA ispLeverCORE Design Process Flow



Running SCUBA from the Command Line

Generally, you should generate ORCA FPGA SCUBA-based modules using the Module/IP Manager in ispLEVER because of the complexity of command line options for Series 4 and later architectures. However, you may wish to quickly generate SCUBA modules for older architectures (i.e., Series 2 and Series 3) directly from the command line using command line options. Use this topic as a reference for SCUBA command line syntax.

***Note**: In addition, we also do not recommend using the command line for newer architectures because error checking on command line options does not exist which may cause **scuba** to quit unexpectedly if incorrect options are specified.*

Syntax

scuba [*options*], where *options* are as follows:

Command	Purpose	Module
-arch (orca2a | orca3)	Selects ORCA 2CA/2TA or Series 3 module generators. (Default is orca2a.)	all
-type <*type*>	2CA/2TA: (aspram | sspram | sdpram | rom | mult | addsub | add | sub | comp | lfsr | fifo) Series 3: (sdpram | sspram | rom | fifo | mpippc | mpi960)	each
-a_flow (north | east | south | west)	SCUBA provides positional attributes for improving multiplier performance. SCUBA creates a rectangular block of a multipler by attaching a locational attribute to each component. A_flow specifies the direction of flow of the A input (default is south). Used with **-b_flow** and **-origin**. A_flow and b_flow must be perpendicular to each other. For example, **-a_flow north -b_flow east** is allowed. However, **-a north -b_flow north** or **-a_flow north -b_flow south** are not allowed because both flows are either in the same direction or in opposite directions.	mult
-addr_flow (north | east | south | west)	SCUBA creates a rectangular block of a memory by attaching a locational attribute to each component. Addr_flow specifies the direction of flow of the address input (default is east). Used with **-data_flow** and **-origin**. Data flow and address flow must be perpendicular to each other. For example, **-data_flow north -addr_flow east** is allowed. However, **-data_flow north -addr_flow north** or **-data_flow north -addr_flow south** are not allowed because data flow and address flow are either in the same direction or in opposite directions.	sdpram sspram aspram rom rommult 3C sdpram 3C sspram fifo
-addr_width <*addr_width*>	The address bus width of the module.	sdpram sspram aspram rom 3C sdpram 3C sspram

-b_flow (north | east | south | west)	SCUBA provides positional attributes for improving multiplier performance. SCUBA creates a rectangular block of a multipler by attaching a locational attribute to each component. B_flow specifies the direction of flow of the B input (default is east). Used with **-a_flow** and **-origin**. A_flow and b_flow must be perpendicular to each other. For example, **-a_flow north -b_flow east** is allowed. However, **-a north -b_flow north** or **-a_flow north -b_flow south** are not allowed because both flows are either in the same direction or in opposite directions.	mult
-bb	Uses ("big endian" (BusA[7 to 0]) bus naming convention for VHDL/Verilog output.	all
-bl	Uses "little endian" (BusA[0 to 7]) bus naming convention for VHDL/Verilog output.	all
-buffer	Inserts a buffer for each I/O signal.	all
-bus_exp (1=BusA(0) | 2=BusA_0 | 3=BusA0 | 4=BusA<0> | 5=BusA[0] | 6 = BusA[0 to 7] | 7=BusA(0 to 7))	(For EDIF only) Selects among the 7 different bus expressions styles in EDIF output. Must be used with **-bb**/**-bl** option.	all
-clk	Use QDO outputs of library elements.	rommult 3C sdpram 3C sspram fifo
-depth	The maximum number of words for the FIFO.	fifo
-data_flow (north | east | south | west)	SCUBA creates a rectangular block of a memory by attaching a locational attribute to each component. Data_flow specifies the direction of flow of the address input (default is south). Used with **-addr_flow** and **-origin**. Data flow and address flow must be perpendicular to each other. For example, **-data_flow north -addr_flow east** is allowed. However, **-data_flow north -addr_flow north** or **-data_flow north -addr_flow south** are not allowed because data flow and address flow are either in the same direction or in opposite directions.	sdpram sspram aspram rom rommult 3C sdpram 3C sspram fifo
-data_width *<data_width>*	The data width of the module.	sdpram sspram aspram rom 3C sdpram 3C sspram
-e	Suppresses EDIF output. SCUBA always creates an EDIF file unless this option is used. To create a VHDL or Verilog file, use the **-lang** option.	all
-fastmode	Uses fast mode memory elements.	sdpram sspram
-h	Prints help and exits.	all
-h *<module type>*	Prints help for the specified *module type* and exits.	all
-inv_reset	Makes the reset port of a pipeline multiplier register active LOW. (Default is active HIGH.)	mult rommult

-memfile <*memfile*>	<*memfile*> is a file containing the contents for the module. If a *memfile* is not used, all the RAM contents are initialized to 0. **Note**: All ORCA memories including ORCA5-EM distributed memory are initialized by use of a memfile with the following syntax: **{hex_address} : {hex_data} [{hex_data}...]** However in ORCA5-EM block memory, the syntax is **{data}** or **{hex_data}** depending on the scuba calling line option. The first line of data will go to address 0 of the memory, and following data must be in order. We strongly recommend generating your ORCA5-EM modules from Module/IP Manager.	sdpram sspram aspram rom rommult 3C sdpram 3C sspram
-ne	Use the inverted clock polarity. The top level clock input is inverted and the inverted clock is used to drive the sequential cells in the module.	sdpram sspram mult rommult
-num_rows <*number of rows*>	Specifies the number of rows of memory needed for the module. For example, if you needed 135 rows of memory, you could use the command **-addr_width** 8 (2^7<135<2^8). However, this would generate unnecessary memory rows. Instead, you can use this option to specify the exact number of rows needed.	sdpram sspram aspram rom 3C sdpram 3C sspram
-mem_mux	Use LUT-based mux decoding.	sspram aspram sdpram rom fifo
-origin <*row*> <*column*>	SCUBA creates a rectangular block of a memory or a multiplier by attaching a locational attribute to each component. Origin specifies the position of top left corner of a memory rectangle or a multiplier rectangle. This option is mandatory if LOC attributes need to be added. Used with **-addr_flow** and **-data_flow** (**-a_flow** and **-b_flow** for multipliers).	sdpram sspram aspram rom rommult 3C sdpram 3C sspram fifo
-pe	Threshold for empty flag.	fifo
-pf	Threshold for full flag.	fifo
-pgroup	Write PGROUP properties used with **-addr_flow** and **-data_flow** (**-a_flow** and **-b_flow** origin options.	fifo

-pipeline <*depth*>	Produces a pipeline multiplier (default is a non-pipeline multiplier). Registers are inserted for every <*depth*> multiplication stage, which is represented by the value <*widtha*>. For example, in an 8x8 register: **-pipeline 1** inserts register in every (all 8) multiplication stage. **-pipeline 2** inserts registers in every other multiplication stage. **-pipeline 4** inserts registers in every fourth stage (stage 4 and stage 8). **-pipeline 8** inserts registers in only the eighth stage. <*depth*> must be greater than 0 and less than (<*widtha*> +1) / 2. The registers have a clock port and a reset port.	mult
-pipe_write_path	Pipelines the write path. Adds pipeline registers and FFs to the data in, write address, write enable, and the optional write port enable.	sspram sdpram rom fifo
-pipe_write_path_rep	In addition to what Pipeline Write Path option does, this option individually pipelines the bottom five (or 4 for 2CA/ 2TA) address lines for each row.This reduces the load on these write address lines. Address is referred to as “Pointer” for FIFOs.	sspram sdpram rom fifo
-pipe_mux_levels	This option is available when the -mem_mux option is used. For 2CA/ 2TA, this options does not depend on the address size. When the address size is less than 7, there is only one level of mux. For such cases, the pipeline is between the DEC16X2 output and the mux input. The address lines used for mux select lines are also pipelined. For Series 3, this option can be used only when the address width is greater than 7. In such cases, there are multiple mux levels, and nets between mux stages are pipelined.	sspram sdpram rom fifo
-pipe_read_addr	Pipelines the read address.	sspram sdpram rom fifo
-pipe_read_addr_rep	The pipeline registers for the lower read address bits are replicated for each row of the memory. Address is referred to as “Pointer” for FIFOs.	sspram sdpram rom fifo
-pipe_final_output	Pipelines the final output. There are only one set of registers. When the placement option is used, the data out registers are placed in the bottom row.	sspram sdpram rom fifo
-port (ci | co | overflow | altb | aeqb | agtb | aleb | aneb | ageb | wpe | rden)	Specialized port for the module.	sdpram sspram aspram addsub add sub comp 3C sdpram 3C sspram
-read_clock	This option is used to generate a separate read clock for dual port memory cells. This option can be used only with the **-pipe_final_output** or a valid **-pipe_mux_levels**.	sdpram
-rom	Specifies a ROM-based multiplier. Use with -type mult.	rommult

-signed	Signed number representation. The default is unsigned.	addsub add sub comp
-stage <*stage_width*>	The stage width of a module.	lfsr 3C lfsr
-synth <*vendor*>	Generates HDL output for *synthesis*. Without this option, the HDL output is generated for *simulation*. For synthesis, attributes are embedded in the HDL design for writing out ORCA Foundry attributes in EDIF. The attributes are supported for Synopsys FPGA/Design Compiler, Mentor Graphics LeonardoSpectrum, and Synplicity Synplify.	all
-unsigned	Unsigned number representation.	addsub add sub comp
-v	Prints version number and exits.	all
-value <*constant factor*>	The constant value of the multiplicand.	mult rommult
-w	Overwrites existing files.	all
-width <*width*>	The width of a module.	addsub add sub comp fifo
-widtha <*widtha*>	The bit width of port ‘a’ (multiplier).	mult rommult
-widthb <*widthb*>	The bit width of port ‘b’ (multiplicand).	mult rommult
-n <*circuitname*>	<*circuitname*> is a module name for Verilog or an entity name for VHDL. <*circuitname*> is also an output file name root as follows: • EDIF <*circuitname*>.edn • VHDL <*circuitname*>.vhd • VHDL instance template <*circuitname*>_tmpl.vhd • Verilog <*circuitname*>.v • Verilog instance template <*circuitname*>_tmpl.v • Report file <*circuitname*>.srp	all
-lang <vhdl | verilog>	Creates a VHDL or Verilog file in addition to an EDIF file.	all

Design Simulation

Running a functional simulation after a design description is complete allows you to verify that the description is functionally correct. Also, by simulating the functionality of your design *before* synthesis, you can find and correct basic design errors sooner. While functional simulation will verify your Boolean equations, it does not indicate timing problems.

The ispLEVER software supports functional simulation for Lattice Semiconductor CPLD and FPGA devices using the Lattice Logic Simulator or *ModelSim* from Mentor Graphics. The RTL design can be simulated for functionality before synthesis using the VHDL or Verilog design description and an input stimulus file.

Simulation Environments

The functional simulators operate in both integrated and stand-alone environments.

Integrated Simulation — To simulate a design inside the current project, the ispLEVER software provides integrated simulation. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the simulation source file, the processes in the tables below are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

For ModelSim, ispLEVER includes scripts that run functional simulation automatically using Lattice-defined settings and preferences. You can customize the run by creating a different script files (DO file), which is a simple script that contain commands that are equivalent to the ModelSim GUI commands. This macro is automatically called when you run ModelSim.

For information about creating your own ModelSim macros, see the *ModelSim User's Manual*, *Chapter 11 Tcl and Macros*, provided with your ispLEVER software (Third-Party Manuals).

CPLD and ispGDX Project Navigator Processes	Simulation Tool Invoked
Functional Simulation	Lattice Logic Simulator
Verilog Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Functional Simulation	ModelSim
Verilog Post-Route Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim
VHDL Post-Route Functional Simulation	ModelSim

Stand-alone Simulation — The ispLEVER software supports stand-alone functional simulation. This provides an easy entry if you need to simulate a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator **Tools** menu.

Design File Descriptions

Lattice Logic Simulator and ModelSim enable you to simulate the operation of your design in the following design entry formats:

- ABEL-HDL format (*design*.abl) — a hierarchical logic description language that supports a variety of behavioral input forms, including high-level equations, state diagrams, and truth tables. (CPLD designs only)
- Schematic format (*design*.sch) — describes your circuit in terms of the components used and how they connect to each other. (CPLD designs only)
- VHDL format (*design*.vhd) — Very High Speed IC Hardware Description Language format.
- Verilog HDL format (*design*.v) — an industry-standard hardware description language used to describe the behavior of hardware that can be implemented directly by logic synthesis tools.

The Lattice Logic Simulator also supports mixed design entry as follows:

- Schematic and ABEL-HDL (CPLD designs only)
- Schematic and VHDL
- Schematic and Verilog HDL

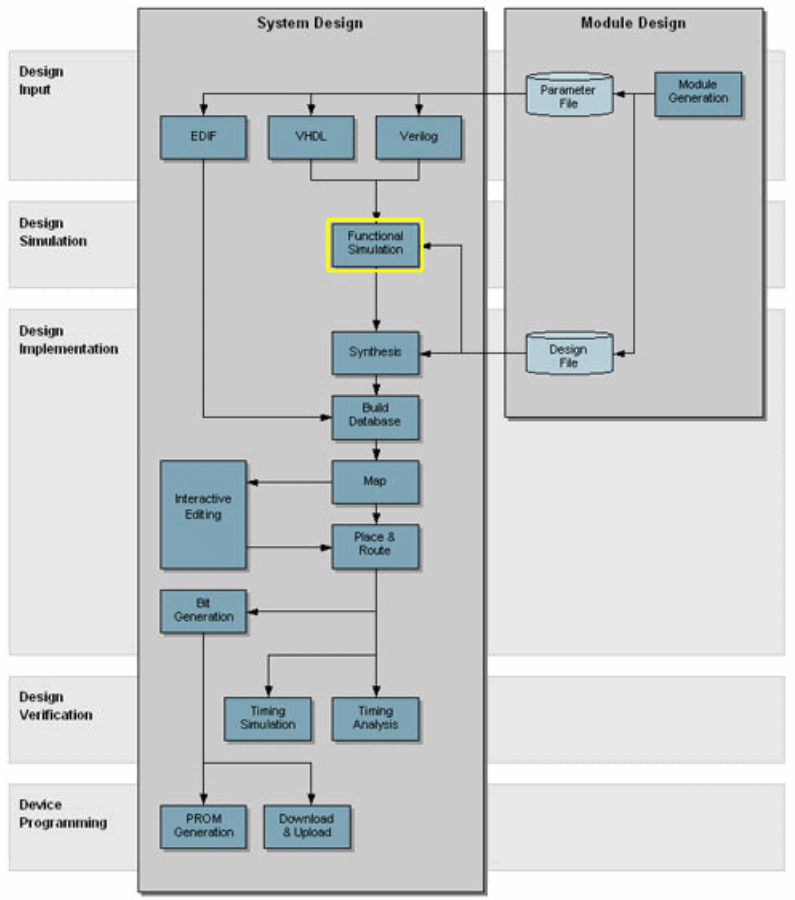
FPGA Test Stimulus Files

The ModelSim simulator supports FPGA functional simulation for HDL design entry methods and requires at least one Verilog test fixture (*.tf) or VHDL test bench (*.vhd) stimulus file.

	ModelSim	
	*.tf	*.vhd
Verilog	X	
VHDL		X

FPGA Simulation Process Flow

The figure below shows functional simulation within the FPGA process flow.



Creating a Verilog Test Fixture from a Template

Verilog test stimulus can be specified either in the top-level HDL source or in a separate test fixture (`.tf`) file. You can create the test fixture manually using a text editor or use a Verilog Test Fixture template (`.tfi`) file.

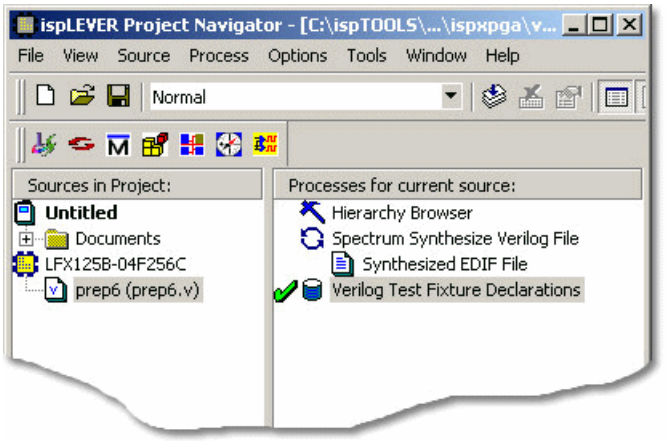
The easiest way to automatically create a Verilog Test Fixture template is using the Project Navigator Verilog Test Fixture Declarations process. After the test fixture template file (`.tfi`) is created, you must add your test vectors and rename it with the extension `.tf` before importing it into your design.

To automatically generate the Verilog test fixture template file and import it into your design:

1. Open your Verilog design in the Project Navigator.
2. In the Sources window, select the top-level Verilog design source (`*.v`) file.

> ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

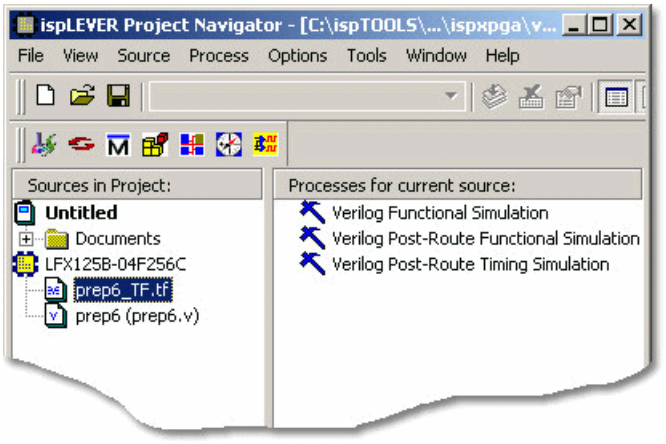
3. In the Processes window, double-click the **Verilog Test Fixture Declarations** process.



   This process creates a template file for a Verilog Test fixture (`<verilog_sourcefile_name>.tfi`). However, in order to use this file as a test fixture in your design, you must edit it and rename it with the extension `.tf`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `prep6_TF.tf`. Add the "`TF`" to the name so that the file will not be overwritten. Change the file extension to "`.tf`" so that it can be imported into the project as a test fixture source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test fixture file. Click **Open**.
7. In the Associate Verilog Test Fixture dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a Verilog Test Fixture. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Creating a VHDL Test Bench from a Template

VHDL test stimulus can be specified either in the top-level HDL source or in a separate test bench (`.vhd`) file. You can create the test bench manually using a text editor or use a VHDL Test Bench template (`.vht`) file.

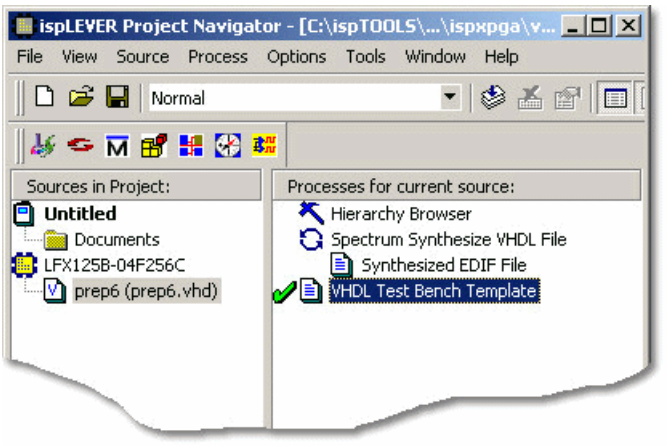
The easiest way to automatically create a VHDL Test Bench template is using the Project Navigator VHDL Test Bench Template process. After the test bench template file is created, you must add your test stimulus and rename it with the extension `.vhd` before importing it into your design.

To generate the VHDL test bench template and import it into your design:

1. Open your VHDL design in the Project Navigator.
2. In the Sources window, select the top-level VHDL design source (`*.vhd`) file.

   ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

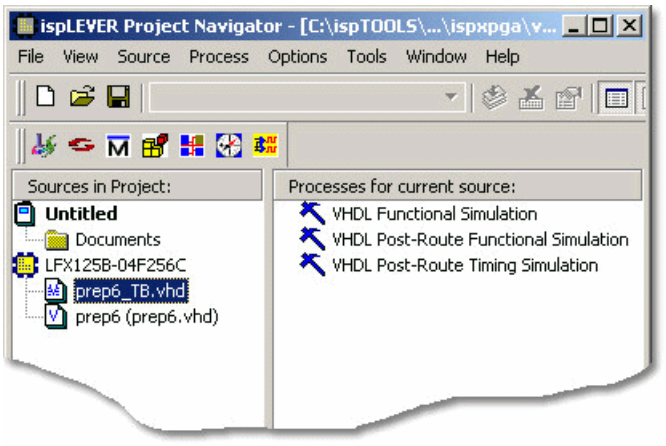
3. In the Processes window, double-click the **VHDL Test Bench Template** process.



   This process creates a template file for a VHDL Test Bench (`<vhdl_sourcefile_name>.vht`). However, to use this file as a test bench in your design, you must edit it and rename it with the extension `.vhd`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `prep6_TB.vhd`. Add the "`TB`" to the name so that the file will not be overwritten. Change the file extension to "`.vhd`" so that it can be imported into the project as a test bench source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test bench file. Click **Open**.
7. In the Import Source Type dialog box, select **VHDL Test Bench** and click **OK**.
8. In the Associate VHDL Test Bench dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a VHDL Test Bench. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Interfacing with ModelSim

ModelSim functional simulation generates several batch files, such as `.fdo`, `.udo`, and `.tdo`. ModelSim users will frequently take advantage of customizing batch files to control their simulation, for example specifying signals to display, run time, and waveform display options.

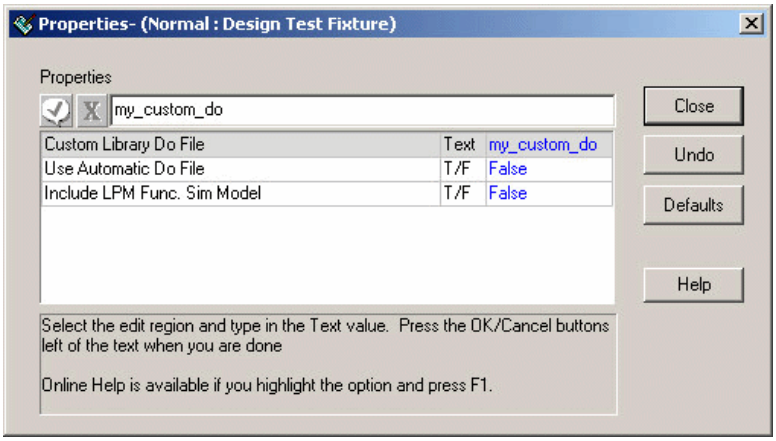
There are two options for creating custom DO files:

Option 1

- In the project folder, edit the `*.udo` file. The Project Navigator will not overwrite this user DO file.

Option 2

1. In the Project Navigator Sources window, select the test bench source.
2. In the Processes window, right-click the functional simulation process to open the Properties dialog box.
3. In the dialog:
   - Type a name for the file in the Custom Library Do File field and click the checkmark icon
   - Set Use Automatic Do File to **False**.
   - Click **Close**.



4. The software will automatically create this file when functional simulation is run. Edit this file as needed.

Design Implementation

Synthesizing

Synthesizing FPGA Designs

For Verilog and VHDL designs, the ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity *Synplify* and Mentor Graphics *LeonardoSpectrum*. You can synthesize your Verilog or VHDL design as a stand-alone process by choosing the synthesis tool from the Lattice Semiconductor program group in your Start menu, or you can synthesize automatically and seamlessly within the Project Navigator.

The ispLEVER Tutorial manual contains synthesis design flow tutorials and is the best place to start if you want to get some hands-on experience. By working through the tutorial lessons, you'll learn how to create sample design projects with some of ispLEVER's most useful and powerful features.

For additional information about synthesis using LeonardoSpectrum or Synplify, see the ispLEVER Third-Party Manuals.

Synthesis Design Flows

You can run the synthesis tools from within the integrated ispLEVER environment, or as a stand-alone process. Whether using LeonardoSpectrum or Synplify, the high-level design flows are basically the same.

Integrated Flow

This approach lets you create, synthesize, import, and implement a design targeting one of the Lattice devices completely from within the ispLEVER Project Navigator environment.

1. Using the Project Navigator, create a new HDL project.
2. Target a device.
3. Using the Text Editor, create the HDL modules.
4. For mixed-mode designs, use the Schematic Editor to create the schematic files.
5. Using the Project Navigator, import the source files.
6. Select a synthesis tool.
7. Fit (Place and Route) the design.

Stand-alone Flow

The stand-alone approach requires you to create or load a VHDL or Verilog HDL design into the synthesis tool environment. Then you synthesize the design and generate an EDIF netlist that you imported into ispLEVER for implementing into a Lattice device.

1. Using your synthesis tool, create a project.
2. Target a device.
3. Load the source files.
4. Synthesize the design to create an EDIF file.
5. Using the ispLEVER Project Navigator, create an EDIF project.
6. Target a device (same as step 2).
7. Import the EDIF source file.
8. Fit (Place and Route) the design.

Integrated Third-Party Tools

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment.

LeonardoSpectrum

Within the LeonardoSpectrum synthesis environment, you can create Lattice device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process.

LeonardoSpectrum from Mentor Graphics combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features.

Synplify

The Synplify solution from Synplicity is a high-performance, sophisticated logic synthesis engine that utilizes proprietary technology to deliver fast, highly efficient FPGA and CPLD designs. Synplify uses Verilog and VHDL Hardware Description Languages as input, and outputs an optimized netlist for the Lattice device.

Selecting the Synthesis Tool

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment. This integrated approach lets you create, synthesize, import, and implement a design targeting a Lattice device completely from within the ispLEVER Project Navigator environment.

To specify the synthesis tool that the ispLEVER software will use:

1. In the Project Navigator, choose **Options > Select RTL Synthesis** to open the dialog box.
2. Select the synthesis tool that you want to use. This tool will be associated with all devices in the current device family. You can also make a synthesis tool the default for all device families.

Building the Database

Building a Lattice Internal Database for an FPGA Design

After you have created a project and imported the design files, the next step is to build an internal database. The input for this database is an EDIF file, which can be generated from a stand-alone synthesis tool and imported into the Project Navigator as your design, or created automatically from a Verilog or VHDL design using the integrated synthesis flow within the ispLEVER software.

When you run the Build Database process, the ispLEVER software converts the EDIF file to a Lattice internal database (`*.ngd`). If the design utilizes parameterized modules and IP cores (Lattice modules and ispLeverCORE IP modules), or user firm macros, the cores are expanded in this process.

Setting Preferences

Setting and Editing Preferences

For FPGA devices, the ispLEVER software supports setting and editing of preference in these ways:

Preference Editor

Many of the FPGA preferences can be edited within the Preference Editor. You can define groups, locations, IO types, as well as various timing preferences for your FPGA design via dialog boxes or directly in the spreadsheets.

To run the Preference Editor:

1. In the Project Navigator Sources window, select the FPGA target device.
2. In the Processes window, do either of the following:
   - Double-click **Pre-Map Preference Editor** to load pre-map preferences.
   - Double-click **Post-Map Preference Editor** to load post-map preferences.
3. The ispLEVER runs the processes and opens the Preference Editor.

FPGA Floorplanner

Using the FPGA Floorplanner after mapping, you have the option of saving one or a combination of constraint types—Pgroups, PIOPgroups, pin assignments, and region assignments—to the design's preference file (`.prf`) using the File > Save Design command. The software then applies your changes to the physical design file (`.ncd`) when you run the Map process or the Place and Route process in the Project Navigator.

To run the FPGA Floorplanner:

1. In the Project Navigator, select the FPGA device in the sources window.
2. In the processes window, perform one of the following actions:
   - Double-click **Pre-Map Logical Design Floorplan**, to build the database and open the Pre-Map Database File (`.ngd`) in the Floorplanner.
   - Double-click **Post-Map Physical Design Floorplan**, to map the design and open the Post-Map Database Map File (`._map.ncd`) in the Floorplanner. This process will also build the database if it has not yet been built.
   - Double-click **Post-PAR Design Floorplan** to place and route the design and open the Post-Map Database File (`.ncd`) in the Floorplanner. This process will also build the database and map the design if these processes have not yet been performed.
3. The ispLEVER software runs the required processes, opens the design, and displays the appropriate windows in the Floorplanner.

Import a Preference File

For those preferences that are not supported by the Preference Editor, you can specify them in a text format Preference file (`*.prf`), and then import it into your FPGA project using the Source > Import Constraint File command in the Project Navigator.

To import an existing PRF file:

1. In the Project Navigator, open the FPGA design you want to work with.
2. Choose **Source > Import Constraint File**.

3. Browse to the PRF file that you want to import, and click **Open**. The selected file will overwrite `<project>.prf` in the project folder.
4. When prompted "Do you want to reset the project update status," click **No** to retain the current process status, or click **Yes** to update it.

Edit an Existing Preference File

You can use the Text Editor (Edit Constraints (ASCII) process) to manually edit the PRF file in your project directory using the text editor of your choice.

You can manually edit the Preference File (`*.prf`) in your project directory by double-clicking the Project Navigator **Edit Constraints (ASCII)** process.

To edit an existing Preference file:

1. In the Project Navigator, open the FPGA design. Make sure the PRF file is already in your project folder.
2. In the Sources window, select the target device.
3. In the Processes window, double-click the **Edit Constraints (ASCII)** process. The `<project.prf>` file opens in the Text Editor.
4. Modify the file as you want, and then save the file.

Mapping

Mapping is the process of converting a design represented as a network of device-independent components (e.g., gates and flip-flops) into a network of device-specific components (e.g., configurable logic blocks).

Using an input EDIF netlist(s), the Map program generates physical descriptions of the logical configuration within the Programmable Logic Cells (PLCs), which include Programmable Function Units (PFUs), Supplemental Logic and Interconnect Cells (SLICs) or tristate buffers (TBUFs), the Programmable Input/Output (PIO) configuration, and special cells (e.g., PCM, GSR, or oscillator).

Depending upon the attributes (i.e., properties) specified in the input netlist, mapping includes absolute placement, logical partitioning (i.e., hierarchical netlists), component group placement, and regional group placement information in the physical description. The resulting physical description is output to a Physical Design file (`.ncd`), in terms of the components in the target architecture. This physical design can then be placed and routed.

You run the Map process from the Project Navigator or from the command line.

Map Input Files

The input to the Map process is an EDIF (Electronic Data Interchange Format) 2 0.0 format netlist. This netlist includes the hierarchy of the input design. The netlist must be a Level 0 EDIF netlist, as defined in the EDIF 2 0.0 specification.

Setting Map Options

From the Project Navigator, Mapping is automatically performed when you run a full design flow, or a partial design flow selecting the Map process.

You should set or change Map options (properties) before running a design flow using the Properties dialog box.

You can also run **map** from the command line.

To set Map options for a design flow in the Project Navigator:

1. In the Project Navigator Sources window, select the **target deice**.
2. In the Processes window, right-click the **Map Design** process and select **Properties** to open the Properties dialog box.
3. Set or change the properties as required.
4. Click **Close** to close the dialog box.

Map Output Files

The following files are created from the Map process.

Physical Design File (`.ncd`) — This file is a physical description of the design in terms of the components in the target device.

Preference File (`.prf`) — This ASCII text file contains the preferences specified during design entry. The preferences in the `.prf` file are expressed in preference language. If no preferences were specified during schematic capture, the Map process creates an empty `.prf` file so that you can enter preferences directly using either a text editor or the EPIC Device Editor.

The Map Report File

The Map report file (`.mrp`) is an ASCII file containing information about the Map command run. The MRP file lists any DRC errors found in the design, details how the design was mapped (e.g., the schematic constraints specified, the logic that was removed or added, and how signals and symbols in the logical design were mapped into signals and components in the physical design).

To view the Map Report File:

- In Project Navigator, right-click on the **Map Report** process in the Processes window and select **View** from the popup menu. The Map report will appear in the output panel at the bottom of the user interface. You can also choose **Window > Report Viewer** to open the Report Viewer and browse for the `.mrp` file in your project directory using the **File > View** command.

  OR

- Use a text editor to open the `.mrp` ASCII file. Search or browse for the file in your project directory.

More on the MRP File

The `.mrp` file also supplies statistics about component usage in the mapped design and contains information about the Map process run. The report is produced whether you have run Map from the Project Navigator or the command line. Although detailed information varies depending on the device to which you have mapped, the format of the file is the same. The report is divided into sections.

Listed below are sections that can appear in an `.mrp` file. Note that depending on the FPGA architecture, these sections can vary.

Design Information — Shows the device to which the design has been mapped and the Map version.

Design Summary — Summarizes the Map run, showing the number of Logical DRC warnings and errors as well as how many of the resources in the target design are used by the mapped device. You will see slightly different elements reported depending in what architecture you are designing. For example, in the ORCA Series 3 Design Summary section could contain reporting on the total number of soft-wired connections, each of which represents a 2-point LUT connection in the PFUs.

DRC Messages — Shows any warnings or errors generated as a result of the Logical DRC tests performed at the beginning of the Map run. These warnings and errors do not depend on the device to which you are mapping.

Mapper Warnings and Errors — Shows any warnings or errors the Map program discovers (for example, a pad is not connected to any logic, or a bidirectional pad is placed in the design but signals only pass in one direction through the pad). These warnings and errors may depend on the device to which you are mapping.

Schematic Attributes — Shows any attributes (properties) specified when the design schematic was created. Some of these attributes also appear as preferences in the preference file that Map produces.

IO (PIO) Attributes — Shows any attributes (properties) specified on PIO logic elements.

Removed Logic — Describes any logic that was removed from the input `.ngd` file when the design was mapped. Logic may be removed because:

- A design uses only part of the logic in a library macro.
- The design has been mapped even though it is not yet complete.
- The Map process has optimized the design logic.
- Unused logic has been created in error during design entry.

This section also indicates which nets were merged (i.e., two nets were combined when a component separating them was removed).

Removed Logic Summary — Summarizes how many blocks and signals were removed from the design. This section reports on these kinds of removed logic:

- Blocks clipped — A “clipped” block is removed because it is along a path that has no driver or no load. Clipping is recursive; that is, if Block A becomes unnecessary because logic to which it is connected has been clipped, then Block A is also clipped.
- Blocks removed — A removed block is removed because it can be eliminated without changing the operation of the design. Removal is recursive; that is, if a Block A becomes unnecessary because logic to which it is connected has been removed, then Block A is also removed.
- Blocks optimized — An “optimized” block is removed because its output remains constant regardless of the state of the inputs (for example, an AND gate with one input tied to ground). Logic generating an input to this optimized block (and to no other blocks) is also removed, and appears in this section.
- Signals removed — Signals that were removed because they were attached only to removed blocks.
- Signals merged — A signal is merged when two signals are combined because a component separating them was removed.

Added Logic — Describes any logic that the Map program added to the design.

Expanded Logic — Describes the mapping of logic that has been added to the database to resolve blocks.

Symbol Cross Reference — Shows where symbols in the logical design were mapped in the physical design.

Signal Cross Reference — Shows where nets in the logical design were mapped in the physical design.

Physical Design Errors and Warnings — Shows errors and warnings associated with problems in assigning components to sites, for example, an attribute on a component specifies that the component must be assigned to site AK, but there is no site AK on the part to which you are mapping. Another example would be a problem in fitting the design to the part where the design maps to 224 logic blocks, but the part only contains 192 logic block sites.

Sample Map Report File - ORCA Series 2

```
               Lattice Mapping Report File for Design 'BIST31'

DRC Messages
------------
WARNING:  logical net 'QMRN' has no load
WARNING:  logical net 'CKN' has no load
WARNING:  logical net 'VN11' has no load
***truncated here for example purposes***


Design Information
------------------
Command line:  c:\isptools\ispfpga\bin\nt\map -a or2c00a -p or2c15a -t M84 -s 4
     series2.ngd -o series2.ncd -f series2.m2t
Target Vendor:  LATTICE
Target Device:  or2c15aM84
Target Speed:   4
Mapper:  or2c00a,  version:  ispLever_v31_Production_Build_Classic (67c)
Mapped on:  11/26/03  11:51:11


Design:  BIST31                                        Date:  11/26/03  11:51:11

Schematic Attributes
--------------------
LOCATION
   "91" on symbol CK_IN.PAD
   "106" on symbol MR_IN.PAD
   "108" on symbol MC1_IN.PAD
   "112" on symbol MC2_IN.PAD
   "117" on symbol MC3_IN.PAD
   "121" on symbol MC4_IN.PAD
   "146" for signal LD_TG_O on symbol LD_TG_O.PAD
   "151" for signal LD_TY_O on symbol LD_TY_O.PAD
   "148" for signal LD_TR_O on symbol LD_TR_O.PAD
   "142" for signal LD_BY_O on symbol LD_BY_O.PAD
```

```
Removed Logic
-------------
Block V11/NEOBUF (NEOINV) undriven or does not drive anything - clipped.
Block S8/SA1/NEOBUF (NEOINV) undriven or does not drive anything - clipped.
Block S8/SA4/NEOBUF (NEOINV) undriven or does not drive anything - clipped.
***truncated here for example purposes***

Removed Logic Summary
---------------------
  42 Blocks clipped
  79 Blocks removed
   6 Blocks optimized away
  42 Signals removed
  79 Signals merged

Expanded Logic
--------------
V11 (FD1S3AX) expanded into these physical comps:  BIT4R2 (PFU), BIT8R2 (PFU),
     V0 (PFU), V1 (PFU), V3 (PFU), V4 (PFU), V8 (PFU)
T4 (CB4) expanded into these physical comps:  T_6 (PFU)
***truncated here for example purposes***

Signal Cross-Reference
----------------------
Signal ALL_CL - Driver Comp: SAND_0/I1
   Load Comp: LD_TY_O
Signal BIT4R2 - Driver Comp: V0
   Load Comp: V5
Signal CK - Driver Comp: QCK1
   Load Comp: QCK1
Signal CKIN - Driver Comp: CK_IN
   Load Comp: QMR
Signal CKSA - Driver Comp: QCK1
   Load Comps: SA_10, SA_13, SA_17, SA_2, SA_21, SA_25, SA_29, SA_3, SA_33,
        SA_37, SA_41, SA_45, SA_49, SA_52, SA_54, SA_56, SA_58, SA_6,
***truncated here for example purposes***
```

```
Symbol Cross-reference
----------------------
BIT10R1 (XNOR2) mapped to:  BIT4R2 (PFU), BIT8R2 (PFU), V0 (PFU), V1 (PFU), V3
(PFU), V4 (PFU), V8 (PFU)
BIT10R2 (OR2) mapped to:  BIT4R2 (PFU), BIT8R2 (PFU), V0 (PFU), V1 (PFU), V3
(PFU), V4 (PFU), V8 (PFU)
BIT11R1 (XNOR2) mapped to:  BIT4R2 (PFU), BIT8R2 (PFU), V0 (PFU), V1 (PFU), V3
(PFU), V4 (PFU), V8 (PFU)
***truncated here for example purposes***

Detailed logic replication cross reference
------------------------------------------
V0 (FD1S3AX) expanded into these physical comps:  V0(PFU)
V6 (FD1S3AX) expanded into these physical comps:  V0(PFU)
BIT6R1 (XNOR2) expanded into these physical comps:  V0(PFU)
BIT4R2 (OR2) expanded into these physical comps:  BIT4R2(PFU)
T3 (AND5) expanded into these physical comps:  BIT4R2(PFU)
***truncated here for example purposes***

Physical Design Errors and Warnings
-----------------------------------
WARNING:  Specified site name "91" is not valid.  Location directive ignored.
WARNING:  Specified site name "106" is not valid.  Location directive ignored.
WARNING:  Specified site name "108" is not valid.  Location directive ignored.
***truncated here for example purposes***

Design Summary
--------------
   Number of warnings:  52
   Number of errors:    0
   Number of QLUTs used for combinational logic:    116
   Number of QLUTs used in ripple mode:    48
   Number of HLUTs used:    50
   Number of HLUTs used in >= 6 input functions:    2
   Number of FFs used:    105
   Number of PFUs in design:  80 out of 400
   Number of internal PICs in design: 0 out of 256
   Number of external PICs in design: 10 out of 64
   Number of TBUFs in design: 4 out of 3200
   Number of RESETN in design: 0
   Number of RDCFG in design: 0
```

```
   Number of JTAG in design: 0
   Number of OSC in design: 0
   Number of GSRs in design: 1
   Number of startup in design: 0
   Number of readback in design: 0
   Number of macros in design: 0


Copyright (c) 1991-1994 by NeoCAD Inc. All rights reserved.
Copyright (c) 1995
     AT&T Corp.   All rights reserved.
Copyright (c) 1995-2001 Lucent
     Technologies Inc.  All rights reserved.
Copyright (c) 2001 Agere Systems
     All rights reserved.
Copyright (c) 2002-2003 Lattice Semiconductor
     Corporation,  All rights reserved.
```

Sample Map Report File - ORCA Series 3

```
                Lattice Mapping Report File for Design 'BIST34'

Design Information
------------------
Command line:   c:\isptools\ispfpga\bin\nt\map -a or3c00 -p or3c55 -t S208
     -s 5 series3.ngd -o series3.ncd -f series3.m2t
Target Vendor:  LATTICE
Target Device:  or3c55S208
Target Speed:   5
Mapper:  or3c00,  version:  ispLever_v31_Production_Build_Classic (67c)
Mapped on:  11/26/03  12:08:11

Design Summary
--------------
   Number of warnings:  0
   Number of errors:    0
   Number of registers:    126
      PFU registers:    126
   Number of PFUs:  45 out of 324 (13%)
   Number of LUT4s: 226
```

```
   Number of ripple logic:  14
   Number of Softwired Connections:  27
   Number of external PIOs: 9 out of 167 (5%)
   Number of SLICs:  13 out of 324 (4%)
   Number of Programmable Clock Managers:  0 out of 2 (0%)
   Number of CLKCNTLs in design:  0 out of 4 (0%)
   Number of GSRs:  1 out of 1 (100%)
   MPI used: No
   JTAG used :  No
   Readback used :  No
   Oscillator used :  No
   Startup used :  No
   Number of clocks:  4
     Net CKVG: 13 loads, 4 NCD loads (Driver: CKVG )
     Net CKSA: 110 loads, 15 NCD loads (Driver: CKSA )
     Net CKIN: 2 loads, 3 NCD loads (Driver: PIO CK_IN )
     Net GCK: 4 loads, 3 NCD loads (Driver: GCK )

IO (PIO) Attributes
-------------------
+---------------------+-----------+-----------+------------+
| IO Name             | Direction | Levelmode | IO         |
|                     |           |           | Register   |
+---------------------+-----------+-----------+------------+
| CK_IN               | INPUT     | TTL       |            |
+---------------------+-----------+-----------+------------+
| LD_BY_O             | OUTPUT    | CMOS      |            |
+---------------------+-----------+-----------+------------+
| MC1_IN              | INPUT     | CMOS      |            |
+---------------------+-----------+-----------+------------+
| MC2_IN              | INPUT     | CMOS      |            |
+---------------------+-----------+-----------+------------+
| MC3_IN              | INPUT     | CMOS      |            |
+---------------------+-----------+-----------+------------+
***truncated here for example purposes***
```

```
Removed logic
-------------
Block S9/SA8/NEOBUF undriven or does not drive anything - clipped.
Block S9/SA6/NEOBUF undriven or does not drive anything - clipped.
Block S9/SA3/NEOBUF undriven or does not drive anything - clipped.
***truncated here for example purposes***

Symbol Cross Reference
----------------------
PFU_0 (PFU) covers blocks: T27/LUT1, T27/LUT2, T27/LUT3, T0
PFU_1 (PFU) covers blocks: T1
PFU_2 (PFU) covers blocks: T2
***truncated here for example purposes***

Signal Cross Reference
----------------------
Signal CKVG - Driver Comp: PFU_29:F2
   Load Comps: PFU_44:CLK, PFU_45:CLK, PFU_46:CLK
Signal V12 - Driver Comp: PFU_46:Q6
   Load Comps: PFU_1:K4_1, PFU_2:K4_1, PFU_3:K4_1, PFU_4:K4_1, PFU_5:K4_1,
        PFU_6:K4_1, PFU_7:K4_1, PFU_8:K4_1, PFU_9:K4_1, PFU_10:K4_1,
        PFU_46:K6_0, PFU_46:K7_0, PFU_47:K5_0, PFU_48:K2_3
***truncated here for example purposes***


Copyright (c) 1991-1994 by NeoCAD Inc. All rights reserved.
Copyright (c) 1995
     AT&T Corp.   All rights reserved.
Copyright (c) 1995-2001 Lucent
     Technologies Inc.  All rights reserved.
Copyright (c) 2001 Agere Systems
     All rights reserved.
Copyright (c) 2002-2003 Lattice Semiconductor
     Corporation,  All rights reserved.
```

Sample Map Report File - Series 4

```
                Lattice Mapping Report File for Design 'pcs'

Design Information
------------------
Command line:   map -s 5 -a or5s00 -p or5s25 -t BM600 pcs.ngd -o pcs.ncd
pcs.prf
Target Vendor:  LATTICE
Target Device:  or5s25BM680
Target Speed:   5
Mapper:  or5s00,  version:  ispLever_v40_Alpha_Build_Classic (41c)
Mapped on:  11/17/03  13:29:34

Design Summary
--------------
   Number of warnings:  1
   Number of errors:    0
   Number of registers:    2718
      PFU registers:    2718
   Number of SLICEs:  2332 out of 12712 (18%)
     Number of LUT4s: 2999
     Number of ripple logic:  40
   Number of external PIOs: 153 out of 520 (29%)
   Number of PLLs:  0 out of 8 (0%)
   Number of Block RAMs:  8 out of 116 (6%)
   Number of GSRs:  1 out of 1 (100%)
   System Bus used :  No
   JTAG used :  No
   Readback used :  No
   Oscillator used :  No
   Startup used :  No
   Number of clocks:  2
     Net clk_156mhz_c: 1455 loads, 1455 rising, 0 falling (Driver: PIO
     clk_156mhz )
     Net clk_161mhz_c: 340 loads, 340 rising, 0 falling (Driver: PIO
     clk_161mhz )
   Number of Clock Enables:  21 (***would show 21 lines below***)
     Net aligner_0/un1_idle_i: 33 loads, 33 SLICEs
     Net aligner_0/idleZ0: 2 loads, 2 SLICEs
```

```
   Number of LSRs:  35
     Net sys_reset_z_c: 4 loads, 4
***truncated here for example purposes***

IO (PIO) Attributes
-------------------
+----------------------+-----------+-----------+------------+
| IO Name              | Direction | Levelmode | IO         |
|                      |           |           | Register   |
+----------------------+-----------+-----------+------------+
| dec6466_data_64b_0   | OUTPUT    | LVCMOS33  |            |
+----------------------+-----------+-----------+------------+
                                   Page 2

Design:  pcs                                        Date:  11/17/03    13:30:14

IO (PIO) Attributes (cont)
--------------------------
| eb_rd_data_0         | INPUT     | LVCMOS33  |            |
+----------------------+-----------+-----------+------------+
| syncerr_tx           | OUTPUT    | LVCMOS33  |            |
+----------------------+-----------+-----------+------------+
| syncerr_rx           | OUTPUT    | LVCMOS33  |            |
+----------------------+-----------+-----------+------------+
| hi_ber               | OUTPUT    | LVCMOS33  |            |
+----------------------+-----------+-----------+------------+
| rx_frm_lock          | OUTPUT    | LVCMOS33  |            |
+----------------------+-----------+-----------+------------+
***truncated here for example purposes***

Removed logic
-------------
Block GND undriven or does not drive anything - clipped.
Block VCC undriven or does not drive anything - clipped.
Block scram_0/GND undriven or does not drive anything - clipped.
Block tx_gearbox_0/write_cnt7_4/RD4P3I/REG3 undriven or does not drive anything
     - clipped.
Block tx_gearbox_0/write_cnt7_4/RD4P3I/REG2 undriven or does not drive anything
     - clipped.
```

```
***truncated here for example purposes***

Symbol Cross Reference
----------------------
SLICE_0 (PFU) covers blocks: aligner_0/CHK_CNT/REGBLOCK0/REG0,
     aligner_0/CHK_CNT/REGBLOCK0/REG1, aligner_0/CHK_CNT/INC0
SLICE_1 (PFU) covers blocks: aligner_0/CHK_CNT/REGBLOCK0/REG2,
     aligner_0/CHK_CNT/REGBLOCK0/REG3, aligner_0/CHK_CNT/INC1
SLICE_2 (PFU) covers blocks: aligner_0/CHK_CNT/REGBLOCK1/REG0,
     aligner_0/CHK_CNT/REGBLOCK1/REG1, aligner_0/CHK_CNT/INC2
SLICE_3 (PFU) covers blocks: aligner_0/CNT_16/RD4P3I/REG0,
     aligner_0/CNT_16/RD4P3I/REG1, aligner_0/CNT_16/INC0
***truncated here for example purposes***

Signal Cross Reference
----------------------
Signal dec6466_data_64b_c_0 - Driver Comp: SLICE_287:O4
   Load Comps: dec6466_data_64b_0:I0
Signal eb_rd_data_c_0 - Driver Comp: eb_rd_data_0:O0
   Load Comps: SLICE_756:I6, SLICE_827:I6, SLICE_869:I4, SLICE_901:I6,
        SLICE_2337:I0
***truncated here for example purposes***

Physical Design Errors and Warnings
----------------------
WARNING: Specified site name "54" is not valid. Location directive ignored.


Copyright (c) 1991-1994 by NeoCAD Inc. All rights reserved.
Copyright (c) 1995
     AT&T Corp.   All rights reserved.
Copyright (c) 1995-2001 Lucent
     Technologies Inc.  All rights reserved.
Copyright (c) 2001 Agere Systems
     All rights reserved.
Copyright (c) 2002-2003 Lattice Semiconductor
     Corporation,  All rights reserved.
```

Mapping Considerations - ORCA Series 3 PIO

For each input, output, or bidirectional buffer, a PIO is created. If bidirectional functionality is required, a bidirectional library element should be instantiated. For example, you should instantiate a BMZ element instead of an OBUFZ and an IBM element. The current release of the mapper does not support piecing together IO buffers. Register ordering applies.

Mapping Considerations - ORCA Series 3 PCM

To use the dedicated PCM pad, any input buffer can be used. A LOC attribute is required to locate the buffer at the proper site.

Because the PCM generates a signal (which is register clock), the characteristics of the signal are required for timing analysis. The following describes how the timing preferences are calculated for each mode.

DLL1X Mode

Eclk and Sclk have the same input frequency as the input clock, but have new duty cycles given by duty data.

- Input data - Frequency, duty
- Fsclk = Fin
- Feclk = Fin
- T = 1/Fin, ns
- duty_hi = T*duty/100.0; ns

Preference generated is DP_PERIOD_ NET with DUTY HIGH data.

DLLPD Mode

Sclk frequency is the same as the input clock but is phase shifted by PDELAY. Eclk frequency is the input clock frequency divided by DIV2 and is phase shifted by PDELAY. This mode assumes a 50 percent duty cycle.

- Input data - Frequency, PDELAY, DIV2
  Feclk = Fin/DIVIDER2, both clocks delayed by DELAY MHz
  Fsclk = Fin MHz
- For Sclk:
  Tin = 1/Fin;
- For PDELAY <= 16
  duty_hi = Tin/2; ns
  duty_low = Tin * PDELAY/32.0; ns
- For PDELAY > 16
  duty_hi = (Tin*PDELAY/32.0) – Tin/2; ns
  duty_low = Tin/2; ns

Preference generated is DP_PERIOD_NET with DUTY HIGH data and DUTY LOW data.

- For Eclk:
- For PDELAY <= 16
  T = 1/(Fin/DIV2);
  duty_hi = T/2; ns
  duty_low = Tin * PDELAY/32.0; ns

- For PDELAY > 16
  duty_hi = (Tin*PDELAY/32.0) + (Tin*DIV2)/2 – Tin; ns
  duty_low = T/2; ns

Preference generated is DP_PERIOD_NET with DUTY HIGH data and DUTY LOW data.

PLL Mode

Eclk and Sclk have frequencies that are a fraction of the input frequency.

- Input data - SCLK, DIV0, DIV1, DIV2
- Feclk = SCLK frequency/DIV2
- Fsclk = (DIV1/DIV0)*DIV2*Fin MHz

Preference generated is DP_FREQUENCY_NET.

PCM-Related Attributes

USER_ID — USER_ID is a string of 1s and 0s for JTAG.

PDELAY — Phase delay of a clock waveform, integer range 0..31. Used in DLLPD mode of PCM, default 1.

DIV0, DIV1, DIV2 — DIV0, DIV1, DIV2 are integer values, 1..8, default is 1. Used in DLLPD and PLL more of PCM.

DUTY — Duty cycle, floating point percentage values. For the DLL1X mode of PCM.

Mapping Considerations - ORCA Series 3 MPI

When using a Microprocessor Interface (MPIi960 or MPIPC), the synthesis tool cannot generate inverters in the DDRVCTL or in the TANTS connection paths. Remove the inverters from the netlist and connect DDRVCTL or TANTS directly to the tristate enable of the output IO buffers.

Extended SWL Formation

The software can recognize more structures for 3-level soft-wired LUTs than in previous versions of ORCA Foundry. The root LUT of three-level SWLs can now be a LUT4 or a LUT5, while in previous software this root LUT had to be a LUT5. This SWL extension can improve timing performance on many designs. Because more SWL connections will be found, it is possible that a given design's area requirements will increase.

***Note**: The default in Map is to optimize for timing. Therefore, more 3-level SWLs are being used which results with an area growth of ~5% on average.*

Mapping Considerations - ORCA Series 4

The items listed in this section are features Map currently supports for the creation of Series 4 designs. Register ordering applies. Please refer to series 2 for register ordering naming conventions.

PFU Items Supported

- All Series 3 features are supported (e.g., register ordering, SWL patterns, and density control) including Guided Map.
- Series 4 packing features:
  - Support nibble-wide packing by default

- 2 CEs allowed per PFU
- 9th FFs share the resources with lower half
- 2 CLKs, LSRs, SELs allowed in register ordering, registers with constraints and other structured cases
- More soft-wired LUT patterns are supported due to the addition of LUT6 library elements
- Command Line option: **-c**
- Single FFs with different control inputs may be merged to meet -c requirement (except for 9th FF)

Map Usage Guidelines - ORCA Series 4

When mapping from the command line, ensure that you follow these guidelines:

- When retargeting Series 3 designs to the Series 4, make sure that the `.edn` file does *not* have the LIBRARY attribute set to **or3c00**.
- Do not run the Series 4 mapper with a Series 3 `.ngd` file without running **edif2ngd** using the **-l or4e00** option to search the appropriate library.
- All the existing Series 3 netlists that have any of the following elements will not work with the Series 4 flow:

  CLKCNTLB, CLKCNTLL, CLKCNTLR, CLKCNTLT, CLKCNTHB, CLKCNTHL, CLKCNTHR, CLKCNTHT, PCMB, and PCMT

If a design contains the CLKCNTL and CLKCNTH elements, remove the element(s) and connect the CLKIN and CLKOUT signals together.

***Note**: The connectivity of OSMUX21 and OEMUX21 elements in Series 4 architecture will be different from Series 3 architecture. In Series 4 the D1 input of the elements can only come from the output register. In Series 3 both inputs D0 and D1 were driven from PFU.*

Mapping PCM to ORCA Series 4

When a design being mapped to ORCA Series 4 contains any of the following ORCA Series 3 Programmable Clock Manager (PCM) elements, use one of the two options described below to get the appropriate behavior.

DLL1XB, DLL1XT, DLLPDB, DLLPDT, PCMBUFB, PCMBUFT, PLLB, and PLLT

Option 1

Instantiate the PPLL element in place of the above mentioned elements and pass the appropriate attributes to get the desired behavior. Possible inputs/ outputs and attributes for these elements are listed on the following pages.

In Series 3, the VCO has 32 taps and in Series 4, it has eight taps. So, the DUTY attribute for DLL1XB and DLL1XT elements has to be replaced by VCOTAP attribute with appropriate value to reflect the actual duty-cycle. Similarly, the PDELAY attribute for DLLPDB and DLLPDT elements should be replaced by the VCOTAP attribute with appropriate value.

For DLL1XB: (FB and INTFB to be tied together)	
INPUTS	CLKIN, FB
OUTPUTS	MCLK, NCLK, LOCK, INTFB
ATTRIBUTES	DIV0: 1, 2, 3, 4, 5, 6, 7, 8 MCLKMODE: DUTYCYCLE NCLKMODE: DUTYCYCLE VCOTAP: 0, 1, 2, 3, 4, 5, 6, 7 DISABLED_GSR: 0, 1

For DLL1XT: (FB and INTFB to be tied together)	
INPUTS	CLKIN, FB
OUTPUTS	MCLK, NCLK, LOCK, INTFB
ATTRIBUTES	DIV0: 1, 2, 3, 4, 5, 6, 7, 8 MCLKMODE: DUTYCYCLE NCLKMODE: DUTYCYCLE VCOTAP: 0, 1, 2, 3, 4, 5, 6, 7 DISABLED_GSR: 0, 1

For DLLPDB: (FB and INTFB to be tied together)	
INPUTS	CLKIN, FB
OUTPUTS	MCLK, NCLK, LOCK, INTFB
ATTRIBUTES	DIV0: 1, 2, 3, 4, 5, 6, 7, 8 DIV2: 1, 2, 3, 4, 5, 6, 7, 8 MCLKMODE: DELAY NCLKMODE: DELAY VCOTAP: 0, 1, 2, 3, 4, 5, 6, 7 DISABLED_GSR: 0, 1

For DLLPDT: (FB and INTFB to be tied together)	
INPUTS	CLKIN, FB
OUTPUTS	MCLK, NCLK, LOCK, INTFB
ATTRIBUTES	DIV0: 1, 2, 3, 4, 5, 6, 7, 8 DIV2: 1, 2, 3, 4, 5, 6, 7, 8 MCLKMODE: DELAY NCLKMODE: DELAY VCOTAP: 0, 1, 2, 3, 4, 5, 6, 7 DISABLED_GSR: 0, 1

For PCMBUFB:	
INPUTS	CLKIN, FB
OUTPUTS	MCLK, NCLK, LOCK, INTFB
ATTRIBUTES	MCLKMODE: BYPASS NCLKMODE: BYPASS

For PCMBUFT:	
INPUTS	CLKIN, FB
OUTPUTS	MCLK, NCLK, LOCK, INTFB
ATTRIBUTES	MCLKMODE: BYPASS NCLKMODE: BYPASS

For PLLB:	
INPUTS	CLKIN, FB
OUTPUTS	MCLK, NCLK, LOCK, INTFB
ATTRIBUTES	DIV0: 1, 2, 3, 4, 5, 6, 7, 8 DIV1: 1, 2, 3, 4, 5, 6, 7, 8 DIV2: 1, 2, 3, 4, 5, 6, 7, 8 MCLKMODE: DELAY NCLKMODE: DELAY DISABLED_GSR: 0, 1

For PLLT:	
INPUTS	CLKIN, FB
OUTPUTS	MCLK, NCLK, LOCK, INTFB
ATTRIBUTES	DIV0: 1, 2, 3, 4, 5, 6, 7, 8 DIV1: 1, 2, 3, 4, 5, 6, 7, 8 DIV2: 1, 2, 3, 4, 5, 6, 7, 8 MCLKMODE: DELAY NCLKMODE: DELAY DISABLED_GSR: 0, 1

In Series 3, ECLK feedback is same as MCLK feedback in Series 4. Similarly, Series 3 SCLK feedback is same as NCLK feedback in Series 4.

Option 2

You can keep the same Series 3 library elements but replace the DUTY attribute for DLL1XT and DLL1XB and replace the PDELAY attribute for DLLPDB and DLLPDT elements with attribute VCOTAP with the appropriate values in the EDIF netlist. For PLLT and PLLB elements, you have to connect the FB input pin.

Map PGROUP/UGROUP Support

PGROUP/UGROUP and its related attributes facilitate the partitioning of the logical design. UGROUP use is intended for grouping blocks in hierarchies or no hierarchy. Refer to the preference section in the Preferences chapter for general information and user-defined PGROUP functionality in the physical preference file.

PGROUP/UGROUP and Related Attributes

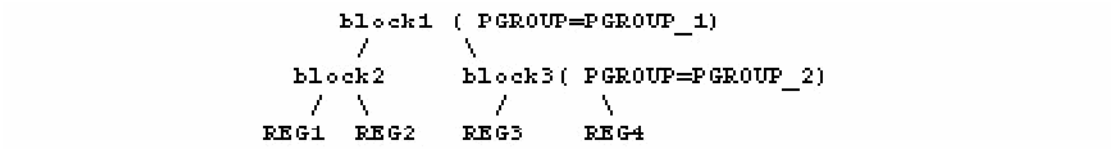
PGROUP/ UGROUP	Indicates a logical partition of a group of components that gets translated into a physical partition for PAR. Syntax: PGROUP = identifier UGROUP = identifier
PLOC	Indicates the physical location of the northwest corner of the PGROUP/UGROUP. This should be attached to same block where PGROUP/UGROUP is specified. This attribute must appear on same block as PGROUP/UGROUP. Syntax: PLOC = <row#col#>
PBBOX	Indicates the Bounding box or the area given in number of rows and columns for a given PGROUP/UGROUP. This attribute must appear on the same block as the PGROUP/UGROUP attribute. Syntax (where H is height, W is width): PBBOX = H,W
PREGION	Indicates the region to which a PGROUP/UGROUP belongs. This attribute must appear on a block that has a PGROUP/ UGROUP attribute. Syntax: PREGION = identifier
PRLOC	Indicates the northwest corner of a region. This attribute must appear on the same block as PREGION. Required if PREGION exists. Syntax: PRLOC = <row#col#>
PRBBOX	Indicates the area size a region. This attribute must appear on the same block as PREGION. Required if PREGION exists. Syntax (where H is height, W is width): PRBBOX = H,W
LOC	Indicates relative locations of PFUs within a PGROUP/ UGROUP. Recommended for use on non-Boolean blocks, e.g. RAMS, registers, and ripple. Syntax: LOC = <*row#col#*> for PFUs For 3C/T/L and 4E family SLICs: LOC = <*SLIC_row#col#*> For 2C/A/T family tristates: LOC = <TRI.row#col#.bank#>
COMP	Optional.

PGROUP Attribute Usage Guidelines

Use the PGROUP attribute on blocks that are to be instantiated multiple times. In addition, observe the following guidelines when using PGROUP attributes:

- PGROUP attribute should appear in the EDIF netlist on high-level blocks.
- All elements within the PGROUP'd block belong to that particular PGROUP.
- Nested PGROUP blocks are considered as unique individual PGROUPs.
- All blocks belong to the PGROUP attached to their nearest ancestor in the hierarchy.
- Multiply instantiated blocks with the same PGROUP are allowed.
- The mapper appends the hierarchical path plus the block instance name to the PGROUP identifier to generate the complete PGROUP name.

For example:



then,

REG1 and REG2 belong to PGROUP_1.
REG3 and REG4 belong to PGROUP_2.

Also, assuming REG1, REG2 are mapped to PFU_0 and REG3, REG4 are mapped to PFU_1, the resulting preferences generated in the preference file are as follows:

```
PGROUP "block1/PGROUP_1"
COMP PFU_0;
PGROUP "block1/block3/PGROUP_2"
COMP PFU_1;
```

If block1 has a PGROUP that is also located, PLOC = R1C1, then the additional preference,

```
LOCATE PGROUP "block1/PGROUP_1" SITE "R1C1"
```

appears in the preference file.

If block1 has a PREGION=ONE, PRLOC=R1C1, PRBBOX=2,2 then the preferences generated are as follows:

```
REGION "ONE" "R1C1" 2 2;
PGROUP "block1/PGROUP_1"
COMP PFU_0;
PGROUP "block1/block3/PGROUP_2"
COMP PFU_1;
LOCATE PGROUP "block1/PGROUP_1" REGION "ONE"
LOCATE PGROUP "block1/block3/PGROUP_2" REGION "ONE"
```

Notice that both PGROUPs are in REGION ONE.

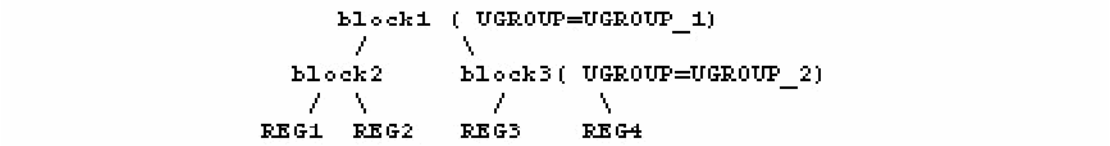
UGROUP Attribute Usage Guidelines

Use the UGROUP attribute to group blocks in different hierarchies or no hierarchy. UGROUP differs from PGROUP in that pre-appending the hierarchy and the block instance does not change its identifier.

Observe the following conditions for proper UGROUP attribute usage:

- All elements within the UGROUP'd block belong to that particular UGROUP.
- Nested UGROUP blocks are considered as unique individual UGROUPs.
- All blocks belong to the UGROUP attached to their nearest ancestor in the hierarchy.
- Unlike PGROUP, the mapper will not append the hierarchical path plus the block instance name.

For example:



then,

REG1 and REG2 belong to UGROUP_1.
REG3 and REG4 belong to UGROUP_2.

Also, assuming REG1, REG2 are mapped to PFU_0 and REG3, REG4 are mapped to PFU_1, the resulting preferences generated in the preference file are as follows:

```
PGROUP "UGROUP_1"
COMP PFU_0;
PGROUP "UGROUP_2"
COMP PFU_1;
```

If block1 has a UGROUP that is also located, PLOC = R1C1, then the additional preference,

```
LOCATE PGROUP "UGROUP_1" SITE "R1C1"
```

appears in the preference file.

If block1 has a PREGION=ONE, PRLOC=R1C1, PRBBOX=2,2 then the preferences generated are as follows:

```
REGION "ONE" "R1C1" 2 2;
PGROUP "UGROUP_1"
COMP PFU_0;
PGROUP "UGROUP_2"
COMP PFU_1;
LOCATE PGROUP "UGROUP_1" REGION "ONE"
LOCATE PGROUP "UGROUP_2" REGION "ONE"
```

Note that UGROUPs also resolved into the PGROUP physical preference.

PFU Logic Mapping with PGROUP/UGROUP

When a PGROUP/UGROUP is defined in an EDIF netlist, Map will prevent logical elements of different PGROUPs/UGROUPs to be mapped into the same Programmable Function Unit (PFU). By doing so, a natural partitioning of the mapped design is achieved.

Map then groups all PFUs of the same PGROUP and writes out PGROUP preferences. The PLOC attribute translates to the LOCATE PGROUP preference. The LOC attribute translates to the relative location within a PGROUP preference.

Map PIOPGROUP Support

PIOPGROUP and its related attributes facilitate the partitioning I/O elements in the logical design.

PIOPGROUP and Related Attributes

PIOPGROUP	Indicates a logical partition of a group of I/O buffer components that get translated into a physical partition for PAR. Syntax: PIOPGROUP = identifier
PIOLOC	Indicates the physical anchor, range, or chip side of PIO sites. This attribute must exist and appear on same block as PIOPGROUP. Syntax: PIOPLOC = <*pin_name*>/<*site*>(anchor) | <*pin_name*>/ <*sitepin_name*>/<*site*>(range) | TOP | BOTTOM | LEFT | RIGHT(chip side)
PIOCOUNT	Indicates the number of PIOs to be considered during placement. This attribute must appear on same block as PIOPGROUP. Syntax: PIOCOUNT = <*integer*>
PIODIRECTION	PIODIRECTION indicates the orientation of the intended PIOs in the group. This attribute must appear on same block as PIOPGROUP. Syntax: PIODIRECTION = CLOCKWISE | COUNTERCLOCKWISE

PIOPGROUP Usage Guidelines

- Use on any hierarchical block containing I/O buffers or on I/O buffers themselves.
- PIOPGROUP attribute can be used in conjunction with PGROUP/ UGROUP as attributes on hierarchical block if logic elements and I/O buffers co-exist within the same sub-hierarchy. PIOPGROUPs *must* be anchored or *must* have a PIOPLOC attribute.

For example:

```
block1 ( PGROUP=PGROUP_1, PIOPGROUP=IOGROUP_1, PIOPLOC=PB12)
                    /       \
                block2      block3
                  /  \        /  \
              REG1  REG2    IBM  OBUFZ
```

Then REG1 and REG2 belong to PGROUP_1 and IBM and OBUFZ belong to IOGROUP_1.

Also, assuming REG1, REG2 are mapped to PFU_0 and IBM, OBUFZ are mapped to PIOs IN and OUT respectively, the resulting preferences generated in the preference file are as follows:

```
PGROUP "block1/PGROUP_1"
COMP PFU_0;
PGROUP "IOGROUP_1"
COMP IN
COMP OUT;
LOCATE PGROUP "IOGROUP_1" RANGE PB12;
```

Timing Driven Re-Mapping

Using the **-td** option in Map from the command line, you can attempt to re-map combinational paths and re-pack paths that TRACE deemed to be error paths. Re-mapping resolves the first 100 paths that TRACE identified as error paths in each preference. Note that guided flow, the **-g** option on the command line, is turned off when timing driven remap is operating.

Inputs include the initial EDIF design compiled into an `.ndg` file, a placed-and-routed physical design (`.ncd`) file, and a preference (`.prf`) with timing preferences. The output is a mapped `.ncd` file.

To run timing driven re-map:

1. Run a normal design flow from Map through Place and Route to generate your physical design (`.ncd`) file.
2. Run **TRACE**. Determine if design is a candidate for re-mapping based the criteria outlined above.
3. Choose a placed and routed `.ncd` file. A good choice is the best seed produced, e.g., best overall frequency performance.
4. Re-run MAP from the command line using the **-td** option and the placed-and-routed `.ncd` file along with the *same* EDIF design.

   Additional syntax:

   -td <routed.ncd> **-o** <outfile[**.ncd**]> <mapped.ncd>

   Example:

   ```
   map -td routed.ncd -o design2 design design.prf -p or4e06 -t ba352
   ```

   where:

   The *routed.ncd* is the previously placed-and-routed physical design file, *design2* is the newly mapped *.ncd* file, and *design* is the initial logic design and -t option specifies the package type. Note that you should specify the device and package type with the **-p** and **-t** options or the mapper will default to the smallest available package for that device.
5. Re-run PAR.

In most cases, timing driven re-mapping will improve results; however, results are design dependent and may not always be optimal.

Packing of Duplicate Registers

Designers often duplicate registers to reduce fan-out to improve timing. Duplicated registers are registers with the same data and control paths. The software keeps duplicated registers from being mapped into the same PFU.

If the COMP= attribute is attached to duplicate registers, it will override duplicate register packing.

Guided Mapping

To decrease place and route runtimes after minor changes to logical design, guided mapping uses a previously generated `.ncd` file to "guide" the mapping of the new logical design. Guided mapping can be performed from the Guide Filename attribute in the Project Navigator Map Design process, or specified using the **-g** option with the file name of guide file.

To perform guided mapping in the Project Navigator:

1. In the Project Navigator Sources window, select the **target deice**.
2. In the Processes window, right-click the **Map Design** process, and then select **Properties** to open the Properties dialog box.
3. Select the **Guide Filename** attribute from the attribute list and type the name of the guide file name in the edit region (*<file_name>*.ncd).
4. Click **Close** to close the dialog box.

Notes on Guided Mapping

All guidance criteria are based on signal name matching. Topology of combinatorial logic is considered when Softwire LUTs (SWLs) exist in the guided file.

Register elements are mapped in two passes. In the first pass, register control signals are matched by name exactly. In the second pass, the control signals names are not matched. This methodology provides a greater chance of matching for registers because control signal names have a tendency to change from successive synthesis runs. Other matching considerations are as follows:

- For combinatorial logic, new SWLs are matched from SWLs extracted from the guide design.
- All unmatched logic is mapped through the regular mapping process.
- The performance of the guided mapped design can be no better than the original.
- A guide report, <design_name>.gpr, gives details of the success that the guided map had in matching with the guide file.

Synthesis Requirements for Guided Mapping

For Synopsys and Synplicity, you must generate hierarchical netlist for guided mapping. For LeonardoSpectrum, see the example script below:

```
# Perform 1st iteration of synthesis (initial optimization)
# guide.scr adheres to the methodology described in the
# Mentor Graphics' app. note.

spectrum -file guide.scr
# Initial map, place & route. guide_route will be used as guide
# file for subsequent guide map and place&route.

# guide_2.scr is used to perform incremental synthesis after
# making changes to the HDL Code.

spectrum -file guide_2.scr
```

Guided mapping will have little effect if many net/signal names change from successive synthesis runs. Subsequently, guided Place and Route is rendered ineffective as well.

Notes on ORCA Series 4 Mapping from the Command Line

When mapping from the command line, ensure that you follow these guidelines:

- When retargeting Series 3 designs to the Series 4, make sure that the `.edn` file does *not* have the **LIBRARY** attribute set to **or3c00**.
- Do not run the Series 4 mapper with a Series 3 `.ngd` file without running **edif2ngd** using the **-l or4e00** option to search the appropriate library.

Running MAP from the Command Line

You can run the Map process from the command line. The Map process can be broken down into three command line steps: EDIF2NGD, NGDBUILD, and Map.

To run MAP from the command line:

1. Type **EDIF2NGD**

   EDIF2NGD accepts FPGA properties if they appear within the input file and passes them through to the output `.ngo` file. All other properties are ignored and are not written to the `.ngo` file. The FPGA-specific properties are described in the **Additional References > Properties** section of Help.

2. Type **NGDBUILD**

   NGDBUILD converts a file or files in the `.ngo` format into a file in the `.ngd` (generic database) format. To create an `.ngd` file, NGDBUILD reduces all of the components in the design to ORCA primitives. Then, NGDBUILD merges components that reference other files (for example, a physical macro file, or another design netlist).

   The resulting `.ngd` file contains a logical description of the design reduced to ORCA primitives, but still retains a description in terms of the original hierarchy expressed in the schematic. After performing the appropriate conversion, NGDBUILD checks the design by running a logical DRC (Design Rule Check), which is a series of tests on the converted logical design.

3. Type **MAP**

   The Map program maps a design to an FPGA device. The input to Map is a `.ngd` file, which contains a logical description of the design in terms of both the hierarchical components used to develop the design and the lower level primitives. The output is an `.ncd` file, which contains a physical description of the design in terms of the components in the target architecture. This `.ncd` file can then be placed and routed.

Interactive Editing

Interactive Editing - FPGA

The Lattice FPGA process flow supports interactive editing with three applications: the Preference Editor, the FPGA Floorplanner, and the EPIC Device Editor.

The Preference Editor

The Preference Editor lets you specify or change placement, routing, and/or timing constraints produced by the user or by the technology mapper. The Preference Editor reads the preference file (`.prf`) and displays the preference settings. Modifications to the PRF file are made via the function dialog boxes. Most of the attributes can be modified directly in the sheet. Pin assignments can be set in the Package View with drag-and-drop functionality.

The FPGA Floorplanner

The FPGA Floorplanner (Floorplanner) provides a graphical interface for managing FPGA device real estate. The Floorplanner can shorten design turn around time and achieve a design that conforms to critical circuit performance requirements.

With the Floorplanner, you can:

- Obtain a graphical display of information provided in the Pack/Place Report and the Route Report.
- Customize placement and routing of your design.
- Use the Floorplanner in Reentrant Flow to help you meet timing requirements and reduce channel congestion. You view the timing results of your design with the Performance Analyst, set the constraints in the Floorplanner, repack and place your design, and check for improved timing results using the Performance Analyst.
- Create Firm Macros and Intellectual Property (IP) that can be instantiated into other designs.
- Use the Timing Widget to perform static timing analysis and display timing paths in the Floorplan View.

EPIC Device Editor

The EPIC Device Editor (EPIC) is a graphical application for displaying and configuring FPGA devices. You can use the EPIC for, but not limited to, the following tasks.

- Place and route critical components before running automatic place and route tools on an entire design.
- Manually finish placement and routing if the routing program was unable to route the design to completion. The editor allows both automatic and manual component placement and routing.
- Read, write, and undo certain preferences in the PRF file.

Placing and Routing

After a design has undergone the necessary translation to bring it into the physical design (`.ncd`) format, it is ready for placement and routing. This phase is done by PAR (Place and Route program). PAR takes a mapped physical design (`.ncd` file), and places and routes the design.

PAR can place and route a design in two different ways: *cost-based* or *timing driven*. You can invoke PAR from the Project Navigator or from the command line.

Cost-Based Place & Route

The standard PAR package is a *cost-based tool*. This means that placement and routing are performed using various cost tables that assign weighted values to relevant factors such as constraints, length of connection, and available routing resources.

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 two stages completes.

During constructive placement, PAR places components into sites based on factors such as:

- 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 also is 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 performs an iterative procedure to converge on a solution that routes the design to completion or minimizes the number of unrouted nets.

During reduction routing, the router takes the result of iterative routing and reroutes some connections to minimize the signal delays within the device. There are two types of reduction (cleanup) routing you can perform:

- A faster cost-based cleanup routing, which makes routing decisions by assigning weighted values to the factors (for example, the type of routing resources used) affecting delay times between sources and loads.
- A more intensive delay-based cleanup routing, which makes routing decisions based on 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 Place & Route

In addition to cost-based PAR, you can run *timing driven* placement and routing using the Timing Wizard. The Timing Wizard is an integrated static timing analysis utility (i.e., it does not depend on input stimulus to the circuit).

This means that placement and routing is executed according to timing constraints (preferences) that you specify up front in the design process. The Timing Wizard interacts with PAR to ensure that the timing preferences you impose on the design are met.

To use timing-driven PAR, you simply write your timing preferences into a preference (`.prf`) file, which serves as input to the Timing Wizard. If PAR finds timing preferences in the preference file, timing-driven placement and routing are automatically invoked.

Below is a brief description of timing preferences that PAR supports. For additional information, see the Preference Editor Help.

FREQUENCY

Identifies the minimum operating frequency for all sequential input pins clocked by a specified net.

PERIOD

Specifies a maximum clock period for all sequential input pins clocked by a specified net.

MAXDELAY

Identifies a maximum total delay for a circuit net, path, path class, or bus in the design. Also specifies a maximum delay from a starting point or to an endpoint.

MAXSKEW

Identifies a signal skew between the loads on a specified signal.

BLOCK

Blocks timing checks on path classes, nets, buses, or component pins that are irrelevant to the timing of the design.

OFFSET

Identifies the external timing relationship between a clock pin and a data pin.

DEFINE STARTPOINT

Identifies a starting point for the MAXDELAY FROM/ TO preference (see MAXDELAY).

DEFINE ENDPOINT

Identifies an ending point for the MAXDELAY FROM/ TO preference (see MAXDELAY).

DEFINE PATH

Specifies a path from a source component to a destination component.

DEFINE BUS

Specifies a grouping of nets.

MULTICYCLE

Allows for relaxation of previously defined PERIOD or FREQUENCY constraints on a path.

Each timing preference can generate many timing constraints for the design. For example, a PERIOD or FREQUENCY preference generates a timing constraint for every data path clocked by a specified net. A MAXDELAY for a designated path class generates a timing constraint for every path in the path class. Depending upon the types of timing preferences specified and the values assigned to the preferences, PAR run time may be increased.

When PAR is complete, you can verify that the design's timing characteristics (relative to the preference file) have been met by running TRACE (Timing Reporter and Circuit Evaluator), ORCA Foundry's timing verification and reporting utility. TRACE, described in detail in the next chapter, issues a report showing any timing warnings and errors and other information relevant to the design.

Place & Route Input Files

PAR uses the following input files:

- Physical design (`.ncd`) file
- Preference (`.prf`) file (optional) — containing placement, routing, and/or timing constraints produced by the user or by the technology mapper.

Setting PAR Properties

PAR properties should be set before running a design flow. You can set PAR properties in the Design Flow Options dialog box with the PAR tab selected. See the Command Line Appendix for instructions on running **par** from the command line. Refer to the TRACE chapter for details on the PAR tab.

To set PAR properties for a design flow:

1. In the Project Navigator Sources window, select the **target device**.
2. In the Processes window, right-click the **Place & Route** process and select **Properties**.
3. Set the properties as desired and then click **Close**. To start the Place & Route Design process, double-click the **Place & Route** process icon.

Running Place & Route

From the Project Navigator, placement and routing is automatically performed when you run a full design flow or a partial design flow selecting just the Place & Route Design process.

To run PAR in Project Navigator:

1. In the **Processes** window in the Project Navigator, right click on **Place & Route Design**.
2. In the pop-up menu, click on **Start** to run the flow through the PAR process. **Force** will rerun the design flow up to the PAR process if selected.

To run PAR from the command line, see Running PAR from the Command Line.

Place & Route Properties

Placement Effort Level	Specifies the effort level of the design from Low (simplest designs) to High (most complex designs). The level is not an absolute; it shows instead relative effort. After you use PAR for a while, you will be better able to estimate whether a design is simple or complex. If you place and route a simple design at a complex level, the design will be placed and routed properly, but the process will take more time than placing and routing at a simpler level. If you place and route a complex design at a simple level, the design may not route to completion or may route less completely (or with worse delay characteristics) than at a more complex level.
Routing Passes	If Automatic is not selected, runs a maximum number (1-1000) of passes of the router, stopping earlier only if the routing goes to 100% completion. By default, Automatic is not selected and is accessible in Advanced PAR options. Each pass is a single attempt to route a placement to completion, and the screen displays a message for each pass.
Disable Timing Driven	Do not use timing-driven option for this PAR run. If unselected, PAR automatically uses the timing-driven option if the Timing Wizard is present and if any timing preferences are found in the preference file. If selected, the timing-driven option is not invoked in any case and cost-based placement and routing are done instead. Two examples of situations in which you might disable this option are: • You have timing preferences specified in your preference file, but you want to execute a quick PAR run without using the timing-driven option to give you a rough idea of how difficult the design is to place and route. • You only have a single license for the timing-driven option but you want to use this license for another application (for example, to perform timing-driven routing within EPIC) that will run at the same time as PAR. This option keeps the license free for the other application.
Create Delay Statistics File	If selected, a Delay Report (`.dly`) file will output on each PAR run. The Delay Report file contains delay information for each net in the design for each run. You can generate this file when you use the **-y** option in the command line.

Advanced Place & Route Properties

Operation	• *Place, Optimize Place and Route* — Performs both placement and routing. • *Place and Optimize Place Only* — Place and Optimize but does not route. • *Optimize Placement Only* — Skip constructive placement and run only optimizing placement. Do not route. • *Optimize Placement Only, then Route* — Run only optimizing placement, then enter the router. (Existing routes are ripped up before routing begins.) • *Route Only* — Route only. Do not place. • *Reentrant Route* — Begin routing leaving the existing routing in place (also called incremental routing). Reentrant routing is useful if you wish to manually route parts of the design and then continue automatic routing. For example, if you halted the route prematurely (with a Control-C) and wish to resume, or if you wish to run additional delay, use the reentrant routing operation.
Run Until Solved	Run until the design is fully routed or until all remaining cost tables are used.
Number of Iterations	The maximum number of placement/routing passes (1-100) to be run (regardless of whether they complete) at the Placement Effort Level. Each iteration will use a different cost table when the design is placed and will produce a different `.ncd` file. If you specify a Starting Cost Table, the iterations begin at that table number.
Placement Start Point (1-99)	Specifies the cost table to use (from 1–100) to begin the PAR run. Default is 1. Cost tables are not an ordered set. There is no correlation between a cost table's number and its relative value. If cost table 100 is reached, placement does not begin at 1 again, even if command options specify that more placements should be performed.
Save Best Runs	If selected, saves the specified number (1-100) of best outputs of the PAR run (defaults to 1). If deselected, saves *all* output designs produced by the PAR run. The best outputs are determined by a scoring system described in the section titled *Scoring the Routed Design*. This option does not care how many iterations you performed or how many effort levels were used. It compares every result to every other result and leaves you with the best number of *.ncd* files.
Automatic Routing Passes	Runs to completion unless the router intelligently determines it cannot complete.
Routing Resource Optimization	Determines the number of cost-based cleanup passes to run. If not used, the router runs one cost-based cleanup pass. If you run both cost-based and delay-based cleanup passes, the cost-based passes run before the delay based passes.
Routing Delay Reduction	Run delay-based cleanup passes.
Reduction Passes	Determines the number of passes to be run for the above two fields. The first delay-based cleanup pass produces the greatest improvement. Running more than five may not show any additional improvement.

Guided Place & Route

To decrease PAR runtimes after minor changes to the physical design file (.ncd), guided PAR uses a previously placed and/or routed .ncd file to "guide" the placement and routing of the new .ncd file. Guided PAR can be performed from the Project Navigator or specified using the **-g** option with the file name of guide file.

For PAR to use a guide file for design, PAR first tries to find a guiding object (i.e., nets, components, and/or macros) in the guide file that corresponds to an object in the new .ncd file. A *guiding object* is an object in the guide file of the same name, type, and connectivity as an object in the new .ncd file. A *guided object* is an object in the new .ncd file that has a corresponding guiding object in the guide file.

After PAR compares the objects in each file, it places and routes each object of the new .ncd file based on the placement/routing of its guiding object. If PAR fails to find a guiding object for a component, for example, PAR will try to find one based on the connectivity. PAR appends the names of all objects that do not have a guiding object in the guide file to .gpr (Guided PAR Report) file.

The matching factor option applies to nets and components only. When matching-factor is 100 (the default), a guiding object must have exactly the same connectivity as the object it is guiding. When a matching factor is specified, the value specified is taken as the minimum percentage of the same connectivity that a *guided object* and its *guiding object* have.

After all guided objects are placed and routed, PAR locks down the locations of all guided components and macros and then proceeds with its normal operation.

Guided PAR supports the following preferences: USE SPINE, USE PRIMARY, USE SECONDARY, USE LONGLINE, USE HALFLINE, LOCATE COMP, LOCATE MACRO, and hard-placed PGROUPs. A guiding object that is specified on any of the above preferences is unplaced/unrouted if it fails to meet the preference.

To perform guided PAR:

1. In the Project Navigator Sources window, select the **target device**.
2. In the Processes window, right-click the **Place & Route** process and select **Properties** to open the dialog box.
3. Under Advanced Options, select the **Guide Filename** attribute and type the name of the file in the text field.
4. Click **Close** to close the dialog box.
5. Double-click the **Place & Route** process. The ispLEVER software runs the process using the specified guide file.

Place & Route Output Files

The following files are the output files generated when the Placement and Routing step is complete.

- Physical Design (.ncd file) — a placed and routed design file (may contain placement and routing information in varying degrees of completion).
- PAR Report (.par file) — a PAR report including summary information of all placement and routing iterations.
- Delay Report (.dly file) — a file containing delay information for each net in the design. This file will be generated when you use the -y option.
- PAD Report (.pad file) — a file containing I/O pin assignments.

Multiple Iterations

If you specify options that produce a single output design file, your output consists of a single `.ncd` file, `.par` file, `.dly` file, and `.pad` file. The `.par`, `.dly`, and `.pad` files all have the same root name as the `.ncd` file.

If you run multiple placement and routing iterations, you produce a `.ncd`, `.par`, `.dly`, and `.pad` file for *each* iteration. After each iteration, the program saves the physical design (`.ncd`) file every time the score has improved. This means that during long PAR runs, the best current `.ncd` file is always available. This behavior prevents a few rare cases where timing scores may have gone up rather than down when performing multiple PAR iterations.

As the **par** command is performed, PAR records one `.par` file (a summary of all placement and routing iterations) at the same level as the directory you specified. It also places within the directory a `.ncd`, `.par`, `.dly` and `.pad` file for each individual iteration.

The file names for the output files use the naming convention *effort-level_cost-table-entry*; for example, 1_1.ncd, 1_1.par. 1_1.dly, and 1_1.pad. In this example, the effort level and cost table entries start at 1 (the default effort level is 5).

The Place & Route Report

The place and route (`.par`) report contains execution information about the PAR command run. The report also shows the steps taken as the program converges on a placement and routing solution. Access the Place & Route Report file in Project Navigator or in your project directory using a text editing program.

To view the Place & Route Report File:

- In Project Navigator, right-click on the **Place & Route Report** process in the Processes window and select **View** from the popup menu. The PAR report will appear in the output panel at the bottom of the user interface. You can also choose **Window > Report Viewer** to open the Report Viewer and browse for the `.par` file in your project directory using the **File > View** command.

  OR

- Use a text editor to open the `.par` ASCII file. Search or browse for the file in your project directory.

More on the .PAR Report File

The Place & Route report has the same elements and structure regardless of the ORCA architecture, which is not the case with Map Report files. See the Sample Place & Route Report. The Place & Route report contains the following sections:

- Time and Date Stamp
- Run Information
- Device Utilization Summary
- Placement
- Routing
- Completion

The following notes refer to the sample place and route report:

- The Placer score is a rating of the relative “cost” of a placement. A lower score indicates a better (that is, less “costly”) placement.
- The Score For This Design is a rating of the routed design.

- Timing score will always be 0 (zero) if all timing preferences have been met. If not, the figure will be other than 0. This tells you immediately whether your timing preferences have been met.
- Underneath the "Completed router resource pre-assignment" line, any warning messages may appear that alert the user that some clock signal may suffer excessive delay or skew due to conflicts in clock routing resources.

  For example, the placer will avoid placing two clock signals in the same PIC pair that share the same clock spine. However, if more than one clock signal is hard located in the same PIC pair that share a common clock spine, then the router will write out a warning message in the PAR Report File (`.par`) at this point.
- The "starting iterative routing" section contains figures in parentheses (125818). This represents the timing score for the design (not to be confused with the PAR score) at the end of the particular iteration. When the timing score reaches 0 (as it does in this example after iteration 2), this means that all timing preferences have been met. This timing score (0) also appears at the end of the Delay Summary Report section. Note that the Delay Summary Report is only generated when you use the -y option in the command line.

  The timing score at the end of the "starting iterative routing" section may not agree with the timing score at the end of the Delay Summary Report. This can occur if a MAXSKEW preference is scored and not met. The score shown in the Delay Summary Report section will always be the correct one.
- Sometimes the design will be completely routed but the router continues to route in the attempt to adhere to timing preferences.
- When you specify the -y option on the command line, the last section of the `.par` file summarizes the delay information for the routed design. The PAR run also produces a `.dly` *(delay) fil*e that contains more detailed timing information. The first column of this section gives the actual averages for the design. The figures in the second column, which are enclosed by parentheses, indicate a "worst case" scenario for the delay.
- For Series 4 designs, there is a section before "Starting Constructive Placer" that will list all selected primary clocks.

Sample Place & Route Report

```
PAR: Place And Route ispLever_v40_Alpha_Build_Classic (41c).
Copyright (c) 1991-1994 by NeoCAD Inc. All rights reserved.
Copyright (c) 1995 AT&T Corp.   All rights reserved.
Copyright (c) 1995-2001 Lucent Technologies Inc.  All rights reserved.
Copyright (c) 2001 Agere Systems   All rights reserved.
Copyright (c) 2002-2003 Lattice Semiconductor Corporation,  All rights
reserved.
Mon Nov 17 13:31:42 2003
par -l 5 -c 0 -w pcs.ncd routed.dir pcs.prf
Preference file: pcs.prf.
Placement level-cost: 5-1.
Start par timer. REAL TIME: 0 secs
Loading design for application par from file pcs.ncd.
   "pcs" is an NCD, version 3.0, vendor LATTICE, device or5s25, package
BM680, speed 5
```

```
Loading device for application par from file 'or5s25.nph' in environment
h:/rel/rtf4_0.41c.
Package: Version 1.17, Status: ALPHA
Dumping design to file C:/tmp/neo_2.

Device utilization summary:
   IO               153/610          25% used
                    153/530          28% bonded
   LOGIC           2332/13702        17% used
   SPECIAL            9/314           2% used
   PIO              153/520          29% used
   SLICE           2332/12712        18% used
   GSR                1/1           100% used
   EBR                8/116           6% used
TIME_REPORT: Starting to place and/or route design at 1 mins 11 secs
WARNING - par: Pulsewidth error at
          rx_gearbox_regs_0/INST/ram512x64_port1/ram512x64_syn0_2_1/ebr512x36
          _0 on net clk_156mhz_c: FREQUENCY PORT "clk_156mhz" 440.000000
          MHz ;
WARNING - par:  Maximum frequency is 250.000 MHz
WARNING - par: Pulsewidth error at
          rx_gearbox_regs_0/INST/ram512x64_port1/ram512x64_syn0_0_3/ebr512x36
          _0 on net clk_156mhz_c: FREQUENCY PORT "clk_156mhz" 440.000000
          MHz ;
WARNING - par:  Maximum frequency is 250.000 MHz
WARNING - par: Pulsewidth error at
          rx_gearbox_regs_0/INST/ram512x64_port0/ram512x64_syn0_2_1/ebr512x36
          _0 on net clk_156mhz_c: FREQUENCY PORT "clk_156mhz" 440.000000
          MHz ;
WARNING - par:  Maximum frequency is 250.000 MHz
WARNING - par: Pulsewidth error at
          rx_gearbox_regs_0/INST/ram512x64_port0/ram512x64_syn0_0_3/ebr512x36
          _0 on net clk_156mhz_c: FREQUENCY PORT "clk_156mhz" 440.000000
          MHz ;
WARNING - par:  Maximum frequency is 250.000 MHz
WARNING - par: Pulsewidth error at
          tx_gearbox_0/INST/ram512x66_port1/ram512x66_syn0_2_1/ebr512x36_0
          on net clk_161mhz_c: FREQUENCY PORT "clk_161mhz" 440.000000 MHz ;
```

```
WARNING - par:  Maximum frequency is 250.000 MHz
WARNING - par: Pulsewidth error at
          tx_gearbox_0/INST/ram512x66_port1/ram512x66_syn0_0_3/ebr512x36_0
          on net clk_161mhz_c: FREQUENCY PORT "clk_161mhz" 440.000000 MHz ;

WARNING - par:  Maximum frequency is 250.000 MHz
WARNING - par: Pulsewidth error at
          tx_gearbox_0/INST/ram512x66_port0/ram512x66_syn0_2_1/ebr512x36_0
          on net clk_161mhz_c: FREQUENCY PORT "clk_161mhz" 440.000000 MHz ;

WARNING - par:  Maximum frequency is 250.000 MHz
WARNING - par: Pulsewidth error at
          tx_gearbox_0/INST/ram512x66_port0/ram512x66_syn0_0_3/ebr512x36_0
          on net clk_161mhz_c: FREQUENCY PORT "clk_161mhz" 440.000000 MHz ;

WARNING - par:  Maximum frequency is 250.000 MHz
The following 2 signals are selected as primary clocks:
    clk_156mhz_c (driver: clk_156mhz, clk load #: 1455)
    clk_161mhz_c (driver: clk_161mhz, clk load #: 340)
Starting Placer Phase 0.  REAL time: 1 mins 28 secs
.................
Finished Placer Phase 0.  REAL time: 1 mins 31 secs
Starting Placer Phase 1.  REAL time: 1 mins 31 secs
Placer score = 1510083.
.....................................................
TIME_REPORT: tw used 0 secs  (called 3 times)
Placer score = 951678.
Finished Placer Phase 1.  REAL time: 2 mins 14 secs
Starting Placer Phase 2.
.
Placer score =  927130
Finished Placer Phase 2.  REAL time: 2 mins 20 secs
Dumping design to file routed.dir/5_1.ncd.
WARNING - par: Pulsewidth error at
          rx_gearbox_regs_0/INST/ram512x64_port1/ram512x64_syn0_2_1/ebr512x36
          _0 on net clk_156mhz_c: FREQUENCY PORT "clk_156mhz" 440.000000
          MHz ;
WARNING - par:  Maximum frequency is 250.000 MHz
WARNING - par: Pulsewidth error at
          rx_gearbox_regs_0/INST/ram512x64_port1/ram512x64_syn0_0_3/ebr512x36
```

```
          _0 on net clk_156mhz_c: FREQUENCY PORT "clk_156mhz" 440.000000
          MHz ;
0 connections routed; 16054 unrouted.

Starting router resource preassignment
Completed router resource preassignment. Real time: 2 mins 27 secs
Starting iterative routing.
For each routing iteration the number inside the parenthesis is the
total time (in picoseconds) the design is failing the timing constraints.
For each routing iteration the router will attempt to reduce this number
until the number of routing iterations is completed or the value is 0
meaning the design has fully met the timing constraints.

End of iteration 1
16054 successful; 0 unrouted; (55059) real time: 2 mins 41 secs
Dumping design to file routed.dir/5_1.ncd.
End of iteration 2
16054 successful; 0 unrouted; (5451) real time: 2 mins 56 secs
Dumping design to file routed.dir/5_1.ncd.
End of iteration 3
16054 successful; 0 unrouted; (5451) real time: 2 mins 57 secs
Giving up.
Total CPU time 2 mins 21 secs
Total REAL time: 2 mins 57 secs
Completely routed.
End of route.  16054 routed (100.00%); 0 unrouted.
No errors found.
Timing score: 5451
Total REAL time to completion: 3 mins
TIME_REPORT: TW used 2 secs
All signals are completely routed.
par done!
```

Scoring the Routed Design

In the Delay Summary Report section of the `.par` file, the SCORE FOR THIS DESIGN is a rating of the routed design. The `.par` file shows the total score as well as the individual factors making up the score. The lower the score, the better the results.

This score takes into account such factors as the number of unrouted nets, the delays on nets and conformance to user timing preferences. The formula that produces the score is:

5000***unr** + 1000***ncst** + 20***acst** + (**delay***weight**)*0.2 + **av***100 + **10w***20

where:

unr The number of unrouted nets

ncst The number of timing constraints not met

acst The amount (expressed in ns) by which the timing constraints were not met

delay Maximum delay on a net with a weight greater than 3

weight Net weights or priorities

av The average of all of the maximum delays on all nets

10w The average of the maximum delays for the ten highest delay nets (10w)

Each factor is weighted by its relative importance.

***Note**: Timing scores in PAR may rise when attempting to meet timing constraints because the router may move a signal to a less favorable path to make a resource available for another signal.*

The Delay File

When using the **-y** option from the command line or selecting the **Create Delay Statistics File** option in the PAR tab of the Design Flow options dialog box, the delay file is output by each PAR run. The delay file contains delay information for each net in the design. It includes:

- A listing of the 20 nets with the longest delays.

  An ’e’ preceding a maximum delay indicates that the delay shown is only approximate The figure in parentheses following each ’e’ delay represents the approximate delay with a certain percentage automatically added to it (a “worst case” situation) when one is specified in the preference (`.prf`) file via EPIC. When Timing Wizard looks at the delays, it uses the value in parentheses rather than the approximate value represented by the ’e’.

- A delay analysis for each net, including the net name, followed by the driver pin and the load pin(s).

Sample Delay File

The following is a portion of a delay file. If this were a complete file, it would show the load delays for *all* nets in the design.

```
PAR: Place And Route ORCA Foundry 2001
Copyright (c) 1991-1994 by NeoCAD Inc. All rights reserved.
Copyright (c) 1995 AT&T Corp. All rights reserved.
Copyright (c) 1996-2001 Lucent Technologies Inc. All rights reserved.
Copyright (c) 2001 Agere Systems. All rights reserved.
Copyright (c) 2002, Lattice Semiconductor Corporation, All rights reserved.
```

```
Tues Jan 15 11:08:38 2002

File: ledspin.dir/5_1.dly

The 20 worst nets by delay are:
+---------------------------------+
| Max Delay | Netname |
+---------------------------------+
9.2 $1N28
8.4 HZ4
8.3 $1N3
5.8 $1N88
4.5 $1N1042
3.5 $1N1370
3.5 $1N1362
3.1 $1N1051
3.0 $1N1036
2.9 $1N1050
2.8 $1N1054
2.7 $1N1049
---------------------------------

------------------------------------------------------------
Net Delays
------------------------------------------------------------

$1I105\Q1
$1N1054.Q1
2.2 $1N1054.A1

$1I105\Q2
$1N1054.Q2

2.2 $1N1054.A2
```

The PAD Specification File

The PAD specification file (`.pad`) contains a listing of all PICs used in the design and their associated primary pins both by port name and by pin names. More specifically, it contains the following information:

- Design Information
- Pinout by Port Name
- VddIO by Bank
- Vref by Bank
- Pinout by Pin Number
- Locate Preferences
- Copyright Information

The report can be divided into three sections described in the following subsections below.

First Section

The first section contains basic design information, *Pinout by Portname* which lists the port names with primary pin designations, *VDDIO by Bank* which shows voltages associated with banks, and *Vref by Bank/Group* which shows voltage reference types per banks/groups . Design information includes *Part type*, *Speed grade*, *Package* and includes version control information for the package pinout. Near the top of the `.pad` file appears a line similar to the one below:

```
Package: Version 1.12, Status: PRODUCTION
```

indicates hardware package status. Any status other than PRODUCTION (e.g., alpha or beta) means that the package pinout for the device has not been finalized, and should not be used for board layout purposes.

For port names, the table also includes the buffer type (input, output, or bidirectional) and any associated properties (i.e., attributes). *Buffer types* information lists PIO modes. For example, a PIO in LVDS or LVPECL mode needs two bonded pads for differential signals that are both included in the `.pad` file.

Because Series 4 devices or higher support several different voltage standards, the bank VDDIOs have been listed separately in the *VDDIO by Bank* section. *Vref by Bank/Group* includes information for reference voltages. All Vref voltage pins are listed in this table.

***Note**: For Series 5 designs, there will be additional information on Load pins in Vref by Bank.*

Second Section

The second section of the report lists the *Pinout by Pin Number* which includes primary pin number, the *Pin info* (e.g., component name or reference voltage type), *Buffer type*, and *Preference* which shows any preferences assigned to the component.

Third Section

The third section of the report lists LOCATE preferences in the `.prf` file format, so these preferences can be added when re-running PAR if necessary. Adding these preferences to the `.prf` file will ensure that the pads will be placed in the same specified site locations. Copyright information is also included here at the end of the file.

***Note**: Earlier architectures, Series 2 and Series 3, do not necessarily contain all of the sections listed above.*

Sample PAD File (first section)

```
PAD Specification File
**************************
PART TYPE:        or4e02
SPEED GRADE:      3
PACKAGE:          FS256
Package: Version 1.16, Status: PRODUCTION
Thu Nov 13 14:46:58 2003

Pinout by Port Name:
+--------------------+----------+--------------------+--------------------------+
|  Port Name         |  Pin     |  Buffer Type       |  Properties              |
+--------------------+----------+--------------------+--------------------------+
| CK_IN              | B9       | input  LVCMOS2     | 12mA SLEW PULLNONE       |
| LD_BY_O            | H14      | output SSTL3       |                          |
| LD_TG_O            | P10      | output HSTL1       |                          |
| LD_TR_O            | J11      | output GTL         |                          |
| LD_TY_O            | B7       | output LVTTL       | 6mA SLEW PULLNONE        |
| MC1_IN             | P7       | input  GTLPLUS     |                          |
| MC2_IN             | J1       | input  HSTL3       |                          |
| MC3_IN             | R8       | input  PCI         |                          |
| MR_IN              | A7       | input  SSTL2       |                          |
+--------------------+----------+--------------------+--------------------------+

VDDIO by Bank:
+--------------------+--------------------+
|  Bank              |  VDDIO             |
+--------------------+--------------------+
| VDDIO0             | 3.3V               |
| VDDIO1             | 2.5V               |
| VDDIO3             | 3.3V               |
| VDDIO5             | 1.5V               |
| VDDIO6             | 3.3V               |
| VDDIO7             | 1.5V               |
+--------------------+--------------------+

Vref by Bank/Group:
+--------------------+----------+--------------------+
|  Bank / Group      |  Pin     |  Vref              |
```

```
+--------------------+----------+--------------------+
| 1 / 6              | D7       | Vref_SSTL2         |
| 6 / 10             | T7       | Vref_GTLPLUS       |
| 7 / 4              | J4       | Vref_HSTL3         |
+--------------------+----------+--------------------+
```

Sample PAD File (second section)

```
Pinout by Pin Number:
+----------+--------------------+--------------------+--------------------+
|  Pin     |  Pin Info          |  Preference        |  Buffer Type       |
+----------+--------------------+--------------------+--------------------+
| A4       |                    |                    |                    |
| A5       |                    |                    |                    |
| A6       |                    |                    |                    |
| A7       | MR_IN              |                    | SSTL2              |
| A8       |                    |                    |                    |
| A11      |                    |                    |                    |
| A12      |                    |                    |                    |
| A13      |                    |                    |                    |
| A14      |                    |                    |                    |
| B4       |                    |                    |                    |
| B5       |                    |                    |                    |
| B6       |                    |                    |                    |
| B7       | LD_TY_O            |                    | LVTTL              |
| B9       | CK_IN              |                    | LVCMOS2            |
| B10      |                    |                    |                    |
| B11      |                    |                    |                    |
| B12      |                    |                    |                    |
| B13      |                    |                    |                    |
| C1       |                    |                    |                    |
| C4       |                    |                    |                    |
| C5       |                    |                    |                    |
| C6       |                    |                    |                    |
| C8       |                    |                    |                    |
| C9       |                    |                    |                    |
| C10      |                    |                    |                    |
| C11      |                    |                    |                    |
| C13      |                    |                    |                    |
| C16      |                    |                    |                    |
```

```
| D6       |                    |                    |                    |
| D7       | Vref_SSTL2         |                    |                    |
| D9       |                    |                    |                    |
| D10      |                    |                    |                    |
```

Sample PAD File (third section)

```
Locate Preferences for each Pin:
LOCATE  COMP  "CK_IN"  SITE  "B9";
LOCATE  COMP  "LD_BY_O"  SITE  "H14";
LOCATE  COMP  "LD_TG_O"  SITE  "P10";
LOCATE  COMP  "LD_TR_O"  SITE  "J11";
LOCATE  COMP  "LD_TY_O"  SITE  "B7";
LOCATE  COMP  "MC1_IN"  SITE  "P7";
LOCATE  COMP  "MC2_IN"  SITE  "J1";
LOCATE  COMP  "MC3_IN"  SITE  "R8";
LOCATE  COMP  "MR_IN"  SITE  "A7";
PAR: Place And Route ispLever_v40_Alpha_Build_Classic (37c).
Copyright (c) 1991-1994 by NeoCAD Inc. All rights reserved.
Copyright (c) 1995 AT&T Corp.   All rights reserved.
Copyright (c) 1995-2001 Lucent Technologies Inc.  All rights reserved.
Copyright (c) 2001 Agere Systems   All rights reserved.
Copyright (c) 2002-2003 Lattice Semiconductor Corporation,  All rights
reserved.
Thu Nov 13 14:46:58 2003
```

Place & Route Considerations - ORCA Series 4

Primary Clock Selection and Placement

Primary clocks are automatically selected according to the load number and driver type. You can also define a signal as a primary clock using the USE PRIMARY preference. The maximum number of primary clocks in a design cannot exceed eight.

In primary clock placement, if the primary clock is a PIO, then the placer will automatically place it into a “sweet site” (i.e., a site that can be routed easily to the center). However, if the primary clock is driven by a PFU (i.e., internally generated primary clock), the placer will attempt to place it on a sweet site or a proximal location. Furthermore, if you locate a primary clock driver on a non-sweet site, the placer will issue a warning. During pre-placement, all PIO constraints are observed (i.e., the pre-placement is always legal).

Additional PAR messages: Placer prints out a list of selected primary clocks for Series 4 designs before the constructive placement. You are advised to check that the primary clocks are selected properly.

Express Clock Placement

A clock is recognized as an Express Clock (ECLK) only if it is driven by a PIO or PLL and at least one of its loads is the EC pin of a PIO.

In an OR4E device, more than one ECLK can be placed on each side. The placer will try to distribute the clock drivers evenly among the four edges. A component is an ECLK load if it is a PIO and the EC pin is driven by an ECLK. The placer will put ECLK loads on the same side as the driver. You can also manually locate the ECLK drivers and allow the placer to place the loads.

Block RAM Placement

Block RAM pre-placement is handled differently from other special blocks because of its complicated component/site relations. For example, the Series 4 device has two kinds of block RAM sites: RAM512 and RAM1024. Block RAMs are grouped into pairs, each pair having a RAM1024 and a RAM512 site.

The legal placement patterns for RAM site pairs are illustrated in the table below:

Pattern	RAM1024 Site	RAM512 Site
#1	RAM1024 COMP	x
#2	RAM512 COMP	RAM512 COMP
#3	x	RAM512 COMP
#4	RAM512 COMP	x
#5	x	x

Legend:
Patterns refer to possible placement combinations.
x = no component placed at site.

A RAM512 site can only contain a RAM512 comp, while a RAM1024 site can contain either RAM512 or RAM1024 comp. When a RAM1024 site contains a RAM512 comp, its pair RAM512 site is still usable. However, when a RAM1024 site contains a RAM1024 comp, its pair RAM512 site is no longer usable.

Router Support - ORCA Series 4

The ORCA Series 4 functionality for the router includes *express clock routing*, *primary clock routing*, and *secondary clock routing*. The express, primary, and secondary clocks are all fully supported.

Running PAR from the Command Line

PAR can also be run through the command line. The Implementation Engine multi-tasking option available in UNIX is explained in detail here because the option is not available for PCs.

Implementation Engine (PAR Multi-Tasking Option)

This option allows you to use multiple machines (nodes) that are networked together for a multi-run PAR job, significantly reducing the total amount of time for completion. You can specify multi-tasking from the UNIX command line. The option is not available for PCs.

Overview

Before the Implementation Engine was developed for ORCA Foundry, PAR could only run multiple jobs in a linear way. The total time required to complete PAR was equal to the amount of time it took for each of the PAR jobs to run.

For example, the PAR command:

```
par -l 5 -n 10 -i 10 -c 1 mydesign.ncd output.dir
```

tells PAR to run 10 place and route passes (**-n 10**) at effort level 5 (**-l 5**), a maximum of 10 router passes (**-i 10**), and one cost-based cleanup pass. It runs each of the 10 jobs consecutively, generating an output . ncd file for each job, i.e., output.dir/ 5_1.ncd, output.dir/5_2.ncd, etc. If each job takes approximately one hour, then the run takes approximately 10 hours.

Suppose, however, that you have five nodes available. The Implementation Engine allows you to use all five nodes at the same time, dramatically reducing the time required for all ten jobs.

Running the Implementation Engine

1. First generate a file containing a list of the node names, one per line as in the following example:

   ```
   # NODE names
   jupiter #Fred's node
   mars #Harry's node
   mercury #Betty's node
   neptune #Pam's node
   pluto #Mickey's node
   ```

2. Now run the job from the command line as follows (or use the PAR shell):

   ```
   par -m nodefile_ name -l 5 -n 10 -i 10 -c l mydesign.ncd output.dir
   ```

This runs the following jobs on the nodes specified.

```
NODE1: par -l 5 -i 10 -cl mydesign.ncd output.dir/5_1.ncd
NODE2: par -l 5 -i 10 -cl mydesign.ncd output.dir/5_2.ncd
NODE3: par -l 5 -i 10 -cl mydesign.ncd output.dir/5_3.ncd
NODE4: par -l 5 -i 10 -cl mydesign.ncd output.dir/5_4.ncd
NODE5: par -l 5 -i 10 -cl mydesign.ncd output.dir/5_5.ncd
```

As the jobs finish, the remaining jobs start on the nodes until all 10 jobs are complete. Since each job takes approximately one hour, all 10 jobs will complete in approximately two hours.

You cannot use the **-a** option (automatic level skipping) with the **-m** nodefile option (the Implementation Engine).

System Requirements

- **rsh** must be located through the path variable.
- The executables required on the machines defined in the nodes file are:
  - /bin/sh
  - PAR (must be located through path variable)
- Required environment variable on remote machines:

  NEOCAD (points at NEOCAD directory structure — must be a path accessible to both the machine from which the Implementation Engine is run and the node)

FOUNDRY (points at FOUNDRY directory structure — must be a path accessible to both the machine from which the Implementation Engine is run and the node)

ATT_ORCA (points at ATT_ORCA directory structure — must be a path accessible to both the machine from which the Implementation Engine is run and the node)

ORCA_SSERVER (points to the security server nodes)

LD_LIBRARY_PATH (supports par path for shared libraries — must be a path accessible to both the machine from which the Implementation Engine is run and the node)

path (contains $ATT_ORCA/bin/$PLATFORM)

To determine if everything is set up correctly, you can run the **rsh** command to the nodes to be used. Type the following:

```
rsh node_name/bin/sh -c par
```

If you get the usage message back on your screen, everything is set correctly.

Environment Variables

Two environment variables are interpreted by the Implementation Engine manager:

- PAR_AUTOMNTPT
- PAR_AUTOMNTTMPPT

You may set PAR_AUTOMNTPT to:

```
setenv PAR_AUTOMNTPT ""
```

The above setting does not issue the **cd** command**;** you must enter full paths for all of the input and output file names. Alternatively, you may wish to use the following setting:

```
setenv PAR_AUTOMNTPT < value>
```

If you set it to a value such as “/nfs,” it is used in place of “/net.” Then, for example, if the current working directory is “/usr/<username>/<design>” on node “mynode”, a command of the form “cd /nfs/mynode/usr/< username>/< design>” is generated.

You may need to specify a different temporary mount point. If, for example, you run a job where the current working directory is “/usr/ < username>/< design>” and you find that the command being generated is “cd /nfs/< remnode>/t_mnt/usr/< username>/< design>”, set the PAR_AUTOMNTTMPPT variable as follows:

```
setenv PAR_AUTOMNTTMPPT /t_mnt
```

This causes the following desired command to be issued:

```
"cd /nfs/< remnode>/usr/< username>/< design>"
```

Security

The Implementation Engine manager (impman) license must be available so that jobs can be allotted to the designated hosts to perform each individual PAR run. If the impman license is not available, you get an error message.

If PAR is able to lock the impman license, each job running on a node tries to lock an Implementation Engine place and route (imppar) license. If it can, the job will automatically be timing-driven and device-independent.

You see a message like this on your screen:

```
Starting job 5_1 on node NODE1
```

If PAR cannot lock an imppar license, you do not see a "starting job" message and PAR reverts to the normal sequence of par, tdpar, and family licensing.

For more information on security, see the ORCA Foundry Installation Guide and the ORCA Foundry Release Notes.

Screen Output

When PAR is running multiple jobs and is not in multi-tasking mode, output from PAR is displayed on the screen as the jobs run. When PAR is running multiple jobs in multi-tasking mode, you only see information regarding the current status of the Implementation Engine. For example, when the job above is executed, the following screen output would be generated:

```
Starting job 5_1 on node NODE1
Starting job 5_2 on node NODE2
Starting job 5_3 on node NODE3
Starting job 5_4 on node NODE4
Starting job 5_5 on node NODE5
```

When one of the jobs finishes, this message will appear:

```
Finished job 5_3 on node NODE3
```

These messages continue until there are no jobs left to run, at which time "Finished" appears on your screen.

You may interrupt the job at any time by pressing Control-C.

For HP workstations, you must have Control-C set as the escape character to halt the Implementation Engine in this manner. To set the escape character, enter **stty ^V^C** in the *.login* file or *.cshrc* file.

If you interrupt the program, you may choose from six options:

- **Continue processing and ignore the interrupt** — PAR continues operating as before the interruption and runs to completion.
- **Normal program exit at next check point** — allows the Implementation Engine to wait for all jobs to finish before terminating. PAR is allowed to generate the master PAR output file (`.par`), which describes the overall run results.

  When you select option 2, a secondary menu offers the following options:

  - Allow jobs to finish — current jobs finish but no other jobs are started if there are any. For example, if you are running 100 jobs (-n 100) and the current jobs running are 5_49 and 5_50, when these jobs finish, job 5_51 is not started.
  - Halt jobs at next checkpoint — all current jobs stop at the next checkpoint; no new jobs are started.
  - Halt jobs immediately — all current jobs stop immediately; no other jobs are started.
- **Exit program immediately** — all running jobs stop immediately (without waiting for running jobs to terminate) and PAR exits the Implementation Engine.
- **Add a node for running jobs** — allows you to dynamically add a node on which you can run jobs.

  When you make this selection, you are prompted to input the name of the node to be added to the list. After you enter the node name, a job starts immediately on that node and a "Starting job" message is displayed.
- **Stop using a node** — allows you to remove a node from the list so that a job does not run on that node.

  If you select **Stop using a node**, you must also enter the number identifying the node you wish to stop using.

If you enter a legal number, you are asked to choose from the following options:

- Terminate the current job immediately and resubmit — halts the job immediately and sets it up again to be run on the next available node. The halted node is not used again unless it is enabled by the "add" function.
- Allow the job to finish — finishes the node's current job, then disables the node from running additional jobs.

The list of nodes described above are not necessarily numbered in a linear fashion. Nodes that are disabled are not displayed. For example, if NODE2 is disabled, the next time "Stop using a node" is opted, it does not appear in the list.

- **Display current status** — displays the current status of the Implementation Engine. It shows the state of nodes and the respective jobs. For example:

ID	Node	Status	Job	Time
1.	NODENAME1	Job Running	5_10	01:30:45
2.	NODENAME2	Job Running	5_11	02:29:03
3.	NODENAME3	Not available		
4.	NODENAME4	Pending Term	5_12	02:20:01
5.	NODENAME5	Job Running	5_13	02:20:01
6.	NODENAME6	Idle		
7.	NODENAME7	Job Running	5_12	25

Each entry is described below:

- `NODENAME1` has been running job 5_10 for approximately 2 1/2 hours.
- `NODENAME2` has been running job 5_11 for approximately 2 1/2 hours.
- `NODENAME3` has been deactivated by the user with the "Stop using a node" option or it was not an existing node or it was not running. Nodes are "pinged" to see if they exist and are running before attempting to start a job.
- `NODENAME4` has been halted "immediately" with job resubmission. The Implementation Engine is waiting for the job to terminate.Once this hap pens the status will be changed to "not available".
- `NODENAME5` has been running job 5_13 for 2 hours 20 minutes.
- `NODENAME6` has finished its current job and is available for another. When you see the "Idle" message, it usually means that no other jobs are available.
- `NODENAME7` is running job 5_12. This job was resubmitted when NODENAME4 was dropped. It has been running for 25 seconds. It is unlikely that you will see the same job listed twice (as in the sample above) since the job pending termination usually finishes very quickly.

There is also a status named "Job Finishing". This appears if the Implementation Engine has been instructed to halt the job at the next checkpoint.

Clock Boosting

Clock Boosting (for ORCA-4, FPSC and Slayer devices), refers to timing optimization of designs through clock skew scheduling to improve the performance in a synchronous system. Clock Boosting is achieved by scheduling clock delay signals at each register to relax critical paths.

Using ispLEVER's clock boosting program (CST), you can improve timing of FPGA designs after running Place & Route.

Given a placed and routed `.ncd` file, Clock Boosting takes advantage of non-zero clock skew available in most components to achieve better timing. The Clock Boosting process attempts to find the optimal clock delay solution, then modifies the `.ncd` files and verifies the new timing for both setup and hold time constraints.

With Clock Boosting, you can generate an optimal `.ncd` for a single preference or near-optimal `.ncd` for multiple constraints in minutes. This may take several hours if you are performing manual changes on multiple preferences.

CST Usage Guidelines

Before running the CST program (using either the Clock Boosting process or the command line), you must run a design through the flow and generate a placed-and-routed `.ncd` file. You can run Clock Boosting before and after running TRACE; however, you should always run TRACE again after running Clock Boosting to get accurate final timing information for your design.

Follow these general guidelines for using Clock Boosting:

- If the placed and routed `.ncd` file fails to meet current constraints, you can run Clock Boosting to:
  - attempt to meet the current constraints and even tighter constraints than the current set (if possible).
  - obtain an optimized .ncd file to meet a relaxed constraint set it failed.
- If the placed and routed `.ncd` met current constraints, you can still run Clock Boosting to get an optimized `.ncd` file to meet a tighter constraint set (if possible).

ASIC Cell Support

Support for the following special cells have also been added to the Clock Boosting functionality:

- SHIFT
- PLL1
- PLL2
- PPLL
- SLAVE
- MASTER
- IODDR
- IOSR2
- IOSR4
- OSR2X2
- ULIS
- ULIM

You can turn off Clock Boosting for special cells using environment variable **CST_NO_ASIC**.

CST Input Files

The following files are input to the CST program:

- Placed and routed files (`.ncd`) that are candidates for clock skew scheduling.
- Preference files (`.prf`) containing user-defined constraints associated with the candidate `.ncd` file.

Optimizing Design Files with Clock Boosting

You can run the CST program from within the Project Navigator using the Clock Boosting process. Note that Clock Boosting can only be used with placed-and-routed ORCA-4, FPSC, or Slayer designs.

To run Clock Boosting on your designs from the Project Navigator:

1. In the Project Navigator Sources window, select the **target device.**
2. In the Processes window, double-click the **Clock Boosting** process. The software runs the CST program and generates the appropriate output files.

***Note**: If you want to name the output file something other than the project name (default), right-click on the **Clock Boosting** process and choose **Properties** to open the Properties dialog box. Type the new name into the text box. Click the check mark to confirm your entry or the red "X" to cancel.*

CST Output Files

The following files are output from the CST program:

- A new `.ncd` file will be generated if all user-defined constraints are met. The default file name of the output is the input file name with the prefix "`new_`" added to the file name. For example, `design1.ncd` would be output as `new_design1.ncd`.
- An optimized `.ncd` file will be generated in the following two cases:
  - when user constraints are met and Cycle Stealing finds a tighter set of constraints.
  - when user constraints are not met but Cycle Stealing finds a solution with relaxed set of constraints.
- A guidelines file (`guidelines.cst`) is generated. The file provides some guidelines for relaxing the constraints in the preference file.
- Under certain conditions, an optimized `.ncd` file with the prefix "`new_`" added to the file name is generated.

Sample CST Guidelines File

A sample `guidelines.cst` file generated by CST is provided here.

Hints for multiple preferences CST

- Set all the preferences to the lower_bound
- Tune up one preference at a time towards upper_bound
- Important preferences first.

Sample File

```
Clock Skew Scheduling met current constraints

Clock delay setting...
comp = 376; bank = 0; delay = 1
comp = 376; bank = 1; delay = 1
comp = 376; bank = 2; delay = 1
comp = 376; bank = 3; delay = 1

Clock Skew Scheduling met tighter constraints
Critical Path for Pref # 1 ==> + 6.03 %

before cst = 158.48(MHz) [ = 6.31 (ns)]
after cst >= 168.04(MHz) [ <= 5.95 (ns)]

Critical Path for Pref # 2

before cst = 163.72(MHz) [ = 6.11 (ns)]
after cst >= 163.72(MHz) [ <= 6.11 (ns)]

Critical Path for Pref # 3

before cst = 161.60(MHz) [ = 6.19 (ns)]
after cst >= 161.60(MHz) [ <= 6.19 (ns)]

Clock delay setting...
comp = 376; bank = 0; delay = 1
comp = 376; bank = 1; delay = 1
comp = 376; bank = 2; delay = 1
comp = 376; bank = 3; delay = 1
comp = 468; bank = 7; delay = 1
comp = 523; bank = 3; delay = 1
comp = 84; bank = 7; delay = 1
comp = 222; bank = 4; delay = 1
comp = 988; bank = 0; delay = 3
comp = 93; bank = 2; delay = 2
comp = 720; bank = 7; delay = 1
comp = 940; bank = 0; delay = 2
comp = 904; bank = 7; delay = 1
comp = 898; bank = 4; delay = 1
comp = 985; bank = 8; delay = 2
comp = 986; bank = 2; delay = 1
comp = 631; bank = 7; delay = 3
comp = 634; bank = 7; delay = 3
comp = 634; bank = 3; delay = 3
```

Running CST from the Command Line

The cycle stealing program (CST) can also be run from the command line. Proper syntax and example of commands are provided in this section.

Syntax

cst [**-o** <*new_design*[**.ncd**]>] <*design*[**.ncd**]> [<*preference*[**.prf**]>] [**-b**]

where:

-o <*new_design*[**.ncd**]>	Designates the output *.ncd* file that will be generated if all timing constraints are met. The default file name of the output is the input file name with the prefix "`new_`" added to the file name. The resulting output file is dependent on the solution CST ascribes to the constraints. Generally, if all constraints are met, relaxed or tightened, CST attaches the "`new_`" prefix to the output *.ncd* file. -b The -b option gets best possible results even if the current preferences are already met. For example, if a timing preference was set to 100 MHz, it may be optimized to 150 MHz or higher. Because the higher frequency could strain other parts of the design, this option may not always be desirable. The default behavior stops when preferences are met.
<*design*[**.ncd**]>	The input physical design (*.ncd*) file that is a candidate for clock skew reduction.
<*preference*[**.prf**]>	The input preference (*.prf*) file that contains your user-defined constraints associated with the candidate *.ncd* file. Note that you do not have to specify this file in the command line if it has the same file name as the input *.ncd* file.

CST Command Line Examples

Following are a few examples of command lines and a description of what each does.

Example 1

This command will output the file `new_um3.ncd` file if timing is met and create a `guidelines.cst` file to refer to for setting preferences. Note that file order is ignored due to -o option indicator.

```
cst um3.ncd um3.prf -o new_um3.ncd
```

Example 2

This shortcut command performs the same function as the one in Example 1, but the preference file must have the same file name as the input `.ncd` file.

```
cst um3
```

Some Notes on CST Functionality

- CST adjusts FFs in PFUs and PIOs completely without violation of special types; FFs in special site types are supported case by case.
- There may be a slight (e.g., one percent) difference between cst and trce with both hold and setup runs due to variation in trace modeling
- CST performs a sanity check on hold times before the optimization stage. CST will issue error messages and ignore the error paths if it fails this hold time check *and* it is not fixable (e.g., the design is useless or the numbers are incorrect). Otherwise, CST will give you a warning and attempt to fix the violation.

Design Verification

Verifying Designs

The ispLEVER software supports two types of timing verification: *static timing analysis* and *dynamic timing simulation*. Both of these methods support all Lattice devices.

Static Timing Analysis

Static timing analysis (timing analysis) is the process of verifying 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/hold time requirements, and the maximum frequency. Lattice has two static timing analysis tools, Performance Analyst and TRACE (for FPGAs).

The primary advantage of timing analysis is that it can be run at any time and requires no input test vectors, which can be very time consuming and tedious to create. Another major advantage of static timing analysis is that it exhaustively checks every possible input-to-output path. One shortcoming of all static timing analysis tools is that they detect false paths that will never be exercised during the course of normal operation of a circuit so that you could spend a lot time instructing the analyzer to ignore those paths. In this process, you could accidentally ignore a real issue. Although timing analysis does not give you a complete timing picture, it is an excellent way to quickly verify the speed of critical paths and identify performance bottlenecks.

Dynamic Timing Simulation

This type of analysis is based on an event-driven simulator and requires you to specify a test vector (waveform). Whereas timing analysis returns partial timing information, dynamic timing simulation (timing simulation) will give you detailed information about gate delays and worst-case circuit conditions. Because total delay of a complete circuit will depend on the number of gates the signal sees and on the way the gates have been placed in the device, timing simulation can only be run after the design has been implemented. Timing simulation also requires several input files to run.

There are two basic types of dynamic timing simulation tools, logic simulators (e.g, Lattice's Logic Simulator) and dynamic simulation analyzers (e.g., MTI's ModelSim™). Logic simulators function in a single delay mode, whereas dynamic simulation analyzers will simulate the ambiguity in delay pairs. Dynamic simulation analyzers typically take longer to process simulation results than logic simulators and considerably longer than static timing analysis tools.

Timing Verification Tools

The ispLEVER software offers timing verification with the following tools:

- Performance Analyst — Static timing analysis tool that runs timing analysis (All CPLD, ispXPLD, ispXPGA, and ispGDX2 devices except ispLSI 1K and 2K).
- Lattice Logic Simulator — Logic simulator that runs timing simulation (ispGDX and CPLD devices only).
- ModelSim for Lattice — Dynamic timing analyzer that runs timing simulation (all devices).
- TRACE — The *Timing Reporter and Circuit Evaluator* (TRACE) is an integrated flow tool that provides static timing analysis based on timing preferences (for FPGA devices only).

Additionally, timing simulation for all devices is supported with these tools:

- Text Editor — Used to create test stimulus files
- Waveform Editor — Used to create test stimulus files graphically
- Waveform Viewer — Used to view the results of simulation

Verification Environments

The timing verification tools operate in both integrated and stand-alone environments.

Integrated Timing Analysis

The Performance Analyst is a static timing analysis tool that lets you quickly determine the performance of designs implemented in any Lattice Semiconductor device. To run timing analysis, launch the Performance Analysis from the Project Navigator. The Performance Analyst traces each logical path in the design and calculates the path delays using the device's timing model and worst-case AC specs supplied in the device data sheet.

The timing analysis results are displayed in a graphical spreadsheet with source signals displayed on the vertical axis and destination signals displayed on the horizontal axis. The worst-case delay value is displayed in a spreadsheet cell if there is at least one delay path between the source and destination. To more easily identify performance bottlenecks, you can double-click a cell to view the path delay details.

Integrated Timing Simulation

To verify a design inside the current project, the ispLEVER software provides integrated verification with the Lattice Logic Simulator and ModelSim™ for Lattice from Mentor Graphics®. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the verification test bench, the following processes are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

CPLD and GDX Project Navigator Process	Simulation Tool Invoked
Timing Simulation	Lattice Logic Simulator
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

Stand-alone Simulation

The ispLEVER software supports stand-alone timing simulation. This provides an easy entry if you need to verify a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator Tools menu.

Required Files for Verification

ModelSim supports Lattice ispXPGA and FPGA device timing simulation. In addition to your design files, you will need at least one test stimulus file, a netlist file, and a timing delay file.

	ModelSim
Test Stimulus File Formats	
• **.tf** (Verilog test fixture)	X
• **.vhd** (VHDL test bench)	X
Netlist File Formats	
• **.vo** (Verilog netlist)	X
• **.vho** (VHDL netlist)	X
Delay File Formats	
• **.sdf** (Standard Delay File)	X

Verification File Descriptions

Test Stimulus Files

- Verilog Test Fixtures (`.tf`) – A Verilog test stimulus file that specifies the input waveforms for simulation in ASCII format.
- VHDL Testbench (`.vhd`) – A VHDL test stimulus file that specifies the input waveforms for simulation in ASCII format.

Netlist Files

- Verilog Netlist (`.vo`) – For Verilog designs, the back-annotated timing simulation netlist named `<design_name>.vo`.
- VHDL Netlist (`.vho`) – For VHDL designs, the back-annotated timing simulation netlist named `<design_name>.vho`.

Timing Delay Files

- Standard Delay Format (`.sdf`) – A file containing delay and timing constraint data for cell instances named `<design_name>.sdf`.

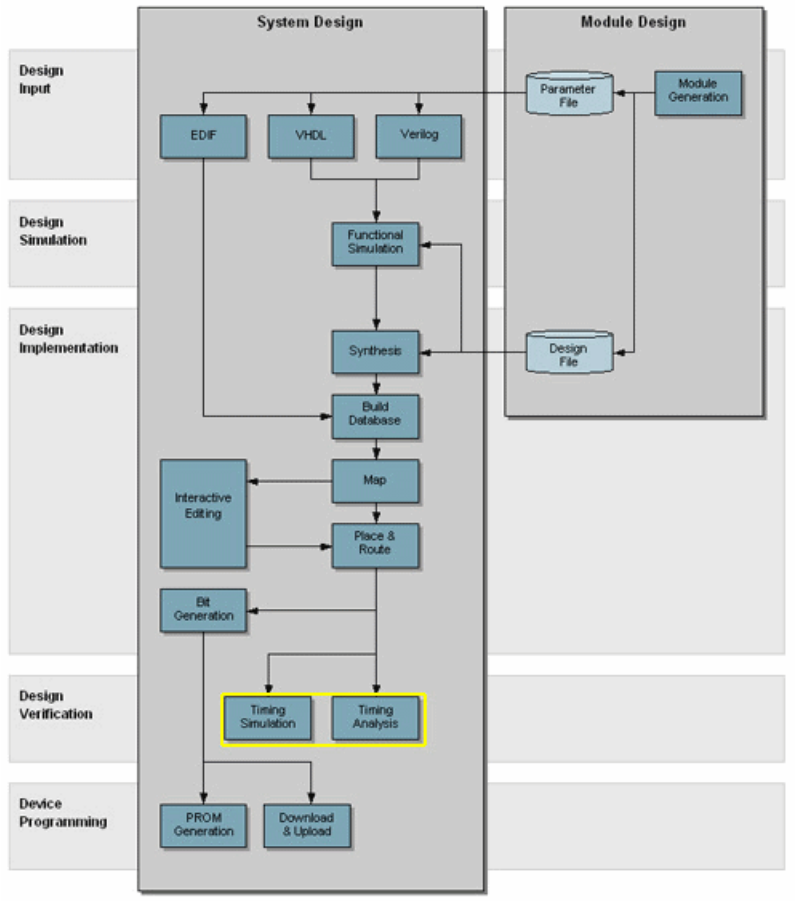
FPGA Verification Summary

The ModelSim simulator supports FPGA timing simulation for EDIF and HDL design entry methods and requires at least one Verilog test fixture (`*.tf`) or VHDL testbench (`*.vhd`) stimulus file. Additionally, ModelSim requires a netlist file (`*.vo` or `*.vho`) and a timing delay file (`*.sdf`).

	ModelSim	
	.tf	.vhd
	.vo + .sdf	.vho + .sdf
EDIF	X	X
Verilog	X	
VHDL		X

FPGA Verification Process Flow

The figure below shows timing analysis and simulation within the FPGA process flow.



Back Annotation

Back annotation is the means by which physical design information (`.ncd` file) is distributed back to the logical design for the purpose of generating a timing simulation file. The ORCA FPGA back annotator takes a design given in the NCD database and converts it into a model that contains functional and timing information about the design. This process uses the appropriate netlist writer to translate the back-annotated information into netlist format for simulation. Output from back annotation uses the ORCA primitive library type in a structural VHDL or Verilog HDL design model and a Standard Delay Format (.sdf file). Timing is annotated at the block level. Annotating timing at the block level ensures that back-annotated timing is in sync with Trace timing analysis.

In addition to back annotating a fully routed design, you can back annotate an unrouted design or create an output netlist to allow simulation of the design at different stages of development in the ispLEVER software environment. For example, if you want to verify that the circuit logic is correct *before* you place and route (PAR) the design, you can use the data in an unmapped `.ngd` design (generic description, Series 2), or unrouted `.ncd` design (Series 3 or 4) as input to back annotation and run a simulation program on the resulting netlist. A netlist backannotated from an unrouted design will not include routing delays.

Back Annotation Input File

The input to the Back Annotation process is the active physical design (`.ncd`) file.

Back Annotation Output Files

The back annotation output files are the following:

- **EDIF (Series 2 only)** — EDIF 2.0.0 netlist file in terms of the Back Annotation NEOPRIM primitive library.
- **Verilog** — Verilog netlist in terms of Back Annotation NEOPRIM (Series 2) or ORCA (Series 3 and 4) primitive library.
- **VHDL** — VHDL netlist file in terms of the Back Annotation NEOPRIM (Series 2) or ORCA (Series 3 and 4) primitive library.
- **SDF — (VHDL/Verilog only)** The SDF file containing timing data; routing interconnect and primitive port delays; and timing checks, including setup, hold, and pulse width checks
- **XRF — (Series 2)** The XNF file containing cross-reference information for VHDL and SDF netlist.

Back Annotation Output Files for ORCA Devices

Architecture	Verilog	VHDL	EDIF-Generic
Series 2	.v, .sdf	.vhd, .xrf or .sdf	.edn
Series 3	.v, .sdf	.v, .sdf	N/A
Series 4	.v, .sdf	.v, .sdf	N/A

Running Back Annotation

You can run back annotation from within the Project Navigator. You can also back annotate from the command line.

To run back annotation using the Project Navigator:

1. In the Project Navigator Source window, select the target FPGA device.
2. In the Processes window, double-click the **Back Annotation** process. The ispLEVER software generates the output files necessary for timing simulation.

Global Reset, PUR, & TSALL in Back Annotation Simulation

For ORCA devices all flop-flops and latches are initialized by a low by a low signal on the GSR input pin of the GSR component. The PUR signal resets the device on power up as a reset and is active HIGH.

A low signal on the TSALL pin on the TSALL component puts all PIC cells on the device in a tri-state condition.

Backannotation will include the functionality of GSR, PUR, and TSALL when these elements are present in the design.

Bus Back Annotation (Series 3 and 4)

The ispLEVER software supports ORCA bus structure output for back annotation of Series 3 and 4 designs. This means that:

if

your input EDIF design was created with top level bus structures (The top level ports have the ARRAY EDIF token)

and

your bus name is of one of the forms of: **a<1:8> a[1:8] a{1:8} a(1:8) or a_8:0**

then

your back annotated design *after* map, place, and route will maintain the bus structure specified in the input.

If one of the above conditions is not met (e.g., the bus was expanded when synthesized), then the bus will continue to be back annotated as separate signals. Map, place, route, and trace still see them as separate signals in either case.

Support for Unknown Inputs (Series 3 and 4)

LUTs and MUXes when back annotated handle unknown logic values (“X”) on their inputs. The back annotation library elements that support this are the MUX21, MUX41, ROM16X1 and ROM32X1 in the Orcaprim version of the VHDL and Verilog back annotation libraries.

LUTs, which are represented in the back-annotated circuit as ROMs, previously remained unknown when any of its inputs remained unknown. Now, the ROM will correctly resolve the output. For example, if addresses “0000” and “0001” both contain a “0”, an input of “000X” will still make the ROM output a “0”. However, if “0001” contains a “1”, then the ROM would continue to output an “X.”

In addition, the MUXes previously went unknown when its select input(s) were unknown. Now it resolves the output correctly, as indicated in the following truth table.

D0	D1	SD	Z
0	0	X	0
1	1	X	1
0	1	X	X
1	0	X	X

Running BACK ANNOTATION from the Command Line

Series 3 and 4 Devices

The LDBANNO program back-annotates physical information (for example, net delays) to the logical design and then writes out the back-annotated design in the desired netlist format.

Input to LDBANNO is a physical design file (*.ncd*)— a mapped and partially or fully placed and/or routed design.

ldbanno [**-w**] [**-sp** <speed grade>] [**-min**] [**-x**] [**-dis** <*delay*> [**-sig** <*sig_file*>]] [**-i**] [**-z**] [**-a**] [**-neg**] [**-n** <*type*>] [**-l** <libtype>] [**-pre** <prefix>] [**-p** <*prffile*[.**prf**.]>] [**-o** <*netlist*>] <*ncdfile*[.**ncd**]>

where:

-w	Overwrite the output file(s).
-sp <*speed grade*>	Re-target back annotation to a different speed grade than the one used to create the *.ncd* file. You are limited to those speed grades available for the port used in the *.ncd* file.
-min	Replaces all timing information for back annotation with the minimum timing for all paths. This option is used for simulation of hold time requirements. Separate simulations are required for hold time verification (**-min** switch) and delay time verification (normal output).
-x	Writes out a trce cross-reference file.
[**-dis** [<*delay*>]] [**-sig** <*sig_file*>]	The **-dis** option distributes routing delays by splitting the signal and inserting buffers. The <delay> value represents the maximum delay number in picoseconds between each buffer (1000 ps by default) The -sig option designates the use of a <sigfile> which is an ASCII file that contains all the top level signals to be split. One signal name can used per line. If -dis is given but -sig is absent, all top level signals will be distributed. These options will enable simulation of high frequency signals (e.g., clock signals), where the interconnection delay may be higher than the clock cycle. In such a case, that signal will be blocked in some simulators if the delay is not distributed.
-I	Inserts a buffer at each block input that has interconnection delay on it. **Note**: This option is valid only when the netlist type is Verilog. It is similar to the "tipd" statement in a VITAL compliant model in VHDL, which can help you to debug the design by isolating the interconnection delay and pin-to-pin delay on each input.
-z	Zero delay (do not calculate or write any delays). See Note for the -a option.

-a	Write all delays, even if they are zero. **Note**: By default, LDBANNO will write all non-zero delays to the appropriate output delay file. If the **-a** option is given, all pin to pin delays in the output netlist will be written even if they are zero. If the **-z** option is given, no delays will be calculated or written to the output file. If both the **-a** and **-z** options are given on the command line, the **-z** option will take precedence and no delays will be written to the output file.
-neg	For better compatibility with simulators, all negative setup/hold delay numbers in the SDF file are set to 0 by default. This may cause some discrepancies between back annotation and the TRACE result. Use the new -neg option to get the negative numbers in the SDF file and back annotation will match the TRACE report. Ensure that the simulator is able to handle negative numbers in the SDF file.
-n <*type*>	Netlist type to write out (*not* case-sensitive): • Verilog generic verilog formal with SDF delay file • VHDL generic VHDL format with SDF delay file
-l <*libtype*>	Valid library types are orca. **Note**: Back Annotation will accept "-l or3c00" for compatability with older scripts, but "-l orca" is preferred.
-o <*netlist*>	(optional) The name of the output file. The extension used for each output depends on the type of netlist being written (specified with the **-n** switch).
-pre <*prefix*>	(optional) Prefix to add to module names to make them unique for multi-chip simulation.
<*preffile*[.**prf**]>	(optional) The preference file in *.prf* format.
<*ncdfile*[.**ncd**]>	The input design file is a mapped and partially or fully placed and/or routed design in *.ncd* format.

Examples

Following are a few examples of LDBANNO command lines and a description of what each does.

Example 1

The following command back annotates *design.ncd* and generates a Verilog file *design.v* and an SDF file *design.sdf*. If the target files exist, they will be overwritten.

```
ldbanno -w -n verilog design.ncd
```

Example 2

The following command back annotates *design.ncd* and generates a VHDL file *backanno.vhd* and an SDF file *backanno.sdf*. Any signal in the design that has an interconnection delay greater than 2000 ps (2 ns) will be split and a series of buffers will be inserted. The maximum interconnection delay between each buffer would be 2000 ps.

```
ldbanno -dis 2000 -n vhdl -o backanno design.ncd
```

Example 3

The following command re-targets back annotation to speed grade -–2, and puts a buffer at each block input to isolate the interconntion delay (ends at that input) and the pin to pin delay (starts from that input).

```
ldbanno -sp 2 -i -n verilog design.ncd
```

Example 4

The following command generates Verilog netlist and SDF files without setting the negative setup/hold delays to 0:

```
ldbanno -neg -n verilog design.ncd
```

Series 2 Devices

When you back-annotate from the command line, you first run the NGDANNO command to back-annotate physical information (for example, net delays) to the logical design. Then you invoke one of the netlist writers NGD2EDIF, NGD2VER, or NGD2VHD) to write out the back-annotated design in the desired netlist format.

When you back-annotate using the MAP/Backanno Shell, the shell runs each of these commands automatically.

NGDANNO

The NGDANNO program distributes delays, setup and hold times, and pulse widths found in the physical *.ncd* design file back to the logical *.ngd* file. It merges mapping information from the *.ngm* file and placement, routing, and timing information from the *.ncd* file and puts all this data in an *.nga* (generic annotated) file.

If you make logical changes to an *.ncd* design from within EPIC that change the functional behavior of the design, NGDANNO cannot correlate the changed objects in the physical design with the objects in the logical design. It recreates the entire *.nga* design from the *.ncd*. You get this warning:

```
WARNING – NCD is out of sync with NGM...creating NGA from NCD
```

Input to the NGDANNO program is:

- *ncd* — a mapped and partially or fully placed and/or routed design
- *ngm* — (optional) a mapped *.ngd* file created by the technology mapper

The *.ngm* file supplies the back-annotation program with information about the logical design (for example, the logical design hierarchy, a gate level description of the design, and the correlation between the logical design and its physical mapping). Information about the physical design (for example, component and net delays, and site assignments for mapped components) is supplied by the input *.ncd* file. The logical information from the *.ngm* file and the physical information from the *.ncd* file complete the database necessary for simulation.

Output from the NGDANNO program is an *.nga* file, which is a back-annotated *.ngd* file. The *.nga* file acts as input to the netlist translation programs.

To run NGDANNO, from the UNIX or DOS command line enter:

ngdanno [**-f** <*command_file*>] [**-o** <*ngafile*[.**nga**]>] [**-p** <*prffile*[.**prf**.]>] < *ncdfile*[.**ncd**]> [<*ngmfile*[.**ngm**]>]

where:

-f <*command_file*>	(optional) Execute command line arguments in the specified *command_file*.
-o <*ngafile*[.nga]>	The output *.nga* file, which is a back-annotated *.ngd* file. If you specify only the *.ncd* file on the command line but do not specify an *.nga* file with the **-o** option, an *.nga* file is generated from the *.ncd* in the same directory. The *.nga* file has the same root name as the *.ncd* file. A physical *.nga* file is created, but there is no logical view of the design.
<*preffile*[.**prf**]>	(optional) The preference file in *.prf* format. You must include the appropriate extension with the file name on the command line.
<*ncdfile*[.**ncd**]>	The input design file is a mapped and partially or fully placed and/or routed design in *.ncd* format.
<*ngmfile*[.**ngm**]>	(optional) The mapped *.ngd* file created by the technology mapper. You must specify the *.ngm* file to have it used.

Netlist Writers

Netlist writers are ORCA programs that translate information contained in *.nga* or *.ngd* files into a specified netlist format.

NGD2EDIF

NGD2EDIF produces an EDIF 2 0 0 netlist in terms of the neoprim primitive set, allowing you to simulate designs before and after routing.

Input can be any of the following files:

- *.nga*, a back-annotated logical design file containing neoprim library components
- *.ngd*, a logical design file containing neoprim library components

Output is an *.edn* file, a netlist in EDIF format. The default *.edn* file produced by NGD2EDIF is generic. If you want to produce EDIF targeted to Mentor Graphics or Logic Modeling Group, you must include the **-v** option on the command line.

For implicit global signals on a device, NGD2EDIF automatically creates extra pins, which can cause interface differences when trying to match the original design.

ngd2edif [**-a** | **-n**] [**-w**] [**-v** <*vendor*>] [**-f** <*command_file*>] <*infile*>**.ngd** | .**nga** [<*outfile*>[**.edn**]]

where:

-a	Write all properties of the design. The default is to write only delay and error checking properties.
-n	Write out a flattened netlist with no properties of the design. Must be used with **-v viewlog**.
-w	Overwrite an existing output file.
-v <*vendor*>	Specifies the CAE vendor toolset to use the resulting EDIF file. Allowable entries are **mentor**, **lmg**, and **viewlog**. You must use the **-n** option when using **-v viewlog**.
-f <*command_file*>	Execute command line arguments in the specified *command_file*.
<*infile*[**.ngd**	**.nga**]>	The input file (*.nga* file comes from NGDANNO).
<*outfile*[.**edn**]>	The output EDIF file. The default output file has the same name as the input file, with an *.edn* extension.

NGD2VER

NGD2VER produces a Verilog netlist and an SDF (Standard Delay Format) file in terms of the neoprim primitive set, allowing you to simulate designs before and after routing.

Input is an *.nga* file, a back-annotated logical design file containing neoprim library components.

Two output files are generated by the NGD2VER program:

- *.v* — a Verilog netlist
- *.sdf* — an SDF file containing timing data; routing interconnect and primitive port delays; and timing checks, including setup, hold, and pulse width checks

The files have the same root name as the *.ngd* or *.nga* file unless you specify otherwise.

ngd2ver [**-w**] [**-n**] [**-f** <*command_file*>] <*infile*>**.ngd** | .**nga** [<*outfile*[.**v**]>

where:

-w	Overwrite an existing output file.
-n	Write out a flattened netlist of the design.
-f <*command_file*>	Execute command line arguments in the specified command_file.
<*infile*[**.nga**]>	The input *.nga* file from NGDANNO or an *.ngd* file.
<*outfile*[.**v**]>	The output Verilog file. The default output file has the same name as the input file. The SDF file has the same name as the Verilog file with an *.sdf* extension. **Note**: The SDF file produced is intended solely for use with the Verilog file. Do not attempt to use the SDF file in conjunction with the original design or the product of another netlist writer.

NGD2VHD

NGD2VHD produces a VHDL netlist and an SDF (Standard Delay Format) file in terms of the neoprim primitive set, allowing you to simulate designs before and after routing.

Input is an *.nga* file, a back-annotated logical design file containing neoprim library components.

Three output files are generated by the NGD2VHD program:

- *.vhd* — a VHDL netlist
- *.sdf* — an SDF file containing timing data; routing interconnect and primitive port delays; and timing checks, including setup, hold, and pulse width checks
- *.xrf* — cross reference file

The files have the same root name as the *.ngd* or *.nga* file unless you specify otherwise.

ngd2vhd [**-n**] [**-p**] [**-t**] [**-w**] [**-x**] [**-f** <*command_file*>] <*infile*>**.ngd** | .**nga** [<*outfile*[.**vhd**]>

where:

-n	Write out a flattened netlist of the design.
-p	Connect a pullup/down device to an unsourced net.
-t	Target this VHDL file to Mentor Time Explorer.
-w	Overwrite an existing output file.
-x	Do not generate the cross reference file.
-f <*command_file*>	Execute command line arguments in the specified *command_file*.
<*infile*[**.nga**]>	The input *.nga* file from NGDANNO or an *.ngd* file.
<*outfile*[.**vhd**]>	The output VHDL file. The default output file has the same name as the input file. The SDF file has the same name as the VHDL file with an *.sdf* extension. **Note**: The SDF file produced is intended solely for use with the VHDL file. Do not attempt to use the SDF file in conjunction with the original design or the product of another netlist writer.

Static Timing Analysis

TRACE

TRACE (*Timing Reporter and Circuit Evaluator*) is an application that provides static timing analysis based on timing preferences for ORCA FPGA devices only. TRACE performs two major processes: timing verification and report generation on timing for analysis. Timing verification is the process of verifying that the design meets your timing preferences. Reporting is the process of enumerating input preference violations and placing them into an accessible file.

TRACE can be run on completely placed and routed designs or on designs that are placed and/or routed to any degree of completion. The report issued by TRACE depends on the completeness of the placement and routing of the input design (see the TRACE reports section).

In Project Navigator, TRACE runs automatically in the Processes window when you run through a design flow and is represented in this window with the **MAP TRACE Report** and **Place & Route TRACE Report** processes. You can set TRACE reporting parameters using **Tools > TRACE Options**. You can also run TRACE independently on a design from the command line. See Running TRACE from the Command Line.

Timing Analysis

TRACE checks the delays in the physical netlist (`.ncd`) against your timing preferences. If delays are exceeded, TRACE issues the appropriate timing error.

The delay for a constrained net is checked to ensure that the constraint is equal to or greater than the route delay (route delay is the signal delay between the driver pin and the load pin(s) on a net). Any nets showing delays that do not meet this condition generate timing errors in the timing report.

The delay through a constrained path is checked to ensure that the constraint is equal to or greater than the sum of logic (component or pin-to-pin) delay, route or wire delay (signal delay between component pins in a path), and setup time (if any). Setup time for clocked paths is the time that data must be present on an input pin before the arrival of the triggering edge of a clock signal. Any paths showing delays that do not meet this condition generate timing errors in the timing report.

Clock signal skew on a clocked net with multiple load pins is the difference between minimum and maximum load delays (delay between the driver pin and a load pin). Clock skew is checked against the specified maximum skew for constrained nets in the preferences (`.prf` file). If the skew is found to exceed the maximum skew constraint, the timing report shows a skew error.

***Note**: Transparent latches are fully supported in ispLEVER. Transparent latch delays appear as *****_DDEL** in TRACE reports.*

Running TRACE

You can run TRACE from the Project Navigator or from the command line using the **trce** command.

In the Project Navigator, you can run TRACE after the design has been mapped (before route) and after the design has been placed and routed (after route) in the design flow. You can set TRACE options and checkpoints before running a design flow.

To analyze timing using TRACE from the Project Navigator:

1. In the Project Navigator, choose **Tools > Timing Checkpoint Options** to open the dialog box. Set the desired run options and then click **OK** to close the dialog box.
2. Choose **Tools > TRACE Options** to open the dialog box. Set the desired reporting options and then click **OK** to close the dialog box.

3. In the Sources window, select the **target device**.
4. To run TRACE *before* the design is routed, in the Processes window, double-click the **Map TRACE Report** process to run TRACE and generate a report based on a mapped design.
5. To run TRACE *after* the design is routed, in the Processes window, double-click the **Place & Route TRACE Report** process to run TRACE and generate a report based on a placed and routed design.

Setting TRACE Timing Checkpoint Options

You can set TRACE Timing Checkpoints before running a design flow. The design flow contains intermediate timing analysis checkpoints that run TRACE. These checkpoints appear after the Map Design and after Place & Route Design processes.

After the mapping and routing stages, you may opt to set the checkpoints to always continue to the next operation, stop, or prompt you with options.

To set TRACE Timing Checkpoint Options:

1. In the Project Navigator, choose **Tools > Timing Checkpoint Options** to open the dialog box.
2. Set the options that you want.
3. Click **OK** to close the dialog box. These options are recognized the next time you run TRACE.

Setting TRACE Options

You can use the TRACE Options dialog box to set various before and after route TRACE reporting options.

To set TRACE report format options:

1. In the Project Navigator, choose **Tools > TRACE Options** to open the dialog box.
2. Set the options that you want.
3. Click **OK** to close the dialog box. These options are recognized the next time you run TRACE.

Running the Design Flow without TRACE

You can run the design flow without running TRACE by using the Timing Checkpoint Options dialog box.

To disable TRACE in the design flow:

1. In the Project Navigator, choose **Tools > Timing Checkpoint Options** to open the dialog box.
2. Under Before Route and After Route, clear **Run Checkpoint**.
3. Click **OK** to close the dialog box. When you run the design flow, TRACE is disabled.

TRACE Input Files

TRACE Input Files include the following:

- A physical design file (`.ncd`). If you are performing timing verification, the design should also be placed and routed.
- A preference file (`.prf`) with timing constraints that the designer specifies. Preferences can indicate such things as clock speed for input signals, the external timing relationship between two or more signals, absolute maximum delay on a design path, or a general timing requirement for a class of pins.

Below are brief descriptions of timing preferences that place and route supports.

FREQUENCY — identifies the minimum operating frequency for all sequential input pins clocked by a specified net.

PERIOD — specifies a maximum clock period for all sequential input pins clocked by a specified net.

MAXDELAY — identifies a maximum total delay for a circuit net, path, path class, or bus in the design. Also specifies a maximum delay from a starting point or to an endpoint.

BLOCK — blocks timing checks on path classes, nets, buses, or component pins that are irrelevant to the timing of the design.

OFFSET — identifies the external timing relationship between a clock pin and a data pin.

DEFINE STARTPOINT — identifies a starting point for the MAXDELAY FROM/ TO preference (see MAXDELAY).

DEFINE ENDPOINT — identifies an ending point for the MAXDELAY FROM/ TO preference (see MAXDELAY).

DEFINE PATH — specifies a path from a source component to a destination component.

DEFINE BUS — specifies a grouping of nets.

MULTICYCLE — allows for relaxation of previously defined PERIOD or FREQUENCY constraints on a path.

TRACE Output Files

TRACE produces a timing report output file (`.twr`), which enables you to see how well the timing constraints for the design have been met.

TRACE Timing Report

A timing report (`.twr`) enables you to determine to what extent the timing constraints for a design have been met.

The timing report produced by TRACE is a report prepared for a particular design. The timing report lists the following:

- Statistics on the design.
  - Timing errors, which indicate absolute timing constraint violations. Timing errors may indicate a need for design modifications and/or multiple placement and/or reentrant routing.
- Timing warnings, which point out potential timing problems.

Two levels of reporting are available:

- Error report
- Verbose report

In each type of report, the header specifies the type of report, the input design name, and the optional input preference file name.

At the end of each report is a timing summary, which includes the following information:

- The number of timing errors found in the design.
- A timing score, showing the total amount of error (in picoseconds) for all timing preferences in the design.
- The number of paths and connections covered by the constraints and the percentage coverage over the whole design.

The error report lists timing errors and associated net/path delay information.

The error and verbose reports also contain an asynchronous loop report that shows all paths that cannot be analyzed.

The verbose report lists delay information for all nets and paths.

For any of the three types of reports, if you specify a preference file that contains invalid data, a list of preference file errors appears at the beginning of the report.

If preferences for process, temperature, or voltage exist in a preference file, these preferences are included in the timing report.

In both error and verbose reports, the details of the paths that contribute to clock skew are reported.

TRACE supports the preferences CLKSKEWDIFF and CLKSKEWDISABLE, which allow you to control clock skew computation. These preferences are described in the Logical and Physical Preferences chapter of this guide.

TRACE Error Report

The main body of the error report lists all timing preferences as they appear in the input `.prf` file. If the preference is met, the report simply states the number of items scored by TRACE, reports no timing errors detected, and issues a brief report line, indicating important information (for example, the maximum delay for the particular preference). If the preference is not met, it gives the number of items scored by TRACE, the number of errors encountered, and a detailed breakdown of the error.

At the top of each type of report is descriptive information such as the software version of the application, the name of the design file, the input preference file name, the device speed and the report level. A timing summary always appears at the end of the report.

In this report, the input design is placed and routed. If the design were placed but not routed, the report would show *estimated* delays signified by the letter **e**. If the input design were neither placed nor routed, estimated delays would all show up as 0 and would also be signified by the letter **e**.

At the bottom of the Delays column, it shows the total amount of delay for the path, the percentages of the total allocated to logic and to route delays (in parentheses), and the number of logic levels (components) involved in the preference.

In addition, the TRACE output format supports logical names in the TRACE report file. The format features are as follows:

- A Logical Details section exists in the trce `.twr` report file. This section reports Circuit Path Source and Destination information:
  - Instance names as they appear in the EDIF file for the following cell types: Registered elements, Ports, RAM instances
  - A description of the type of function performed at the destination (e.g., Data in, Write enable, or Carry in)
  - Indication of the triggering edge of the clock using (+/-)

The section includes the following information: Source component logical name, cell type, pin type, as well as the clock name and edge, destination component logical name, cell type, pin type*, as well as the clock name and edge (*source pin types: Q, O).

In addition, TRACE reports clock skew details for placed/routed designs.

A sample Error Report (`.twr` report file) illustrates the current format of the output file. The Logical Details section, which reports circuit path source and destination information, appears at the beginning.

Sample Error Report

```
--------------------------------------------------------------------
TRACE, Version ORCA Foundry 2001 Beta (65)
Copyright (c) 1991-1994 by NeoCAD Inc. All rights reserved.
Copyright (c) 1995 AT&T Corp. All rights reserved.
Copyright (c) 1996-2001 Lucent Technologies Inc. All rights reserved.
Copyright (c) 2001 Agere Systems All rights reserved.
Copyright (c) 2002, Lattice Semiconductor Corporation, All rights reserved.

Design file: Par_Rev_1.ncd
Preference file: tope.prf
Device,speed: or4e06,2
Report level: error report
--------------------------------------------------------------------

====================================================================
Preference: FREQUENCY PORT "clock" 290.000000 MHz ;
  45 items scored, 7 timing errors detected.
--------------------------------------------------------------------

Error: The following path exceeds requirements by 0.368ns

Logical Details: Cell type Pin type Cell name (clock net +/-)
Source:  Port  Pad  din(0)
Destination: FF  Data in MEMORY_ix37_ix31(to clock_int +)
   FF    MEMORY_ix37_ix31
   FF    MEMORY_ix37_ix31
   FF    MEMORY_ix37_ix31

Delay: 4.046ns (39.1% logic, 60.9% route), 1 logic levels.

Constraint Details:
4.046ns physical path delay din(0) to PFU_1 exceeds
3.448ns delay constraint less
-0.230ns WD_SET requirement (totaling 3.678ns) by 0.368ns

Physical Path Details:
Name  Fanout Delay(ns) Site  Resource
IN_DEL --- 1.583  V1.PAD to V1.INDD din(0)
ROUTE  1  2.463  V1.INDD to R32C2.DIN0 din(0)_int(to
clock_int)
   --------
4.046 (39.1% logic, 60.9% route), 1 logic levels.

Error: The following path exceeds requirements by 0.001ns
```

```
Logical Details: Cell type Pin type Cell name (clock net +/-)
Source: FF Q CNTR_count_value_ix5 (from lock_int +)
Destination: FF Read Addr MEMORY_ix37_ix31 (to clock_int +)
FF MEMORY_ix37_ix31
FF MEMORY_ix37_ix31
FF MEMORY_ix37_ix31

Delay: 2.629ns (33.2% logic, 66.8% route), 1 logic levels.

Constraint Details:
2.629ns physical path delay PFU_0 to PFU_1 exceeds
3.448ns delay constraint less
0.607ns skew and
0.213ns RA_SET requirement (totaling 2.628ns) by 0.001ns

Physical Path Details:
Name Fanout Delay (ns) Site Resource
REG_DEL --- 0.873 R32C3.CLK0 to R32C3.Q3 PFU_0 (from clock_int)
ROUTE 10 1.756 R32C3.Q3 to R32C2.K3_3 count_addr(3) (to clock_int)
--------
2.629 (33.2% logic, 66.8% route), 1 logic levels.

Clock Skew Details:
Source Clock:
Delay Connection
2.443ns U3.INCK to R32C3.CLK0

Destination Clock:
Delay Connection
1.836ns U3.INCK to R32C2.CLK0

Warning: 262.055MHz is the maximum frequency for this preference.

1 preference not met.

Timing summary:
---------------

Timing errors: 7 Score: 873

Constraints cover 45 paths, 4 nets, and 47 connections (90.4% coverage)
```

***Note**: The above error report has been shortened for example only. An actual error report would be much longer.*

If a path begins or ends at a sequential element (a flip-flop, for example), the clock signal associated with the flip-flop is reported in the path delay report. For example:

```
10.150ns delay CLKD1 to CLKD1' exceeds
 10.000ns delay constraint less
  3.000ns setup requirement (totaling 7.000ns) by 3.150ns

  Delay   Site    Resource
 3.000ns CLB_R2C1.K to CLB_R2C1.XQ CLKD1 (from CLK)
 1.310ns CLB_R2C1.XQ to CLB_R2C1.G2 CLKD1
 4.500ns CLB_R2C1.G2 to CLB_R2C1.Y CLKD1
 1.340ns CLB_R2C1.Y to CLB_R2C2.C1 CLKD1D2 (to CLK)
 -------
 10.150ns (73.9% logic, 26.1% route), 2 logic levels.
```

Source and destination clock signals are listed because the path originates at a flip-flop (CLKD1.K, signal CLK) and terminates at a flip-flop with a 3 ns setup requirement relative to the signal CLK. This allows you to determine what the source and destination clock signals are for sequential paths.

The following example has source and destination clock signals that are different. Identifying these clocks allows you to determine whether or not the timing constraint on the path needs modification based on the phase relationships of the different clock signals.

```
4.830ns delay CLKD2 to CLKD2' meets
 20000ns delay constraint less
 3.000ns setup requirement (totaling 17.000ns) by 12.170ns

  Delays   Site     Resource
 3.000ns CLB_R3C1.K to CLB_R3C1.XQ CLKD2 (from CLK2)
 1.830ns CLB_R3C1.XQ to CLB_R3C2.C1 CLKD2 (to CLK)
 -------
 4.830ns (62.1% logic, 37.9% route), 1 logic levels.
```

TRACE Verbose Report

The verbose report is similar to the error report but provides more details on delays for all constrained paths and nets in the design. As in the other two types of reports, descriptive material appears at the top.

The body of the verbose report enumerates each preference as it appears in the input preference file, the number of items scored by TRACE for that preference, and the number and description of errors detected for the preference. A report line for each preference provides important information, such as the amount of delay on a net and by how much the constraint is met.

For path preferences, if there is an error, the report indicates the amount by which the preference is exceeded. If there are no errors, the report indicates that the preference passed and by how much. Each logic and route delay is analyzed, totaled, and reported. Paths that are blocked using the BLOCK PATH preference will be reported in the verbose mode and connections will be removed from the **-c** listing (uncovered listing).

In the sample, only a few preferences and their accompanying reports are shown. If this were an actual report, *all* timing preferences for the design would be enumerated and the report would be much longer. Note that the report limit can be specified to limit the number of items reported for each preference in either error or verbose mode.

Also note that as in the error report, five columns follow the FREQUENCY preference.

***Note**: For ORCA 2C devices, the TRACE error report and verbose report list the Clk-to-Q logic delay that is different from the data sheet. This is because ORCA Foundry uses a switch box, whose delay is reported not as part of Clk-to-Q delay but as part of routing delay. The total delay from Clk to the output of the switch box complies with the specification.*

Sample Verbose Report

```
--------------------------------------------------------------------
TRACE, Version ORCA Foundry 2001 Beta (65)
Copyright (c) 1991-1994 by NeoCAD Inc. All rights reserved.
Copyright (c) 1995 AT&T Corp. All rights reserved.
Copyright (c) 1996-2001 Lucent Technologies Inc. All rights reserved.
Copyright (c) 2001 Agere Systems All rights reserved.
Copyright (c) 2002, Lattice Semiconductor Corporation, All rights reserved.

Design file: Par_Rev_1.ncd
Preference file: topv.prf
Device,speed: or4e06,2
Report level: verbose report
--------------------------------------------------------------------

====================================================================
Preference: FREQUENCY PORT "clock" 50.000000 MHz ;
45 items scored, 0 timing errors detected.
--------------------------------------------------------------------
Passed: The following path meets requirements by 16.184ns

Logical Details: Cell type Pin type Cell name (clock net +/-)
Source: Port Pad din(0)
Destination: FF Data in MEMORY_ix37_ix31 (to clock_int +)
FF MEMORY_ix37_ix31
FF MEMORY_ix37_ix31
FF MEMORY_ix37_ix31

Delay: 4.046ns (39.1% logic, 60.9% route), 1 logic levels.

Constraint Details:
4.046ns physical path delay din(0) to PFU_1 meets
20.000ns delay constraint less
-0.230ns WD_SET requirement (totaling 20.230ns) by 16.184ns

Physical Path Details:
Name Fanout Delay (ns) Site Resource
IN_DEL --- 1.583 V1.PAD to V1.INDD din(0)
```

```
ROUTE 1 2.463 V1.INDD to R32C2.DIN0 din(0)_int (to clock_int)
--------
4.046 (39.1% logic, 60.9% route), 1 logic levels.

Passed: The following path meets requirements by 17.476ns

Logical Details: Cell type Pin type Cell name (clock net +/-)
Source: FF Q CNTR_count_value_ix11 (from clock_int +)
Destination: FF Write Addr MEMORY_ix37_ix31 (to clock_int +)
FF MEMORY_ix37_ix31
FF MEMORY_ix37_ix31
FF MEMORY_ix37_ix31

Delay: 2.012ns (43.4% logic, 56.6% route), 1 logic levels.

Constraint Details:
2.012ns physical path delay PFU_0 to PFU_1 meets
20.000ns delay constraint less
0.607ns skew and
-0.095ns WA_SET requirement (totaling 19.488ns) by 17.476ns

Physical Path Details:
Name Fanout Delay (ns) Site Resource
REG_DEL --- 0.873 R32C3.CLK0 to R32C3.Q1 PFU_0 (from
clock_int)
ROUTE 10 1.139 R32C3.Q1 to R32C2.DIN3 count_addr(1) (to clock_int)
--------
2.012 (43.4% logic, 56.6% route), 1 logic levels.

Clock Skew Details:
Source Clock:
Delay Connection
2.443ns U3.INCK to R32C3.CLK0

Destination Clock :
Delay Connection
1.836ns U3.INCK to R32C2.CLK0

Report: 262.055MHz is the maximum frequency for this preference.

All preferences were met.

Timing summary:
---------------
```

```
Timing errors: 0 Score: 0

Constraints cover 45 paths, 4 nets, and 47 connections (90.4% coverage)

Created Tue Jul 10 12:57:13 2001
```

***Note**: The above error verbose report has been shortened for example only. An actual verbose report would be much longer.*

TRACE Loop Report

This section of the report lists all paths that cannot be analyzed because of asynchronous loops in the circuit. Use the **-p** option to report the number of asynchronous loops.

Running TRACE from the Command Line

Syntax

trce [-e | -v[<limit>]] [-p] [-sp <speed>] [-c] [-hld] <file_name[.ncd]> [-o <report[.twr]>] [<file_name[.prf] >] [-f <command_file>] [-sm [-hld] [-v]<stamp_file> <ncd_file> [<prf_file>]]

where:

-e and **-v** <*limit*>	Respectively, generates an error report or a verbose (full) timing report file in the current working directory. If neither of these switches is specified, you get a summary report, sent to standard output. <*limit*> limits the number of items reported for each timing preference). Enter 0 or greater (0 = no limit).
-p	Reports the number of asynchronous loops in verbose mode.
-sp <*speed*>	Overrides the default speed grade.
-c	Prints uncovered connections. The -c option should be used with the verbose timing report option.
<-v> **-hld** <*file_name*[.**ncd**]> <*file_name*[.**prf**]>	Performs hold time checks on FREQUENCY, CLOCK_TO_OUT, INPUT_SETUP and OFFSET preferences and outputs a verbose timing report in your working directory.
<*file_name*[.**ncd**]>	Specifies the physical design file.
-o <*file_name*[.**twr**]>	The name of the output file to contain the timing report. If no output name is specified, the report name defaults to the same root name as the *.ncd* file.
<*file_name*[.**prf**]>	The *.prf* file containing timing constraints for the design file. If a preference file is not specified, TRACE looks for one with the same root name as the *.ncd* file.
-f <*command_file*>	Execute command line arguments in the specified *command_file*.
[**-sm** [-hld] [-v]<*stamp_file*> <*ncd_file*> [<*prf_file*>]]	The **-sm** option enables STAMP model generation capability which simplifies verifying timing of a design at the board level. By default, TRACE will not generate STAMP model files. When **trce** is run with the **-sm** option a *.mod* file and a *.data* file are generated in addition to the *.twr* file.

Examples

Following are a few examples of TRACE command lines and a description of what each does.

Example 1

The following command verifies the timing characteristics of the design named *design1.ncd*, generating a summary timing report. Timing preferences contained in the file *group1.prf* are the timing constraints for the design. This generates the report file *design1.twr*.

```
trce design1.ncd group1.prf
```

Example 2

The following command produces a file listing all delay characteristics for the design named *design1.ncd*. Timing preferences contained in the file *group1.prf* are the timing constraints for the design. The file *output.twr* is the name of the verbose report file.

```
trce -v design1.ncd group1.prf -o output.twr
```

Example 3

The following command analyzes the file *design1.ncd* and reports on the three worst errors for each preference in *timing.prf*. The report is called *design1.twr*.

```
trce -e 3 design1.ncd timing.prf
```

Example 4

The following command analyzes the file *design1.ncd* and produces a verbose report to check on hold times on any FREQUENCY, CLOCK_TO_OUT, INPUT_SETUP and OFFSET preferences in the *timing.prf* file. With the output report file name unspecified here, a file using the root name of the *.ncd* file (i.e., *design1.twr*) will be output by default.

```
trce -v -hld design1.ncd timing.prf
```

Example 5

The following example commands summarize the three ways in which you can use the STAMP model generation **-sm** option for Trace.

The command line syntax to generate STAMP model files with max delay information and setup requirements is as follows:

```
trce -sm stamp.file design1.ncd
```

The command line syntax to generate STAMP files with min delay information and hold requirements is as follows:

```
trce -hld -sm stamp.file design1.ncd
```

The command line syntax to generate a conditional STAMP output in which certain paths are blocked is as follows:

```
trce -v -sm stamp.file design1.ncd timing.prf
```

Device Programming

Bit Generation

Generating a Bitstream

The Bit generation (**bitgen**) takes a fully routed physical design, in the form of a circuit description (`.ncd`) file, as input and produces a configuration bitstream (bit images). The bitstream file contains all of the configuration information from the physical design defining the internal logic and interconnections of the ORCA FPGA, as well as device-specific information from other files associated with the target device.

The data in the bitstream can then be downloaded directly into the ORCA FPGA's memory cells or used to generate files for PROM programming. You can run **bitgen** from the Project Navigator by double-clicking the Generate Bitstream Data process or from the command line.

Generating Bitstream Files

From the Project Navigator, bitstream generation is automatically performed when you run a full design flow or a partial design flow selecting the Generate Bitstream Data process. You can set BIT options before running a design flow by setting properties.

To generate bitstream files from the Project Navigator:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window right-click the **Generate Bitstream Data** process and select **Properties** to open the Properties dialog box.
3. Set the Generate Bitstream Data properties that you want and then click **Close** to close the dialog box.

   ***Note**: To view a Process definition, select an ispLEVER Process and press the F1 key on your keyboard. To view a Property definition, in the Properties dialog box select a Property and press the F1 key.*

4. In the Project Navigator, double-click the **Generate Bitstream Data** process to generate the bitstream files.

Bit Generation Output Files

The bit generation output includes the following file formats:

- Bit File (binary) — binary (`.bit`) bitstream. Binary bitstream files are the default output of the bitstream process and contain the configuration information in bitstream (zeroes and ones) that is represented in the physical design (`.ncd`) file.
- Raw Bit File (ASCII) — ASCII (`.rbt`) bitstream. The Raw Bit File is a text file containing ASCII ones and zeros representing the bits in the bitstream file. If you are using a microprocessor to configure a single ORCA FPGA, you can include the Raw Bit file in the source code as a text file to represent the configuration data. The sequence of characters in the Raw Bit file is the same as the bit sequence that will be written into the ORCA FPGA. The `.rbt` file differs from the `.bit` file in that it contains design information in the first six lines.
- Mask File — mask (`.msk`) bitstream. Used to compare relevant bit locations for executing a read back of configuration data contained in an operating ORCA FPGA.

JTAG Setup

Using the **bitgen** program from the command line, you can generate setup bitstreams (`.jbt`) files to set JTAG port read and write the ORCA FPGA device. Downloading these setup bitstreams requires the serial cable with a JTAG cable connector.

Programming Series 4 System Block RAM

You must program Series 4 system block RAM using the `.mif` file generated in SCUBA as **bitgen** input or by specifying INITVAL properties in the `.ncd` file. The former can be accomplished using the **-r** option in **bitgen** on the command line, or in the Project Navigator using the **Generate Bitstream Data** process, **MIF Filename** property.

To program Series 4 system block RAM:

1. Generate a block RAM SCUBA module to create a `.mif` file. SCUBA will automatically generate a `.mif` file with a file name identical to your module name.
2. Use the **-r** option in **bitgen** to initialize the system block RAM, or select the `.mif` file from the properties window of the Generate Bitstream Data Data process. To initialize a system bus block using the MIF Filename property from the command line, type the following command syntax:

   ```
   bitgen -r <sysbus.mif> <mydesign.ncd> <mydesign.bit>
   ```

   Using the SCUBA-generated `sysbus.mif` file, this command runs bitgen on `mydesign.ncd` and writes out `mydesign.bit` to initialize the system bus block RAMs.

***Note**: If block RAMs are already initialized by INITVAL properties in the `.ncd` file, addresses that are explicitly addressed in the `.mif` file will be overridden by the INITVAL properties. In addition, an 8X8 multiplier `.mif` file will override a system bus block RAM if conflicts occur in competing `.mif` files.*

Bit Generation Considerations

Note the following user information that may require additional steps to implement BIT options properly.

- If the device is to be configured in other than a serial (master or slave) mode, there are some limitations involving startup options and configuration pins used as outputs after configuration.
- If a parallel mode is planned, the **bitgen** program command line option **-g OutputsActive:DoneIn** (in the Project Navigator Generate Bitstream Data/PROM Data process, the I/O Tri-State Release On: **DoneIn**) property is recommended.

Running BIT GENERATION from the Command Line

Syntax

bitgen [**-b**] [**-d**] [**-f** <*command_file*>] [**-h** [<*architecture*>]] [**-a**] [**-o** <*outfile*>] [**-x**] [**-j**] [**-m** <*format*>] [**-w**] [**-g** *option:value*] [**-J r** | **w** <*infile1*[**.ncd**] {<*infile2*[**.ncd**]}] [**-r** <*infile*[**.mif**]> <*infile*[**.ncd**]> <*outfile*>] <*infile*[**.ncd**]> [<*outfile*>] [<*preffile*[.**prf**]>]

where:

-b	Create a “rawbits” (*.rbt*) file instead of a binary file. Do not use **-b** with **-m** if you want both a rawbits file and a mask file (see the **-m** command below). Instead run BITGEN twice once with the **-b** option and once with the **-m** option.
-d	Do not run DRC.

-f <command_file>	Execute command line arguments in the specified *command_file*.
-h <*architecture*> or **-help** <*architecture*>	Display available BITGEN command options for the specified *architecture*. The **bitgen -h** command with no architecture specified will display a list of valid architectures.
-a	Outputs an ASCII file.
-o <outfile>	Specifies the output file name.
-j	Do not create a bitstream file.
-m <*format*>	Create a mask file. Format value equal 0, 1, or 2. The value 0 is the generic format, 1 is design-specific, and 2 is design-specific plus user RAMs.
-w	Overwrite an existing *.bit*, *.msk*, or *.rbt* output file.
-x	Byte mirror. Swaps 0s with 7s, etc.
-g option**:**value	Set ORCA configuration options. Options and values are case-insensitive. Available options are: ADDRESS (Series 3 and 4 only) Bitstream addressing mode. Options are **Increment** (default) for general use and **Explicit** for testing and debugging. CFGMODE Options are **Enable** (default) and **Serial**. DONEACTIVE Select the event that activates the DONE signal. Settings are **C1**, **C2**, **C3**, **C4** (default) (first CCLK rising edge after the length count is met, second CCLK rising edge after the length count is met, etc.). DONEPIN Options are **Pullup** (default) and **Pullnone**. EXTERNALCLK Options are **Yes** and **No**. GSRINACTIVE Select the event that releases the internal set/reset to the latches and flip-flops. Settings are **C1**, **C2** (default), **C3**, **C4**, and **Donein**. JTAG Enables or disables JTAG (boundary scan) after configuration. Settings are **Enable** and **Disable** (default). OSCILLATOR Enables the on-chip oscillator after configuration. Options are **Enable**, **Disable** (default), and **EnableDiv8** (divide the speed by eight). OUTPUTSACTIVE Select the event that releases the I/O from 3-state condition and turns the configuration-related pins operational. Settings are **C1**, **C2**, **C3**, **C4**, and **Donein** (when the DONEIN signal goes high the default). OUTPUTSCFG If enabled, the FPGA will not drive outputs during configuration/reconfiguration. Options are **Enable** (default) and **Disable**. PARITY (2CA/2TA only) Enables or disables a parity check by the FPGA of the configuration bitstream. Settings are **Enable** (default) and **Disable**. RAMCFG Options are **Reset** (default) and **No Reset**. Reset reinitializes the device when you download bitstream. No reset retains the current configuration and allows additional bitstream configuration.

READBACK
Allows you to extract the configuration data stored in an FPGA in order to verify the configuration. Settings are **Disable** (default), **Once** (allow one readback only; after that, readback cannot be invoked again), and **Command** (multiple readbacks).

READCAPTURE
Enable or disable readback of the configuration bitstream. Settings are **Disable** (default), **Read**, **UserNet**, and **Both**.

REGISTERCFG
Reset the PFU registers before configuration. Values are **Enable** (default) and **Disable**.

STARTUPCLK
Use CCLK or a User clock during startup. Options are **Cclk** (default) or **UserClk**.

SYNCTODONE
Synchronize the startup sequence to the external DONE signal. Options are **Yes** and **No** (default).

ZEROFRAMES
(Series 3 and 4) Option to specifiy that data frames with all zeros be written into the bitstream. Options are No and **Yes** (default). In Series 4, this option is used with Increment Address mode and in Series 3 it is used in Explicit Address mode.

Series 4 Addressing Modes

When **Yes** is specified with Increment Address mode, every data frame is written out with no address frames. When **No** is specified with Increment Address mode, all sequentional non-zero data frames are written without data frames. When a (zero) data frame is skipped, the address frame forthe next (non-zero) data frame is written.

Series 3 Addressing Modes

When **Yes** is specified with the Explict Address mode, every address frame is written out followed by every data frame. This is good for partial reconfiguration, where bitstream may only include a few frames. **No** is not allowed. A warning will be generated that *Zeroframes:No* is not compatible with Explicit addressing and the option will be ignored.

SYSBUSCFG
(Series 4 only) Options are **Reset** and **NoReset**. The system bus is either reset or not reset at reconfiguration.

SYSBUSCLKCFG (Series 4 only) Options are **Reset** and **NoReset**. The system bus clock is either reset or not reset at reconfiguration.

WAITSTATETIMEOUT (Series 4 only) Controls an internal system bus timeout counter. This counter counts the number of HCLK (sysbus clk) cycles goes by while the system bus is in wait states. Possible values are **0, 1, 2...15**.

0 - Forever (never time out)
1 - 2^2 HCLK cycles
2 - 2^4 HCLK cycles
3 - 2^6 HCLK cycles
4 - 2^8 HCLK cycles
5 - 2^10 HCLK cycles
6 - 2^12 HCLK cycles
7 - 2^14 HCLK cycles
8 - 2^16 HCLK cycles
9 - 2^18 HCLK cycles
10- 2^20 HCLK cycles
11 - 2^22 HCLK cycles
12 - 2^24 HCLK cycles

	13 - 2^26 HCLK cycles 14 - 2^28 HCLK cycles 15 - 2^31 HCLK cycles GRANTTIMEOUT (Series 4 only) Controls an internal system bus grant counter which counts the number of HCLK (sysbus clk) cycles goes by before the grant signal is taken away from the master.This prevents the system bus from locking up if anything goes wrong. Possible values are **0, 1, 2...15**. See WAITSTAITTIMEOUT. LENGTHBITS (Series 4 only) Specifies the number of bits in the bitstream length field. Options are 24 (default) and 32.
-J r | **w** <*infile1*[**.ncd**]> {<*infile2*[**.ncd**]}	JTAG setup. Allows the user to set JTAG port read and write on the FPGA chip by providing them with the setup bitstreams. Users will use tools other than ORCA Foundry to download these bitstreams. Example syntax: [**-J w** <*infile1.ncd*> {<*infile2.ncd*>}] [**-J r**] [**-a**] [**-o** <*outfile*>] where: **-J r** generates a JTAG read setup bitstream.The default output is jtagread.jbt. **-J w** generates JTAG write setup bitstream. **-a** outputs an ASCII file. **-o** <outfile> specifies the output file name. The example syntax for JTAG setup will generate JTAG write setup bitstream that specifies *infile1.ncd* as the target design to be downloaded, so the correct initialization bits will be added to the bitstream. If more than one design is on the command line, the bitstreams for daisy-chained devices will be generated using JTAG port. The default output is *infile1.jbt*.
[**-r** <*infile*[**.mif**]> <*infile*[**.ncd**]> <*outfile*>]	(*for Series 4 system block RAM only*) Use this option to designate bitgen to program Series 4 system block RAM using the *.mif* file generated in SCUBA for input. For example, initialize a system bus block RAM by entering the following command in the command line: **bitgen -r** <*sysbus.mif*> <*mydesign.ncd*> <*mydesign.bit*> Using the SCUBA-generated sysbus *.mif* file, this command runs bitgen on *mydesign.ncd* and writes out *mydesign.bit* to initialize the system bus block RAMs. SCUBA will automatically generate a *.mif* file with a file name identical to your module name. Note that if block RAMs are already initialized by INITVAL properties in the *.ncd* file, addresses that are explicitly addressed in the *.mif* file will be overidden by the INITVAL properties.
<*infile*[.**ncd**]>	SCUBA will automatically generate a *.mif* file with a file name identical to your module name. Note that if block RAMs are already initialized by INITVAL properties in the *.ncd* file, addresses that are explicitly addressed in the *.mif* file will be overidden by the INITVAL properties.
<*outfile*>	The output file. If you do not specify an output file, BITGEN creates one in the input file's directory. If you specify **m** on the command line, the extension is *.msk*. If you specify **-b**, the extension is *.rbt*. Otherwise the extension is *.bit*. A report (*.bgn*) file containing all of BITGEN's output is automatically created under the same directory as the output file.

PROM Generation

Generating PROMs

After creating a bitstream file with the Bit Generation (**bitgen**) program, you can store the configuration information in a PROM file. PROM generation takes one or more bitstreams (`.bit` or `.rbt` files) and creates a PROM-formatted file in one of three widely used PROM formats: Intel's MCS-86, Motorola's EXORMACS, or Tektronix's TEKHEX.

There are three common storage methods:

- One design in one PROM.
- Individual designs stored separately in one PROM. This is useful when you want to use a single PROM to reprogram the same device with different designs.
- Several different designs concatenated in the same or multiple PROMs. This approach is likely when you want to program several devices in a daisy chain or when using a PROM that is too small to store the complete bitstream for the target FPGA.

PROM Generation Input Files

The input to the Prom Generation process is one or more bitstreams (`.bit` or `.rbt` files). These files are generated in the Generate Bitstream Data process during a design flow run in the Project Navigator or by using the PROM GENERATION (**promgen**) command from the command line.

Generating PROM Files

After creating a bitstream file with bit generation, you may want to store the configuration information in a PROM. You can use the bitstream file you have created by running the design flow as your input for the PROM Generation. When using the Project Navigator, the bitstream file will be in the project directory.

PROM options should be set before running the PROM generator. The PROM generator takes bitstreams and creates PROM image files for PROM configuration. You can set PROM options in the Generate PROM Data process properties dialog box.

To generate PROM files from the Project Navigator:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window, right-click the **Generate PROM Data** process and select **Properties** to open the Properties dialog box.
3. Set the Generate Bitstream Data properties that you want and then click **Close** to close the dialog box.
4. In the Project Navigator, double-click the **Generate PROM Data** process to generate the bitstream files.

PROM Generation Output Files

The PROM generation process creates two files:

- A PROM file containing configuration information used to program the PROM. This file will have an extension of `.mcs`, `.exo`, or `.tek`, depending upon the selected PROM format.
- A PROM image file containing a memory map of the newly created PROM file showing the starting and ending PROM address for each bitstream (`.bit` file) loaded. This file has a `.prm` extension.

Running PROM GENERATION from the Command Line

Syntax

promgen [**-b**] [**-f**] [**-h**] [**-n** <*bitfile*> {<*bitfile*>}] [**-o** <*outfile*>] [**-p** <*format*>] [**-s** <*size*>] [**-t** <*size*>] {**-u** <*hexaddr*> <*bitfile*> {<*bitfile*>}} {**-d** <*hexaddr*> <*bitfile*>{<*bitfile*>}}

where:

-b	Specifies byte-wide mode (no bit mirroring).
-f <command_file>	Execute command line arguments in the specified *command_file*.
-h or -help	Displays available PROMGEN command options.
-n <bitfile> {<bitfile>}	Load *.bit* or *.rbt* file(s) up or down from next available PROM address following the previous load. This option requires that a previous **-u** or **-d** option be given. You can use multiple **-n** options, (for example, **-n** *bitfile* [**-n** *bitfile*...]). Filenames entered after the **-n** option are not daisy chained to previous files.
-o <outfile>	The output PROM file. If you do not specify a file name, the PROM file will have the same root name as the first bit file loaded, with the extension specified with the **-p** option. If you do not use the **-p** option, the extension name for the PROM file defaults to *.mcs*. A report (*.prm*) file containing all of PROMGEN's output is automatically created under the same directory as the output file.
-p <format>	Specifies the PROM format as **mcs**, **exo**, or **tek**. If used, must precede **-u, -d,** or **n** on the command line. The default is **mcs**.
-s <*size*>	Specifies the PROM size in kilobytes. Must be a power of 2. If not specified, the smallest possible size will be selected.
-t <*size*>	Split the file into multiple prom-files with user specified size. The resulting files are named as <file>00.<ext>, <file>01.<ext>, <file>02.<ext>, ..., etc. Cannot be used with switch **-s, -n,** or **-d**.
-u <hexaddr> <bitfile> {<bitfile>}	Load bit file(s) up from the specified PROM address, that is, into successively higher addresses from the starting point. Entering multiple filenames after the **-u** option concatenates the files in a daisy chain. You may use several **-u** options to load files at various addresses, (for example, **-u** <*hexaddr*> <*bitfile*> **-u** <*hexaddr*> <*bitfile*>). You must specify a starting address.
-d <hexaddr> <bitfile> {<bitfile>}	Load bit file(s) down from the specified PROM address, that is, into successively lower addresses from the starting point. Entering multiple filenames after the **-d** option concatenates the files in a daisy chain. You may use several **-d** options to load files at various addresses, (for example, **-d** <*hexaddr*> <*bitfile*> **-d** <*hexaddr*> <*bitfile*>). You must specify a starting address.
<bitfile>	Specifies the name(s) of the *.bit* or *.rbt* file(s) to be loaded. The default extension is *.bit*.

Notes

Previously, the –t option required a file argument:

```
promgen -p exo -t 256 orca.bit
```

However, now it is a separate parameter. To do the same action, do the following:

```
promgen -p exo -t 256 -u 0 orca.bit
```

Examples

The following command loads the file *orca.bit* down from Hex address 0x0300 in a PROM file with a Motorola format:

```
promgen -p exo -d 300 orca.bit
```

The following command daisy chains files *att1.bit* and *att2.bit* up starting at the next available PROM address:

```
promgen -u 0F00 att1.bit att2.bit
```

The following command daisy chains *orca1.bit*, *orca2.bit*, and *orca3.bit* up from PROM address 0x0000 and loads *orca4.bit* (not daisy chained to the other files) at the next available address, using a Tektronix format and naming the output file *promtest*:

```
promgen -p tek -u 0 orca1.bit orca2.bit orca3.bit n  orca4.bit -o promtest
```

Download/Upload

Programming FPGA Devices

After you have created and verified your design, you can use the final output bitstream files or PROM files to download/upload bitstream to/from an FPGA device using one of the device programming tools for intended use with FPGA devices. To download/upload a bitstream into and from the target device, use the DEVPROG program or the ispVM System software. This operation allows you to transmit configuration data directly into an FPGA from a host computer's (PC or UNIX) serial communication from a programmed FPGA device for either debugging or reconfiguration of another device.

***Note**: You can daisy chain multiple bitstream files using the command line.*

Download/Upload Input Files

DOWNLOAD/UPLOAD accepts the following files as input for the download cable:

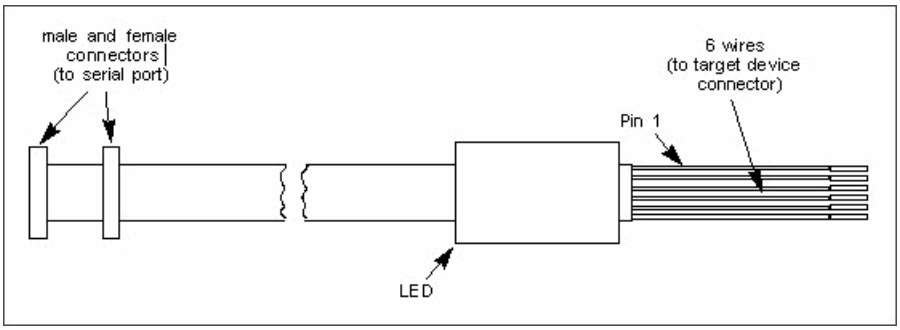
- Bit File (`.bit` file)
- Raw Bit File (`.rbt` file)
- Intel MCS-86 (`.mcs` PROM file)
- ExorMax (`.exo` PROM file)
- TekHex (`.tek` PROM file)

Connecting the Download Cable

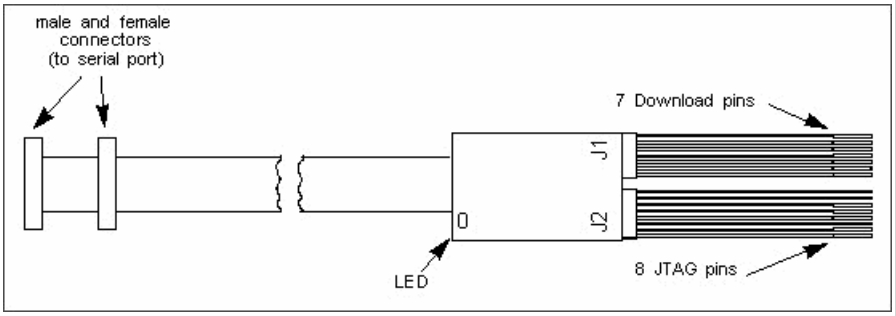
This procedure is intended for use with the **devprog** program. See the ispVM System Help for any special instructions for download cable connections.

1. Attach either the female or male D-type connector on the download/upload cable to the host computer serial port.

   Lucent Cable



   Serial Cable



2. Attach the other end of the download cable to the target device connector according to the tables below for newer JTAG download/upload cables and the older download/upload cables. It is recommended that all the pins be connected, as this is the only way to ensure full functionality of download/ upload. The LED will turn on when there is power to the cable.

   For newer JTAG download/upload cables, fly lead J1 mates to a 7-pin connector AMP P/N-103906-6 using the following configuration:

Pin...	Connects To...
1	GND
2	NC
3	PRGMN
4	DONE
5	DO
6	CCLK
7	VDD

For newer JTAG download/upload cables, fly lead J2 mates to an 8-pin connector AMP P/N-103906-7 using the following configuration:

Pin...	Connects To...
1	GND
2	PINITN
3	RD_CFGN
4	TDO
5	TCK
6	TMS
7	TDI
8	VDD

For older download cables, AMP 103635-6 is the recommended connector using the following configuration:

Pin	Connects To	Color
1	Vcc	Red
2	CCLK	Green
3	Din	Yellow
4	Done	Blue
5	PRGM	Orange or Purple
6	GND	Black
7	GND	Black

Notes on Older Cable Configuration

- Ensure that the device is in slave serial mode before downloading (true for all cable types). This is controlled via the MODE pins on the FPGA.
- Supply Vcc to the cable from your board (true for all cable types).
- The cable does not control the PRGM pin. To initiate the download, you must manually pulse the Program pin LOW on the FPGA.
- The INIT pin on the FPGA is only necessary for power-up and is controlled by the FPGA. Typically it is connected to a 10K pull-up resistor. This will allow the pin to be pulled high after initialization. Please refer to the ORCA FPGA data sheet or contact technical support for more information.

Running DEVPROG from the Command Line

You can run DOWNLOAD/UPLOAD from the command line using the **devprog** program. Entering **devprog –h** at a DOS or UNIX prompt will display the command line usage for devprog.

Programming Devices using ispVM System

Lattice supports device programming for all programmable logic devices with the following GUI tools, which are briefly described below and covered in detail in their respective Help. They are listed in alphabetical order.

ispVM System

The ispVM™ System software (ispVM) supports both serial and concurrent (turbo) programming of all Lattice devices in a PC environment. The ispVM System software is built around a graphical user interface. Device chains can be scanned automatically. Any required JEDEC ISC or Bitstream data files are selected by browsing with a built-in file manager. Non-Lattice devices that are compliant with IEEE 1149.1 can be bypassed once their instruction register length is defined in the chain description. Programmable devices from other vendors can be programmed through the vendor-supplied SVF file. See the ispVM System Help for more information about this tool.

Model 300 Programmer

The ISP Engineering Kit Model 300 programmer is an engineering device programmer that supports prototype development by allowing single-device programming directly from a PC. The Model 300 programmer supports all JTAG devices produced by Lattice, with device Vcc of 1.8, 2.5, 3.3, and 5.0V. See the Model 300 Programmer Help for more information about this tool.

SVF Debugger

The SVF Debugger can be used with ispVM System software to help you debug a Serial Vector Format (SVF) file. The SVF Debugger software allows you to program a device, and then edit, check syntax, debug and trace the process of an SVF file. See the SVF Debugger Help for more information about this tool.

Universal File Writer

The Universal File Writer (UFW) is a separate application that generates bitstream files or an SVF data file for a single device. Using JEDEC or ISC files, the software generates bitstream PCM, Intel Hex and Motorola Hex data files concurrently. It can also generate an SVF file using the parameters you select. You can run the Universal File Writer from the ispVM System toolbar or separately. See the Universal File Writer Help for more information about this tool.

Running FPGA Tools from the Command Line

Running FPGA Design Tools from the Command Line

General Comments

You can run the ORCA FPGA design tools from the command line. The following are general guidelines that apply.

- Files are position-dependent. Generally, they follow the convention `[options] <infile> <outfile> <preference_file>` (although order of `<outfile>` and `<infile>` are sometimes reversed). Use the **-h** command line option to check the exact syntax; for example, **par** -h.
- Command line options are entered on the command line in any order, preceded by a hyphen (-), and separated by spaces.
- Most command line options are case-sensitive and must be entered in lowercase letters. When an option requires an additional parameter, the option and the parameter must be separated by spaces or tabs (for example, **-l 5** is correct, **-l5** is not).
- Options that don't require an additional parameter may be grouped together preceded by a single hyphen (for example, **-arw** is the same as **-a -r -**w).
- Options can appear anywhere on the command line. Arguments that are bound to a particular option must appear after the option. For example, **-f** <command_file> is legal; <command_file> **-f** is not.
- For options that may be specified multiple times, in most cases the option letter must precede each parameter. For example, **-b romeo juliet** is not acceptable, while **-b romeo -b juliet** is.
- If you enter the ORCA Foundry application name on the command line with no arguments and the application requires one or more arguments (**par,** for example), you get a brief usage message consisting of the command line format string.

The following conventions are used when commands are described:

Convention	Meaning
( )	Encloses a logical grouping for a choice between sub-formats.
[ ]	Encloses items that are optional. (Do not type the brackets.) Note that `<infile[.ncd]>` indicates that the `.ncd` extension is optional but that the extension must be ncd.
{ }	Encloses items that may be repeated zero or more times.
|	Logical OR function. You must choose one or a number of options. For example, if the command syntax says **pan up | down | right | left** you enter **pan up** or **pan down** or **pan right** or **pan left**.
< >	Encloses a variable name or number for which you must substitute information.
, (comma)	Indicates a range for an integer variable.
- (dash)	Indicates the start of an option name.
:	The bind operator. Binds a variable name to a range.
bold text	Indicates text to be taken literally. You type this text exactly as shown (for example, “Type **autoroute -all -i 5** in the command area.”) Bold text is also used to indicate the name of an EPIC command, a UNIX command, or a DOS command (for example, “The **playback** command is used to execute the macro you created.”).

Convention	Meaning
Italic text or text enclosed in angle brackets <>	Indicates text that is not to be taken literally. You must substitute information for the enclosed text. Italic text is also used for the names of documents, files, and directories (for example, “Edit the `autoexec.bat` file” or “The file is in the */data* directory”).
`Monospace`	Indicates text that appears on the screen (for example, "`File already exists.`"). Monospace text is also used to show text from UNIX or DOS text files.

To get a brief usage message plus a verbose message that explains each of the options and arguments, enter the ORCA Foundry application name on the command line followed by **-help** or **-h**. For example, enter **edif2ngd -h** for option descriptions for the **edif2ngd** program.

To redirect this message to a file (to read later or to print out), enter this command:

```
command_name -help | -h > filename
```

The usage message is redirected to the filename that you specify.

For those ORCA Foundry applications that have architecture-specific command lines (e.g., or2c00a), you must enter the application name plus **-help** (or **-h**) plus the architecture to get the verbose usage message specific to that architecture. If you fail to specify the architecture, you get a message similar to the following:

```
Use '<appname> -help <architecture>' to get detailed usage for a particular
architecture.
```

The –f Option

For any ORCA Foundry executable, you may store arguments (that is, filenames and command options) in a file and then execute them at any time by entering the UNIX or DOS command line followed by the name of the file containing the arguments. This can be useful if you frequently execute the same arguments each time you perform the command, or if the command line becomes too long. This is the recommended way to get around the DOS command line length limitation of 127 characters. (Equivalent to specifying a shell Options file.)

The command file is an ASCII file containing the command arguments. Arguments are separated by space and can be spread across one or more lines within the file. You can put new lines or tabs anywhere white space would otherwise be allowed on the UNIX or DOS command line. You can put all arguments on the same line, or one argument per line, or anything in between. There is no line length limitation within the file. All carriage returns and other non-printable characters are treated as space and ignored. Comments are designated by a # (pound sign) and go to the end of the line.

Command File Example

This is an example of a command file:

```
#command line options for par for design mine.ncd
-a -n 10
-w
-l 5
-s 2 #will save the two best results
/home/users/jimbob/designs/mine.ncd
#output design name
/home/users/jimbob/designs/output.dir
#use timing preference file
/home/users/jimbob/designs/mine.prf
```

Using the Command

You can use the command file in two ways:

- To supply all of the command arguments as in this example:

  ```
  par -f <command_file>
  ```

  where **<command_file>** is the name of the file containing the command-line arguments.

- To insert certain command line arguments within the command line as in the following example:

  ```
  par -i 33 -f placeoptions -s 4 -f routeoptions design_i.ncd design_o.ncd
  ```

  where

  placeoptions is the name of a file containing placement command arguments.

  routeoptions is the name of a file containing routing command arguments.

Running SCUBA from the Command Line

Generally, you should generate ORCA FPGA SCUBA-based modules using the Module/IP Manager in ispLEVER because of the complexity of command line options for Series 4 and later architectures. However, you may wish to quickly generate SCUBA modules for older architectures (i.e., Series 2 and Series 3) directly from the command line using command line options. Use this topic as a reference for SCUBA command line syntax.

***Note**: In addition, we also do not recommend using the command line for newer architectures because error checking on command line options does not exist which may cause **scuba** to quit unexpectedly if incorrect options are specified.*

Syntax

scuba [*options*], where *options* are as follows:

Command	Purpose	Module
-arch (orca2a | orca3)	Selects ORCA 2CA/2TA or Series 3 module generators. (Default is orca2a.)	all
-type <*type*>	2CA/2TA: (aspram | sspram | sdpram | rom | mult | addsub | add | sub | comp | lfsr | fifo) Series 3: (sdpram | sspram | rom | fifo | mpippc | mpi960)	each
-a_flow (north | east | south | west)	SCUBA provides positional attributes for improving multiplier performance. SCUBA creates a rectangular block of a multipler by attaching a locational attribute to each component. A_flow specifies the direction of flow of the A input (default is south). Used with **-b_flow** and **-origin**. A_flow and b_flow must be perpendicular to each other. For example, **-a_flow north -b_flow east** is allowed. However, **-a north -b_flow north** or **-a_flow north -b_flow south** are not allowed because both flows are either in the same direction or in opposite directions.	mult

Command	Purpose	Module
-addr_flow (north | east | south | west)	SCUBA creates a rectangular block of a memory by attaching a locational attribute to each component. Addr_flow specifies the direction of flow of the address input (default is east). Used with **-data_flow** and **-origin**. Data flow and address flow must be perpendicular to each other. For example, **-data_flow north -addr_flow east** is allowed. However, **-data_flow north -addr_flow north** or **-data_flow north -addr_flow south** are not allowed because data flow and address flow are either in the same direction or in opposite directions.	sdpram sspram aspram rom rommult 3C sdpram 3C sspram fifo
-addr_width <*addr_width*>	The address bus width of the module.	sdpram sspram aspram rom 3C sdpram 3C sspram
-b_flow (north | east | south | west)	SCUBA provides positional attributes for improving multiplier performance. SCUBA creates a rectangular block of a multipler by attaching a locational attribute to each component. B_flow specifies the direction of flow of the B input (default is east). Used with **-a_flow** and **-origin**. A_flow and b_flow must be perpendicular to each other. For example, **-a_flow north -b_flow east** is allowed. However, **-a north -b_flow north** or **-a_flow north -b_flow south** are not allowed because both flows are either in the same direction or in opposite directions.	mult
-bb	Uses (“big endian” (BusA[7 to 0]) bus naming convention for VHDL/Verilog output.	all
-bl	Uses “little endian” (BusA[0 to 7]) bus naming convention for VHDL/Verilog output.	all
-buffer	Inserts a buffer for each I/O signal.	all
-bus_exp (1=BusA(0) | 2=BusA_0 | 3=BusA0 | 4=BusA<0> | 5=BusA[0] | 6 = BusA[0 to 7] | 7=BusA(0 to 7))	(For EDIF only) Selects among the 7 different bus expressions styles in EDIF output. Must be used with **-bb**/**-bl** option.	all
-clk	Use QDO outputs of library elements.	rommult 3C sdpram 3C sspram fifo
-depth	The maximum number of words for the FIFO.	fifo
-data_flow (north | east | south | west)	SCUBA creates a rectangular block of a memory by attaching a locational attribute to each component. Data_flow specifies the direction of flow of the address input (default is south). Used with **-addr_flow** and **-origin**. Data flow and address flow must be perpendicular to each other. For example, **-data_flow north -addr_flow east** is allowed. However, **-data_flow north -addr_flow north** or **-data_flow north -addr_flow south** are not allowed because data flow and address flow are either in the same direction or in opposite directions.	sdpram sspram aspram rom rommult 3C sdpram 3C sspram fifo

Command	Purpose	Module
-data_width <*data_width*>	The data width of the module.	sdpram sspram aspram rom 3C sdpram 3C sspram
-e	Suppresses EDIF output. SCUBA always creates an EDIF file unless this option is used. To create a VHDL or Verilog file, use the **-lang** option.	all
-fastmode	Uses fast mode memory elements.	sdpram sspram
-h	Prints help and exits.	all
-h <*module type*>	Prints help for the specified *module type* and exits.	all
-inv_reset	Makes the reset port of a pipeline multiplier register active LOW. (Default is active HIGH.)	mult rommult
-memfile <*memfile*>	<*memfile*> is a file containing the contents for the module. If a *memfile* is not used, all the RAM contents are initialized to 0. *NOTE*: All ORCA memories including ORCA5-EM distributed memory are initialized by use of a memfile with the following syntax: **{hex_address} : {hex_data} [{hex_data}...]** However in ORCA5-EM block memory, the syntax is **{data}** or **{hex_data}** depending on the scuba calling line option. The first line of data will go to address 0 of the memory, and following data must be in order. We strongly recommend generating your ORCA5-EM modules from Module/IP Manager.	sdpram sspram aspram rom rommult 3C sdpram 3C sspram
-ne	Use the inverted clock polarity. The top level clock input is inverted and the inverted clock is used to drive the sequential cells in the module.	sdpram sspram mult rommult
-num_rows <*number of rows*>	Specifies the number of rows of memory needed for the module. For example, if you needed 135 rows of memory, you could use the command **-addr_width** 8 (2^7<135<2^8). However, this would generate unnecessary memory rows. Instead, you can use this option to specify the exact number of rows needed.	sdpram sspram aspram rom 3C sdpram 3C sspram
-mem_mux	Use LUT-based mux decoding.	sspram aspram sdpram rom fifo

Command	Purpose	Module
-origin <*row*> <*column*>	SCUBA creates a rectangular block of a memory or a multiplier by attaching a locational attribute to each component. Origin specifies the position of top left corner of a memory rectangle or a multiplier rectangle. This option is mandatory if LOC attributes need to be added. Used with **-addr_flow** and **-data_flow** (**-a_flow** and **-b_flow** for multipliers).	sdpram sspram aspram rom rommult 3C sdpram 3C sspram fifo
-pe	Threshold for empty flag.	fifo
-pf	Threshold for full flag.	fifo
-pgroup	Write PGROUP properties used with **-addr_flow** and **-data_flow** (**-a_flow** and **-b_flow** origin options.	fifo
-pipeline <*depth*>	Produces a pipeline multiplier (default is a non-pipeline multiplier). Registers are inserted for every <*depth*> multiplication stage, which is represented by the value <*widtha*>. For example, in an 8x8 register: **-pipeline 1** inserts register in every (all 8) multiplication stage. **-pipeline 2** inserts registers in every other multiplication stage. **-pipeline 4** inserts registers in every fourth stage (stage 4 and stage 8). **-pipeline 8** inserts registers in only the eighth stage. <*depth*> must be greater than 0 and less than (<*widtha*> +1) / 2. The registers have a clock port and a reset port.	mult
-pipe_write_path	Pipelines the write path. Adds pipeline registers and FFs to the data in, write address, write enable, and the optional write port enable.	sspram sdpram rom fifo
-pipe_write_path_rep	In addition to what Pipeline Write Path option does, this option individually pipelines the bottom five (or 4 for 2CA/ 2TA) address lines for each row.This reduces the load on these write address lines. Address is referred to as “Pointer” for FIFOs.	sspram sdpram rom fifo
-pipe_mux_levels	This option is available when the -mem_mux option is used. For 2CA/ 2TA, this options does not depend on the address size. When the address size is less than 7, there is only one level of mux. For such cases, the pipeline is between the DEC16X2 output and the mux input. The address lines used for mux select lines are also pipelined. For Series 3, this option can be used only when the address width is greater than 7. In such cases, there are multiple mux levels, and nets between mux stages are pipelined.	sspram sdpram rom fifo
-pipe_read_addr	Pipelines the read address.	sspram sdpram rom fifo
-pipe_read_addr_rep	The pipeline registers for the lower read address bits are replicated for each row of the memory. Address is referred to as “Pointer” for FIFOs.	sspram sdpram rom fifo

Command	Purpose	Module
-pipe_final_output	Pipelines the final output. There are only one set of registers. When the placement option is used, the data out registers are placed in the bottom row.	sspram sdpram rom fifo
-port (ci | co | overflow | altb | aeqb | agtb | aleb | aneb | ageb | wpe | rden)	Specialized port for the module.	sdpram sspram aspram addsub add sub comp 3C sdpram 3C sspram
-read_clock	This option is used to generate a separate read clock for dual port memory cells. This option can be used only with the **-pipe_final_output** or a valid **-pipe_mux_levels**.	sdpram
-rom	Specifies a ROM-based multiplier. Use with -type mult.	rommult
-signed	Signed number representation. The default is unsigned.	addsub add sub comp
-stage <*stage_width*>	The stage width of a module.	lfsr 3C lfsr
-synth <*vendor*>	Generates HDL output for *synthesis*. Without this option, the HDL output is generated for *simulation*. For synthesis, attributes are embedded in the HDL design for writing out ORCA Foundry attributes in EDIF. The attributes are supported for Synopsys FPGA/Design Compiler, Mentor Graphics LeonardoSpectrum, and Synplicity Synplify.	all
-unsigned	Unsigned number representation.	addsub add sub comp
-v	Prints version number and exits.	all
-value <*constant factor*>	The constant value of the multiplicand.	mult rommult
-w	Overwrites existing files.	all
-width <*width*>	The width of a module.	addsub add sub comp fifo
-widtha <*widtha*>	The bit width of port ‘a’ (multiplier).	mult rommult
-widthb <*widthb*>	The bit width of port ‘b’ (multiplicand).	mult rommult

Command	Purpose	Module
-n <*circuitname*>	<*circuitname*> is a module name for Verilog or an entity name for VHDL. <*circuitname*> is also an output file name root as follows: • EDIF <*circuitname*>.edn • VHDL <*circuitname*>.vhd • VHDL instance template <*circuitname*>_tmpl.vhd • Verilog <*circuitname*>.v • Verilog instance template <*circuitname*>_tmpl.v • Report file <*circuitname*>.srp	all
-lang <vhdl | verilog>	Creates a VHDL or Verilog file in addition to an EDIF file.	all

Running EDIF2NGD from the Command Line

Syntax

edif2ngd [**-l** <*libname*> **-r -s** <*file*[**.dc**]> **-f** <*command_file*>] <*edif_file*[.**edn**]> [<*outfile*[.**ngo**]>]

where:

-l <*libname*>	Specifies libraries to search when determining what library components were used to build the design. NGDBUILD uses this information to resolve the components to ORCA primitives. The allowable entries for *libname* are: • **orca** (required if your design contains components from the ORCA Macro Library targeted to 2TA/2TB architecture) • **or3c00** (required if your design contains components from the ORCA Macro Library targeted to 3C/3T/3L architecture) • **or4e00** (required if your design contains components from the ORCA Macro Library targeted to 4E architecture)
-r	Filters out all LOCATION properties (LOC=) from the design. This can be used when you are migrating to a different device or architecture.
-s *synfile*[**.dc**]	Reads the constraints in a Synopsys setup (*.dc*) file.
-f <*command_file*>	Executes the command line arguments in the specified *command_file*.
<*edif_file*[**.edn**] | [**.edf**]>	Name of the input netlist file in EDIF 2 0 0 format. If the file has an extension, you must enter the extension as part of *edif_file*. If you enter a filename without an extension, EDIF2NGD looks for an *.edn* file with the name you specified.
<*outfile*[**.ngo**]>	Output design file in *.ngo* format. The output filename, its extension, and its location are determined in this way: • No output filename specified: The output file has the same name as the input file, with an *.ngo* extension. • Output filename specified but with no extension: EDIF2NGD appends the *.ngo* extension to the filename. • Output filename specified with an extension other than *.ngo*: Produces an error message and EDIF2NGD will not run. • No full pathname: The output file is placed in the directory from which you ran EDIF2NGD. If the output file already exists, it will be overwritten with the new file.

Running NGDBUILD from the Command Line

Before you run NGDBUILD, you must convert the top-level design file, and all files referenced in the top-level file, to *.ngo* format.

Syntax

ngdbuild -a <*architecture*> {**-p** *search_path*} [**-f** <*command_file*>] [<*ngo_file.*[**ngo**]>] [<*outfile.*[**ngd**]>]

where:

-a <*architecture*>	Specifies the architecture of the device to which your design will be mapped (and into which the design will eventually be programmed). When you specify the *architecture*, the output *.ngd* file is optimized for mapping into that architecture. Valid *architectures* are **or4e00**, **or3t00**, **or3c00**, **or2t00a** or **or2c00a**.
-p <*search_path*>	Adds the specified *search_path* to the list of directories to search when resolving file references (i.e., files specified in the design with a FILE= *filename* attribute). You need not specify a search path if the necessary *.ngo* or *.nmc* file is in the directory containing the top-level *.ngo* file or if the FILE attribute in the design gives a complete path name for the file (instead of a relative path name). To specify multiple **-p** options, precede each with **-p**; you cannot combine multiple *search_path* specifiers after one **p**.
-f <*command_file*>	Executes the command line arguments in the specified *command_file*.
<*ngo_file*[.**ngo**] >	Name of the input top-level design file in *.ngo* format. If you enter a filename with no extension, NGDBUILD looks for an *.ngo* file with the name you specify. If you specify a filename with an extension other than *.ngo*, you get an error message and NGDBUILD will not run.
<*outfile*[.**ngd**]>	Output design file in *.ngd* format. The output filename, its extension, and its location are determined in this way: • No output filename specified: The output file has the same name as the input file, with an *.ngd* extension. • Output filename specified but with no extension: EDIF2NGD appends the *.ngd* extension to the filename. • Output filename specified with an extension other than *.ngd*: Produces an error message and NGDBUILD will not run. If the output file already exists, it will be overwritten with the new file.

Running MAP from the Command Line

Syntax

map [**-a** <*architecture*> **-p** <*device*> **-t** <*pkgname*> **-s** <*speed*> **-c** <*packfactor*>
-k -l -u -m -r -hier -swl [<*swlpattern*>:*1-10*>] **-g** <*guide_file*[**.ncd**]> **-o** <*outfile*[**.ncd**]> **-f** <*command_file*>
-h <*architecture*> **-pr** <*oprffile*[**.prf**]>] <*infile*[**.ngd**]> [<*pref_file*[**.prf**]>] [**-td** <*routed .ncd*>]

where:

-a <*architecture*>	Specifies the architecture of the device to which you will map the design. Not needed if you are mapping to the architecture specified in the input *.ngd* file. The **-a** overrides the architecture specified in the input file. Valid *architectures* are **or4e00**, **or3t00**, **or3c00**, **or2t00a**, or **or2c00a**. Note that this is unnecessary to specify if you use the **-p** option.
-p <*device*>	Specifies the part number for the device. May be used in either of two ways: • Device name only (for example, **or3c55**). • Part number (for example, **or3c55s208**) Since the part number specifies both a device and a package, you need not enter a **t** option (specifying a package). If you do not specify a part number using the **-p** option, MAP selects a part in this way: • If the design's netlist contains properties that define a device, MAP selects the part specified by the properties. • If no device properties were specified, MAP uses the default part number, for example: **or3t80s208or3c55s208** or **2t04as208or2co4as208** Device names and part numbers are given in the *Devices* appendix. **Note**: You can only enter a part number or device name from a device library you have installed on your system. For example, if you have not installed the ORCA3C device library, you cannot create a design using the **or3c55s208** part.
-t <*pkgname*>	Specifies the package (e.g., **s208, fs256,** or **bm680**) for the device to which you will map. Package names are given for devices in the *Devices* appendix.
-s <*speed*>	Specifies the *speed* (toggle rate) of the part. Applicable speeds are given in the *Devices* appendix. If not specified, MAP selects the speed as follows: • If the design's netlist file contains a attribute that defines the device speed, MAP selects the speed specified by the attribute. • If no device speed attribute is specified, MAP uses the default speed.
-k	(2CA/2TA only) Map to 6-input functions.
-l	(2CA/2TA only) Do *not* perform logic replication.
-u	(2CA/2TA only) Do not remove unused logic.
-m	Allow overmapped NCDs.
-r	(2CA/2TA only) No register ordering. (**Note**: for Series 3, register ordering is always used.) See register ordering namine conventions in 2CA/2TA section.

-hier	Use the **-hier** option to designate hierarchical mapping instead of flat mapping. With the option designated, MAP will not pack components from different top level modules into the same PFU. This affords PAR better placement choices through improved logic grouping. Keep the following in mind when using the -hier option: • The hierarchical mapping option will override signal sharing packing of inputs or outputs. • SWLs or direct connections have higher priority than **-hier** option. The default is flat mapping, same as in previous releases. In addition, the **COMP=** properties will now have hierarchical paths appended to their data strings. This allows the multiple instantiation of blocks with **COMP=** properties to be unique and separate.
-pr <*oprffile*[**.prf**]>	Optional preference file output. Prevents changing the input preference file.
-swl [<*swlpattern*>: *1-10*>]	(Series 3 only) By default, MAP tries to implement drive-load LUTs using one of the soft-wired lookup table (SWL) connections in the PFU.This significantly reduces routing delays, since the PFU SWL has nearly 0 delay. Otherwise, the LUTs must use outside routing resources which have larger delays. The **-swl** option turns off this soft-wired LUT feature and also specifies degree of Softwire LUT (SWL) usage in the design with a predefined subset of SWL patterns ranging from empty set to full set. Currently, there are six different SWL pattern sets that are executed: • **-swl 0** - No SWL; most area efficient • **-swl 1** - Full PFU three-level SWLs, Half PFU two-level SWLs; local area efficient • **-swl 2** - All above, plus all two-level SWLs; less area efficient • **-swl (3-5)** - All above, plus three-level SWLs with 6 or 7 LUT4s; less area efficient • **-swl (6-8)** - All above, plus three-level SWLs with 5 or less LUT4s; less area efficient • **-swl (9-10)** - All above, plus extended SWLs with some LUT4s upgraded to LUT5s; least area efficient Using smaller SWL pattern sets will typically result in denser packing. This is especially true when combined with -c option. Due to fragmentation, this trade-off is non-linear. Using larger SWL pattern sets results in better performance reflected in PAR run times and timing. The default SWL option, **-swl,** is the same as using the **-swl 8** option.
-g <*guide_file*[**.ncd**]>	To decrease PAR runtimes after minor changes to logical design, guided mapping uses a previously generated *.ncd* file to “guide” the mapping of the new logical design. This is specified using the **-g** option with the file name of guide file. This option is only available for Series 3C/L/T designs. All guidance criteria is based on signal name matching. Topology of combinatorial logic is considered when Softwire LUTs (SWLs) exist in the guided file. Register elements are mapped in two passes. In the first pass, register control signals are matched by name exactly. In the second pass, the control signals names are not matched. This methodology provides a greater chance of matching for registers since control signal names have a tendency to change from successive synthesis runs. Other matching considerations are as follows: • For combinatorial logic, new SWLs are matched from SWLs extracted from the guide design. • All unmatched logic are mapped through the regular mapping process. • The performance of the guided mapped design can be no better than the original. • A guide report, <design_name>.gpr, gives details of the success guided map had in matching with the guide file.

-c [<*packfactor*>: *1-100*>]	(2CA/2TA) Pack logic blocks (*unrelated* PFUs). Used as a compression switch. MAP will try to compress the density down to the specified <*packfactor*>, the relative density (of available PFUs) at which the PFUs within a device are to be packed. If **-c** is not used, MAP will pack the design to the densest possible. For example, suppose a design, without using **-c**, packs to 53 PFUs out of 100; **-c 70** will still pack it to 53 PFUs. If another design (without using **-c**) packs to 98 PFUs out of 100, then **-c 60** will try (hard) to pack it down to 60 PFUs, probably unsuccessfully. If **-c** is used with no <*packfactor*>, the default is 97% of total PFUs. (Series 3) For Series 3, the **-c** switch has greater control of the packing density. If no **-c** is used, the result will be close to the densest packing. If **-c** is used without a <*packfactor*>, the densest mapping is performed (MAP better than without **-c**). If a <*packfactor*> is supplied, MAP will map to that density. In fact, it is possible in Series 3 to "loosen" the packing density. For example, if a design could map to 120 PFUs out of 324 (in a 3C55), **-c 70** will cause MAP to "expand" it to 227 PFUs (0.7 x 324 = 227 PFUs). Note that if the <*packfactor*> is set too low, MAP will map to its densest but may not achieve the factor requested. On the other hand, if the <*packfactor*> is set too high, it may not meet that request either. For example, a small circuit cannot be mapped into 300 PFUs.
-o <*outfile*[**.ncd**]>	Specifies the output design file in *.ncd* format. The *.ncd* extension is optional. If the output file already exists, it is overwritten with the new *.ncd* file. The output filename and its location are determined as follows: • Not specified the output file has the same name as the input file, with an *.ncd* extension. The file is placed in the input file's directory. • Output filename specified with no path (for example, *cpu_dec.ncd* instead of */home/designs/cpu_dec.ncd*) the *.ncd* file is placed in the current working directory. • Output filename specified with a full path (for example, */home/designs/cpu_dec.ncd*) the output file is placed in the specified directory.
-f <*command_file*>	Execute command line arguments in the specified *command_file*.
-h <*architecture*> or **-help** <*architecture*>	Displays all of the available MAP command options for mapping to the specified *architecture*. Values for *architecture* are the same as for the **-a** option.
<*infile*[.**ngd**]>	The design file (in *.ngd* format) to be mapped. You must enter an input design, but the *.ngd* extension is optional.
<*preffile*[.**prf**]>	The preference file in *.prf* format. A preference filename is optional on the command line, but if one is entered it must be entered after the input filename. You need not enter the *.prf* extension.

The *.mrp* and *.ngm* files produced by a MAP run both have the same name as the output file, with the proper extension. If the *.mrp* or *.ngm* (*.ldb*) files already exist, they are overwritten by the new files.

-td routed .ncd 4E only. Timing driven remap based on error paths in the previous routed *.ncd*.

Running PAR from the Command Line

Syntax

All filenames without special switches must be in the order <*infile*> < *outfile*> <*preffile*>. Options may exist in any order.

par [general_options] [placement_options] [routing_options] <infile[.**ncd**]> <outfile> [<preffile[.**prf**]>]

General Options

-f <*command_file*>	Execute command line arguments in the specified *command_file*.
-k	Skip constructive placement. Run only optimizing placement, then enter the router. Existing routes are ripped up before routing begins. Use only on a design that has already been placed. To run re-entrant (also called incremental) routing, use the options **-kp** to skip placement (constructive and optimizing) and begin routing, leaving the existing routing in place.
-p	Do not run placement (constructive or optimizing) route only. Existing routes are ripped up before routing begins. To run re-entrant (also called incremental) routing, use the options **-kp** to skip placement and begin routing, leaving the existing routing in place.
-r	Do not route the design.
-sp <*speed*>	Overrides the default speed grade.
-w	If the specified output file already exists, over-write the existing file (including the input file). If the specified output is a directory, overwrite files in the directory. With this option, any *.par* , *.pad* , and *.dly* files generated will overwrite existing *.par* , *.pad* , and *.dly* files.
-x	Do not use timing-driven option for this PAR run.
-g <*guide_file*[**.ncd**]>	To specify a guide file for guided PAR use the **-g** option with the file name of the guide file. Guided PAR decreases PAR runtime after minor changes to the physical design file (*.ncd*) by using a previously placed and/or routed *.ncd* file to "guide" the placement and routing of the new *.ncd* file. For PAR to use a guide_file for design, PAR first tries to find a guiding object (i.e., nets, components, and/ or macros) in the guide_file that corresponds to an object in the new *.ncd* file. A *guiding object* is an object in the guide_file of the same name, type, and connectivity as an object in the new *.ncd* file. A *guided object* is an object in the new *.ncd* file that has a corresponding guiding object in the guide_file. Syntax: **-g** <*guide_file*[**.ncd**]> **-mf** <*matching_factor*> **-gv** where: **-mf** specifies the matching factor or percentage of the same connectivity that guiding and guided objects must have. <*matching_factor*> is a number ranging from 1 to 100 and is defaulted to 100. **-gv** specifies that the names of all guided objects are also listed in the *.gpr* file. After PAR compares the objects in each file, it places and routes each object of the new *.ncd* file based on the placement/routing of its guiding object. If PAR fails to find a guiding object for a component, for example, PAR will try to find one based on the

	connectivity. PAR appends the names of all objects which do not have a guiding object in the guide_file to the Guided PAR Report (*.gpr*) file. After all guided objects are placed and routed, PAR locks down the locations of all guided components and macros and then proceeds with its normal operation.
-mf <*matching_factor*>	Specifies the matching factor or percentage of the same connectivity that guiding and guided objects must have during Guided PAR. *See the -g option in PAR.* <*matching_factor*> is a number ranging from 1 to 100 and is defaulted to 100. The matching factor option applies to nets and components only. When matching factor is 100 (the default), a guiding object must have exactly the same connectivity as the object it is guiding. When **-mf** is provided, the value specified is taken as the minimum percentage of the same connectivity that a *guided object* and its *guiding object* have.
-gv	In Guided PAR, the **-gv** option specifies that the names of all guided objects are also listed in the Guided PAR Report (*.gpr*) file. *See the -g option in PAR.*
-y	Adds the Delay Summary Report in the *.par* file and creates the delay file (in *.dly* format) at the end of the par run.

Placement Options

-a	Automatic level skipping for constructive placement. When you use **-a**, the **-n** option controls the maximum number of iterations per level. You cannot use **-a** with **-m** or **-r** options.
-l <*1-5*>	Specifies the effort level of the design from 1 (simplest designs) to 5 (most complex designs). Default is level 5.
-m <*nodefile_name*>	Multi-tasking mode. Cannot use **-m** with **-a**.
-n <*0-100*>	(Cost tables). Number of iterations (place and route passes) performed at the effort level specified by the **-l** option. Each iteration uses a different cost table during placement and will produce a different *.ncd* file. If you enter **n 0**, PAR continues to run until solved, stopping either after the design is fully routed and all constraints are met or after completing the iteration at cost table 100. If you do not specify the **-n** option, one place and route iteration runs. If you specify a **-t** option, the iterations begin at the cost table specified by **-t**.
-q	Minimize the clock skew on all clock nets.
-s <*1-100*>	Save the *number* of best results for this run. If not used, all results are saved.
-t <*1-100*>	Start placement at the specified cost table. Default is 1.

Routing Options

-c <*number*>	Run *number* of cost-based cleanup passes of the router.
-d <*number*>	Run *number* of delay-based cleanup passes of the router. To run delay-based cleanup passes only on designs that have been routed to completion (100% routed), use the **-e** option instead of **-d**.
-e <*number*>	Run *number* of delay-based cleanup passes of the router on *completely-routed designs* only.
-i <*1-1000*>	Run a maximum *number* of passes, stopping earlier only if the routing goes to 100% completion and all constraints are met. If not used, runs to completion or until router decides it cannot complete.
<*infile*[.**ncd**]>	The design file (in *.ncd* format) to be placed and routed. You must enter an input design, but the *.ncd* extension is optional.

<*outfile*>	The target design file which is written after PAR is finished. If options you specify yield a single output design file, *outfile* may have an extension of *.ncd* or *.dir*. An *.ncd* extension generates an output file in *.ncd* format; the *.dir* extension directs PAR to create a directory in which to place the output file (in *.ncd* format). If options you specify generate more than one place and route iteration (that is, the **-n** option described above), you must specify a *.dir* extension*;* the results of each iteration are written into a separate file in the directory. These multiple files are in *.ncd* format. If the file or directory you specify already exists, you get an error message and the operation does not run. You can override this protection and automatically overwrite existing files by using the **-w** option.
<*preffile*[.**prf**]>	The preference file in *.prf* format containing design constraints such as timing, path specification, locking, etc. If you do not enter the name of a preference file and the current directory contains an existing preference file with the *infile* name and a *.prf* extension, PAR will use it. A preference filename is optional on the command line, but if one is entered it must be entered after the input filename. You need not enter the *.prf* extension.

Examples

Following are a few examples of PAR command lines and a description of what each does.

Example 1

The following command places and routes the design in the file *input.ncd* and writes the placed and routed design to *output.ncd*.

```
par input.ncd output.ncd
```

Example 2

The following command skips the placement phase and preserves all routing information without locking it (re-entrant routing). Then it runs up to 999 passes of the router or stops upon completion and conformance to timing preferences found in the *pref.prf* file. Then it runs three delay-based cleanup router passes. If the design is already completely routed, the effect of this command is to just run three delay-based cleanup passes.

```
par -kp -i 999 -c 0 -d 3 input.ncd output.ncd pref.prf
```

Example 3

The following command runs 20 place and route iterations at effort Level 3. The iterations begin at cost table entry 5. Only the best 3 output design files are saved. The output design files (in *.ncd* format) are placed into a directory called *results.dir*.

```
par -n 20 -l 3 -t 5 -s 3 input.ncd results.dir
```

Example 4

The following command copies the input design to the output design using the -pr option. The placement and routing phases are skipped completely. You generate a delay file using the -y option, which creates a delay file that allows the user to check the delay times in the design without having PAR change any of the design's placement or routing.

```
par -pr -y input.ncd output.ncd
```

Example 5

The following example consists of two command lines. For the first, use the **-q** option if the design is prone to clock skew problems.

Suppose that you have determined the best level for the design to be level 3. If you wanted PAR to save the best three results running at level 3 with 20 place-and-route iterations, you would enter this command:

```
par -l 3 -n 20 -w -s 3 input.ncd output.dir
```

Now, if you wanted to run two passes of cost-based and delay-based cleanup on the three designs saved (without running placement), you would enter this command for each design:

```
par -kp -i 0 -c 2 -d 2 input.ncd output.ncd
```

Example 6

The following command below runs optimized placement and routing. The constructive placement results from a previous run are kept and refined with placement optimization. The router starts from scratch.

```
par -k input.ncd output.ncd
```

Example 7

The command below allows re-entrant routing. Use this command when your design is only partially routed and you want to complete it. Placement and placement optimization are skipped. In this case up to 30 router passes are run (you could run up to 999).

```
par -kp -i 30 input.ncd output.ncd
```

Example 8

The command below gives you a delay report for a placed and routed file without modifying the file.

```
par -pwr input.ncd input.ncd
```

Running CST from the Command Line

The cycle stealing program (CST) can also be run from the command line. Proper syntax and example of commands are provided in this section.

Syntax

cst [**-o** <*new_design*[**.ncd**]>] <*design*[**.ncd**]> [<*preference*[**.prf**]>] [**-b**]

where:

-o <*new_design*[**.ncd**]>	Designates the output `.ncd` file that will be generated if all timing constraints are met. The default file name of the output is the input file name with the prefix "`new_`" added to the file name. The resulting output file is dependent on the solution CST ascribes to the constraints. Generally, if all constraints are met, relaxed or tightened, CST attaches the "`new_`" prefix to the output `.ncd` file. **-b** The -b option gets best possible results even if the current preferences are already met. For example, if a timing preference was set to 100 MHz, it may be optimized to 150 MHz or higher. Because the higher frequency could strain other parts of the design, this option may not always be desirable. The default behavior stops when preferences are met.
<*design*[**.ncd**]>	The input physical design (*.ncd*) file that is a candidate for clock skew reduction.

<*preference*[**.prf**]>	The input preference (*.prf*) file that contains your user-defined constraints associated with the candidate *.ncd* file. Note that you do not have to specify this file in the command line if it has the same file name as the input *.ncd* file.

Running TRACE from the Command Line

Syntax

trce [-e | -v[<limit>]] [-p] [-sp <speed>] [-c] [-hld] <file_name[.ncd]> [-o <report[.twr]>] [<file_name[.prf]>] [-f <command_file>] [-sm [-hld] [-v]<stamp_file> <ncd_file> [<prf_file>]]

where:

-e and **-v** <*limit*>	Respectively, generates an error report or a verbose (full) timing report file in the current working directory. If neither of these switches is specified, you get a summary report, sent to standard output. <*limit*> limits the number of items reported for each timing preference). Enter 0 or greater (0 = no limit).
-p	Reports the number of asynchronous loops in verbose mode.
-sp <*speed*>	Overrides the default speed grade.
-c	Prints uncovered connections. The -c option should be used with the verbose timing report option.
<-v> **-hld** <*file_name*[.**ncd**]> <*file_name*[.**prf**]>	Performs hold time checks on FREQUENCY, CLOCK_TO_OUT, INPUT_SETUP and OFFSET preferences and outputs a verbose timing report in your working directory.
<*file_name*[.**ncd**]>	Specifies the physical design file.
-o <*file_name*[.**twr**]>	The name of the output file to contain the timing report. If no output name is specified, the report name defaults to the same root name as the *.ncd* file.
<*file_name*[.**prf**]>	The *.prf* file containing timing constraints for the design file. If a preference file is not specified, TRACE looks for one with the same root name as the *.ncd* file.
-f <*command_file*>	Execute command line arguments in the specified *command_file*.
[**-sm** [-hld] [-v]<*stamp_file*> <*ncd_file*> [<*prf_file*>]]	The **-sm** option enables STAMP model generation capability which simplifies verifying timing of a design at the board level. By default, TRACE will not generate STAMP model files. When **trce** is run with the **-sm** option a *.mod* file and a *.data* file are generated in addition to the *.twr* file.

Examples

Following are a few examples of TRACE command lines and a description of what each does.

Example 1

The following command verifies the timing characteristics of the design named *design1.ncd*, generating a summary timing report. Timing preferences contained in the file *group1.prf* are the timing constraints for the design. This generates the report file *design1.twr*.

```
trce design1.ncd group1.prf
```

Example 2

The following command produces a file listing all delay characteristics for the design named *design1.ncd*. Timing preferences contained in the file *group1.prf* are the timing constraints for the design. The file *output.twr* is the name of the verbose report file.

```
trce -v design1.ncd group1.prf -o output.twr
```

Example 3

The following command analyzes the file *design1.ncd* and reports on the three worst errors for each preference in *timing.prf*. The report is called *design1.twr*.

```
trce -e 3 design1.ncd timing.prf
```

Example 4

The following command analyzes the file *design1.ncd* and produces a verbose report to check on hold times on any FREQUENCY, CLOCK_TO_OUT, INPUT_SETUP and OFFSET preferences in the *timing.prf* file. With the output report file name unspecified here, a file using the root name of the *.ncd* file (i.e., *design1.twr*) will be output by default.

```
trce -v -hld design1.ncd timing.prf
```

Example 5

The following example commands summarize the three ways in which you can use the STAMP model generation **-sm** option for Trace.

The command line syntax to generate STAMP model files with max delay information and setup requirements is as follows:

```
trce -sm stamp.file design1.ncd
```

The command line syntax to generate STAMP files with min delay information and hold requirements is as follows:

```
trce -hld -sm stamp.file design1.ncd
```

The command line syntax to generate a conditional STAMP output in which certain paths are blocked is as follows:

```
trce -v -sm stamp.file design1.ncd timing.prf
```

Running BACK ANNOTATION from the Command Line

Series 3 and 4 Devices

The LDBANNO program back-annotates physical information (for example, net delays) to the logical design and then writes out the back-annotated design in the desired netlist format.

Input to LDBANNO is a physical design file (*.ncd*)— a mapped and partially or fully placed and/or routed design.

ldbanno [**-w**] [**-sp** <speed grade>] [**-min**] [**-x**] [**-dis** <*delay*> [**-sig** <*sig_file*>]] [**-i**] [**-z**] [**-a**] [**-neg**] [**-n** <*type*>] [**-l** <libtype>] [**-pre** <prefix>] [**-p** <*prffile*[.**prf**.]>] [**-o** <*netlist*>] <*ncdfile*[.**ncd**]>]

where:

-w	Overwrite the output file(s).
-sp <*speed grade*>	Re-target back annotation to a different speed grade than the one used to create the *.ncd* file. You are limited to those speed grades available for the port used in the *.ncd* file.
-min	Replaces all timing information for back annotation with the minimum timing for all paths. This option is used for simulation of hold time requirements. Separate simulations are required for hold time verification (**-min** switch) and delay time verification (normal output).
-x	Writes out a trce cross-reference file.
[**-dis** [<*delay*>]] [**-sig** <*sig_file*>]	The **-dis** option distributes routing delays by splitting the signal and inserting buffers. The <delay> value represents the maximum delay number in picoseconds between each buffer (1000 ps by default) The -sig option designates the use of a <sigfile> which is an ASCII file that contains all the top level signals to be split. One signal name can used per line. If -dis is given but -sig is absent, all top level signals will be distributed. These options will enable simulation of high frequency signals (e.g., clock signals), where the interconnection delay may be higher than the clock cycle. In such a case, that signal will be blocked in some simulators if the delay is not distributed.
-I	Inserts a buffer at each block input that has interconnection delay on it. **Note**: This option is valid only when the netlist type is Verilog. It is similar to the "tipd" statement in a VITAL compliant model in VHDL, which can help you to debug the design by isolating the interconnection delay and pin-to-pin delay on each input.
-z	Zero delay (do not calculate or write any delays). See Note for the -a option.
-a	Write all delays, even if they are zero. **Note**: By default, LDBANNO will write all non-zero delays to the appropriate output delay file. If the **-a** option is given, all pin to pin delays in the output netlist will be written even if they are zero. If the **-z** option is given, no delays will be calculated or written to the output file. If both the **-a** and **-z** options are given on the command line, the **-z** option will take precedence and no delays will be written to the output file.
-neg	For better compatibility with simulators, all negative setup/hold delay numbers in the SDF file are set to 0 by default. This may cause some discrepancies between back annotation and the TRACE result. Use the new -neg option to get the negative numbers in the SDF file and back annotation will match the TRACE report. Ensure that the simulator is able to handle negative numbers in the SDF file.
-n <*type*>	Netlist type to write out (*not* case-sensitive): • Verilog generic verilog formal with SDF delay file • VHDL generic VHDL format with SDF delay file

-l <*libtype*>	Valid library types are orca. **Note**: Back Annotation will accept "-l or3c00" for compatability with older scripts, but "-l orca" is preferred.
-o <*netlist*>	(optional) The name of the output file. The extension used for each output depends on the type of netlist being written (specified with the **-n** switch).
-pre <*prefix*>	(optional) Prefix to add to module names to make them unique for multi-chip simulation.
<*preffile*[.**prf**]>	(optional) The preference file in *.prf* format.
<*ncdfile*[.**ncd**]>	The input design file is a mapped and partially or fully placed and/or routed design in *.ncd* format.

Examples

Following are a few examples of LDBANNO command lines and a description of what each does.

Example 1

The following command back annotates *design.ncd* and generates a Verilog file *design.v* and an SDF file *design.sdf*. If the target files exist, they will be overwritten.

```
ldbanno -w -n verilog design.ncd
```

Example 2

The following command back annotates *design.ncd* and generates a VHDL file *backanno.vhd* and an SDF file *backanno.sdf*. Any signal in the design that has an interconnection delay greater than 2000 ps (2 ns) will be split and a series of buffers will be inserted. The maximum interconnection delay between each buffer would be 2000 ps.

```
ldbanno -dis 2000 -n vhdl -o backanno design.ncd
```

Example 3

The following command re-targets back annotation to speed grade -–2, and puts a buffer at each block input to isolate the interconntion delay (ends at that input) and the pin to pin delay (starts from that input).

```
ldbanno -sp 2 -i -n verilog design.ncd
```

Example 4

The following command generates Verilog netlist and SDF files without setting the negative setup/hold delays to 0:

```
ldbanno -neg -n verilog design.ncd
```

Series 2 Devices

When you back-annotate from the command line, you first run the NGDANNO command to back-annotate physical information (for example, net delays) to the logical design. Then you invoke one of the netlist writers NGD2EDIF, NGD2VER, or NGD2VHD) to write out the back-annotated design in the desired netlist format.

When you back-annotate using the MAP/Backanno Shell, the shell runs each of these commands automatically.

NGDANNO

The NGDANNO program distributes delays, setup and hold times, and pulse widths found in the physical *.ncd* design file back to the logical *.ngd* file. It merges mapping information from the *.ngm* file and placement, routing, and timing information from the *.ncd* file and puts all this data in an *.nga* (generic annotated) file.

If you make logical changes to an *.ncd* design from within EPIC that change the functional behavior of the design, NGDANNO cannot correlate the changed objects in the physical design with the objects in the logical design. It recreates the entire *.nga* design from the *.ncd*. You get this warning:

```
WARNING – NCD is out of sync with NGM...creating NGA from NCD
```

Input to the NGDANNO program is:

- *ncd* — a mapped and partially or fully placed and/or routed design
- *ngm* — (optional) a mapped *.ngd* file created by the technology mapper

The *.ngm* file supplies the back-annotation program with information about the logical design (for example, the logical design hierarchy, a gate level description of the design, and the correlation between the logical design and its physical mapping). Information about the physical design (for example, component and net delays, and site assignments for mapped components) is supplied by the input *.ncd* file. The logical information from the *.ngm* file and the physical information from the *.ncd* file complete the database necessary for simulation.

Output from the NGDANNO program is an *.nga* file, which is a back-annotated *.ngd* file. The *.nga* file acts as input to the netlist translation programs.

To run NGDANNO, from the UNIX or DOS command line enter:

ngdanno [**-f** <*command_file*>] [**-o** <*ngafile*[.**nga**]>] [**-p** <*prffile*[.**prf**.]>] < *ncdfile*[.**ncd**]> [<*ngmfile*[.**ngm**]>]

where:

-f <*command_file*>	(optional) Execute command line arguments in the specified *command_file*.
-o <*ngafile*[.nga]>	The output *.nga* file, which is a back-annotated *.ngd* file. If you specify only the *.ncd* file on the command line but do not specify an *.nga* file with the **-o** option, an *.nga* file is generated from the *.ncd* in the same directory. The *.nga* file has the same root name as the *.ncd* file. A physical *.nga* file is created, but there is no logical view of the design.
<*preffile*[.**prf**]>	(optional) The preference file in *.prf* format. You must include the appropriate extension with the file name on the command line.
<*ncdfile*[.**ncd**]>	The input design file is a mapped and partially or fully placed and/or routed design in *.ncd* format.
<*ngmfile*[.**ngm**]>	(optional) The mapped *.ngd* file created by the technology mapper. You must specify the *.ngm* file to have it used.

Netlist Writers

Netlist writers are ORCA programs that translate information contained in *.nga* or *.ngd* files into a specified netlist format.

NGD2EDIF

NGD2EDIF produces an EDIF 2 0 0 netlist in terms of the neoprim primitive set, allowing you to simulate designs before and after routing.

Input can be any of the following files:

- *.nga*, a back-annotated logical design file containing neoprim library components
- *.ngd*, a logical design file containing neoprim library components

Output is an *.edn* file, a netlist in EDIF format. The default *.edn* file produced by NGD2EDIF is generic. If you want to produce EDIF targeted to Mentor Graphics or Logic Modeling Group, you must include the **-v** option on the command line.

For implicit global signals on a device, NGD2EDIF automatically creates extra pins, which can cause interface differences when trying to match the original design.

ngd2edif [**-a** | **-n**] [**-w**] [**-v** <*vendor*>] [**-f** <*command_file*>] <*infile*>**.ngd** | .**nga** [<*outfile*>[**.edn**]]

where:

-a	Write all properties of the design. The default is to write only delay and error checking properties.
-n	Write out a flattened netlist with no properties of the design. Must be used with **-v viewlog**.
-w	Overwrite an existing output file.
-v <*vendor*>	Specifies the CAE vendor toolset to use the resulting EDIF file. Allowable entries are **mentor**, **lmg**, and **viewlog**. You must use the **-n** option when using **-v viewlog**.
-f <*command_file*>	Execute command line arguments in the specified *command_file*.
<*infile*[**.ngd** | **.nga**]>	The input file (*.nga* file comes from NGDANNO).
<*outfile*[.**edn**]>	The output EDIF file. The default output file has the same name as the input file, with an *.edn* extension.

NGD2VER

NGD2VER produces a Verilog netlist and an SDF (Standard Delay Format) file in terms of the neoprim primitive set, allowing you to simulate designs before and after routing.

Input is an *.nga* file, a back-annotated logical design file containing neoprim library components.

Two output files are generated by the NGD2VER program:

- *.v* — a Verilog netlist
- *.sdf* — an SDF file containing timing data; routing interconnect and primitive port delays; and timing checks, including setup, hold, and pulse width checks

The files have the same root name as the *.ngd* or *.nga* file unless you specify otherwise.

ngd2ver [**-w**] [**-n**] [**-f** <*command_file*>] <*infile*>**.ngd** | .**nga** [<*outfile*[.**v**]>

where:

-w	Overwrite an existing output file.
-n	Write out a flattened netlist of the design.
-f <*command_file*>	Execute command line arguments in the specified command_file.
<*infile*[**.nga**]>	The input `.nga` file from NGDANNO or an .ngd file.

<*outfile*[.**v**]>	The output Verilog file. The default output file has the same name as the input file. The SDF file has the same name as the Verilog file with an *.sdf* extension. **Note**: The SDF file produced is intended solely for use with the Verilog file. Do not attempt to use the SDF file in conjunction with the original design or the product of another netlist writer.

NGD2VHD

NGD2VHD produces a VHDL netlist and an SDF (Standard Delay Format) file in terms of the neoprim primitive set, allowing you to simulate designs before and after routing.

Input is an *.nga* file, a back-annotated logical design file containing neoprim library components.

Three output files are generated by the NGD2VHD program:

- *.vhd* — a VHDL netlist
- *.sdf* — an SDF file containing timing data; routing interconnect and primitive port delays; and timing checks, including setup, hold, and pulse width checks
- *.xrf* — cross reference file

The files have the same root name as the *.ngd* or *.nga* file unless you specify otherwise.

ngd2vhd [**-n**] [**-p**] [**-t**] [**-w**] [**-x**] [**-f** <*command_file*>] <*infile*>**.ngd** | .**nga** [<*outfile*[.**vhd**]>

where:

-n	Write out a flattened netlist of the design.
-p	Connect a pullup/down device to an unsourced net.
-t	Target this VHDL file to Mentor Time Explorer.
-w	Overwrite an existing output file.
-x	Do not generate the cross reference file.
-f <*command_file*>	Execute command line arguments in the specified *command_file*.
<*infile*[**.nga**]>	The input *.nga* file from NGDANNO or an *.ngd* file.
<*outfile*[.**vhd**]>	The output VHDL file. The default output file has the same name as the input file. The SDF file has the same name as the VHDL file with an *.sdf* extension. **Note**: The SDF file produced is intended solely for use with the VHDL file. Do not attempt to use the SDF file in conjunction with the original design or the product of another netlist writer.

Running BIT GENERATION from the Command Line

Syntax

bitgen [**-b**] [**-d**] [**-f** <*command_file*>] [**-h** [<*architecture*>]] [**-a**] [**-o** <*outfile*>] [**-x**] [**-j**] [**-m** <*format*>] [**-w**] [**-g** *option:value*] [**-J r** | **w** <*infile1*[**.ncd**] {<*infile2*[**.ncd**]}] [**-r** <*infile*[**.mif**]> <*infile*[**.ncd**]> <*outfile*>] <*infile*[**.ncd**]> [<*outfile*>] [<*preffile*[.**prf**]>]

where:

-b	Create a "rawbits" (*.rbt*) file instead of a binary file. Do not use **-b** with **-m** if you want both a rawbits file and a mask file (see the **-m** command below). Instead run BITGEN twice once with the **-b** option and once with the **-m** option.
-d	Do not run DRC.
-f <command_file>	Execute command line arguments in the specified *command_file*.
-h <*architecture*> or **-help** <*architecture*>	Display available BITGEN command options for the specified *architecture*. The **bitgen -h** command with no architecture specified will display a list of valid architectures.
-a	Outputs an ASCII file.
-o <outfile>	Specifies the output file name.
-j	Do not create a bitstream file.
-m <*format*>	Create a mask file. Format value equal 0, 1, or 2. The value 0 is the generic format, 1 is design-specific, and 2 is design-specific plus user RAMs.
-w	Overwrite an existing *.bit*, *.msk*, or *.rbt* output file.
-x	Byte mirror. Swaps 0s with 7s, etc.
-g option**:**value	Set ORCA configuration options. Options and values are case-insensitive. Available options are: ADDRESS (Series 3 and 4 only) Bitstream addressing mode. Options are **Increment** (default) for general use and **Explicit** for testing and debugging. CFGMODE Options are **Enable** (default) and **Serial**. DONEACTIVE Select the event that activates the DONE signal. Settings are **C1**, **C2**, **C3**, **C4** (default) (first CCLK rising edge after the length count is met, second CCLK rising edge after the length count is met, etc.). DONEPIN Options are **Pullup** (default) and **Pullnone**. EXTERNALCLK Options are **Yes** and **No**. GSRINACTIVE Select the event that releases the internal set/reset to the latches and flip-flops. Settings are **C1**, **C2** (default), **C3**, **C4**, and **Donein**. JTAG Enables or disables JTAG (boundary scan) after configuration. Settings are **Enable** and **Disable** (default).

OSCILLATOR
Enables the on-chip oscillator after configuration. Options are **Enable**, **Disable** (default), and **EnableDiv8** (divide the speed by eight).

OUTPUTSACTIVE
Select the event that releases the I/O from 3-state condition and turns the configuration-related pins operational. Settings are **C1**, **C2**, **C3**, **C4**, and **Donein** (when the DONEIN signal goes high the default).

OUTPUTSCFG
If enabled, the FPGA will not drive outputs during configuration/reconfiguration. Options are **Enable** (default) and **Disable**.

PARITY (2CA/2TA only)
Enables or disables a parity check by the FPGA of the configuration bitstream. Settings are **Enable** (default) and **Disable**.

RAMCFG
Options are **Reset** (default) and **No Reset**. Reset reinitializes the device when you download bitstream. No reset retains the current configuration and allows additional bitstream configuration.

READBACK
Allows you to extract the configuration data stored in an FPGA in order to verify the configuration. Settings are **Disable** (default), **Once** (allow one readback only; after that, readback cannot be invoked again), and **Command** (multiple readbacks).

READCAPTURE
Enable or disable readback of the configuration bitstream. Settings are **Disable** (default), **Read**, **UserNet**, and **Both**.

REGISTERCFG
Reset the PFU registers before configuration. Values are **Enable** (default) and **Disable**.

STARTUPCLK
Use CCLK or a User clock during startup. Options are **Cclk** (default) or **UserClk**.

SYNCTODONE
Synchronize the startup sequence to the external DONE signal. Options are **Yes** and **No** (default).

ZEROFRAMES
(Series 3 and 4) Option to specifiy that data frames with all zeros be written into the bitstream. Options are No and **Yes** (default). In Series 4, this option is used with Increment Address mode and in Series 3 it is used in Explicit Address mode.

Series 4 Addressing Modes

When **Yes** is specified with Increment Address mode, every data frame is written out with no address frames. When **No** is specified with Increment Address mode, all sequentional non-zero data frames are written without data frames. When a (zero) data frame is skipped, the address frame forthe next (non-zero) data frame is written.

Series 3 Addressing Modes

When **Yes** is specified with the Explict Address mode, every address frame is written out followed by every data frame. This is good for partial reconfiguration, where bitstream may only include a few frames. **No** is not allowed. A warning will be generated that *Zeroframes:No* is not compatible with Explicit addressing and the option will be ignored.

SYSBUSCFG
(Series 4 only) Options are **Reset** and **NoReset**. The system bus is either reset or not reset at reconfiguration.

	SYSBUSCLKCFG (Series 4 only) Options are **Reset** and **NoReset**. The system bus clock is either reset or not reset at reconfiguration. WAITSTATETIMEOUT (Series 4 only) Controls an internal system bus timeout counter. This counter counts the number of HCLK (sysbus clk) cycles goes by while the system bus is in wait states. Possible values are **0, 1, 2...15**. 0 - Forever (never time out) 1 - 2^2 HCLK cycles 2 - 2^4 HCLK cycles 3 - 2^6 HCLK cycles 4 - 2^8 HCLK cycles 5 - 2^10 HCLK cycles 6 - 2^12 HCLK cycles 7 - 2^14 HCLK cycles 8 - 2^16 HCLK cycles 9 - 2^18 HCLK cycles 10- 2^20 HCLK cycles 11 - 2^22 HCLK cycles 12 - 2^24 HCLK cycles 13 - 2^26 HCLK cycles 14 - 2^28 HCLK cycles 15 - 2^31 HCLK cycles GRANTTIMEOUT (Series 4 only) Controls an internal system bus grant counter which counts the number of HCLK (sysbus clk) cycles goes by before the grant signal is taken away from the master.This prevents the system bus from locking up if anything goes wrong. Possible values are **0, 1, 2...15**. See WAITSTAITTIMEOUT. LENGTHBITS (Series 4 only) Specifies the number of bits in the bitstream length field. Options are 24 (default) and 32.
-J r | **w** <*infile1*[**.ncd**]> {<*infile2*[**.ncd**]}	JTAG setup. Allows the user to set JTAG port read and write on the FPGA chip by providing them with the setup bitstreams. Users will use tools other than ORCA Foundry to download these bitstreams. Example syntax: [**-J w** <*infile1.ncd*> {<*infile2.ncd*>}] [**-J r**] [**-a**] [**-o** <*outfile*>] where: **-J r** generates a JTAG read setup bitstream.The default output is jtagread.jbt. **-J w** generates JTAG write setup bitstream. **-a** outputs an ASCII file. **-o** <outfile> specifies the output file name. The example syntax for JTAG setup will generate JTAG write setup bitstream that specifies *infile1.ncd* as the target design to be downloaded, so the correct initialization bits will be added to the bitstream. If more than one design is on the command line, the bitstreams for daisy-chained devices will be generated using JTAG port. The default output is *infile1.jbt*.

[**-r** <*infile*[**.mif**]> <*infile*[**.ncd**]> <*outfile*>]	(*for Series 4 system block RAM only*) Use this option to designate bitgen to program Series 4 system block RAM using the *.mif* file generated in SCUBA for input. For example, initialize a system bus block RAM by entering the following command in the command line: **bitgen -r** <*sysbus.mif*> <*mydesign.ncd*> <*mydesign.bit*> Using the SCUBA-generated sysbus *.mif* file, this command runs bitgen on *mydesign.ncd* and writes out *mydesign.bit* to initialize the system bus block RAMs. SCUBA will automatically generate a *.mif* file with a file name identical to your module name. Note that if block RAMs are already initialized by INITVAL properties in the *.ncd* file, addresses that are explicitly addressed in the *.mif* file will be overidden by the INITVAL properties.
<*infile*[.**ncd**]>	SCUBA will automatically generate a *.mif* file with a file name identical to your module name. Note that if block RAMs are already initialized by INITVAL properties in the *.ncd* file, addresses that are explicitly addressed in the *.mif* file will be overidden by the INITVAL properties.
<*outfile*>	The output file. If you do not specify an output file, BITGEN creates one in the input file's directory. If you specify **m** on the command line, the extension is *.msk*. If you specify **-b**, the extension is *.rbt*. Otherwise the extension is *.bit*. A report (*.bgn*) file containing all of BITGEN's output is automatically created under the same directory as the output file.

Running BITTOOL from the Command Line

The **bittool** program is used for incremental bitstream generation.

Syntax

bittool -d <*file1*> <*file2*> {<*file3*>} **-f -h** [**-z**] [**-b**] [**-o** <*outfile*>] [**-r**]

where:

-d <*file1*> <*file2*> {<*file3*>}	Compares the two bit or raw bit files. <*file 1*> is the original file and <*file 2*> is the updated file. If <*file 1*> does not have reconfiguration ram set, bittool will output a file with this bit set. The default file name for this file is *file1_changed.*[**rbt** | **bit**] unless <*file3*> is provided. This option is required.
-f <command_file>	Execute command line arguments in the specified *command_file*.
-h or -help	Displays available BITTOOL command options.
-z	Preserves zero frames for *file1_changed* or <*file3*> if <*file1*> is of explicit address format. Default is **No Zeroframes**.
-b	Specifies that output file will be a raw bit file (*.rbt*).
-o <outfile>	Specifes <outfile> as the output file name. The default output file name is *file1_file2.*[**rbt** | **bit**] by outfile.
-r	Set the “NoReset RAMs during reconfiguration” bit in output file. Default is “Reset RAMs during reconfiguration”.

Running CHIPDEBUG from the Command Line

Syntax

chipdebug -a <*architecture*> **-p** <*part*> [**-f**] [**-h**] [**-all**] [**-no_reg**] [**-no_ram**] **-file** <*readback_file*> {<*output_file*>}

where:

-a <architecture>	Specifies the architecture for the readback file. This option is required.
-p <part>	Specifies the device. This option is required.|
-f <command_file>	Execute command line arguments in the specified *command_file*.
-h or -help	Displays available CHIPDEBUG command options.
-all	Specifies that all registers and RAMs including those with zero values appear in the output file. The default is to output only output registers and RAMs with no zero values.
-no_reg	Specifies that no register values appear in the output file.
-no_ram	Specifies that no RAM values are in the output file.
-file <readback_file> {<outfile>}	Specifies the readback file to use for output. That is, readback_file is readback file name and {<*outfile*>} is the output file name you specify (default name is readback_file.pos if not specified). The **-file** option is required.

Running PROM GENERATION from the Command Line

Syntax

promgen [**-b**] [**-f**] [**-h**] [**-n** <*bitfile*> {<*bitfile*>}] [**-o** <*outfile*>] [**-p** <*format*>] [**-s** <*size*>] [**-t** <*size*>] {**-u** <*hexaddr*> <*bitfile*> {<*bitfile*>}} {**-d** <*hexaddr*> <*bitfile*>{<*bitfile*>}}

where:

-b	Specifies byte-wide mode (no bit mirroring).
-f <command_file>	Execute command line arguments in the specified *command_file*.
-h or -help	Displays available PROMGEN command options.
-n <bitfile> {<bitfile>}	Load *.bit* or *.rbt* file(s) up or down from next available PROM address following the previous load. This option requires that a previous **-u** or **-d** option be given. You can use multiple **-n** options, (for example, **-n** *bitfile* [**-n** *bitfile*...]). Filenames entered after the **-n** option are not daisy chained to previous files.
-o <outfile>	The output PROM file. If you do not specify a file name, the PROM file will have the same root name as the first bit file loaded, with the extension specified with the **-p** option. If you do not use the **-p** option, the extension name for the PROM file defaults to *.mcs*. A report (*.prm*) file containing all of PROMGEN's output is automatically created under the same directory as the output file.
-p <format>	Specifies the PROM format as **mcs**, **exo**, or **tek**. If used, must precede **-u, -d,** or **n** on the command line. The default is **mcs**.
-s <*size*>	Specifies the PROM size in kilobytes. Must be a power of 2. If not specified, the smallest possible size will be selected.
-t <*size*>	Split the file into multiple prom-files with user specified size. The resulting files are named as <file>00.<ext>, <file>01.<ext>, <file>02.<ext>, ..., etc. Cannot be used with switch **-s, -n,** or **-d**.

-u <hexaddr> <bitfile> {<bitfile>}	Load bit file(s) up from the specified PROM address, that is, into successively higher addresses from the starting point. Entering multiple filenames after the **-u** option concatenates the files in a daisy chain. You may use several **-u** options to load files at various addresses, (for example, **-u** <*hexaddr*> <*bitfile*> **-u** <*hexaddr*> <*bitfile*>). You must specify a starting address.
-d <hexaddr> <bitfile> {<bitfile>}	Load bit file(s) down from the specified PROM address, that is, into successively lower addresses from the starting point. Entering multiple filenames after the **-d** option concatenates the files in a daisy chain. You may use several **-d** options to load files at various addresses, (for example, **-d** <*hexaddr*> <*bitfile*> **-d** <*hexaddr*> <*bitfile*>). You must specify a starting address.
<bitfile>	Specifies the name(s) of the *.bit* or *.rbt* file(s) to be loaded. The default extension is *.bit*.

Notes

Previously, the –t option required a file argument:

```
promgen -p exo -t 256 orca.bit
```

However, now it is a separate parameter. To do the same action, do the following:

```
promgen -p exo -t 256 -u 0 orca.bit
```

Examples

The following command loads the file *orca.bit* down from Hex address 0x0300 in a PROM file with a Motorola format:

```
promgen -p exo -d 300 orca.bit
```

The following command daisy chains files *att1.bit* and *att2.bit* up starting at the next available PROM address:

```
promgen -u 0F00 att1.bit att2.bit
```

The following command daisy chains *orca1.bit*, *orca2.bit*, and *orca3.bit* up from PROM address 0x0000 and loads *orca4.bit* (not daisy chained to the other files) at the next available address, using a Tektronix format and naming the output file *promtest*:

```
promgen -p tek -u 0 orca1.bit orca2.bit orca3.bit n  orca4.bit -o promtest
```

Running DOWNLOAD/UPLOAD from the Command Line

Syntax

devprog [**-a** <*architecture*>] [**-b** <*baud_rate*>] [**-c** <*cabletype*>] [**-d** <device>] [**-f** <*command_file*>] [**-h**] [**-j w** | **r**] [**-p** <*port*>] <*file_name*[***.ext***]> {<*file_name*[***.ext***]>}

where:

-a <architecture>	Specifies the architecture of the device you are programming. You must enter **-a** if you will be programming the device from a PROM file, but you do not have to enter an **-a** option if you will be programming from a *.bit* or *.rbt* bitstream configuration file.
-b <baud_rate>	Specifies the baud rate at which the port will operate for programming as follows: • Workstation: Solaris **9600** (default), **19200**, **38400** • PC 9600, 19200, 38400 (default), 57600, 115200

-c <cabletype>	Specifies the download cable type: • **serial** | **SERIAL** serial cable • **luc** | **LUC** Lucent Technologies cable (default) • **neo** | **NEO** NeoCAD cable
-d <device>	Specifies the device. It is required for upload in **-j r** option usage.
-f <command_file>	Execute command line arguments in the specified *command_file*.
-h or –help	Displays available DEVPROG command options.
-j w | r	Allows the use of Lucent JTAG cable for downloading bitstreams and uploading the configurations to and from the chip. **w** upload to the specified file name. **r** download from the file name.
-p <*port*>	Specifies the port to which you have attached the download cable. Acceptable values for *port* are: • Solaris **a** (default), **b** • PC **com1**, **com2** (default), **com3**, **com4** You can specify ports other than these if they are available; for example, **-p /dev/ttyc**.
<file_name[.ext]> {<file_name[.ext]>}	Specifies the name of the input data file containing the configuration information for the target device or the readback file name fo the **-j r** option. You can also daisy chain multiple bitstreams. Note that you cannot daisy chain PROM files with **devprog**. *ext* signifies the type of configuration file (input file format): **.bit** bitstream (the default) **.rbt** rawbits file **.mcs** Intel PROM file format must be created at address 0 and counting up **.exo** ExorMax PROM file format **.tek** TekHex PROM file format **.xxx** Readback file

Examples

The following example shows a Series 3 OR3C80 design bitstream for download using the Solaris port “a” at 38400 baud rate through a Neocad cable. Note that port “a” does not have to be specifed since it is the command line default.

```
devprog or3c00 or3c80 -b 38400 -c neo design1.bit
```

The following example shows three Series 4 OR4E06 design bitstreams daisy chained for download using a serial cable at 57600 baud rate through port “b.” Note that the **-a** and **-d** switches are not needed at all if bitstream files are used, because the device information is already embedded in the *.bit* files. These switches need only be specified when PROM (*.mcs*, *.tek*, *.exo*) files are downloaded.

```
devprog or4e00 or4e06 -b 57600 -c serial -p b design1a.bit design1b.bit
design1c.bit
```

The final example shows the same three OR4E06 design bitstreams daisy chained for download using the JTAG port at 115200 baud rate through port “com2.”

```
devprog -b 115200 -c serial -j w -p com2 design1a.bit design1b.bit
design1c.bit
```

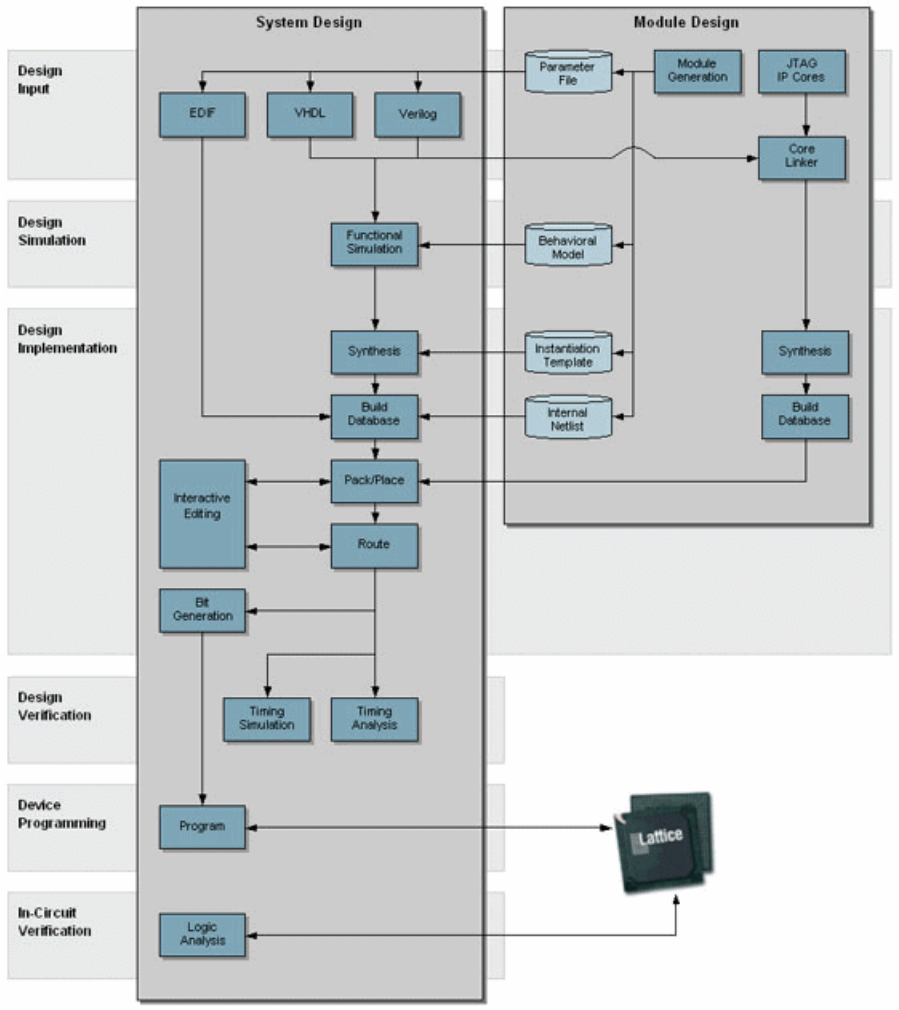
Daisy Chaining Bitstream Guidelines

- Only .rbt and .bit files may be daisy chained.
- *Only the serial cable supports JTAG download.* When daisy chaining with JTAG, all devices in the chain must be from the same architecture family. In particular, it is not legal to include a mix of Series 4, Series 3, and Series 2 in one chain when downloading via JTAG.
- *No daisy chaining of PROM files is allowed.* PROM files may continue to contain multiple bitstreams, but only a single PROM file may be used with **devprog**. Attempting to daisy chain PROM files will result in a fatal user error.
- *A daisy chained PROM file (multiple bitstreams added during **promgen**) may not be downloaded with a JTAG download.* It may, however, but downloaded a regular serial download. Attempts to use this configuration will result in only the first device in the chain being programmed. Note that the software will not warn you or produce an error when you attempt to daisy chain a PROM in a JTAG download.
- If the first design in a daisy chain is a Series 4 device and uses the 32-bit address option (default is 24-bit address), then all designs in the chain must also be capable of using the 32-bit address, that is, all the designs must be Series 4 or later. The software will detect if this rule is violated and error out with an appropriate message.

CHAPTER 2 *ispXPGA Process Flow*

Introduction

ispXPGA Design



Lattice's ispLEVER® 4.0 design tool allows you easy implementation of designs using ispXPGA devices. Synthesis library support is available for the major logic synthesis tools. The ispLEVER tool takes the output from these common synthesis packages and places and routes the design in the ispXPGA device. The tool allows floor planning and the management of other constraints within the device. The tool also provides outputs to common timing analysis tools for timing analysis.

To increase designer productivity, Lattice provides a variety of pre-designed modules referred to as IP cores. These IP cores let you concentrate on the unique portions of your design while using pre-designed blocks to implement standard functions such as bus-interfaces, standard communication-interfaces, and memory-controllers.

Supported Device Families

- ispXPGA

Overview of ispLEVER for ispXPGA

The ispLEVER program offers an integrated environment consisting of several tools necessary to implement Lattice ispXPGA devices. These tools are briefly described in the following paragraphs and covered in detail in their respective Help. They are listed in alphabetical order.

Constraint Editor

The Constraint Editor lets you specify pin and node location assignments, group assignments, I/O types settings, power level settings, resource reservations, sysCLOCK attributes, as well as output slew rates and JEDEC file options. The Constraint Editor reads the constraint file and displays the constraint settings. Modifications to the constraint file are made via the function dialogs. See the Constraint Editor Help for more information about this tool.

ispXPGA Floorplanner

The ispXPGA Floorplanner provides graphical user interface to facilitate the management of ispXPGA device real estate to conform to critical circuit performance requirements and to shorten design turn around time. See the ispXPGA Floorplanner Help for more information about this tool.

ispEXPLORER

The ispEXPLORER lets you run multiple passes of your design using different combinations of Fitter/Optimizer settings and critical timing constraints to achieve the best solution. Results are summarized in a single spreadsheet and detailed reports for each run are accessible. See the ispEXPLORER Help for more information about this tool.

ispTRACY IP Manager

The ispTRACY tool helps you debug your ispXPGA circuitry by measuring the logic behavior of internal nodes inside the devices. The ispTRACY tool analyzes the behavior of internal nodes that are captured into the trace memory that is implemented by the ispXPGA internal block RAMs. The behavior of these internal nodes can then be written out through the ispXPGA JTAG port when certain trigger conditions are matched.

The RTL templates and source code of ispTRACY IP cores for ispLA, ispJTAG and JTAG_PORT are created through ispTRACY IP Core Generator. The easy-to-use GUI allows user to specify the IP core parameters to customize the IP core. The IP source code is not pre-compiled but is generated dynamically using the ispTRACY Core Compiler engine. The RTL source code with pre-defined black box instances can be encrypted in order to protect the IP property.

See the ispTRACY IP Manager Help for more information about this tool.

ispTRACY Logic Analyzer

The ispTRACY Logic Analyzer interfaces directly to the ispLA and ispJTAG cores in the design. You can set up triggers, select capture modes, enable external trigger input/output, and run or stop the triggers.

The ispTRACY Logic Analyzer can launch the ispVM programming software to configure the specified device. When the acquisition data is displayed in the Signal Analysis waveform viewer.

The ispTRACY Logic Analyzer manipulates the raw silicon bit data based on the setting information contained in the Tracy Core Generator (`.tcg`) file.

The Logic Analyzer allows user to set trigger configuration and extract ispTRACY information from a programmed device through the JTAG ports. The GUI allows you to trace the signal/bus in waveform viewer. You can also the import and export to communicate with external Logic Analyzer.

See the ispTRACY Logic Analyzer Help for more information about this tool.

LeonardoSpectrum for Lattice

The LeonardoSpectrum™ synthesis environment from Mentor Graphics® lets you can create Lattice ispXPGA device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process. LeonardoSpectrum combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features. See the LeonardoSpectrum for Lattice documentation supplied by the manufacturer for more information about this tool.

ModelSim for Lattice

The ispLEVER software supports third-party HDL simulation and verification with ModelSim™ from Mentor Graphics®. Using this integrated package, you can simulate single Verilog or VHDL designs within one environment. See the ModelSim for Lattice documentation supplied by the manufacturer for more information about this tool.

Module/IP Manager

The Module/IP Manager enables you to generate cores (parameterized modules and IP cores) from templates supplied by Lattice Semiconductor and third-party vendors. After generating the module, the Module/IP Manager automatically stores the instantiation template and related files to your project folder. See the Module/IP Manager Help for more information about this tool.

Performance Analyst

The Performance Analyst analyzes the performance of your design after it has been optimized and implemented by the Fitter. See the Performance Analyst Help for more information about this tool.

Project Navigator

The Project Navigator is the primary interface for ispLEVER and provides an integrated environment for managing the project elements and processes, as well as accessing all ispLEVER tools. See the Project Navigator Help for more information about this tool.

Report Viewer

You can use the Report Viewer to view, but not edit, the various report files generated by ispLEVER. See the Report Viewer Help for more information about this tool.

Synplify for Lattice

Synplify® for Lattice is a logic synthesis tool for ispXPGA devices, developed by Synplicity®. Synplify starts with high-level designs written in Verilog or VHDL hardware description languages (HDLs). Synplify then converts the HDL into small, high-performance, design netlists that are optimized for Lattice devices. See the Synplify for Lattice documentation supplied by the manufacturer for more information about this tool.

Tcl Editor

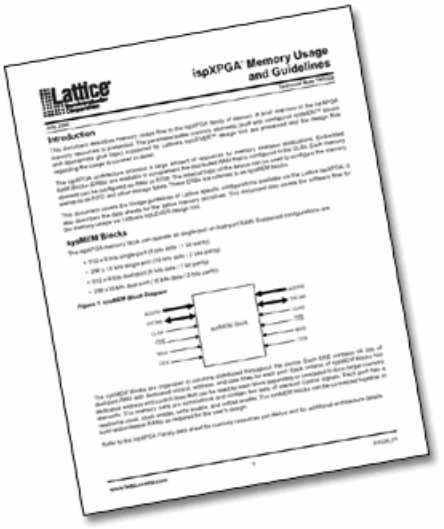
The Tcl Editor is a design tool that allows you to create, edit, and run a series of Tool Control Language (TCL) commands. These commands invoke individual ispLEVER processes for generating a complete programmable logic solution. The default text editor permits you to edit TCL code, and it highlights key TCL language elements in different colors to clarify the nature and use of each language element. You can generate a Tcl script for a project in Project Navigator, open it in the Tcl Editor, edit the script, and run it. The Tcl Editor, together with the robust features of the language, gives you more control over the design environment. See the Tcl Editor Help for more information about this tool.

Text Editor

The Text Editor is the ispLEVER text entry tool. You use this tool to create and edit text-based files, such as ABEL-HDL files, test stimulus files, and project documentation files. See the Text Editor Help for more information about this tool.

ispXPGA Application Notes

The Lattice Semiconductor web site lists several Application Notes for ispXPGA devices.



Design Entry

Verilog HDL Design Entry

The ispLEVER software supports Verilog HDL, a hardware description language used to design and document electronic systems. Verilog HDL allows designers to design at various levels of abstraction.

All Lattice devices support Verilog HDL design.

Adding a Verilog HDL Module to Your Design

To add a Verilog HDL module to a design, you can either import a `.v` file, or create a new Verilog HDL module file with the Text Editor.

Creating a New Verilog HDL Module

You can use the Text Editor to create a new Verilog HDL module.

To create a new Verilog HDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. Select **Verilog Module** and click **OK**. The Text Editor window appears together with the New Verilog Module dialog box.
3. In the dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new Verilog HDL file appears in the Text Editor window.
5. Use the commands on the Edit menu to Cut, Copy, Paste, Find, or Replace text.

Synthesizing Your Verilog HDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor Graphics LeonardoSpectrum. You can synthesize your Verilog HDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

VHDL Design Entry

VHDL is a language for describing the structure and function of integrated circuits. VHDL allows you to:

- Describe the hierarchical structure and interconnect of a design.
- Specify the function of designs using familiar programming language forms.
- Simulate the design before being manufactured, so that design alternatives can be quickly compared and tested.

All Lattice devices support VHDL design.

Adding a VHDL Module to Your Design

To add a VHDL module to a design, you can either import a `.vhd` file, or create a new VHDL module file with the Text Editor.

Creating a New VHDL Module

You can use the Text Editor to create a new VHDL module.

To create a new VHDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. In the dialog box, select VHDL Module and click **OK**. The Text Editor window appears together with the New VHDL Source dialog box.
3. In the New VHDL Source dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new VHDL file appears in the Text Editor window.
5. Use the commands in the Edit menu to Cut, Copy, Paste, or Replace text.

Synthesizing Your VHDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor graphics LeonardoSpectrum. You can synthesize your VHDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

EDIF Design Entry

The Electronic Design Interchange Format (EDIF) is a format used to exchange design data between different ECAD systems.

The EDIF format is designed to be written and read by computer programs that are constituent parts of EDA systems or tools. Its syntax has been designed for easy machine parsing and is similar to LISP.

The ispLEVER software supports EDIF Version 2 0 0.

All Lattice devices support EDIF design entry.

Importing an EDIF Netlist

You can import a design netlist description into the ispLEVER software from a third-party synthesis or schematic tool if the design file is formatted as EDIF 2 0 0.

***Note**: The project that you are importing the netlist into must be an EDIF project.*

To import an EDIF netlist into your project:

1. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
2. Change Files of type to **EDIF Netlist** (`*.ed*`), and then select the EDIF file that you want to import.
3. Click **Open** to open the Import EDIF dialog box.
4. The default setting for power and ground in the ispLEVER software are the VCC and GND symbols. If you know that the EDIF generated by other tools uses a different convention, you can change it in the window. Select **Custom**. Select either **Symbol** or **Net** representation. Then type the new names for VCC and GND.
5. If you are following the recommendation from Lattice for generating the EDIF file from the supported third party design kit, select **CAE Vendors**. From the list, choose the vendor that generated the EDIF file: Mentor, Synopsys, Synplicity, or Viewlogic.
6. Click **OK**. The software adds the selected EDIF file to the project sources.

Translating EDIF Properties

By default, the ispLEVER software ignores EDIF properties. If you want the ispLEVER software to translate EDIF properties to design constraints for the Fitter, do the following:

1. In Project Navigator, choose **Tools > Import Source Constraint Option** to open the dialog box. The Import Source Constraints Option dialog box lets you import constraints, such as Location (pin/node) Assignments, Group Assignments, and Output Slew Rate, from source files (ABEL, schematic, or EDIF).
2. In the dialog box, select **Auto Import Source Constraints**.
3. Click **OK**.

When you select this option, the ispLEVER software displays a confirmation dialog box prior to implementing the function. This confirmation dialog box appears every time you run the Fit Design process, unless you select the Do Not Import Source Constraints option.

On the warning message dialog box, if you click **Yes**, the constraints from the source files are written into the project constraint file.

***Important**: Constraints from source files and existing constraints in the project constraint file are not merged; existing constraints are overridden by the new constraints.*

Existing constraints (only Location Assignments, Group Assignments, and Output Slew Rate are affected) in the project are cleared, regardless of constraints that might exist in the source file. If there are constraints in the source file, the new constraints are written into the project constraint file. If there are no constraints in the source file, no constraints are written into the file.

EDIF Properties

The ispLEVER software will take design-specific constraints from the properties in the EDIF netlist. The following is the list of properties that the Fitter supports.

PIN LOCATION Property

Name: LOC

Value: {PIN # }

Example: LOC = P20

Scope: IO PORT, net connect to the IO port.

GROUPING Property

Name: GROUPING

Value: GROUP NAME

Example: Use the following command to assign signal locations in your design. In this case, you have a list of internal nodes: a, b, and c, and you want to assign them into a group “mg.” The location of this group needs to be Block “A”, Segment “2”:

On net a, grouping = mg

On net a, loc = "A, 2"

On net b, grouping = mg

On net b, loc = "A, 2"

On net c, grouping = mg

On net c, loc = "A, 2"

OUTPUT SLEW Property

Name: SLEW

Value: {Fast, Slow}

Example: To set port A to high slew, put the following property on the net or port: SLEW=Fast

Scope: OUTPUT PORT/NET

SIGNAL OPTIMIZATION Property

Name: OPT

Value: {KEEP, COLLAPSE}

Scope: On any net of the design.

OPEN DRAIN Property

Name: OPENDRAIN

Value: {On/Off}

Example: To set port A to an open drain, put the following property on the net or port: OPENDRAIN=on

Scope: OUTPUT PORT/NET

PULL Property

Name: PULL

Value: {On/Off/Hold}

Example: To set port A to pull up, put the following property on the net or port: PULL=on

Scope: OUTPUT PORT/NET.

OUTPUT VOLTAGE Property

Name: VOLTAGE

Value: {VCC/VCCIO}

Example: To set port A to output voltage at VCCIO level, put the following property on the net or port: VOLTAGE=VCCIO

Scope: OUTPUT PORT/NET.

Module and IP Design Entry

The Module/IP Manager helps you build large designs using Lattice Parameterized Modules (modules) and intellectual property cores (IPs). Including a module is as easy as:

- Selecting a module or IP core from the module tree
- Specifying parameters in the configuration window
- Instantiating the module or IP core with your text editor.

The Module/IP Manager is fully integrated with every level of the ispLEVER software, simplifying module and IP management within your design project.

Two Approaches to Modules and IP Cores

The ispLEVER software lets you choose the most convenient method for placing parameterized modules and IP cores into your design database. In both methods the software produces the instantiation template and its associated files and saves all of them in the project directory.

GUI Approach

The Module/IP Manager graphic user interface (GUI) simplifies the process of selecting and configuring a module or IP core. The interface lists all available modules by type and displays the parameters in an easy-to-use dialog box. The GUI approach requires more time for editing a design than the PMI approach.

PMI Approach

The Parameterized Module Instantiation template (PMI), though more difficult to use than the GUI, gives you the flexibility of editing a module at any time without having to recreate it. Using the PMI template, you embed the core description of your module into your HDL source files.

Design Simulation

Running a functional simulation after a design description is complete allows you to verify that the description is functionally correct. Also, by simulating the functionality of your design *before* synthesis, you can find and correct basic design errors sooner. While functional simulation will verify your Boolean equations, it does not indicate timing problems.

The ispLEVER software supports functional simulation for Lattice Semiconductor CPLD and FPGA devices using the Lattice Logic Simulator or *ModelSim* from Mentor Graphics. The RTL design can be simulated for functionality before synthesis using the VHDL or Verilog design description and an input stimulus file.

Simulation Environments

The functional simulators operate in both integrated and stand-alone environments.

Integrated Simulation — To simulate a design inside the current project, the ispLEVER software provides integrated simulation. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the simulation source file, the processes in the tables below are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

For ModelSim, ispLEVER includes scripts that run functional simulation automatically using Lattice-defined settings and preferences. You can customize the run by creating a different script files (DO file), which is a simple script that contain commands that are equivalent to the ModelSim GUI commands. This macro is automatically called when you run ModelSim.

For information about creating your own ModelSim macros, see the *ModelSim User's Manual*, *Chapter 11 Tcl and Macros*, provided with your ispLEVER software (Third-Party Manuals).

CPLD and ispGDX Project Navigator Processes	Simulation Tool Invoked
Functional Simulation	Lattice Logic Simulator
Verilog Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Functional Simulation	ModelSim
Verilog Post-Route Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim
VHDL Post-Route Functional Simulation	ModelSim

Stand-alone Simulation — The ispLEVER software supports stand-alone functional simulation. This provides an easy entry if you need to simulate a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator **Tools** menu.

Design File Descriptions

Lattice Logic Simulator and ModelSim enable you to simulate the operation of your design in the following design entry formats:

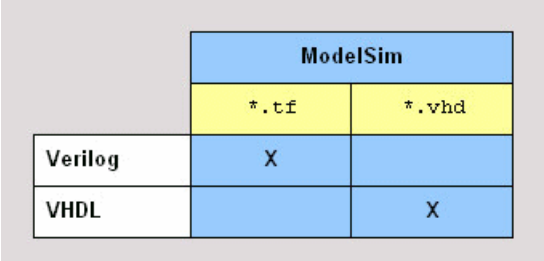
- ABEL-HDL format (*design*.abl) — a hierarchical logic description language that supports a variety of behavioral input forms, including high-level equations, state diagrams, and truth tables. (CPLD designs only)
- Schematic format (*design*.sch) — describes your circuit in terms of the components used and how they connect to each other. (CPLD designs only)
- VHDL format (*design*.vhd) — Very High Speed IC Hardware Description Language format.
- Verilog HDL format (*design*.v) — an industry-standard hardware description language used to describe the behavior of hardware that can be implemented directly by logic synthesis tools.

The Lattice Logic Simulator also supports mixed design entry as follows:

- Schematic and ABEL-HDL (CPLD designs only)
- Schematic and VHDL
- Schematic and Verilog HDL

ispXPGA Test Stimulus Files

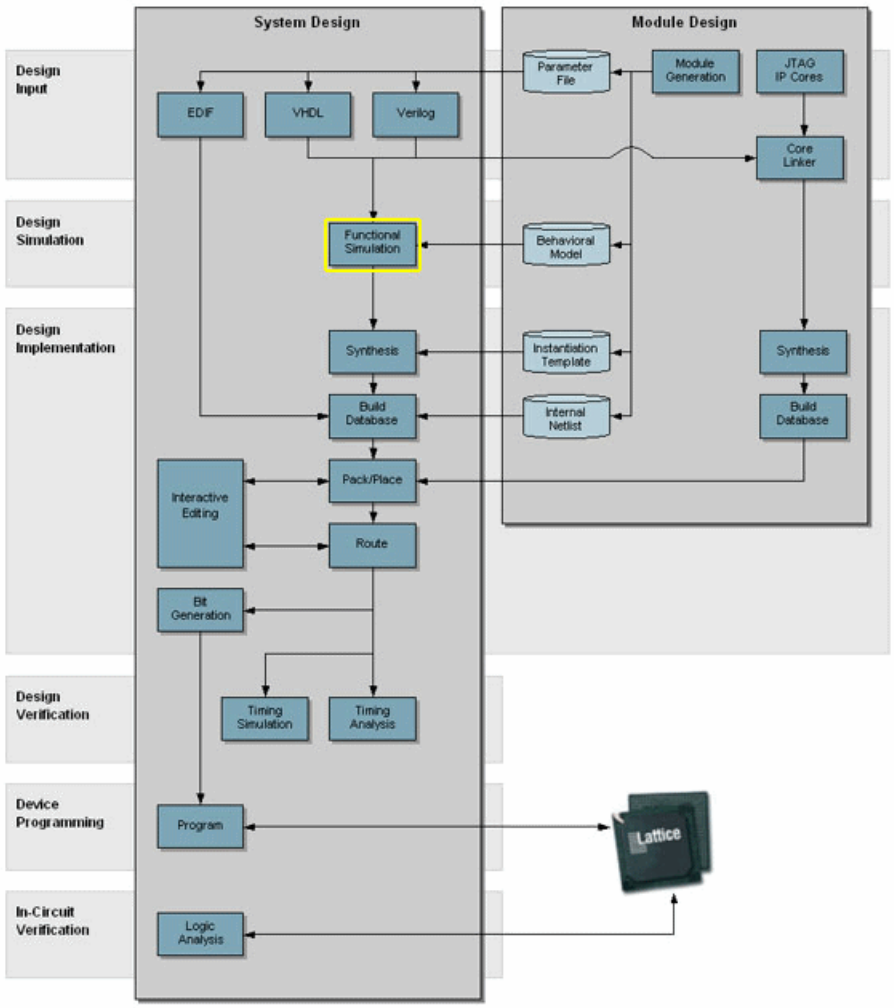
Once you have completed your design (or a module of the design), you can test it to confirm that it behaves the way you expect it to. The ModelSim simulator supports ispXPGA functional simulation for HDL design entry methods and requires at least one Verilog test fixture (*.tf) or VHDL test bench (*.vhd) stimulus file.



	ModelSim	
	*.tf	*.vhd
Verilog	X	
VHDL		X

ispXPGA Simulation Process Flow

The figure below shows functional simulation within the ispXPGA process flow.



Creating a Verilog Test Fixture from a Template

Verilog test stimulus can be specified either in the top-level HDL source or in a separate test fixture (`.tf`) file. You can create the test fixture manually using a text editor or use a Verilog Test Fixture template (`.tfi`) file.

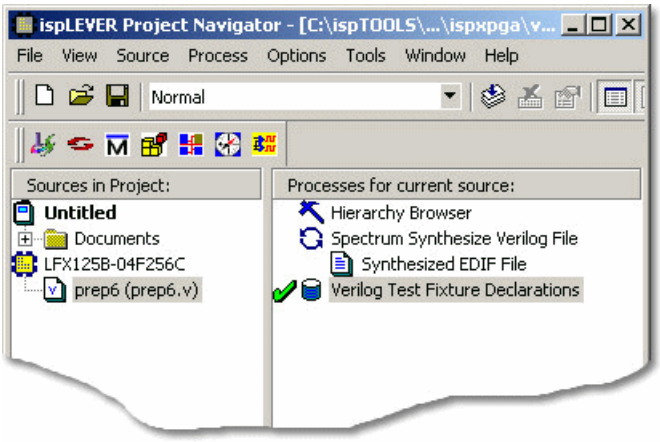
The easiest way to automatically create a Verilog Test Fixture template is using the Project Navigator Verilog Test Fixture Declarations process. After the test fixture template file (`.tfi`) is created, you must add your test vectors and rename it with the extension `.tf` before importing it into your design.

To automatically generate the Verilog test fixture template file and import it into your design:

1. Open your Verilog design in the Project Navigator.
2. In the Sources window, select the top-level Verilog design source (`*.v`) file.

***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

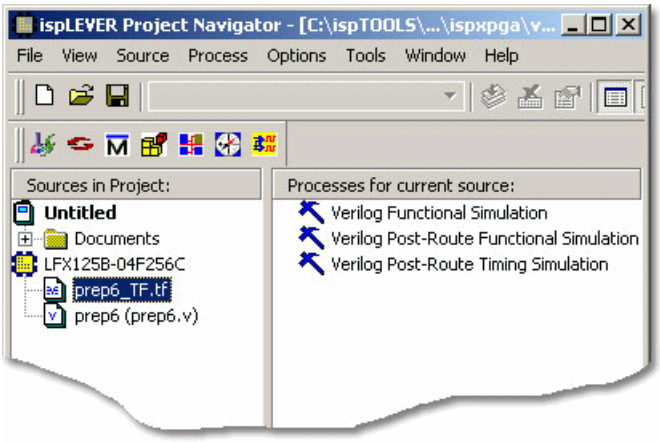
3. In the Processes window, double-click the **Verilog Test Fixture Declarations** process.



   This process creates a template file for a Verilog Test fixture (`<verilog_sourcefile_name>.tfi`). However, in order to use this file as a test fixture in your design, you must edit it and rename it with the extension `.tf`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `prep6_TF.tf`. Add the "`TF`" to the name so that the file will not be overwritten. Change the file extension to "`.tf`" so that it can be imported into the project as a test fixture source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test fixture file. Click **Open**.
7. In the Associate Verilog Test Fixture dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a Verilog Test Fixture. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Creating a VHDL Test Bench from a Template

VHDL test stimulus can be specified either in the top-level HDL source or in a separate test bench (`.vhd`) file. You can create the test bench manually using a text editor or use a VHDL Test Bench template (`.vht`) file.

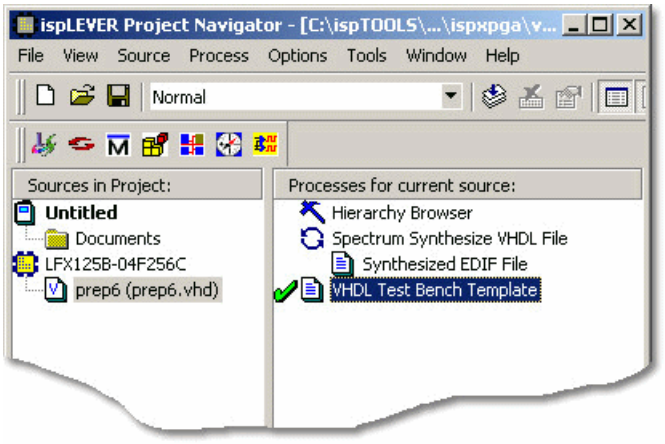
The easiest way to automatically create a VHDL Test Bench template is using the Project Navigator VHDL Test Bench Template process. After the test bench template file is created, you must add your test stimulus and rename it with the extension `.vhd` before importing it into your design.

To generate the VHDL test bench template and import it into your design:

1. Open your VHDL design in the Project Navigator.
2. In the Sources window, select the top-level VHDL design source (`*.vhd`) file.

   ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

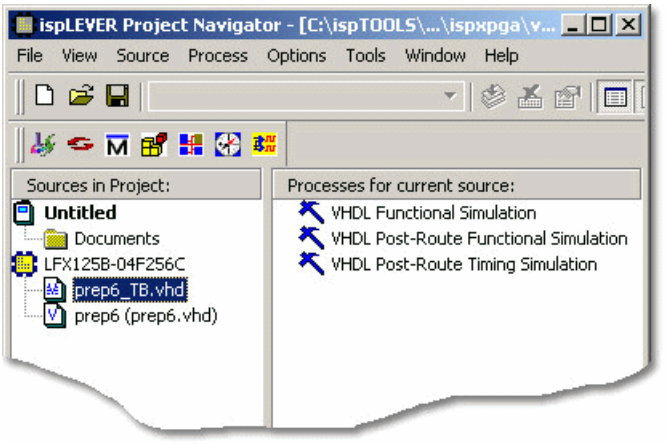
3. In the Processes window, double-click the **VHDL Test Bench Template** process.



   This process creates a template file for a VHDL Test Bench (`<vhdl_sourcefile_name>.vht`). However, to use this file as a test bench in your design, you must edit it and rename it with the extension `.vhd`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `prep6_TB.vhd`. Add the "`TB`" to the name so that the file will not be overwritten. Change the file extension to "`.vhd`" so that it can be imported into the project as a test bench source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test bench file. Click **Open**.
7. In the Import Source Type dialog box, select **VHDL Test Bench** and click **OK**.
8. In the Associate VHDL Test Bench dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a VHDL Test Bench. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Interfacing with ModelSim

ModelSim functional simulation generates several batch files, such as `.fdo`, `.udo`, and `.tdo`. ModelSim users will frequently take advantage of customizing batch files to control their simulation, for example specifying signals to display, run time, and waveform display options.

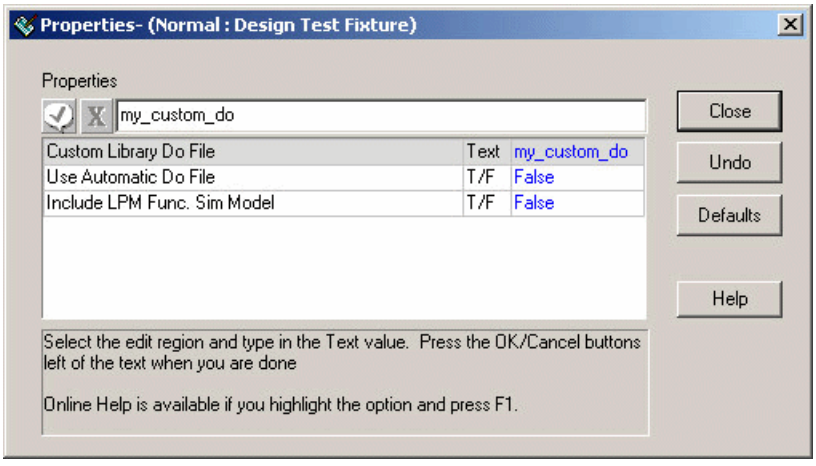
There are two options for creating custom DO files:

Option 1

- In the project folder, edit the `*.udo` file. The Project Navigator will not overwrite this user DO file.

Option 2

1. In the Project Navigator Sources window, select the test bench source.
2. In the Processes window, right-click the functional simulation process to open the Properties dialog box.
3. In the dialog:
   - Type a name for the file in the Custom Library Do File field and click the checkmark icon
   - Set Use Automatic Do File to **False**.
   - Click **Close**.



4. The software will automatically create this file when functional simulation is run. Edit this file as needed.

Design Implementation

Synthesizing

Synthesizing ispXPGA Designs

For Verilog and VHDL designs, the ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity *Synplify* and Mentor Graphics *LeonardoSpectrum*. You can synthesize your Verilog or VHDL design as a stand-alone process by choosing the synthesis tool from the Lattice Semiconductor program group in your Start menu, or you can synthesize automatically and seamlessly within the Project Navigator.

The ispLEVER Tutorial manual contains synthesis design flow tutorials and is the best place to start if you want to get some hands-on experience. By working through the tutorial lessons, you'll learn how to create sample design projects with some of ispLEVER's most useful and powerful features.

For additional information about synthesis using LeonardoSpectrum or Synplify, see the ispLEVER Third-Party Manuals.

Synthesis Design Flows

You can run the synthesis tools from within the integrated ispLEVER environment, or as a stand-alone process. Whether using LeonardoSpectrum or Synplify, the high-level design flows are basically the same.

Integrated Flow

This approach lets you create, synthesize, import, and implement a design targeting one of the Lattice devices completely from within the ispLEVER Project Navigator environment.

1. Using the Project Navigator, create a new HDL project.
2. Target a device.
3. Using the Text Editor, create the HDL modules.
4. For mixed-mode designs, use the Schematic Editor to create the schematic files.
5. Using the Project Navigator, import the source files.
6. Select a synthesis tool.
7. Fit (Place and Route) the design.

Stand-alone Flow

The stand-alone approach requires you to create or load a VHDL or Verilog HDL design into the synthesis tool environment. Then you synthesize the design and generate an EDIF netlist that you imported into ispLEVER for implementing into a Lattice device.

1. Using your synthesis tool, create a project.
2. Target a device.
3. Load the source files.
4. Synthesize the design to create an EDIF file.
5. Using the ispLEVER Project Navigator, create an EDIF project.
6. Target a device (same as step 2).
7. Import the EDIF source file.
8. Fit (Place and Route) the design.

Integrated Third-Party Tools

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment.

LeonardoSpectrum

Within the LeonardoSpectrum synthesis environment, you can create Lattice device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process.

LeonardoSpectrum from Mentor Graphics combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features.

Synplify

The Synplify solution from Synplicity is a high-performance, sophisticated logic synthesis engine that utilizes proprietary technology to deliver fast, highly efficient FPGA and CPLD designs. Synplify uses Verilog and VHDL Hardware Description Languages as input, and outputs an optimized netlist for the Lattice device.

Selecting the Synthesis Tool

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment. This integrated approach lets you create, synthesize, import, and implement a design targeting a Lattice device completely from within the ispLEVER Project Navigator environment.

To specify the synthesis tool that the ispLEVER software will use:

1. In the Project Navigator, choose **Options > Select RTL Synthesis** to open the dialog box.
2. Select the synthesis tool that you want to use. This tool will be associated with all devices in the current device family. You can also make a synthesis tool the default for all device families.

Building the Database

Building a Lattice Internal Database for an ispXPGA Design

After you have created a project and imported the design files, the next step is to build an internal database. The input for this database is an EDIF file, which can be generated from a stand-alone synthesis tool and imported into the Project Navigator as your design, or created automatically from a Verilog or VHDL design using the integrated synthesis flow within the ispLEVER software.

When you run the Build Database process, the ispLEVER software converts the EDIF file to a Lattice internal database (`*.ld1`). If the design utilizes parameterized modules and IP cores (Lattice modules and ispLeverCORE IP modules), or user firm macros, the cores are expanded in this process.

Setting Constraints

Setting and Editing Constraints

For ispGDX, CPLD, ispXPLD, and ispXPGA devices, the ispLEVER software supports setting and editing of constraints in these ways:

Constraint Editor

Many ispGDX, CPLD, ispXPLD, and ispXPGA constraints can be edited within the Constraint Editor. You can specify pin and node assignments, group assignments, resource reservations, power level settings, output slew-rates, and nodal constraints, as well as PLL and HSI attributes. Modifications to the constraint file are made via the function dialog boxes or directly in the appropriate spreadsheets.

To run the Constraint Editor:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window, double-click the **Constraint Editor** process.

   Any invalid attribute or incorrect assignment is displayed in red in the Constraint Editor constraint sheet as well as in all the dialog boxes of the Constraint Editor. All the default values are displayed in blue. However, you can change the system default colors by choosing View > Set Colors.

Optimization Constraint Editor

For most CPLD and ispXPLD devices, the Optimization Constraint Editor lets you specify the global constraints used in optimization. It reads the constraint file and displays the constraint settings in the Opt Global Constraints sheet. You can directly modify the optimization constraints in the sheet.

To edit global optimization constraints using the Optimization Constraint Editor:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window, double-click the **Optimization Constraint** process. The software runs the process and opens the Optimization Constraint Editor.
3. In the Opt Global Constraints sheet, double-click a table cell in the **Constraint Value** column.
4. Select a desired option from the list, or directly type your setting in the edit box.
5. Choose **File > Save** to save the edits to the project constraint file (`.lci`).

ispXPGA Floorplanner

For ispXPGA designs using the ispXPGA Floorplanner, you have the option of saving one or a combination of constraint types—PFU, group assignments, pin assignments, region assignments—to the design's constraint file (`.lct`) using teh File > Save Constraints command. The software overwrites the current LCT file with the new constraints and clears the checkmarks from the processes displayed in the Project Navigator Processes window. The software will apply the changes to the physical design file (`.ld2` or `.ld3`) the next time you run Pack & Place or Route.

To save constraints in the ispFloorplanner:

1. Choose **File > Save Constraints**.
2. In the Save Constraints dialog box, select **Region**, **Pin Assignments**, and/or **PFU Packing/Placement**, depending on the types of changes you have made.
3. In the File name box, accept the default name and file type and click **Save**.

4. Click **Yes** to confirm that you want to replace the current LCT file. The Floorplanner saves the changes to the constraint file.

Packing and Placing

After an initial Lattice internal database is generated (`*.ld1`), the Pack & Place Design process packs the design instances or cells into Programmable Functional Unit (PFUs) and then places them on the ispXPGA device. The output of this process is the post-place design database (`*.ld2`).

The default constraints allow you to run Pack & Place Design for quick evaluation, whether the design fits in the target ispXPGA or not. To obtain better results, you can modify the Pack & Place Design constraints by using the Global Constraints Sheet of the Constraint Editor.

Keeping Track of Processes

The Project Navigator automatically keeps track of your design's processes for you. For example, it knows which processes should be run for a targeted device, a selected source, or for the entire design. Also, you can choose to run any process step, and the Project Navigator will run all other processes required to complete that selected step without running further, unnecessary steps.

The Project Navigator lists all processes for a selected source in the Processes window. Device-related processes are shown in the Processes window after you select a target device and highlight it in the Sources window.

The Pack/Place Report

This process opens the Pack/Place Report (plain text format) in the Project Navigator's Report Viewer. The report gives details about the design's packing and placement before routing and includes the following sections.

Pack and Place Options

Lists the options that were passed from the LCT file to pack and place.

Design Summary

Summarizes the design statistics.

Design Resource Summary

Summarizes the resources of the device and utilization of those resources.

PLL Summary

Summarizes the configuration of the sysCLOCK PLLs.

HSI Summary

Summarizes the configuration of the sysHSI blocks.

EBR Summary (Embedded Block RAMS)

Summarizes the configuration of the sysMEM blocks.

Input Signal List

Shows how the inputs of the design were implemented. The Input Signal List also includes an Input Signal List Cross-reference for long signal names that have been replaced with a number.

Differential Input Signal Listing

Shows how the differential inputs of the design were implemented. The Differential Input Signal List is sorted by signal name, in descending numbers for busses, with the paired signals placed together in the list. The list includes a Differential Input Signal List Cross-reference for long signal names that have been replaced with a number.

Output Signal List

Shows how the outputs of the design were implemented. The Output Signal List is sorted by signal name and includes an Output Signal List Cross-reference for long signal names that have been replaced with a number.

Differential Output Signal Listing

Shows how the differential inputs of the design were implemented. The list is sorted by signal name, in descending numbers for busses, with the paired signals placed together in the list. The list includes a Differential Output Signal List Cross-reference for long signal names that have been replaced with a number.

Bi-directional Signal List

Shows how the bi-directional signals of the design were implemented. The list is sorted by signal name and includes a Bi-directional Signal List Cross-reference for long signal names that have been replaced with a number.

Differential Bi-directional Signal Listing

Shows how the differential inputs of the design were implemented. The list is sorted by signal name, in descending numbers for busses, with the paired signals placed together in the list. The list includes a Differential Bi-directional Signal List Cross-reference for long signal names that have been replaced with a number.

Internal Signal List

Shows how the internal nodes of the design were implemented. The list is sorted by signal name and includes an Internal Signal List Cross-reference for long signal names that have been replaced with a number.

FanIn/FanOut List

Shows the signals that are used to create other signals in the design. The list is sorted by signal name and includes a FanIn/FanOut List Cross-reference for long signal names that have been replaced with a number.

Device Pin-out List

Lists all pins of the device, sorted by pin number. For each pin, the list shows pin type and corresponding PIO where applicable. An asterisk in the Assigned Pin section indicates a pin that is user-assigned. The IO_TYPES and Signal name columns indicate the I/O standard and signal name associated with the pin.

Signal Cross-reference

Lists any signals that had name changes when processed by the Lattice tools. This does not include long signal names that were placed with numbers.

Symbol Cross-reference

Lists any symbols that had name changes when processed by the Lattice tools. This does not include long signal names that were placed with numbers

Removed Logic

Describes all logic that was removed from the design during the Pack and Place process.

Added Logic

Describes any logic added to the design by the Lattice tools to improve implementation results.

Compilation Time

Describes the processor time to complete the process.

The HTML Pack/Place Report

This process opens the Pack/Place Report (HTML format) using the default web-browser tool.

Post-Place Pinouts

This process opens the Constraint Editor to display pin assignments made by the placer.

Post-Place Design Floorplan

This process opens the Floorplanner with the placed design netlist. There is no routing information here. The process gives you control over the placement of PFUs before routing, allowing you to optimize performance. For example, you can select an individual PFU and move it closer to the PIO of a critical path.

You can also select a region by dragging the pointer around a group of PFUs. This is especially useful for developing macros and intellectual property. You can then adjust the aspect ratio of your design.

Use the Post-Place Design Floorplan and the Post-Place timing report to optimize performance prior to routing.

***Note**: For designs that use Carry-Chains cells (CCU_*), moving the PFUs may result in Placer error.*

The Post-Place Timing Report

This process opens the timing report and lets you review pre-route timing. The report provides a best-case estimate (in most cases, best-case timing) before the time-consuming Route Design process. It gives details about the specific static timing information for each path in the design and indicates whether the timing constraints were met. Use both the Post-Place Design Floorplan and the Post-Place timing report to optimize the performance prior to Routing.

Interactive Editing

The Lattice ispXPGA process flow supports interactive editing with two applications: the Constraint Editor and the Floorplanner.

The Constraint Editor

The Constraint Editor lets you specify or change pin and node assignments, group assignments, pin reservations, power level settings, and output slew-rates. It reads the constraint file and displays the constraint settings. Modifications to the constraint file are made via the function dialog boxes. Most of the attributes can be modified directly in the sheet. Pin assignments can be set in the Package View with drag-and-drop functionality.

The Floorplanner

The Floorplanner provides a graphical interface for managing ispXPGA device real estate. The Floorplanner can shorten design turn around time and achieve a design that conforms to critical circuit performance requirements.

With the Floorplanner, you can:

- Obtain a graphical display of information provided in the Pack/Place Report and the Route Report.
- Customize placement and routing of your design.
- Use the Floorplanner in Reentrant Flow to help you meet timing requirements and reduce channel congestion. You view the timing results of your design with the Performance Analyst, set the constraints in the Floorplanner, repack and place your design, and check for improved timing results using the Performance Analyst.
- Create Firm Macros and Intellectual Property (IP) that can be instantiated into other designs.
- Use the Timing Widget to perform static timing analysis and display timing paths in the Floorplan View.

Routing

The ispLEVER software routes the design in the ispXPGA after placement. The routing algorithm takes full advantage of the Lattice ispXPGA architecture to achieve maximum performance. It uses congestion-driven routing to achieve a fit with minimal congestion, and it uses timing-driven routing to achieve maximum performance.

In timing-driven routing, the router determines the critical path and optimizes it during the routing process. You have the option of specifying a path for the router to concentrate on. After routing, you can examine the route report for errors and use the post-route design floorplan to route the design incrementally.

The Route Report

This process displays the Routing Report, which shows how a design was routed in the target device and informs you of the success or failure of the route. If the route was not successful, you can use the Post-Route Design Floorplan and the Query Widget to determine where the unrouted nets are.

Post-Route Design Floorplan

This process opens the Floorplanner with the routed design netlist. Note that if a design failed to successfully complete the routing (i.e. the report specifies unrouted nets), you can still open the Floorplanner to determine the location of the unrouted nets. This capability lets you modify the design to achieve successful routing. Each time you move a PFU or group of PFU's, you unroute the nets, which are then displayed in red. To route these broken nets incrementally within Floorplanner, choose **Edit > Route**.

To route the design globally, use the Project Navigator.

Design Verification

Verifying Designs

The ispLEVER software supports two types of timing verification: *static timing analysis* and *dynamic timing simulation*. Both of these methods support all Lattice devices.

Static Timing Analysis

Static timing analysis (timing analysis) is the process of verifying 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/hold time requirements, and the maximum frequency. Lattice has two static timing analysis tools, Performance Analyst and TRACE (for FPGAs).

The primary advantage of timing analysis is that it can be run at any time and requires no input test vectors, which can be very time consuming and tedious to create. Another major advantage of static timing analysis is that it exhaustively checks every possible input-to-output path. One shortcoming of all static timing analysis tools is that they detect false paths that will never be exercised during the course of normal operation of a circuit so that you could spend a lot time instructing the analyzer to ignore those paths. In this process, you could accidentally ignore a real issue. Although timing analysis does not give you a complete timing picture, it is an excellent way to quickly verify the speed of critical paths and identify performance bottlenecks.

Dynamic Timing Simulation

This type of analysis is based on an event-driven simulator and requires you to specify a test vector (waveform). Whereas timing analysis returns partial timing information, dynamic timing simulation (timing simulation) will give you detailed information about gate delays and worst-case circuit conditions. Because total delay of a complete circuit will depend on the number of gates the signal sees and on the way the gates have been placed in the device, timing simulation can only be run after the design has been implemented. Timing simulation also requires several input files to run.

There are two basic types of dynamic timing simulation tools, logic simulators (e.g, Lattice's Logic Simulator) and dynamic simulation analyzers (e.g., MTI's ModelSim™). Logic simulators function in a single delay mode, whereas dynamic simulation analyzers will simulate the ambiguity in delay pairs. Dynamic simulation analyzers typically take longer to process simulation results than logic simulators and considerably longer than static timing analysis tools.

Timing Verification Tools

The ispLEVER software offers timing verification with the following tools:

- Performance Analyst — Static timing analysis tool that runs timing analysis (All CPLD, ispXPLD, ispXPGA, and ispGDX2 devices except ispLSI 1K and 2K).
- Lattice Logic Simulator — Logic simulator that runs timing simulation (ispGDX and CPLD devices only).
- ModelSim for Lattice — Dynamic timing analyzer that runs timing simulation (all devices).
- TRACE — The *Timing Reporter and Circuit Evaluator* (TRACE) is an integrated flow tool that provides static timing analysis based on timing preferences (for FPGA devices only).

Additionally, timing simulation for all devices is supported with these tools:

- Text Editor — Used to create test stimulus files
- Waveform Editor — Used to create test stimulus files graphically

- Waveform Viewer — Used to view the results of simulation

Verification Environments

The timing verification tools operate in both integrated and stand-alone environments.

Integrated Timing Analysis

The Performance Analyst is a static timing analysis tool that lets you quickly determine the performance of designs implemented in any Lattice Semiconductor device. To run timing analysis, launch the Performance Analysis from the Project Navigator. The Performance Analyst traces each logical path in the design and calculates the path delays using the device's timing model and worst-case AC specs supplied in the device data sheet.

The timing analysis results are displayed in a graphical spreadsheet with source signals displayed on the vertical axis and destination signals displayed on the horizontal axis. The worst-case delay value is displayed in a spreadsheet cell if there is at least one delay path between the source and destination. To more easily identify performance bottlenecks, you can double-click a cell to view the path delay details.

Integrated Timing Simulation

To verify a design inside the current project, the ispLEVER software provides integrated verification with the Lattice Logic Simulator and ModelSim™ for Lattice from Mentor Graphics®. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the verification test bench, the following processes are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

CPLD and GDX Project Navigator Process	Simulation Tool Invoked
Timing Simulation	Lattice Logic Simulator
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

Stand-alone Simulation

The ispLEVER software supports stand-alone timing simulation. This provides an easy entry if you need to verify a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator Tools menu.

Required Files for Verification

ModelSim supports Lattice ispXPGA and FPGA device timing simulation. In addition to your design files, you will need at least one test stimulus file, a netlist file, and a timing delay file.

	ModelSim
Test Stimulus File Formats	
• **.tf** (Verilog test fixture)	X
• **.vhd** (VHDL test bench)	X
Netlist File Formats	
• **.vo** (Verilog netlist)	X
• **.vho** (VHDL netlist)	X
Delay File Formats	
• **.sdf** (Standard Delay File)	X

Verification File Descriptions

Test Stimulus Files

- Verilog Test Fixtures (`.tf`) — A Verilog test stimulus file that specifies the input waveforms for simulation in ASCII format.
- VHDL Testbench (`.vhd`) — A VHDL test stimulus file that specifies the input waveforms for simulation in ASCII format.

Netlist Files

- Verilog Netlist (`.vo`) — For Verilog designs, the back-annotated timing simulation netlist named `<design_name>.vo`.
- VHDL Netlist (`.vho`) — For VHDL designs, the back-annotated timing simulation netlist named `<design_name>.vho`.

Timing Delay Files

- Standard Delay Format (`.sdf`) — A file containing delay and timing constraint data for cell instances named `<design_name>.sdf`.

Generating Timing Simulation Files

After you have fit the design, the ispLEVER software lets you export the netlist and delays for timing simulation. For netlist files, ispLEVER supports VHDL, EDIF, and Verilog formats. For timing delay files, ispLEVER supports the standard SDF and Viewlogic DTB timing formats.

To choose a simulation file format:

1. In the Project Navigator, choose **Tools > Generate Timing Simulation Options** to open the dialog box.
2. Select the format options that you want.
   - For netlist formats, you have a choice of Verilog, VHDL, or EDIF (Version 2.0.0). If you choose the EDIF format, you can customize the Power/Ground representation by selecting either Cell or Net.
   - For timing format, SDF (version 2.1) and Viewlogic DTB format are supported.
3. Click **OK** to close the dialog.
4. When you run the **Generate Timing Simulation Files** process, ispLEVER generates the files in the specified formats.

Viewing the Simulation Input Files

You can use the Report Viewer to view simulation input files.

1. In the Project Navigator Sources window, select the target device.
2. In the Project Navigator Processes window, double-click **Report File**. Notice the two output files (Netlist and Delay) listed at the top of the report. The ispLEVER software generates these files and places them in the project directory.
3. To view these files, choose **File** > **View** and select the file that you want to view.

***Note**: You cannot modify files using the Report Viewer. You can only view them.*

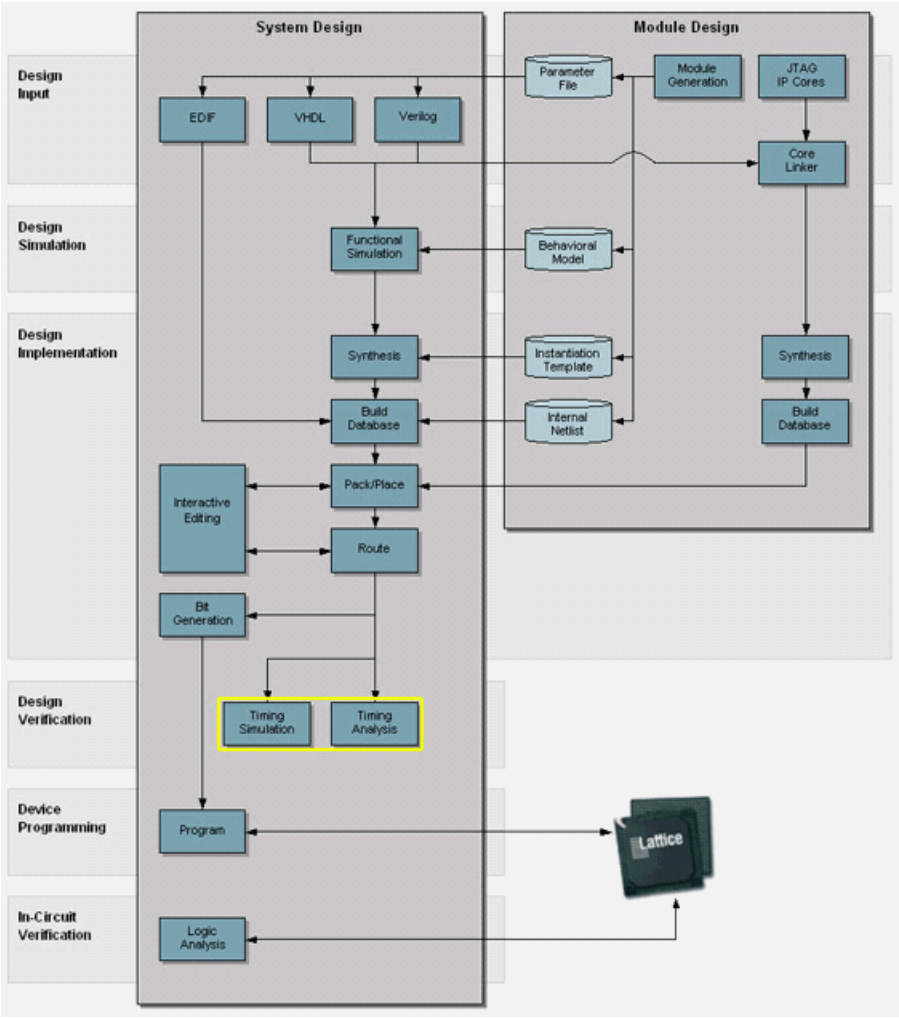
ispXPGA Verification Summary

The ModelSim simulator supports ispXPGA timing simulation for EDIF and HDL design entry methods and requires at least one Verilog test fixture (`*.tf`) or VHDL testbench (`*.vhd`) stimulus file. Additionally, ModelSim requires a netlist file (`*.vo` or `*.vho`) and a timing delay file (`*.sdf`).

	ModelSim	
	.tf	.vhd
	.vo + .sdf	.vho + .sdf
EDIF	X	X
Verilog	X	
VHDL		X

ispXPGA Verification Process Flow

The figure below shows timing analysis and simulation within the ispXPGA process flow.



Device Programming

Programming Devices

Lattice supports device programming for all programmable logic devices with the following tools, which are briefly described below and covered in detail in their respective Help. They are listed in alphabetical order.

ispVM System

The ispVM™ System software (ispVM) supports both serial and concurrent (turbo) programming of all Lattice devices in a PC environment. The ispVM System software is built around a graphical user interface. Device chains can be scanned automatically. Any required JEDEC ISC or Bitstream data files are selected by browsing with a built-in file manager. Non-Lattice devices that are compliant with IEEE 1149.1 can be bypassed once their instruction register length is defined in the chain description. Programmable devices from other vendors can be programmed through the vendor-supplied SVF file. See the ispVM System Help for more information about this tool.

Model 300 Programmer

The ISP Engineering Kit Model 300 programmer is an engineering device programmer that supports prototype development by allowing single-device programming directly from a PC. The Model 300 programmer supports all JTAG devices produced by Lattice, with device Vcc of 1.8, 2.5, 3.3, and 5.0V. See the Model 300 Programmer Help for more information about this tool.

SVF Debugger

The SVF Debugger can be used with ispVM System software to help you debug a Serial Vector Format (SVF) file. The SVF Debugger software allows you to program a device, and then edit, check syntax, debug and trace the process of an SVF file. See the SVF Debugger Help for more information about this tool.

Universal File Writer

The Universal File Writer (UFW) is a separate application that generates bitstream files or an SVF data file for a single device. Using JEDEC or ISC files, the software generates bitstream PCM, Intel Hex and Motorola Hex data files concurrently. It can also generate an SVF file using the parameters you select. You can run the Universal File Writer from the ispVM System toolbar or separately. See the Universal File Writer Help for more information about this tool.

In-Circuit Verification

Debugging and Verifying In-Circuit

Real-time access to internal signals makes verifying chip functionality and timing much easier, especially for leadless packages. Lattice supports in-circuit verification for all ispXPGA devices with the following tools, which are briefly described below and covered in detail in their respective Help.

ispTRACY IP Manager

The ispTRACY tool helps you verify and debug your ispXPGA circuitry by measuring the logic behavior of internal nodes inside the devices. The ispTRACY tool analyzes the behavior of internal nodes that are captured into the trace memory that is implemented by the ispXPGA internal block RAMs. The behavior of these internal nodes can then be written out through the ispXPGA JTAG port when certain trigger conditions are matched.

The RTL templates and source code of ispTRACY IP cores for ispLA, ispJTAG and JTAG_PORT are created through ispTRACY IP Manager. The easy-to-use GUI allows user to specify the IP core parameters to customize the IP core. The IP source code is not pre-compiled but is generated dynamically using the ispTRACY Core Compiler engine. The RTL source code with pre-defined black box instances can be encrypted in order to protect the IP property.

See the ispTRACY IP Manager Help for more information about this tool.

ispTRACY Core Linker

As an integral component of ispTRACY IP Manager, the ispTRACY Core Linker links, or instantiates, the ispTRACY cores with user design. The ispTRACY IP cores can be integrated with user design using RTL instantiation, EDIF merge, or Firm IP insertion.

See the ispTRACY IP Manager Help for more information about this tool.

ispTRACY Logic Analyzer

The ispTRACY Logic Analyzer allows you to set trigger configurations and extract ispTRACY information from programmed device through the JTAG ports. The intuitive GUI makes user to trace the signal/bus in waveform viewer much easily. Also the importing and exporting capabilities are available to communicate with external Logic Analyzer. See the ispTRACY Logic Analyzer Help for more information about this tool.

Running ispLEVER from the Command Line

Running from the Command Line

You can run the ispLEVER software from the command line on PC and UNIX using **ispflow** command line software. The ispflow software will attempt to fit the design to the specified part.

***Note**: All examples are shown in PC format. For UNIX, use forward slash instead of back slash.*

Syntax

The format of the command is as follows (on one line):

```
ispflow [-i <design>] [-d <device>] [-imp <yes\no>]
[-sdf <edif\verilog\vhdl\off>]
[-edf <mentor\synopsys\synplicity\viewlogic>]
[-syn <spectrum\synplify>] [-h] [-v]
```

where [ ] denotes optional parameters.

Definitions

-i <design name>	EDIF, Verilog, or VHDL design name. The name must include the appropriate file extension in the design name, such as `.edf`, `.v`, or `.vhd`.
-d <device name>	Specifies the device part number that the design will be fitted to. For example, `LFX125B-04F256C`. This option is required if the `<design name>.lci` file does not exist. After running **ispflow**, the specified device will appear in the newly created LCI file.
-imp <yes/no>	Import source constraints. Default is Yes.
-sdf <edif>	Outputs an SDF file in EDIF format.
-sdf <verilog>	Outputs an SDF file in Verilog format.
-sdf <vhdl>	Outputs an SDF file in VHDL format. Default.
-sdf <off>	Switch off SDF output.
-edf <mentor>	Specifies a Mentor Graphics-generated EDIF file. Default.
-edf <synopsys>	Specifies a Synopsys-generated EDIF file.
-edf <synplicity>	Specifies a Synplicity-generated EDIF file.
-edf <viewlogic>	Specifies a Viewlogic-generated EDIF file.
-syn <spectrum>	Specifies a LeonardoSpectrum synthesize file. Default.
-syn <synplify>	Specifies a Synplify synthesize file.
-spd <yes/no/fmax>	Speed: yes (speed), no (area), fmax.
-mpts	Max_PTerm_Split value.
-mptc	Max_PTerm_Collapse value.
-mptl	Max_PTerm_limit value.
-mfan	Max_fanin value.
-msym	Max_symbols value.
-fmll	Fmax_Logic_Level value.
-svf <on/off>	Switch on SVF generation. Default is off.
-r <*.lct/*.lci>	Refit using the constraint file.

-c <yes/no>	Specifies to run vcick (vc checker). Default is no.
-h	Help.
-v	Displays version number.

The input source file switch (**-i**) is mandatory. All other switches are not.

If a device name (**-d** <device name>) is specified from the command line, and a Lattice Constraint File (.lci) file exists, the device specified from the command line takes precedence. The **ispflow** software will not run if a device name is not specified from either the command line or in a LCI file.

Specifying Options and Pin Assignments

To specify optimization options and pin assignments, you must create or modify a <design_name>.lci file. If there is no LCI file, the software creates a default file for the specified device. In this case, the **-d** option is required.

***Note 1**: The LCI file is case-sensitive. Once an LCI file is created, the –d option can be omitted from the command line, because the device information will be obtained from the LCI file. The –d option takes precedence over the LCI file. If the –d option is used again with a different device part number, the device information part of the LCI file will be updated to reflect the changes.*

***Note 2**: Pin assignments and certain optimization options in the LCI file could be incorrect if the device is changed on the Command Line.*

The ispflow software runs through the complete flow, including timing analysis.

Input Formats

Acceptable input formats are non-hierarchical EDIF, VHDL, and Verilog HDL source files. The source files must be named with the .edf, .vhd, or .v extensions respectively.

Log Files

A log file of the process will be generated named *<design_name>*.batch.log file.

Batch Mode Example

The following example is for fitting a new design.

***Note**: The example is shown in PC format. For UNIX, use forward slash instead of back slash.*

To fit a new design:

1. Create a new directory and copy the input files needed.

   <isptools>\ispcpld\examples\ispxpga\edif\design\traffic\traffic.edf

   <isptools>\ispcpld\examples\ispxpga\edif\design\traffic\traffic.lci

2. Run ispflow -i traffic.edf.

   If you have a Lattice Constraint File (*<design_name>*.lci), the software will get the device from the LCI file and fit the design. The LCI file can be varied to an optimizer setting that you prefer. See Lattice Constraint File Description for additional information.

OR

3. If you do not have a LCI file, run ispflow -i traffic.edf -d LFX1200B-04F900C.

The software fits the design into the specified ispXPGA device.

Retaining Pin Assignments Using Batch Mode

You can retain pin assignments by copying LOCATION ASSIGNMENTS from your output Lattice Constraint File (`.lco`) into your input Lattice Constraint File (`.lci`).

To retain pin assignments using batch mode:

1. In the project directory, using a text editor, open the *<design_name>*`.lco` file.
2. In the LCO file, copy the LOCATION ASSIGNMENTS section.
3. In the project directory, using a text editor, open the *<design_name>*`.lci` file.
4. Replace the LOCATION ASSIGNMENTS section in the LCI file with the LOCATION ASSIGNMENTS section you copied from the LCO file.

LCI Files

The Lattice Constraint File (`.lci`) contains the constraints for Part selection as well as the Optimization and Placing and Routing processes.

You do not need to generate a LCI file on the first fit. If you specify a device (`-d`) with ispflow, this will generate a LCI file.

Command Line FAQs

The following are Frequently Asked Questions about processing designs in Command Line Mode using ispflow software.

Q. What report files are available to view after processing a design using Command Line mode?

A. Using a text editor, you can view the Timing Report File (`.trp`) and the Fitting Report File (`.rpt`) in the project directory after command line batch mode processing.

The RPT file displays details about how the current design was fit into the target device. If a fit was not accessible, the RPT file explains the problems preventing a fit. The file is located in your project directory, and is named *<design_name>*`.rpt`

The TRP file contains all information for stamp model generator. The file is located in your project directory, and is named *<design_name>*`.trp`.

Q. Where do I find my programming (JEDEC) file?

A. The programming file can be found in the project directory after command line batch mode processing. The file is located in your project directory, and is named *<design_name>*`.jed`.

Q. Where do I find my back-annotated timing simulation netlist?

A. The back-annotated timing simulation netlist is located in the project directory. The extension of the netlist depends upon the source files used to process your design.

- For VHDL designs, the back-annotated timing simulation netlist is named *<design_name>*`.vho`.
- For Verilog designs, the back-annotated timing simulation netlist is named *<design_name>*`.vo`.
- For EDIF designs, the back-annotated timing simulation netlist is named *<design_name>*`.edo`.

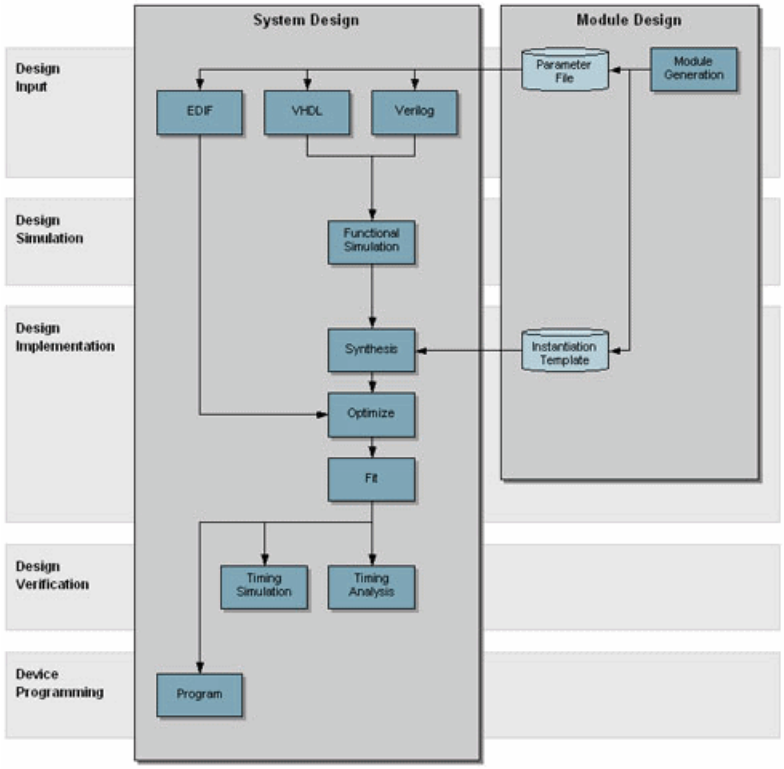
Q. Where do I find my timing delay file?

A. The timing delay file can be found in the project directory after command line batch mode processing. The file is located in your project directory, and is named <*design_name*>.sdf.

CHAPTER 3 *ispXPLD Process Flow*

Introduction

ispXPLD Design



The ispXPLD™ 5000MX family represents a new class of devices from Lattice Semiconductor called eXpanded Programmable Logic Devices (XPLDs). These devices are built around a new building block, the Multi-Function Block (MFB). These blocks can be individually configured as SuperWIDE™ (136-input) logic, single- or dual-port memory, FIFO, or CAM depending on your application.

This unparalleled PLD flexibility is combined with sysIO™ interfaces for support of leading edge standards such as LVDS, HSTL, and SSTL, along with the more familiar LVCMOS standards. sysCLOCK™ PLLs allow easy clock management. ispXPLD 5000MX devices provide expanded in-system programming referred to as ispXP. As such, ispXPLD devices are non-volatile and infinitely reconfigurable. They can be programmed through an industry standard IEEE 1532 interface or reconfigured through the Lattice sysCONFIG™ microprocessor interface. Devices are available to support 3.3, 2.5, and 1.8-volt power supply operation.

Supported Device Families

- ispXPLD 5000MX

Overview of ispLEVER for ispXPLD

The ispLEVER program offers an integrated environment consisting of several tools necessary to implement Lattice ispXPLD devices. These tools are briefly described in the following paragraphs and covered in detail in their respective Help. They are listed in alphabetical order.

Constraint Editor

The Constraint Editor lets you specify pin and node location assignments, group assignments, I/O types settings, power level settings, resource reservations, PLL attributes, as well as output slew rates and JEDEC file options. The Constraint Editor reads the constraint file and displays the constraint settings. Modifications to the constraint file are made via the function dialogs. See the Constraint Editor Help for more information about this tool.

ispEXPLORER

The ispEXPLORER lets you run multiple passes of your design using different combinations of Fitter/Optimizer settings and critical timing constraints to achieve the best solution. Results are summarized in a single spreadsheet and detailed reports for each run are accessible. See the ispEXPLORER Help for more information about this tool.

LeonardoSpectrum for Lattice

The LeonardoSpectrum™ synthesis environment from Mentor Graphics® lets you can create Lattice CPLD device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process. LeonardoSpectrum combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features. See the LeonardoSpectrum for Lattice documentation supplied by the manufacturer for more information about this tool.

ModelSim for Lattice

The ispLEVER software supports third-party HDL simulation and verification with ModelSim™ from Mentor Graphics®. Using this integrated package, you can simulate single Verilog or VHDL designs within one environment. See the ModelSim for Lattice documentation supplied by the manufacturer for more information about this tool.

Module/IP Manager

The Module/IP Manager enables you to generate cores (parameterized modules and IP cores) from templates supplied by Lattice Semiconductor and third-party vendors. After generating the module, the Module/IP Manager automatically stores the instantiation template and related files to your project folder. See the Module/IP Manager Help for more information about this tool.

Optimization Constraint Editor

The Optimization Constraint Editor lets you specify the global constraints used in optimization. It reads the constraint file and displays the constraint settings in the Opt Global Constraints sheet. You can directly modify the optimization constraints in the sheet. The Optimization Constraint Editor is opened prior to optimization, whereas the Constraint Editor is opened after optimization. See the Optimization Constraint Editor Help for more information about this tool.

Performance Analyst

The Performance Analyst analyzes the performance of your design after it has been optimized and implemented by the Fitter. See the Performance Analyst Help for more information about this tool.

Pin Migration Tool

The Pin Migration Tool is used to migrate pin assignment from an old device to a new one that uses the similar package. When you want to implement an old design onto a new device with the similar package, the tool will suggest alternative devices that are in similar pin packages, generate a new project targeted at the selected new device, and convert all compatible pin assignment of the old project to the new one. After the migration, all you have to do is to open the new project, re-assign any pins that are released, and re-compile the design. See the Pin Migration Tool Help for more information about this tool.

Project Navigator

The Project Navigator is the primary interface for ispLEVER and provides an integrated environment for managing the project elements and processes, as well as accessing all ispLEVER tools. See the Project Navigator Help for more information about this tool.

Report Viewer

You can use the Report Viewer to view, but not edit, the various report files generated by ispLEVER. See the Report Viewer Help for more information about this tool.

Synplify for Lattice

Synplify® for Lattice is a logic synthesis tool for CPLD devices, developed by Synplicity®. Synplify starts with high-level designs written in Verilog or VHDL hardware description languages (HDLs). Synplify then converts the HDL into small, high-performance, design netlists that are optimized for Lattice devices. See the Synplify for Lattice documentation supplied by the manufacturer for more information about this tool.

Tcl Editor

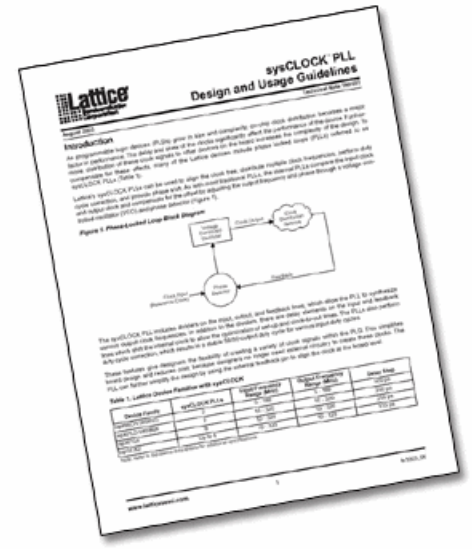
The Tcl Editor is a design tool that allows you to create, edit, and run a series of Tool Control Language (TCL) commands. These commands invoke individual ispLEVER processes for generating a complete programmable logic solution. The default text editor permits you to edit TCL code, and it highlights key TCL language elements in different colors to clarify the nature and use of each language element. You can generate a Tcl script for a project in Project Navigator, open it in the Tcl Editor, edit the script, and run it. The Tcl Editor, together with the robust features of the language, gives you more control over the design environment. See the Tcl Editor Help for more information about this tool.

Text Editor

The Text Editor is the ispLEVER text entry tool. You use this tool to create and edit text-based files, such as ABEL-HDL files, test stimulus files, and project documentation files. See the Text Editor Help for more information about this tool.

ispXPLD Application Notes

The Lattice Semiconductor web site lists several Application Notes for ispXPLD devices.


sysCLOCK PLL
Design and Usage Guidelines

Design Entry

Verilog HDL Design Entry

The ispLEVER software supports Verilog HDL, a hardware description language used to design and document electronic systems. Verilog HDL allows designers to design at various levels of abstraction.

All Lattice devices support Verilog HDL design.

Adding a Verilog HDL Module to Your Design

To add a Verilog HDL module to a design, you can either import a `.v` file, or create a new Verilog HDL module file with the Text Editor.

Creating a New Verilog HDL Module

You can use the Text Editor to create a new Verilog HDL module.

To create a new Verilog HDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. Select **Verilog Module** and click **OK**. The Text Editor window appears together with the New Verilog Module dialog box.
3. In the dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new Verilog HDL file appears in the Text Editor window.
5. Use the commands on the Edit menu to Cut, Copy, Paste, Find, or Replace text.

Synthesizing Your Verilog HDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor Graphics LeonardoSpectrum. You can synthesize your Verilog HDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

VHDL Design Entry

VHDL is a language for describing the structure and function of integrated circuits. VHDL allows you to:

- Describe the hierarchical structure and interconnect of a design.
- Specify the function of designs using familiar programming language forms.
- Simulate the design before being manufactured, so that design alternatives can be quickly compared and tested.

All Lattice devices support VHDL design.

Adding a VHDL Module to Your Design

To add a VHDL module to a design, you can either import a `.vhd` file, or create a new VHDL module file with the Text Editor.

Creating a New VHDL Module

You can use the Text Editor to create a new VHDL module.

To create a new VHDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. In the dialog box, select VHDL Module and click **OK**. The Text Editor window appears together with the New VHDL Source dialog box.
3. In the New VHDL Source dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new VHDL file appears in the Text Editor window.
5. Use the commands in the Edit menu to Cut, Copy, Paste, or Replace text.

Synthesizing Your VHDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor graphics LeonardoSpectrum. You can synthesize your VHDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

EDIF Design Entry

The Electronic Design Interchange Format (EDIF) is a format used to exchange design data between different ECAD systems.

The EDIF format is designed to be written and read by computer programs that are constituent parts of EDA systems or tools. Its syntax has been designed for easy machine parsing and is similar to LISP.

The ispLEVER software supports EDIF Version 2 0 0.

All Lattice devices support EDIF design entry.

Importing an EDIF Netlist

You can import a design netlist description into the ispLEVER software from a third-party synthesis or schematic tool if the design file is formatted as EDIF 2 0 0.

***Note**: The project that you are importing the netlist into must be an EDIF project.*

To import an EDIF netlist into your project:

1. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
2. Change Files of type to **EDIF Netlist** (`*.ed*`), and then select the EDIF file that you want to import.
3. Click **Open** to open the Import EDIF dialog box.
4. The default setting for power and ground in the ispLEVER software are the VCC and GND symbols. If you know that the EDIF generated by other tools uses a different convention, you can change it in the window. Select **Custom**. Select either **Symbol** or **Net** representation. Then type the new names for VCC and GND.
5. If you are following the recommendation from Lattice for generating the EDIF file from the supported third party design kit, select **CAE Vendors**. From the list, choose the vendor that generated the EDIF file: Mentor, Synopsys, Synplicity, or Viewlogic.
6. Click **OK**. The software adds the selected EDIF file to the project sources.

Translating EDIF Properties

By default, the ispLEVER software ignores EDIF properties. If you want the ispLEVER software to translate EDIF properties to design constraints for the Fitter, do the following:

1. In Project Navigator, choose **Tools > Import Source Constraint Option** to open the dialog box. The Import Source Constraints Option dialog box lets you import constraints, such as Location (pin/node) Assignments, Group Assignments, and Output Slew Rate, from source files (ABEL, schematic, or EDIF).
2. In the dialog box, select **Auto Import Source Constraints**.
3. Click **OK**.

When you select this option, the ispLEVER software displays a confirmation dialog box prior to implementing the function. This confirmation dialog box appears every time you run the Fit Design process, unless you select the Do Not Import Source Constraints option.

On the warning message dialog box, if you click **Yes**, the constraints from the source files are written into the project constraint file.

***Important**: Constraints from source files and existing constraints in the project constraint file are not merged; existing constraints are overridden by the new constraints.*

Existing constraints (only Location Assignments, Group Assignments, and Output Slew Rate are affected) in the project are cleared, regardless of constraints that might exist in the source file. If there are constraints in the source file, the new constraints are written into the project constraint file. If there are no constraints in the source file, no constraints are written into the file.

EDIF Properties

The ispLEVER software will take design-specific constraints from the properties in the EDIF netlist. The following is the list of properties that the Fitter supports.

PIN LOCATION Property

Name: LOC

Value: {PIN # }

Example: LOC = P20

Scope: IO PORT, net connect to the IO port.

GROUPING Property

Name: GROUPING

Value: GROUP NAME

Example: Use the following command to assign signal locations in your design. In this case, you have a list of internal nodes: a, b, and c, and you want to assign them into a group “mg.” The location of this group needs to be Block “A”, Segment “2”:

On net a, grouping = mg

On net a, loc = "A, 2"

On net b, grouping = mg

On net b, loc = "A, 2"

On net c, grouping = mg

On net c, loc = "A, 2"

OUTPUT SLEW Property

Name: SLEW

Value: {Fast, Slow}

Example: To set port A to high slew, put the following property on the net or port: SLEW=Fast

Scope: OUTPUT PORT/NET

SIGNAL OPTIMIZATION Property

Name: OPT

Value: {KEEP, COLLAPSE}

Scope: On any net of the design.

OPEN DRAIN Property

Name: OPENDRAIN

Value: {On/Off}

Example: To set port A to an open drain, put the following property on the net or port: OPENDRAIN=on

Scope: OUTPUT PORT/NET

PULL Property

Name: PULL

Value: {On/Off/Hold}

Example: To set port A to pull up, put the following property on the net or port: PULL=on

Scope: OUTPUT PORT/NET.

OUTPUT VOLTAGE Property

Name: VOLTAGE

Value: {VCC/VCCIO}

Example: To set port A to output voltage at VCCIO level, put the following property on the net or port: VOLTAGE=VCCIO

Scope: OUTPUT PORT/NET.

Module and IP Design Entry

The Module/IP Manager helps you build large designs using Lattice Parameterized Modules (modules) and intellectual property cores (IPs). Including a module is as easy as:

- Selecting a module or IP core from the module tree
- Specifying parameters in the configuration window
- Instantiating the module or IP core with your text editor.

The Module/IP Manager is fully integrated with every level of the ispLEVER software, simplifying module and IP management within your design project.

Two Approaches to Modules and IP Cores

The ispLEVER software lets you choose the most convenient method for placing parameterized modules and IP cores into your design database. In both methods the software produces the instantiation template and its associated files and saves all of them in the project directory.

GUI Approach

The Module/IP Manager graphic user interface (GUI) simplifies the process of selecting and configuring a module or IP core. The interface lists all available modules by type and displays the parameters in an easy-to-use dialog box. The GUI approach requires more time for editing a design than the PMI approach.

PMI Approach

The Parameterized Module Instantiation template (PMI), though more difficult to use than the GUI, gives you the flexibility of editing a module at any time without having to recreate it. Using the PMI template, you embed the core description of your module into your HDL source files.

Design Simulation

Running a functional simulation after a design description is complete allows you to verify that the description is functionally correct. Also, by simulating the functionality of your design *before* synthesis, you can find and correct basic design errors sooner. While functional simulation will verify your Boolean equations, it does not indicate timing problems.

The ispLEVER software supports functional simulation for Lattice Semiconductor CPLD and FPGA devices using the Lattice Logic Simulator or *ModelSim* from Mentor Graphics. The RTL design can be simulated for functionality before synthesis using the VHDL or Verilog design description and an input stimulus file.

Simulation Environments

The functional simulators operate in both integrated and stand-alone environments.

Integrated Simulation — To simulate a design inside the current project, the ispLEVER software provides integrated simulation. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the simulation source file, the processes in the tables below are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

For ModelSim, ispLEVER includes scripts that run functional simulation automatically using Lattice-defined settings and preferences. You can customize the run by creating a different script files (DO file), which is a simple script that contain commands that are equivalent to the ModelSim GUI commands. This macro is automatically called when you run ModelSim.

For information about creating your own ModelSim macros, see the *ModelSim User's Manual*, *Chapter 11 Tcl and Macros*, provided with your ispLEVER software (Third-Party Manuals).

CPLD and ispGDX Project Navigator Processes	Simulation Tool Invoked
Functional Simulation	Lattice Logic Simulator
Verilog Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Functional Simulation	ModelSim
Verilog Post-Route Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim
VHDL Post-Route Functional Simulation	ModelSim

Stand-alone Simulation — The ispLEVER software supports stand-alone functional simulation. This provides an easy entry if you need to simulate a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator **Tools** menu.

Design File Descriptions

Lattice Logic Simulator and ModelSim enable you to simulate the operation of your design in the following design entry formats:

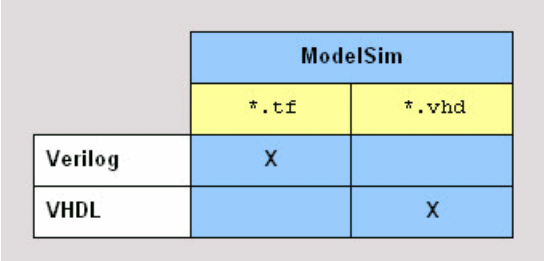
- ABEL-HDL format (*design*.abl) — a hierarchical logic description language that supports a variety of behavioral input forms, including high-level equations, state diagrams, and truth tables. (CPLD designs only)
- Schematic format (*design*.sch) — describes your circuit in terms of the components used and how they connect to each other. (CPLD designs only)
- VHDL format (*design*.vhd) — Very High Speed IC Hardware Description Language format.
- Verilog HDL format (*design*.v) — an industry-standard hardware description language used to describe the behavior of hardware that can be implemented directly by logic synthesis tools.

The Lattice Logic Simulator also supports mixed design entry as follows:

- Schematic and ABEL-HDL (CPLD designs only)
- Schematic and VHDL
- Schematic and Verilog HDL

ispXPLD Test Stimulus Files

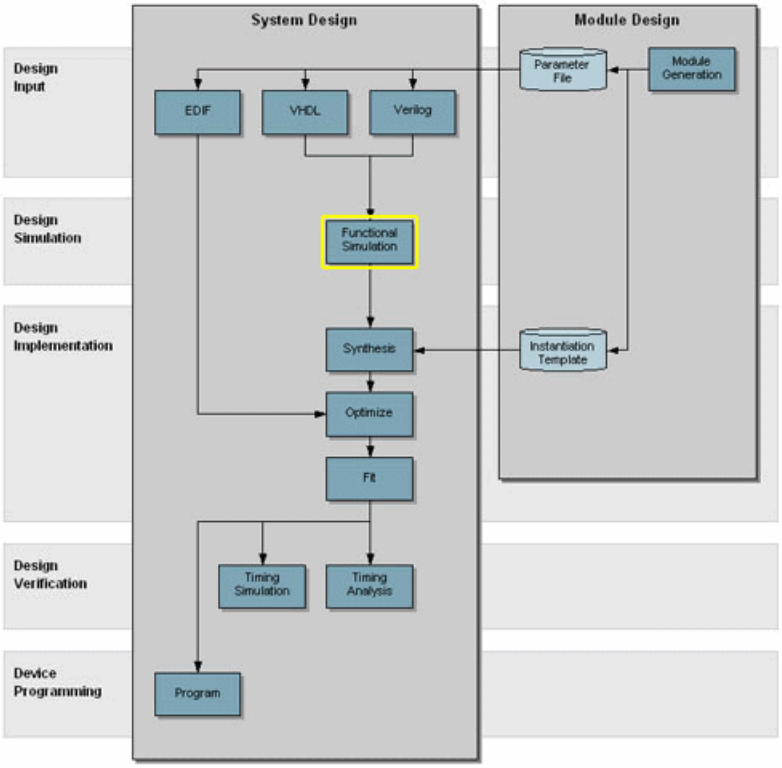
Once you have completed your design (or a module of the design), you can test it to confirm that it behaves the way you expect it to. The ModelSim simulator supports ispXPLD functional simulation for HDL design entry methods and requires at least one Verilog test fixture (*.tf) or VHDL test bench (*.vhd) stimulus file.



	ModelSim	
	*.tf	*.vhd
Verilog	X	
VHDL		X

ispXPLD Simulation Process Flow

The figure below shows functional simulation within the ispXPLD process flow.



Creating a Verilog Test Fixture from a Template

Verilog test stimulus can be specified either in the top-level HDL source or in a separate test fixture (`.tf`) file. You can create the test fixture manually using a text editor or use a Verilog Test Fixture template (`.tfi`) file.

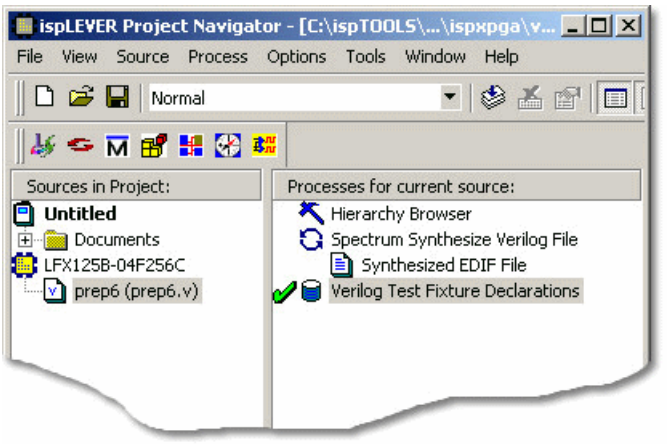
The easiest way to automatically create a Verilog Test Fixture template is using the Project Navigator Verilog Test Fixture Declarations process. After the test fixture template file (`.tfi`) is created, you must add your test vectors and rename it with the extension `.tf` before importing it into your design.

To automatically generate the Verilog test fixture template file and import it into your design:

1. Open your Verilog design in the Project Navigator.
2. In the Sources window, select the top-level Verilog design source (`*.v`) file.

> ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

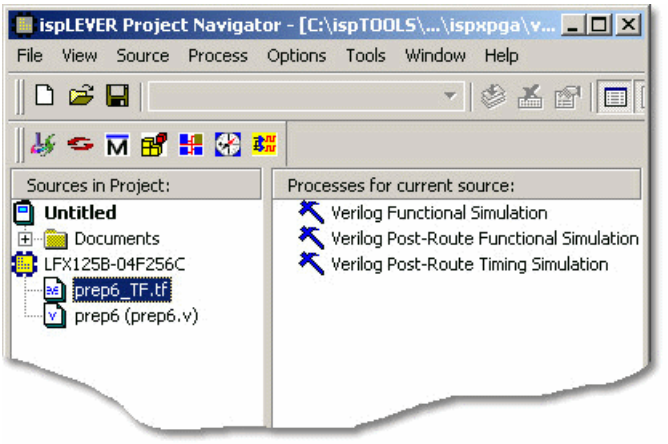
3. In the Processes window, double-click the **Verilog Test Fixture Declarations** process.



This process creates a template file for a Verilog Test fixture (`<verilog_sourcefile_name>.tfi`). However, in order to use this file as a test fixture in your design, you must edit it and rename it with the extension `.tf`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `prep6_TF.tf`. Add the "`TF`" to the name so that the file will not be overwritten. Change the file extension to "`.tf`" so that it can be imported into the project as a test fixture source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test fixture file. Click **Open**.
7. In the Associate Verilog Test Fixture dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a Verilog Test Fixture. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Creating a VHDL Test Bench from a Template

VHDL test stimulus can be specified either in the top-level HDL source or in a separate test bench (`.vhd`) file. You can create the test bench manually using a text editor or use a VHDL Test Bench template (`.vht`) file.

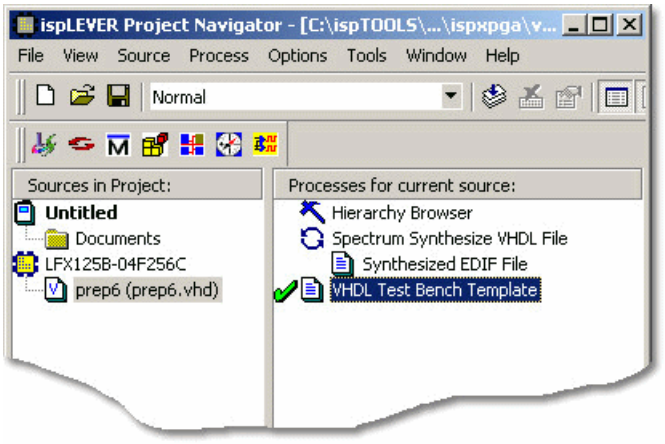
The easiest way to automatically create a VHDL Test Bench template is using the Project Navigator VHDL Test Bench Template process. After the test bench template file is created, you must add your test stimulus and rename it with the extension `.vhd` before importing it into your design.

To generate the VHDL test bench template and import it into your design:

1. Open your VHDL design in the Project Navigator.
2. In the Sources window, select the top-level VHDL design source (`*.vhd`) file.

   ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

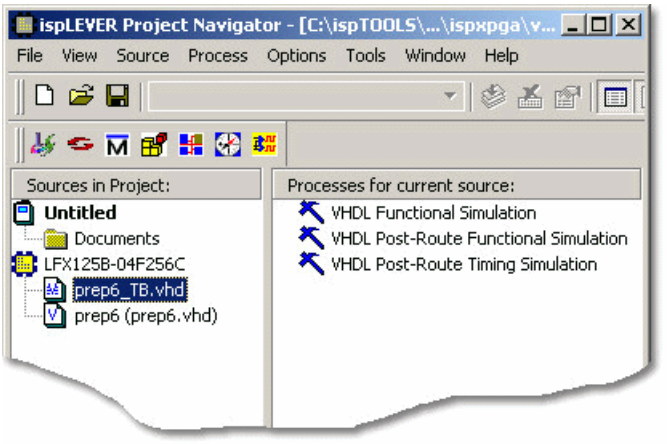
3. In the Processes window, double-click the **VHDL Test Bench Template** process.



   This process creates a template file for a VHDL Test Bench (`<vhdl_sourcefile_name>.vht`). However, to use this file as a test bench in your design, you must edit it and rename it with the extension `.vhd`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `prep6_TB.vhd`. Add the "`TB`" to the name so that the file will not be overwritten. Change the file extension to "`.vhd`" so that it can be imported into the project as a test bench source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test bench file. Click **Open**.
7. In the Import Source Type dialog box, select **VHDL Test Bench** and click **OK**.
8. In the Associate VHDL Test Bench dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a VHDL Test Bench. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Interfacing with ModelSim

ModelSim functional simulation generates several batch files, such as `.fdo`, `.udo`, and `.tdo`. ModelSim users will frequently take advantage of customizing batch files to control their simulation, for example specifying signals to display, run time, and waveform display options.

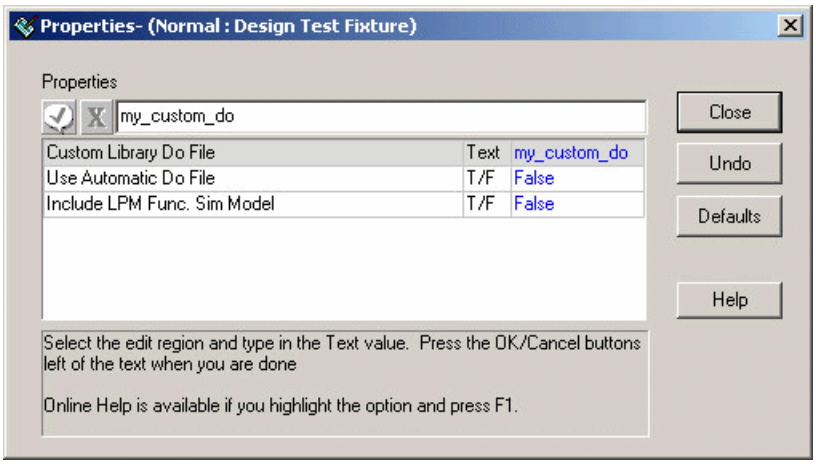
There are two options for creating custom DO files:

Option 1

- In the project folder, edit the `*.udo` file. The Project Navigator will not overwrite this user DO file.

Option 2

1. In the Project Navigator Sources window, select the test bench source.
2. In the Processes window, right-click the functional simulation process to open the Properties dialog box.
3. In the dialog:
   - Type a name for the file in the Custom Library Do File field and click the checkmark icon
   - Set Use Automatic Do File to **False**.
   - Click **Close**.



4. The software will automatically create this file when functional simulation is run. Edit this file as needed.

Design Implementation

Synthesizing

Synthesizing ispXPLD Designs

For Verilog and VHDL designs, the ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity *Synplify* and Mentor Graphics *LeonardoSpectrum*. You can synthesize your Verilog or VHDL design as a stand-alone process by choosing the synthesis tool from the Lattice Semiconductor program group in your Start menu, or you can synthesize automatically and seamlessly within the Project Navigator.

For Verilog and VHDL mixed-mode designs, the HDL portions are synthesized using the selected synthesis tool. For ABEL mixed-mode designs, both the ABEL and schematic portions are compiled. You can only do mixed-mode design using the Project Navigator integrated flow.

The ispLEVER Tutorials contain synthesis design flow tutorials and is the best place to start if you want to get some hands-on experience. By working through the tutorial lessons, you'll learn how to create sample design projects with some of ispLEVER's most useful and powerful features.

For additional information about synthesis using LeonardoSpectrum or Synplify, see the ispLEVER Third-Party Manuals.

Synthesis Design Flows

You can run the synthesis tools from within the integrated ispLEVER environment, or as a stand-alone process. Whether using LeonardoSpectrum or Synplify, the high-level design flows are basically the same.

Integrated Flow

This approach lets you create, synthesize, import, and implement a design targeting one of the Lattice devices completely from within the ispLEVER Project Navigator environment.

1. Using the Project Navigator, create a new HDL project.
2. Target a device.
3. Using the Text Editor, create the HDL modules.
4. For mixed-mode designs, use the Schematic Editor to create the schematic files.
5. Using the Project Navigator, import the source files.
6. Select a synthesis tool.
7. Fit (Place and Route) the design.

Stand-alone Flow

The stand-alone approach requires you to create or load a VHDL or Verilog HDL design into the synthesis tool environment. Then you synthesize the design and generate an EDIF netlist that you imported into ispLEVER for implementing into a Lattice device.

1. Using your synthesis tool, create a project.
2. Target a device.
3. Load the source files.
4. Synthesize the design to create an EDIF file.
5. Using the ispLEVER Project Navigator, create an EDIF project.

6. Target a device (same as step 2).
7. Import the EDIF source file.
8. Fit (Place and Route) the design.

Integrated Third-Party Tools

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment.

LeonardoSpectrum

Within the LeonardoSpectrum synthesis environment, you can create Lattice device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process.

LeonardoSpectrum from Mentor Graphics combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features.

Synplify

The Synplify solution from Synplicity is a high-performance, sophisticated logic synthesis engine that utilizes proprietary technology to deliver fast, highly efficient FPGA and CPLD designs. Synplify uses Verilog and VHDL Hardware Description Languages as input, and outputs an optimized netlist for the Lattice device.

Selecting the Synthesis Tool

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment. This integrated approach lets you create, synthesize, import, and implement a design targeting a Lattice device completely from within the ispLEVER Project Navigator environment.

To specify the synthesis tool that the ispLEVER software will use:

1. In the Project Navigator, choose **Options > Select RTL Synthesis** to open the dialog box.
2. Select the synthesis tool that you want to use. This tool will be associated with all devices in the current device family. You can also make a synthesis tool the default for all device families.

Setting Constraints

Setting and Editing Constraints

For ispGDX, CPLD, ispXPLD, and ispXPGA devices, the ispLEVER software supports setting and editing of constraints in these ways:

Constraint Editor

Many ispGDX, CPLD, ispXPLD, and ispXPGA constraints can be edited within the Constraint Editor. You can specify pin and node assignments, group assignments, resource reservations, power level settings, output slew-rates, and nodal constraints, as well as PLL and HSI attributes. Modifications to the constraint file are made via the function dialog boxes or directly in the appropriate spreadsheets.

To run the Constraint Editor:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window, double-click the **Constraint Editor** process.

   Any invalid attribute or incorrect assignment is displayed in red in the Constraint Editor constraint sheet as well as in all the dialog boxes of the Constraint Editor. All the default values are displayed in blue. However, you can change the system default colors by choosing View > Set Colors.

Optimization Constraint Editor

For most CPLD and ispXPLD devices, the Optimization Constraint Editor lets you specify the global constraints used in optimization. It reads the constraint file and displays the constraint settings in the Opt Global Constraints sheet. You can directly modify the optimization constraints in the sheet.

To edit global optimization constraints using the Optimization Constraint Editor:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window, double-click the **Optimization Constraint** process. The software runs the process and opens the Optimization Constraint Editor.
3. In the Opt Global Constraints sheet, double-click a table cell in the **Constraint Value** column.
4. Select a desired option from the list, or directly type your setting in the edit box.
5. Choose **File > Save** to save the edits to the project constraint file (`.lci`).

ispXPGA Floorplanner

For ispXPGA designs using the ispXPGA Floorplanner, you have the option of saving one or a combination of constraint types—PFU, group assignments, pin assignments, region assignments—to the design's constraint file (`.lct`) using teh File > Save Constraints command. The software overwrites the current LCT file with the new constraints and clears the checkmarks from the processes displayed in the Project Navigator Processes window. The software will apply the changes to the physical design file (`.ld2` or `.ld3`) the next time you run Pack & Place or Route.

To save constraints in the ispFloorplanner:

1. Choose **File > Save Constraints**.
2. In the Save Constraints dialog box, select **Region**, **Pin Assignments**, and/or **PFU Packing/Placement**, depending on the types of changes you have made.
3. In the File name box, accept the default name and file type and click **Save**.
4. Click **Yes** to confirm that you want to replace the current LCT file. The Floorplanner saves the changes to the constraint file.

Compiling

The ispLEVER software accepts several design entry formats. With the exception of EDIF, all CPLD designs must be either synthesized or compiled before going to the Fitter.

For ABEL-HDL and Schematic designs, the compilation process is an integrated part of the ispLEVER process flow. When you compile a design, you are changing your design entry format into Boolean equations, which serve as input to simulation and device implementation programs. In general, compiling a design involves running every process after design entry. These processes include compiling and optimizing steps that can be performed on a single source or on the entire design.

Keeping Track of Processes

The Project Navigator automatically keeps track of your design's processes for you. For example, it knows which processes should be run for a targeted device, a selected source, or for the entire design. Also, you can choose to run any process step and the Project Navigator will run all other processes required to complete that selected step, but not run further, unnecessary steps.

The Project Navigator lists all processes for a selected source in the Processes window. Device-related processes, such as fitting the design, are shown in the Processes window after you select a target device, and highlight it in the Sources window.

Understanding the Compilation Process

The Project Navigator processes each logic module, schematic file, or EDIF netlist to obtain an intermediate file that can later be linked together before fitting the design into a Lattice device.

There are more processes required to compile a logic source than a schematic source, primarily because logic designs are language-based and are stored in ASCII format. This means that the ispLEVER software must check the language syntax and process equation statements that are within the logic file.

The processing steps required to compile a design are listed below in the order in which they run:

- Compile (for logic, schematic, EDIF, or test vector files)
- Check Syntax (for logic files)
- Compiler Listing (for logic files)
- Compiled Equations (for logic, schematic, and EDIF files)
- Signal Cross Reference (for EDIF files)

Compiling Logic, Schematic, or EDIF

This comprehensive process compiles a logic module, a schematic design file, or an EDIF netlist. Design compilation steps differ between source types, as described below:

Compile Logic (for logic sources)

- Checks for and flags syntax errors
- Converts state diagrams and truth tables into equations
- Expands macros
- Converts equations with sets to equations without sets
- Replaces all operators with equivalent operations using only NOTs, ANDs, ORs and XORs
- ORs together equations that cause multiple assignments to the same identifier
- Performs simple logic reduction
- Translates the equations into the OPEN-ABEL-2.0 file format

Compile Schematic (for schematic sources)

- Compiles the schematic to produce a BLIF format file, including any attributes or properties specified. Schematics compiled this way should use only the Device-Independent symbol library.

***Note**: You should run design rule checking (Schematic Editor: DRC > Consistency Check) before compiling the schematic.*

Compile EDIF (EDIF)

- Compiles the EDIF file to produce a BLIF format file, including any attributes or properties specified.

Check Syntax

Checks the syntax of a logic module. No compilation is run. If there are syntax errors, the errors can be viewed in the Process Log File. If you want to see the errors in a compiler listing report format, use the Compiler Listing process.

Compiler Listing

The Compiler Listing process gives a record of the compilation of your source file. The report shows your logic file by line numbers, with errors and warnings below the line on which they occurred.

Compiled Equations

This process shows the Boolean equations produced by the compiler. The equations are shown in sum-of-products form. Positive and reverse polarity equations are displayed, along with product term and fan-in/fan-out summaries for each signal.

Signal Cross-Reference (EDIF)

This option displays a cross-reference of the old names to the new names (converting long and hierarchical to mangled names). Names that are more than 32 characters long or that contain one or more of the characters ‘/’, ‘>’, or ‘@’ will not be displayed properly.

Compiling Source Files

The Project Navigator Auto-Update feature reprocesses sources when they are needed to perform the process you request.

However, you can compile individual source files by selecting the file in the Sources window, and then double-clicking **Compile Logic** in the Processes window. Alternatively, you can double-click a report in the Processes window, and the software will compile the source automatically.

Optimizing

Optimizing Designs

The default options in the ispLEVER software are set up to achieve the highest possible performance in the smallest possible device, for most designs. You can choose to maximize design flexibility by spreading out logic or exercise tighter control over the fitting process to achieve your design goals.

Each clock signal is evaluated and classified as a global clock or a non-global clock. The Fitter attempts to place all global clock signals at global clock pins (check the log file for the status of all clock signals after optimization). The Fitter assigns all other clock signals to I/O pins and implements them as Product Term clocks, if the architecture supports Product Term clocks. Input pins and nodes that are defined but not referenced (not used by another equation) are discarded from the design during optimization (warning messages are generated).

Design Resources Check

Information about the internal architecture of the specified device is loaded and resource checks are performed on the design. Errors are reported if the design exceeds the device's product term, macrocell, pin, clock, set, reset, or output enable control resources.

Logic Synthesis Options

Logic Synthesis options allow you to control how logic functions are optimized before partitioning takes place.

Boolean Logic Reduction

This option removes redundant product terms from each equation. Unless your equations have redundant logic to prevent problems (for example, in combinatorial functions), you should always leave this option selected.

D/T Synthesis

This option lets the optimizer automatically choose between a D-Type or T-Type register, thereby reducing the product term requirements. In some cases, the speed of the design may improve if only D-Type registers are used in an M4A device. This option should be selected for most designs.

Input Register Optimization

This option allows the Fitter to automatically place single-variable registered functions in input pad registers. This option should be selected for M4A devices unless you are trying to prevent the use of input registers.

XOR Synthesis

This option enables or disables exclusive OR synthesis. When this option is selected, the optimizer synthesizes XOR equations, if this can be achieved in the design. When this option is cleared, the sum-of-product equations will be generated. This option is device-dependent. Default state = Enabled.

Node Collapsing

This option allows the optimizer to collapse intermediate combinatorial nodes into registers and output pins, thus speeding up the design. Unless you have handcrafted each equation in your design, you should leave this option selected. This option should always be selected for designs that have been synthesized or described in low-level combinatorial gates.

Speed

This option collapses all nodes up to the set Product Term limit, globally optimized, without regard for the path.

Area

This option collapses all nodes up to the set Product Term limit, without increasing area cost.

Fmax

This option causes the Logic Optimizer to automatically identify all critical paths between any pair of registers, from clock-pin of one register to data-pin of the other register (or the same register). The Logic Optimizer then attempts to collapse/combine the logic nodes along the critical paths, reduce the logic level, and allow the chip to run at a higher frequency.

Collapsing Max. Product Term

This option lets you control the Fitter optimization process by setting a maximum limit on the number of Product Terms (PT) in each equation. In other words, the Optimizer shapes the equations relative to the set number of PT. For example, if the value is set to 35, the Optimizer stops collapsing equations when it exceeds 35 PT.

This option works the opposite of Splitting Max. Product Term.

Collapsing Max. Input

This option lets you control the Fitter optimization process by setting a maximum limit on the number of inputs in each equation. For example, if the value is set to 32, the Optimizer stops collapsing inputs when it exceeds 32 inputs.

Splitting Max. Product Term

This option lets you control the Fitter optimization process by setting a maximum limit on the number of Product Terms (PT) in each equation. In other words, the Optimizer shapes the equations relative to the set number of PT. For example, if the value is set to 35, the Optimizer splits equations if it has more than 35 PT.

This option works the opposite of Collapsing Max. Product Term.

Example

An M4A-32 design consists of six equations having 12 product terms each, and one equation having 21 product terms. (An M4A-32 macrocell can implement up to 20 product terms without equation splitting.) The Fitter can implement each of the six smaller equations as single-macrocell equations, but the one larger equation must be implemented using two macrocells. In its default mode, the optimizer will split the 21-product term equation into one equation of 20 product terms and one equation of 2 product terms (the extra product term is required to accept feedback from the second macrocell).

Reducing the equation-splitting threshold to 12 will result in less of an imbalance in the number of product terms placed at each macrocell. Each of the original 12 product term equations remains at a single macrocell, while the 21 product term equations is split into two macrocells: one with 12 product terms and one with 10 product terms. Thus, none of the equations are using the maximum capacity of its macrocell, which improves the odds of fitting the design and makes it easier to add logic to the design later.

***Note**: Do not reduce the equation splitting threshold if doing so will cause many equations to be split. If, for instance, the preceding example's six smaller equations had contained 15 product terms each, setting the gate-splitting threshold to 12 would have caused all seven equations to be split, resulting in 16 under-utilized macrocells.*

Example

Consider the following:

- A synchronous registered equation with 22 product terms
- Splitting Max. Product Term field set to 20

The equation will be split into two equations, one with 20 product terms and one with 3 product terms. It will take two passes through the array to implement the new equations.

Utilization Options

Utilization options let you specify the percentage of device resources available during each fitter run. You can choose to reduce the device resources available during the initial fitter run, back annotate the pins, and then increase the available device resources when making design changes.

Reducing available device resources during the fitter run may increase the fitter runtime. For example, when the maximum number of block inputs is set too low (<60%). This condition may cause the Fitter to take a long time grouping (i.e. partitioning) logic equations into blocks because the blocks have fewer available resources.

Logic Grouping

You can use logic grouping to exercise manual control over the Partitioner. Logic grouping lets you manually pack selected portions of your design into the same block while setting up the global optimization options to spread out the rest of the logic.

Logic grouping can also be used to group selected inputs, outputs, and buried logic functions into the same block or segment to achieve performance goals.

In cases where the Partitioner is unable to find a solution, manually grouping a small portion of the design may aid in the fitting process.

If you do attempt manual grouping, try to place logic with common inputs and feedback in the same block. This minimizes the number of signals crossing between blocks, which results in a lower demand for interconnection resources and an increased likelihood of a successful fit.

Fitting

Fitting Designs

The ispLEVER software has a single user interface with all options preset to deliver the highest possible push-button performance. At the end of a successful fitter run, the ispLEVER software generates a JEDEC file, as well as a fitter report, so that you can see how the ispLEVER software has routed the design and utilized resources on the part.

Performing Multiple Runs

For CPLD, ispXPLD, and ispXPGA devices, you can use the ispEXPLORER to try many combinations of constraints to achieve a fit. This is especially useful for designs that the software cannot fit because of the constraints. The ispEXPLORER produces a spreadsheet summary of results and settings for each run, making it easy to compare one group of settings with another.

The Fitting Process

After you have entered your design, you are ready to run the Fitter. The fitting process consists of four phases. An understanding of each phase can help you choose the best corrective action if the design does not fit, or your performance criteria are not met.

- Initialization
- Optimization
- Partitioning
- Fitting (Placement and Routing)

Initialization

At the beginning of a new project, the ispLEVER software automatically copies a default constraint file from the ispLEVER directory into your project directory. For first-time users, the default settings allow most designs to achieve a First-Time Fit (FTF). For users requiring more control, the default settings can be easily changed to achieve better fitting density or performance.

You can control the contents of the constraint file using the Global Constraints dialog box and the Location Assignments dialog box.

Using the Global Constraints Dialog Box to Control Optimization

Using the Global Constraints dialog box, you can pack your design, spread your design, or use other advanced options such as specifying device utilization levels.

Using the Location Assignments Dialog Box to Pre-assign Pins and Nodes

The Location Assignments dialog box in the Constraint Editor lets you specify pin and block locations, group signals in specific blocks, or even reserve pins for later use.

Assigning Pin and Node Locations

The ispLEVER software lets you pre-assign pin and node locations. You can use the Location Assignment dialog box in the Constraint Editor to assign input and output pins and buried nodes. The Macrocell, Block, and Segment list boxes are context sensitive to the selected device; only applicable features are available.

You can also use the drag and drop feature in the Package View of the Constraint Editor to assign input, output and bi-directional pins.

Pin and Node Pre-Assignment

Pre-assigning pins lets you lay out your board at the same time as you are doing logic design, thus shortening the design cycle. Pre-assigning nodes is usually not required and is not recommended.

Pin Assignment Guidelines

If you want to pre-place signals (not recommended unless pinout configuration is important), follow these guidelines:

- Do not place large equations to macrocells or pins at the beginning or end of a block.
- Signals that share many common inputs should generally be grouped in the same block (the Partitioner does this automatically). Signals that do not share many common inputs should generally be distributed across several blocks to avoid overburdening the switch matrix for a single block.

Large Functions at the End of a Block

The macrocells at the end of a block have access to fewer product terms than other macrocells.

- Cell number 0, the first cell in all devices, can access the product term clusters from adjacent, higher-numbered cells, but it cannot access any lower-numbered cells (cell 0 being the lowest-numbered cell in the block).
- The last cell in a block can access the product term cluster from the adjacent lower-numbered cell, but it cannot access any higher-numbered cells.

If signals have not been assigned to macrocells, the Fitter will find a macrocell replacement for all the signals that satisfy their product term requirements.

Adjacent Macrocell Use

In MACH devices, adjacent macrocells can share clusters. Therefore, with designs having equations that use a high number of product terms, it is a good idea not to place them in adjacent macrocells.

Modifying Assignments

You can use the Location Assignment dialog box of the Constraint Editor to modify the current location assignment. Modifications can also be made directly in the Pin Attributes sheet of the Constraint Editor.

Deleting Assignments

You can delete project assignments via the Constraint Editor. To do this, select the entire row whose existing assignment(s) you want to delete. From the Edit menu, select **Delete Row(s)**. You may also right-click and select **Delete Row(s)**. In cases where you no longer want any of the current assignments, you can delete all of them at the same time.

Ignoring Assignments

There may be times when you want to ignore, but not delete, the current assignments. For example, after you complete a design, you may want to try fitting it into a different device. In this case, the current pin assignments may not be valid for the new device. The ispLEVER software lets you ignore current constraints for the next Fitter run.

Power Control

Using the ispLEVER software, you can control power settings for your device. By default, the device is always set to high power, high speed. However, you can set the device or blocks of the device to low power mode. This setting results in slightly decreased speed, but increased power savings. This is useful for handheld and battery-operated devices.

Slew Rate Control

For the majority of Lattice devices, you can set the slew rate to either Slow or Fast. By default, the slew rate is set to Fast. However, changing it to Slow can result in less board noise.

Optimization

If your design failed to fit, or it did not meet your performance criteria, you can apply optimization procedures your design again.

Partitioning

After optimization, the design is partitioned into individual blocks on the specified device. Partitioning is achieved by assigning logic to specific blocks, based on the following considerations:

- Individual signal pre-placements and Grouping assignments
- A block's available internal resources (free macro cells, product terms, clock signals, and so forth)
- The switch-matrix interconnect resources available to the block

The Partitioner considers commonality of signals, macro cell requirements, Set/Reset requirements, product-term requirements, and other factors to determine which partition is most likely to succeed in fitting the design. Only partitions that are likely to succeed (according to the Partitioner's rules) are attempted.

Balanced Partitioning

Controlling how the Partitioner works can be very important. There is one important strategy for partitioning and that is called Balanced Partitioning. By selecting the Balanced Partitioning option in the Global Constraints dialog box, you are telling the Partitioner to spread all of the signals among all the blocks in the device, rather than trying to fill a few blocks to their maximum potential.

There are advantages to either side of the strategy. If you turn balanced partitioning on, you can save room in the device for any future functionality you might want to add to existing logic. However, turning balanced partitioning off lets you "pack" as much logic into the minimum number of blocks in the device as possible, leaving some free blocks for future design enhancements.

Place and Route (Fitting)

Placement is the assignment of physical block resources such as I/O pins, XORs, registers, and product-term clusters to logic equations. *Routing* is the assignment of switch-matrix interconnect resources to logic equations, after the logic equations are placed.

Placement

In the placement phase of the fitting process, individual equations are assigned to physical resources, as follows:

- Logic equations that have been pre-assigned to pins are assigned first.
- Buried logic functions are placed in the remaining unused macrocells.
- Inputs are assigned to any available pin. These pins can be dedicated inputs pins, clock/input pins, or I/O pins that correspond to macrocells that are either unused or used to implement buried logic functions.
- Outputs can be assigned to any unused I/O pin.

Spread Placement

Controlling how the Placer works is also important. When you select the Spread Placement option, you are telling the Placer to spread the signals in the block as far out as possible.

Routing

In the routing phase, the Fitter attempts to route input, output, and feedback signals to and from the physical resources assigned in the placement phase. If the Fitter fails to route all signals, it tries another placement. The Fitter continues trying different placements, and different routing attempts within each placement, until a successful fit is found or the time allotted for fitting is exceeded.

Fitter Options

The Global Constraints dialog box lets you set options for the Fitter. Using the Global Constraints dialog box, you can tell the Fitter to pack as much logic into the device as possible, spread the logic across the entire device, or use other advanced options such as specifying device utilization. The following sections describe these options.

Pack Design

The Pack Design option lets you pack as much logic into the device as possible. This option allows you to achieve the highest possible performance in the smallest possible device, for most designs. Each block may be completely filled, leaving less room for any design changes or logic additions.

Spread Design

The Spread Design option spreads all of the logic across the entire device rather than trying to fill each block to its maximum potential. This option allows you to achieve the highest possible performance, while leaving room for any additional functionality that you may want to add in the future. The fitter leaves room to accommodate design changes to existing logic. Because each block may be incompletely filled, the design may *or may not* require a larger device to achieve a successful fit.

Advanced Options

The advanced options let you individually control the partitioning and placement algorithms.

Balance Partitioning

The Balance Partitioning advanced option partitions the design evenly among all the blocks in the device, so each block should have the same amount of resources used. When this option is cleared, the software partitions the design block-by-block, filling up one block at a time. This means that some blocks may be filled up completely, while others may be unused.

Spread Placement

The Spread Placement advanced option places the signals evenly, or spreads them out, among macrocells in the block. Spreading out the placement lets you make minor changes to the existing output and node signals in the block. When this option is cleared, the software assigns design signals to the first available macrocell, making it easier to add new outputs or nodes to a block.

Fitter Effort

The Fitter Effort option is used to instruct the Fitter how much effort to apply to a fit. The Low option enables a faster fitting process, but will be more likely to result in failures to fit when the utilization gets higher. The High option provides the most exhaustive search of the solution space, but takes more time.

Fitter Report Formats

Two Fitter Report formats are available in the ispLEVER software, text and HTML.

- If you select the Fitter Report process associated with the target device, the Fitter Report is opened in the Output Panel of the Project Navigator or in the Report Viewer.

  ***Note**: By default, the ispLEVER software opens Fitter Report in the Output Panel of the Project Navigator. If you want it opened in the Report Viewer, select **Using Report Viewer** in the Log tab of the Environment Options dialog box (Project Navigator: Options > Environment).*

- If you select the HTML Fitter Report process associated with the target device, the Fitter Report is opened with your local Internet Browser.

Formatting the Fitter Report

For CPLD devices, you can select various options that determine the information in the Fitter report using the Fitter Report Options dialog box.

Understanding the ispXPLD Fitter Report

The Fitter Report displays statistics and information on the fitting process of your design, including utilization numbers, pin assignments, etc. The Fitter Report is also written into HTML format to allow user to browse through the report easily.

The Fitter Report is divided into several sections, each briefly described below.

Project Summary

The Project Summary Section contains basic information about your project. Information about the ispLEVER project name, project working directory, and basic device parameters are presented in this section.

Two of the entries in this section require some further clarification. The first is the MFB Input Mux Size. Each ispXPLD MFB has 68 inputs. Each of these inputs is preceded by a multiplexer that permits the routing of signals from I/O pins and from macrocell nodes. The MFB Input Mux Size describes the size of the multiplexer, which feeds each one of the 68 input pins of the MFB. For the XPLD 5512MX each input signal to the MFB has a 48 pin multiplexer providing routing from the I/O and macrocell nodes.

The second entry is the Available Blocks, which provides information about the number of MFB's contained within the XPLD device.

Compilation Times

This section tells you how long it took the Fitter to fit the design in the specified device. The name for each process step is listed, as well as the total elapsed time. Prefit Time consists mainly of run-times of the design compilation and optimization phases. Total Fit Time is the total run-time of the design compilation, optimization, partition, placement and routing phases.

Design Summary

This section describes the overall device utilization. Each resource in the design is represented on its own line. Each line describes the resource type, the number/amount of that resource, the amount of that resource used by the design, and the amount available for additional logic.

Example 1 shows the entire Device Resource Summary section for an ispXPLD design. As can be seen in the example, there are two dedicated clock pins, two dedicated clock/clock enable pins, two dedicated global enable pins, and one dedicated global reset pin. Example 1 shows all dedicated clock and clock/clock enable pins have been used in the design.

Some of the entries are easy to understand, as there is a one to one correlation to items described in the datasheet. Other items in this section require some detailed description, as there is no description of these elements in the datasheet. Each of the derived items will be described in turn.

Logic Macrocells: This is a simple arithmetic addition of the total number of logic macrocells in a XPLD device with the number of I/O pins available in the package. In Example 1, a XPLD 5512MX 484 BGA part is described. A 5512MX has 512 macrocells, and a 484 BGA has 253 I/O pins. Thus the number of logic macrocells is 512+253 = 765. This is used as a rough approximation of the amount of logic available to a design, since a macrocell register and I/O pin can be used independently.

MFB Inputs: This is another simple ratio, and is the multiplication of the number of inputs to a MFB to the number of MFB's in the device. The 5000MX series has 68 inputs per MFB, a 5512MX has 16 MFB, or 68 * 16 = 1088. The number of MFB's will vary by the size of XPLD device.

Logical Product Terms: This is the number of Product Terms per MFB multiplied by the number of MFB's in the design. For the 5512MX this is 160 product terms * 16 = 2560 logical product terms. This number is a pre-placement calculation.

Occupied MFBs: This field indicates how many of the MFB's have at least one macrocell used within them. For each MFB that has a single macrocell placed within it, the Used column is incremented by one.

Occupied Macrocells: The entries below this header all need to refer to Figure 1. Figure 1 shows the basic XPLD MFB structure. Within the left-most dotted section is the And Array, the center dotted area is the Dual-Or Array, also known as a cluster, and the rightmost dotted section is the Macrocell. The Occupied Macrocell section only describes how many of the macrocells described in the rightmost dotted section have been used.

Two Function Macrocells: This section describes how many of the macrocells have used both the macrocell register and the I/O pin. The register and I/O pin must be assigned independent equations (i.e. the register does not directly drive the I/O pin).

One Function Macrocells: This section will normally contain the largest percentage of the macrocell usage in most designs. A One Function Macrocell is one where the macrocell register has been used, or the macrocell I/O has been used.

Zero Function Macrocells: This section describes when a macrocell has not been used, but its shareable product term has been used. The Dual-Or Array is used and then exported to adjacent macrocells via the Product Term Sharing Array (PTSA), or to the N+7th macrocell.

Memory Macrocells: Each memory block used in an XPLD uses up one complete MFB. Each MFB contains 32 macrocells. The used memory macrocells is the product of the 32 macrocells (i.e. one MFB) and the Total Memory Blocks value found in the Design_Summary section. For example, if Total Memory Blocks is 2 then the used Memory Macrocells will be 2*32 = 64.

Arithmetic Macrocells: This is the number of macrocell registers used for arithmetic functions. Typically this will be a count of the number of counter bits in the design.

Occupied Product Terms: This entry is, once again, just a simple summation. The number of available product terms here is simply the number of Logic Product Terms and the number of Control Product Terms. There are four shared control product terms per MFB (see Figure 1), and for a 5512MX device there are 16 MFB's, giving a total of 64 Control Product Terms.

Available Macrocells: ADD NEW INFO

Control Product Terms: The control product terms describe resource usage on PT Clock/Clock Enable, PT Reset, PT Preset, or PT OE. In effect this counts product terms that are not block, segment, or global.

Segment Product Term Enable: This is the number of Shared PT Output Enable available in the XPLD device. There is one Shared PT Output Enable per segment. See the MFB_Resource_Summary section for the description of a segment.

MFB Clocks: This is the Shared PT Clock, one is present for each MFB.

MFB Clock Enables: This is the Shared PT Clock Enable, one is present for each MFB.

MFB Resets: This is the Shared PT Reset to the macrocell. There is one present for each MFB.

Macrocell Clocks: This is the PT Clock in each macrocell, one is present for each macrocell.

Macrocell Clock Enables: This is the PT Clock Enable in each macrocell, one is present for each macrocell.

Macrocell Enables: This is the PT Output Enable in each macrocell.

Macrocell Resets: This is the PT Reset in each macrocell.

Macrocell Presets: This is the PT Preset in each macrocell.

Example 1

	Total Available	Used	Available	Utilization
Dedicated Pins:				
Clock Pins	2	2	0	100
Clock/Clock Enable Pins	2	2	0	100
Enable Pins	2	0	2	0
Reset Pins	1	0	1	0
I/O Pins	253	84	169	33
Logic Macrocells	765	328	437	42
Input Registers	253	0	253	0
Unusable Macrocells	..	0	..	..
MFB Inputs	1088	353	735	32
Logical Product Terms	2560	901	1659	35
Occupied MFBs	16	15	1	93
Occupied Macrocells	512	312	200	60
Two Function Macrocells	..	0	..	..
One Function Macrocells	..	195	..	..
Zero Function Macrocells	..	33	..	..
Memory Macrocells	..	64	..	..
Arithmetic Macrocells	..	20	..	..
Occupied Product Terms	2624	962	1662	36
Available Macrocells	512	75	437	14
Available 1 PT Macrocells	..	16	..	..
Available 2 PT Macrocells	..	7	..	..
Available 3 PT Macrocells	..	20	..	..
Available 4 PT Macrocells	..	74	..	..
Available 5 PT Macrocells	..	320	..	..
Control Product Terms:				
Segment Product Term Enable	16	0	16	0
MFB Clocks	16	0	16	0
MFB Clock Enables	16	4	12	25
MFB Resets	16	13	3	81
Macrocell Clocks	512	0	512	0
Macrocell Clock Enables	512	50	462	9
Macrocell Enables	512	0	512	0
Macrocell Resets	512	0	512	0
Macrocell Presets	512	12	500	2
Global Routing Pool	765	267	498	34
GRP from IFB	..	47	..	..
(from input signals)	..	47	..	..
(from output signals)	..	0	..	..

	Total Available	Used	Available	Utilization
(from bidir signals)	..	0	..	..
GRP from MFB	..	220	..	..

Device Resource Summary

This section lists all of the resources available within the device and how much of each resource has been used by the design. It also reports how much of each resource is still available.

GLB Resource Summary

This section lists various GLB (and segment) level resource counts, such as fan-in (or array inputs), I/O pins, input registers, macrocells, logic product terms and product term clusters.

GLB Control Summary

This section lists the totals for all control signals, and how much of each is utilized by individual GLBs.

Optimizer and Fitter Options

This section displays all of the settings that were used to fit and optimize the design. These include things such as Ignoring Constraints to the type of flip-flop synthesis you have chosen. The information in this section is set with the Constraints Options dialog box.

Pinout Listing

This section lists the I/Os and control signals on the device, and how they are assigned.

(Input, Output, Bidir, Buried) Signal List

This section reports information on individual I/Os, such as I/O type, location assignment, the fan-out, and other signal attributes.

Signals Fan-out List

This section lists signal resources and the functions they fan-out to.

GLB (GLB name) Cluster Steering Tables

This section shows information about how functions and inputs are placed in a GLB. It shows how product terms are steered to a macrocell on which a function has been placed. It also contains information about what type of control signal has been used.

GLB (GLB name) Logic Array Fan-in

This section shows how design signals are mapped to individual GLB block inputs.

Product Term Histogram

This section lists and sorts the equations according to the number of product terms they use (in the logic only).

GLB Input Histogram

This section lists and sorts the equations according to their number of inputs, which includes the logic and the ctrl signals, but not the global signals (dedicated routing).

Post-Fit Equations

This section reports the equations in your design, after fitting. It begins with a product term histogram and a GLB input histogram.

Back Annotating Assignments

You can back annotate (write) assignments from the Fitter output to the project constraint file using the Back Annotation tab on the Constraints Options dialog box. This feature lets you retain the assignments made by the Fitter so that they can be used in a future fitting process.

You can back annotate the following location and constraint assignment options:

- Pin Assignments — Only pin assignments are back annotated to the project constraint file. This option lets you retain the fitter pin assignments. All buried nodes assignments are not retained. Existing buried node assignments are removed from the project constraint file.
- Pin and GLB Assignments — Only pin and GLB assignments are back annotated to the project constraint file. The GLB assignments for the back annotated buried nodes are retained, but the associated macrocell assignments are removed. This allows the buried nodes to be placed in the same blocks, but it does not force the Fitter to use the same macrocell assignments.
- Pin, GLB and Macrocell Assignments — The pin, GLB, and macrocell assignments for the design are back annotated. This option retains all the pin and macrocell assignments.
- IO Types — You can also back annotate IO Types constraint settings.

You can only back annotate project assignments after the Fit Design process has been successfully completed. An error message appears if the ispLEVER software detects that this process did not complete successfully.

Design Verification

Verifying Designs

The ispLEVER software supports two types of timing verification: *static timing analysis* and *dynamic timing simulation*. Both of these methods support all Lattice devices.

Static Timing Analysis

Static timing analysis (timing analysis) is the process of verifying 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/hold time requirements, and the maximum frequency. Lattice has two static timing analysis tools, Performance Analyst and TRACE (for FPGAs).

The primary advantage of timing analysis is that it can be run at any time and requires no input test vectors, which can be very time consuming and tedious to create. Another major advantage of static timing analysis is that it exhaustively checks every possible input-to-output path. One shortcoming of all static timing analysis tools is that they detect false paths that will never be exercised during the course of normal operation of a circuit so that you could spend a lot time instructing the analyzer to ignore those paths. In this process, you could accidentally ignore a real issue. Although timing analysis does not give you a complete timing picture, it is an excellent way to quickly verify the speed of critical paths and identify performance bottlenecks.

Dynamic Timing Simulation

This type of analysis is based on an event-driven simulator and requires you to specify a test vector (waveform). Whereas timing analysis returns partial timing information, dynamic timing simulation (timing simulation) will give you detailed information about gate delays and worst-case circuit conditions. Because total delay of a complete circuit will depend on the number of gates the signal sees and on the way the gates have been placed in the device, timing simulation can only be run after the design has been implemented. Timing simulation also requires several input files to run.

There are two basic types of dynamic timing simulation tools, logic simulators (e.g, Lattice's Logic Simulator) and dynamic simulation analyzers (e.g., MTI's ModelSim™). Logic simulators function in a single delay mode, whereas dynamic simulation analyzers will simulate the ambiguity in delay pairs. Dynamic simulation analyzers typically take longer to process simulation results than logic simulators and considerably longer than static timing analysis tools.

Timing Verification Tools

The ispLEVER software offers timing verification with the following tools:

- Performance Analyst — Static timing analysis tool that runs timing analysis (All CPLD, ispXPLD, ispXPGA, and ispGDX2 devices except ispLSI 1K and 2K).
- Lattice Logic Simulator — Logic simulator that runs timing simulation (ispGDX and CPLD devices only).
- ModelSim for Lattice — Dynamic timing analyzer that runs timing simulation (all devices).
- TRACE – The Tim*ing Reporter and Circuit Evaluator* (TRACE) is an integrated flow tool that provides static timing analysis based on timing preferences (for FPGA devices only).

Additionally, timing simulation for all devices is supported with these tools:

- Text Editor — Used to create test stimulus files
- Waveform Editor — Used to create test stimulus files graphically

- Waveform Viewer — Used to view the results of simulation

Verification Environments

The timing verification tools operate in both integrated and stand-alone environments.

Integrated Timing Analysis

The Performance Analyst is a static timing analysis tool that lets you quickly determine the performance of designs implemented in any Lattice Semiconductor device. To run timing analysis, launch the Performance Analysis from the Project Navigator. The Performance Analyst traces each logical path in the design and calculates the path delays using the device's timing model and worst-case AC specs supplied in the device data sheet.

The timing analysis results are displayed in a graphical spreadsheet with source signals displayed on the vertical axis and destination signals displayed on the horizontal axis. The worst-case delay value is displayed in a spreadsheet cell if there is at least one delay path between the source and destination. To more easily identify performance bottlenecks, you can double-click a cell to view the path delay details.

Integrated Timing Simulation

To verify a design inside the current project, the ispLEVER software provides integrated verification with the Lattice Logic Simulator and ModelSim™ for Lattice from Mentor Graphics®. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the verification test bench, the following processes are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

CPLD and GDX Project Navigator Process	Simulation Tool Invoked
Timing Simulation	Lattice Logic Simulator
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

Stand-alone Simulation

The ispLEVER software supports stand-alone timing simulation. This provides an easy entry if you need to verify a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator Tools menu.

Required Files for Verification

The Lattice Logic Simulator and ModelSim support Lattice ispGDX and CPLD device timing simulation. In addition to your design files, you will need at least one test stimulus file, a netlist file, and a timing delay file.

	Lattice Logic Simulator	ModelSim
Test Stimulus File Formats		
• **.wdl** (Graphic waveform)	X	
• **.abv/.abl** (Test vector)	X	
• **.tf** (Verilog test fixture)		X
• **.vhd** (VHDL test bench)		X
Netlist File Formats		
• **.vo** (Verilog netlist)		X
• **.vho** (VHDL netlist)		X
• **.edo/.sim** (EDIF/simulation)	X	
Delay File Formats		
• **.sdf** (Standard Delay File)	X	X

Verification File Descriptions

Test Stimulus Files

- Graphic Waveforms (`.wdl`) — A file created by the Waveform Editor file that graphically represents a waveform as a sequence of signal states separated by time intervals.
- Test Vectors (`.abv/.abl`) — Test vectors are sets of input stimulus values and corresponding expected outputs that can be used with both functional and timing simulators. Test vectors can be specified either in a top-level ABEL-HDL source or in a separate ABEL-HDL test vector format (`.abv`) file. The ABV file is considered a text document and is kept above the device level in the Sources window. Whether the test vectors are part of a top-level ABEL-HDL source (`.abl`) or are in a separate file, they will be compiled and passed to the simulator.
- Verilog Test Fixtures (`.tf`) — A Verilog test stimulus file that specifies the input waveforms for simulation in ASCII format.
- VHDL Testbench (`.vhd`) — A VHDL test stimulus file that specifies the input waveforms for simulation in ASCII format.

Netlist Files

- Verilog Netlist (`.vo`) — For Verilog designs, the back-annotated timing simulation netlist is named `<design_name>.vo`.
- VHDL Netlist (`.vho`) — For VHDL designs, the back-annotated timing simulation netlist is named `<design_name>.vho`.
- EDIF/Simulation Netlist (`.edo/.sim`) — For EDIF designs, the back-annotated timing simulation netlist is named `<design_name>.edo`.

Timing Delay Files

- Standard Delay Format (`.sdf`) — A file containing delay and timing constraint data for cell instances named `<design_name>.sdf`.

Generating Timing Simulation Files

After you have fit the design, the ispLEVER software lets you export the netlist and delays for timing simulation. For netlist files, ispLEVER supports VHDL, EDIF, and Verilog formats. For timing delay files, ispLEVER supports the standard SDF and Viewlogic DTB timing formats.

To choose a simulation file format:

1. In the Project Navigator, choose **Tools > Generate Timing Simulation Options** to open the dialog box.
2. Select the format options that you want.
   - For netlist formats, you have a choice of Verilog, VHDL, or EDIF (Version 2.0.0). If you choose the EDIF format, you can customize the Power/Ground representation by selecting either Cell or Net.
   - For timing format, SDF (version 2.1) and Viewlogic DTB format are supported.
3. Click **OK** to close the dialog.
4. When you run the **Generate Timing Simulation Files** process, ispLEVER generates the files in the specified formats.

Viewing the Simulation Input Files

You can use the Report Viewer to view simulation input files.

1. In the Project Navigator Sources window, select the target device.
2. In the Project Navigator Processes window, double-click **Report File**. Notice the two output files (Netlist and Delay) listed at the top of the report. The ispLEVER software generates these files and places them in the project directory.
3. To view these files, choose **File** > **View** and select the file that you want to view.

***Note**: You cannot modify files using the Report Viewer. You can only view them.*

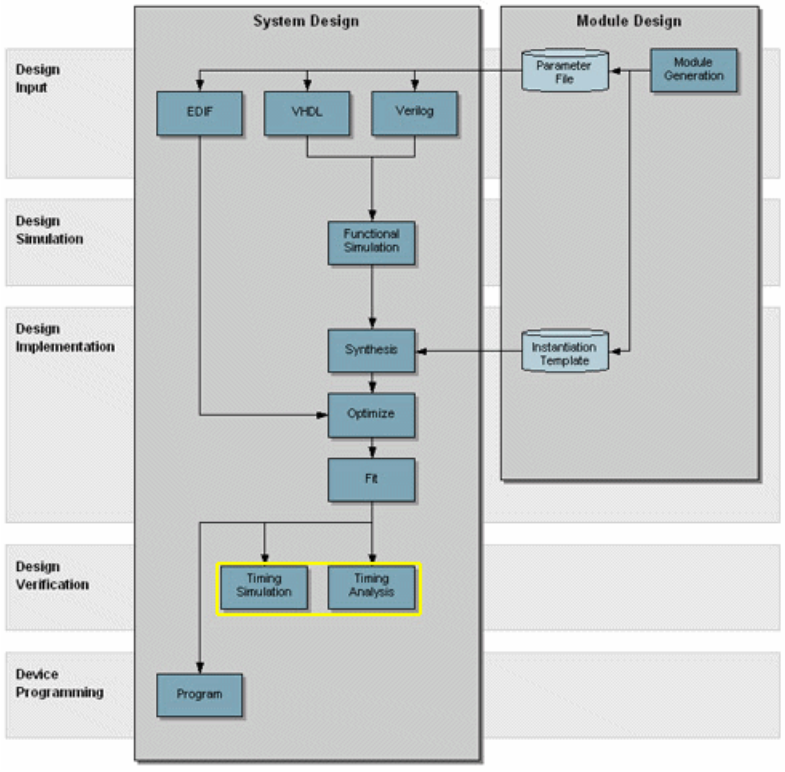
ispXPLD Verification Summary

The ModelSim simulator supports ispXPLD timing simulation for EDIF and HDL design entry methods and requires at least one Verilog test fixture (`*.tf`) or VHDL testbench (`*.vhd`) stimulus file. Additionally, ModelSim requires a netlist file (`*.vo` or `*.vho`) and a timing delay file (`*.sdf`).

	ModelSim	
	.tf	.vhd
	.vo + .sdf	.vho + .sdf
EDIF	X	X
Verilog	X	
VHDL		X

ispXPLD Verification Process Flow

The figure below shows timing analysis and simulation within the ispXPLD process flow.



Stamp Model

Introduction to Static Timing Analysis Tool

In the era of high-performance electronics, timing continues to be the number one issue in CPLD design. The designers are spending an increasingly large percentage of their time addressing CPLD performance.

Static Timing Analysis Tool provides solutions to the complexity, the performance, and even the time-to-market crises. Static Timing Analysis separates performance analysis from functional verification. It assumes that the design is functionally correct and analyzes performance only.

Functionality of Static Timing Analysis Tool

Static Timing Analysis Tool can be used as the board-level static timing analyzer. You can obtain results of static timing analysis and eliminate the timing errors in significantly less time for a board-level design. Its modes are mainly chip-level.

Static Timing Analysis Tool can also be integrated with synthesis tools. After the integration, it can be used for timing optimization of synthesis. There are four modeling levels: black box, gray box, gate-level, and transistor-level.

Moreover, Static Timing Analysis Tool is used as a gate-level timing analyzer, which enables you to get some timing information such as critical path, maximum operating frequency, setup and hold time for flip-flop, delay of combination circuit, pin-to-pin delay. The Performance Analyst is used for this purpose.

Stamp Modeling Language

Stamp is a new modeling language developed to enable the accurate, concise, and complete specification of timing models for large cores and blocks.

Stamp Model Generator Command Line

```
stamppar -i input_file [-o output_file][-tech tech_file][-vl][-gui][-log
automake.err][-help]
```

Input Files (produced by the Performance Analyst):

`design_name.trp` (Produced by the Performance Analyst and contain all information for stamp model generator)

Output Files

`design_name.mod` (Stamp model file of a design)

`design_name.data` (Stamp model data file of a design)

Technology Files

Technology file allows you to specify technology-dependent constraint values for some Stamp timing arcs.

Report files

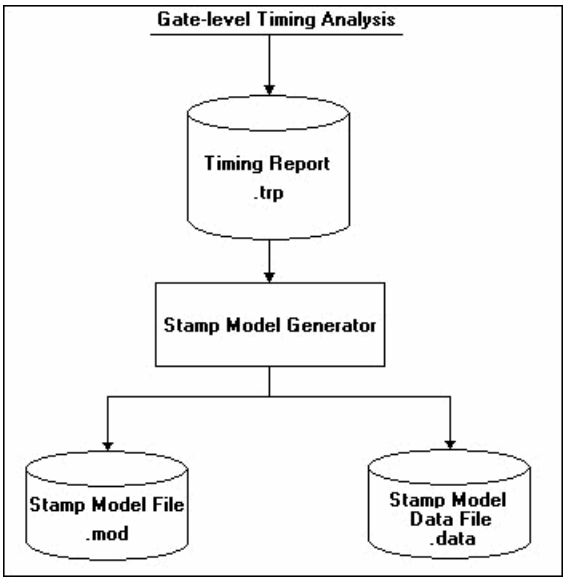
`automake.log` (-gui mode, integration in ispLEVER)

`stamppar.log` (in stand-alone mode)

Design Flow

Lattice Stamp Model Generator is used to build board-level (or chip-level) Stamp Models of a design with any Lattice Semiconductor device for board-level static timing analysis tools. With board-level Stamp Models for PMS (Processor, Memories, Switch, and peripheral circuit) parts such as CPU, memories, ispMACH chip of a design, a board-level static timing analysis tool enables you to manage large, high performance designs while minimizing development time. With the Stamp models, you accelerate board-level design.

Design Flow of Lattice Timing Model Generator



Running the Lattice Stamp Model Generator

You can run Lattice Stamp Model Generator either in its integration mode or in its stand-alone mode.

Integration Mode

After running the Performance Analyst, with the target device selected in the Sources window of Project Navigator, double-click Generate Board-level Stamp Model in the Processes window. When the process has completed successfully, the Stamp Model File (`.mod`) and the Stamp Model Data File (`.data`) are generated. You can view the two files in the Report Viewer.

Stand-alone Mode

If you have some timing data other than the current project, you can also invoke the stand-alone Stamp Timing Model Generator using the following command line:

```
stamppar -i design_name [-o new_design_name][-tech tech_file][-vl]
```

The default Stamp files are `<design_name>.mod` and `<design_name>.data`. You can use the `[-o new_design_name]` option to produce new stamp files `<new_design_name>.mod` and `<new_design_name>.data`.

The `[-tech tech_file]` option allows you to specify technology-dependent constraint values for some Stamp timing arcs.

The `[-vl]` option forces the `design_name` and port name in Stamp Model to be upper-case so that they can correspond to those in the board-level netlists created by Viewdraw of Viewlogic.

All timing paths of your design can be produced after running Lattice Stamp Timing Model in its stand-alone mode.

***Note**: In this release of ispLEVER, the stand-alone Stamp Model only supports ispGDX and ispMACH devices.*

Structure of Stamp Model File and Stamp Model Data File

Stamp Model File (`.mod`)	Stamp Model Data File (`.data`)
Header	Header
Port Definitions	Port Data
Arc Definitions	Arc Data

Sample Stamp Model File (.mod)

```
MODEL
MODEL_VERSION "1.0";
DESIGN "and_ff2";
DATE "Fri Dec 01 16:43:56 2000";
VENDOR "Lattice Semiconductor Corporation";
PROGRAM "STAMP Model Generator";

/* port name and type */
INPUT Clk;
INPUT input_1;
INPUT input_2;
OUTPUT output_q;

/* timing arc definitions */
Clk_output_q_delay: DELAY Clk output_q;

/* timing check arc definitions */
input_1_Clk_setup: SETUP(POSEDGE) input_1 Clk;
input_1_Clk_hold: HOLD(POSEDGE) input_1 Clk;
input_2_Clk_setup: SETUP(POSEDGE) input_2 Clk;
input_2_Clk_hold: HOLD(POSEDGE) input_2 Clk;

ENDMODEL
```

Sample Stamp Model Data File (.data)

```
MODELDATA
MODELDATA_VERSION "1.0";
DESIGN "and_ff2";
DATE "Fri Dec 01 16:43:56 2000";
VENDOR "Lattice Semiconductor Corporation";
PROGRAM "STAMP Model Generator";

/* port drive, max transition and max capacitance */
PORTDATA
Clk: MAXTRANS(0.0);
input_1: MAXTRANS(0.0);
input_2: MAXTRANS(0.0);
output_q: MAXTRANS(0.0);
ENDPORTDATA

/* timing arc data */
TIMINGDATA

ARCDATA
Clk_output_q_delay:
CELL_RISE(scalar) {
VALUES(6.4);
}
CELL_FALL(scalar) {
VALUES(6.4);
}
ENDARCDATA

ARCDATA
input_1_Clk_setup:
RISE_CONSTRAINT(scalar) {
VALUES(0.6);
}
FALL_CONSTRAINT(scalar) {
VALUES(0.6);
}
ENDARCDATA
ENDTIMINGDATA
ENDMODELDATA
```

Device Programming

Programming Devices

Lattice supports device programming for all programmable logic devices with the following tools, which are briefly described below and covered in detail in their respective Help. They are listed in alphabetical order.

ispVM System

The ispVM™ System software (ispVM) supports both serial and concurrent (turbo) programming of all Lattice devices in a PC environment. The ispVM System software is built around a graphical user interface. Device chains can be scanned automatically. Any required JEDEC ISC or Bitstream data files are selected by browsing with a built-in file manager. Non-Lattice devices that are compliant with IEEE 1149.1 can be bypassed once their instruction register length is defined in the chain description. Programmable devices from other vendors can be programmed through the vendor-supplied SVF file. See the ispVM System Help for more information about this tool.

Model 300 Programmer

The ISP Engineering Kit Model 300 programmer is an engineering device programmer that supports prototype development by allowing single-device programming directly from a PC. The Model 300 programmer supports all JTAG devices produced by Lattice, with device Vcc of 1.8, 2.5, 3.3, and 5.0V. See the Model 300 Programmer Help for more information about this tool.

SVF Debugger

The SVF Debugger can be used with ispVM System software to help you debug a Serial Vector Format (SVF) file. The SVF Debugger software allows you to program a device, and then edit, check syntax, debug and trace the process of an SVF file. See the SVF Debugger Help for more information about this tool.

Universal File Writer

The Universal File Writer (UFW) is a separate application that generates bitstream files or an SVF data file for a single device. Using JEDEC or ISC files, the software generates bitstream PCM, Intel Hex and Motorola Hex data files concurrently. It can also generate an SVF file using the parameters you select. You can run the Universal File Writer from the ispVM System toolbar or separately. See the Universal File Writer Help for more information about this tool.

Running ispLEVER from the Command Line

Running from the Command Line

You can run the ispLEVER software from the command line on PC and UNIX using **ispflow** command line software. The ispflow software will attempt to fit the design to the specified part.

***Note**: All examples are shown in PC format. For UNIX, use forward slash instead of back slash.*

Syntax

The format of the command is as follows (on one line):

```
ispflow [-i <design>] [-d <device>] [-imp <yes\no>]
[-sdf <edif\verilog\vhdl\off>]
[-edf <mentor\synopsys\synplicity\viewlogic>]
[-syn <spectrum\synplify>] [-h] [-v]
```

where [ ] denotes optional parameters.

Definitions

-i <design name>	EDIF, Verilog, or VHDL design name. The name must include the appropriate file extension in the design name, such as `.edf`, `.v`, or `.vhd`.
-d <device name>	Specifies the device part number that the design will be fitted to. For example, `LFX125B-04F256C`. This option is required if the `<design name>.lci` file does not exist. After running **ispflow**, the specified device will appear in the newly created LCI file.
-imp <yes/no>	Import source constraints. Default is Yes.
-sdf <edif>	Outputs an SDF file in EDIF format.
-sdf <verilog>	Outputs an SDF file in Verilog format.
-sdf <vhdl>	Outputs an SDF file in VHDL format. Default.
-sdf <off>	Switch off SDF output.
-edf <mentor>	Specifies a Mentor Graphics-generated EDIF file. Default.
-edf <synopsys>	Specifies a Synopsys-generated EDIF file.
-edf <synplicity>	Specifies a Synplicity-generated EDIF file.
-edf <viewlogic>	Specifies a Viewlogic-generated EDIF file.
-syn <spectrum>	Specifies a LeonardoSpectrum synthesize file. Default.
-syn <synplify>	Specifies a Synplify synthesize file.
-spd <yes/no/fmax>	Speed: yes (speed), no (area), fmax.
-mpts	Max_PTerm_Split value.
-mptc	Max_PTerm_Collapse value.
-mptl	Max_PTerm_limit value.
-mfan	Max_fanin value.
-msym	Max_symbols value.
-fmll	Fmax_Logic_Level value.
-svf <on/off>	Switch on SVF generation. Default is off.
-r <*.lct/*.lci>	Refit using the constraint file.
-c <yes/no>	Specifies to run vcick (vc checker). Default is no.

-h	Help.
-v	Displays version number

The input source file switch (**-i**) is mandatory. All other switches are not.

If a device name (**-d** <device name>) is specified from the command line, and a Lattice Constraint File (.lci) file exists, the device specified from the command line takes precedence. The **ispflow** software will not run if a device name is not specified from either the command line or in a LCI file.

Specifying Options and Pin Assignments

To specify optimization options and pin assignments, you must create or modify a *<design_name>*.lci file. See Lattice Constraint File Description for more information on the LCI file format. If there is no LCI file, the software creates a default file for the specified device. In this case, the **-d** option is required.

***Note 1**: The LCI file is case-sensitive. Once an LCI file is created, the –d option can be omitted from the command line, because the device information will be obtained from the LCI file. The –d option takes precedence over the LCI file. If the –d option is used again with a different device part number, the device information part of the LCI file will be updated to reflect the changes.*

***Note 2**: Pin assignments and certain optimization options in the LCI file could be incorrect if the device is changed on the Command Line.*

The ispflow software runs through the complete flow, including timing analysis.

Input Formats

Acceptable input formats are non-hierarchical EDIF, VHDL, and Verilog HDL source files. The source files must be named with the .edf, .vhd, or .v extensions respectively.

Log Files

A log file of the process will be generated named *<design_name>*.batch.log file.

Batch Mode Example

The following example is for fitting a new design.

***Note**: The example is shown in PC format. For UNIX, use forward slash instead of back slash.*

To fit a new design:

1. Create a new directory and copy the input files needed.

   <isptools>\ispcpld\examples\ispxpga\edif\design\traffic\traffic.edf

   <isptools>\ispcpld\examples\ispxpga\edif\design\traffic\traffic.lci

2. Run ispflow -i traffic.edf.

   If you have a Lattice Constraint File (*<design_name>*.lci), the software will get the device from the LCI file and fit the design. The LCI file can be varied to an optimizer setting that you prefer. See Lattice Constraint File Description for additional information.

OR

3. If you do not have a LCI file, run ispflow -i traffic.edf -d LFX1200B-04F900C.

The software fits the design into the specified ispXPGA device.

Retaining Pin Assignments Using Batch Mode

You can retain pin assignments by copying LOCATION ASSIGNMENTS from your output Lattice Constraint File (`.lco`) into your input Lattice Constraint File (`.lci`).

To retain pin assignments using batch mode:

1. In the project directory, using a text editor, open the *<design_name>*`.lco` file.
2. In the LCO file, copy the LOCATION ASSIGNMENTS section.
3. In the project directory, using a text editor, open the *<design_name>*`.lci` file.
4. Replace the LOCATION ASSIGNMENTS section in the LCI file with the LOCATION ASSIGNMENTS section you copied from the LCO file.

LCI Files

The Lattice Constraint File (`.lci`) contains the constraints for Part selection as well as the Optimization and Placing and Routing processes.

You do not need to generate a LCI file on the first fit. If you specify a device (`-d`) with ispflow, this will generate a LCI file.

Command Line FAQs

The following are Frequently Asked Questions about processing designs in Command Line Mode using ispflow software.

Q. What report files are available to view after processing a design using Command Line mode?

A. Using a text editor, you can view the Timing Report File (`.trp`) and the Fitting Report File (`.rpt`) in the project directory after command line batch mode processing.

The RPT file displays details about how the current design was fit into the target device. If a fit was not accessible, the RPT file explains the problems preventing a fit. The file is located in your project directory, and is named *<design_name>*`.rpt`

The TRP file contains all information for stamp model generator. The file is located in your project directory, and is named *<design_name>*`.trp`.

Q. Where do I find my programming (JEDEC) file?

A. The programming file can be found in the project directory after command line batch mode processing. The file is located in your project directory, and is named *<design_name>*`.jed`.

Q. Where do I find my back-annotated timing simulation netlist?

A. The back-annotated timing simulation netlist is located in the project directory. The extension of the netlist depends upon the source files used to process your design.

- For VHDL designs, the back-annotated timing simulation netlist is named *<design_name>*`.vho`.
- For Verilog designs, the back-annotated timing simulation netlist is named *<design_name>*`.vo`.
- For EDIF designs, the back-annotated timing simulation netlist is named *<design_name>*`.edo`.

Q. Where do I find my timing delay file?

A. The timing delay file can be found in the project directory after command line batch mode processing. The file is located in your project directory, and is named `<design_name>.sdf`.

EDIF2BLF Command Line Options

The following are EDIF2BLD command line options:

stamppar

Usage: stamppar [options]

Options:

-i [file] — Specifies input file name

-o [file] — Specifies output file name

-tech [file] — Specifies technolegy file name

-log [file] — Specifies log file name

-help — Displays command line options

synpwrap

Usage: synpwrap [options]

Options:

-e [file] — Specifies command file name

-rem — Removes header file

-part — Specifies part name

-target — Specifies device name

mblifopt

Usage: mblifopt [-i] input_file {options}

Options:

-collapse [all | none | fmax] — Speed/area/fmax mode node collapse

-errlog err_file — Specifies error file name

-help — Displays command line option

-sweep — Promote nodes

-keepwires — Collapse wires

-mergefb — Merge feedbacks

-o output_file[.bl1 | .tt2] — Specifies output file name

-oxrf cross_reference_file[.xrf] — Specifies cross reference file name

-pla — Output type: .bl? | .tt?

-pterms n — Maximum product terms for every equation

-plimit n — Pterms limit for every cluster

-nmax n — Maximum fanin for every equation

-nfanlimit n — Fanin limit for every cluster

-areamax n — PLD MC count for reference

-delaymax n — Logic level

-cluster n — Pterms per MC

-nxor — No exor gate

-nxt — No exor on T-FF

-xorsyn — Exor synthesis

-x1 n — Maximum pterms for exor gate x1 leg

-reduce none | [group|bypin] [fixed|choose] [exact]

group: term sharing optimization (FPLA)

bypin: single output optimization (PLD, FPGA)

fixed: polarity determined by 'pos' and 'neg'

choose: automatic polarity selection

exact: quines McCluskey minimization algorithm

Group Fixed is default for FPLA, Bypin Choose for other PLDs and FPGAs

abelvci

Program to read logic optimization options in `*.lci` (Lattice Constraint Input) file and write to command line files for the following programs

1. mdiofft — for D/T FF transformation

   Usage: abelvci -vci constraint_input_file -diofft response_file_name(.d0)

2. mblifopt — for no reduce, node_collapse

   Usage: abelvci -vci constraint_input_file –blifopt response_file_name(.b2_)

3. prefit — for on/off set selection, xor synthesis, equation splitting ...etc.

   Usage: abelvci -vci constraint_input_file –prefit response_file_name(.l0)

mdiofft

Usage: mdiofft [-i] input_file -args transformation_rule {options}

Options:

-o output_file

-define [Y|N] signal_list

-device device_name

-idevice device_file

-log log_file

-oxrf cross_reference_file[.xrf]

-pla

-cluster n

-help

mblflink

Usage: mblflink [-i] top_module -mod submod0...submodN {options}

Options:

-errlog err_file[.err]

-help

-ipo (Instantiate primitives only)

-lib blif_library(ies)

-o flattened_design[.bl2]

prefit

Usage: -input filename.tt2 [options]

Options :

-ck [On|off|min]

-ce [On|off|min]

-ar [On|off|min]

-ap [On|off|min]

-oe [On|off|min]

— Control equations synthesis for ck: clock; ce: clock enable; ar: reset; ap:preset; oe: output enable.

on: allow 1 product term in ON set of the control equation;

off: allow 1 product term in OFF set of the control equation;

min: allow 1 product term in the minimum of ON or OFF set equation;

Default to ON set for all control equations.

-output filename (output BLIF/BPLA file, default follows input file).

-blif (output BLIF file `*.bl*` instead of BPLA file `*.tt*`).

-log filename (default to stdout).

-err filename (default to stderr).

-mod modulename.

-ff_inv [yes|No]: specifies the flipflop is before the inverter. FF is after the inverter by default.

-sr [Yes|no]: swaps Set/Reset to follow output polarity, effective when "-ff_inv yes".

-percent filename: applies the percent sign '%' to invert the logic. and output an additional PLA file without the inversion character '%'.

-split <maxpt>: maxpt : maximum number of product terms per equation, default to 16.

-share <num>: num: minimum number of equations and product terms to share for equation splitting, default to 2 when -split is specified.

-clust <num>: num: number of PT per cluster.

-speed [yes|No]: splits equation with speed priority.

-reduce none.

-noopt (no optimization at all).

-invert [Yes|no]: allows inversion on outputs for Demorgan.

-ifb [Yes|no]: uses internal feedback for combinatorial pin feedback.

-xor [on|Off]: on : turns on EXOR synthesis; off: turn off EXOR synthesis;

-x1 <num>: num: number of PT on X1 cluster, default to 1.

-nxt (No Xor synthesis on T-FF).

-nowrap: prevents line wrapping in the output pla file.

@option_file.

pmtool.exe

[file] -project file name (`*.syn`)

simcp.exe

[@simcp._sp]

The command options are stored in the response file. Its format is:

Simcp.[pre1|pre2|pre4|pre5|pre6|post1|post2] ini simcpls.ini [unit|max]
simcp.[pre1|pre2|pre4|pre5|pre6|post1|post2] cfg [fdk_file] [lts file name|ltv file name] map [lsi file name]

latls.exe

Usage: latls -c cmdfile [-p pc] [-p pr] [-p sc] [-p sr] [-i (dc|un)] -(e|f|t|n|v|a|o) [-h] [-m namemapfile]

-c : leading the cmd filename (`*.lts`)

-p pc: allow don't care propagate through non-memory primitve

-p pr: allow don't care propagate through memory primitve

-p sc: not allow don't care propagate through non-memory primitve

-p sr: not allow don't care propagate through memory primitve

-n : netlist entry is EDIF format and file ext. is edf

-e : netlist entry is EDIF format and file ext. is edn

-t : netlist entry is SIM format

-f : netlist entry is BLIF format

-v : netlist entry is TT4 format

-a : netlist entry is TTE format

-o : netlist entry is EDO format

-h : open switch of hazard report

-m : leading the name mapping file

-i dc : decide the initial value as don't care\n"

-i un : decide the initial value as unknown, default setting

-i zr : decide the initial value as zero

-i on: decides the initial value as one

lpfmx, lpf, lpf4k

Usage: lpfmx, lpf, lpf4k [options]

Options:

-i [file] — Specifies input file name

-lci [file] — Specifies constraint input file name

-d [device] — Specifies device name

-lco [file] — Specifies constraint output file name

-fti [file] — Specifies FTI file name

-fmt — Specify the format of the tto file. See -tto option. Valid formats are PLA, HURL or BLIF.\n\

-tto [file] — Specifies output TT4 file name

-eqn [file] — Specifies output eq3 file name

-tmv [file] — Specifies tmv file name

-html_rpt — Output HTML fitter report

vlog2jhd : <verilog source> -p <library definition dir> -predefine <predirective file>

vhd2jhd : <vhdl file> -o <jhd file> -m <library map file> -p <library root dir>

synsvf: -exe <svf generator path> -prj <project name> -if <jed ile> -j2s|-j2i -log <log file> -gui

j2s: jedec to svf; j2i: jedec to ieee 1532

impsrc

Usage: impsrc.exe -prj <prj name> -lci <lct file> -log <log file> -err <errout file> -tti <input netlist> -dir <prj path>

synedit

Usage: synedit [options]

Options: filename

synview

Usage: synview [options]

Options: filename

lpfgdx

Usage: lpfgdx [options]

Options:

-inp [file] — Specifies input netlist file name

-lci [file] — Specifies constraint input file name

-d [device] — Specifies device name

-lco [file] — Specifies constraint output file name

-fti [file] — Specifies FTI file name

-jed [jedfile] — Specify the JEDEC file name.

-eqn [eqnfile] — Write out an equation file

wrap

Usage: wrap.exe -i gdf_file -p device -src type [-simport exffile]

-i gdf_file — Specifies GDF file

-p device — GDX device type

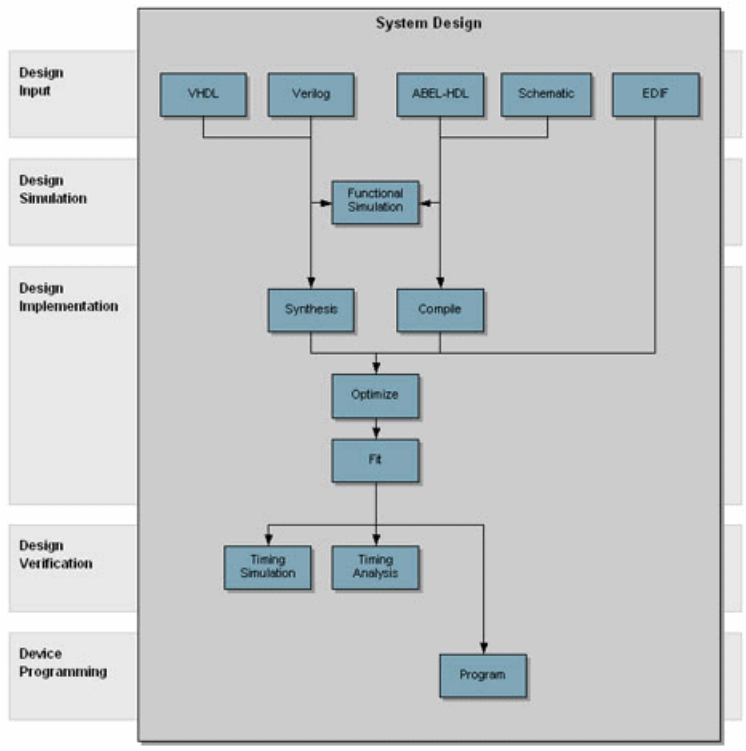
-src type — HDL : project type is HDL; GDF: project type is GDF

-simport exffile — Specifies EXF file. Not available id src type is GDF.

CHAPTER 4 *CPLD Process Flow*

Introduction

CPLD Design



Lattice's ispLEVER® 4.0 is a new generation of design tool that provides a complete system for CPLD design including all ispMACH, ispLSI, and MACH 5 devices.

The ispLEVER software includes a fully integrated, push-button design environment and advanced features for schematic and HDL design entry, functional simulation, synthesis, implementation, optimization, and debug. Furthermore, integrated third-party tools let you get an accurate and complete simulation of your design.

Supported Device Families

- ispLSI
- ispMACH
- MACH 5

Overview of ispLEVER for CPLD

The ispLEVER program offers an integrated environment consisting of several tools necessary to implement Lattice CPLD devices. These tools are briefly described in the following paragraphs and covered in detail in their respective Help. They are listed in alphabetical order.

Constraint Editor

The Constraint Editor lets you specify pin and node location assignments, group assignments, I/O types settings, power level settings, resource reservations, PLL attributes, as well as output slew rates and JEDEC file options. The Constraint Editor reads the constraint file and displays the constraint settings. Modifications to the constraint file are made via the function dialogs. See the Constraint Editor Help for more information about this tool.

Hierarchy Browser

The Hierarchy Browser allows you to navigate through a design consisting of any combination of schematic and HDL modules. In contrast with the Hierarchy Navigator, the Hierarchy Browser works with designs whose top level is either a schematic or HDL source. Additionally, you can cross probe between design sources and their appropriate tool. See the Hierarchy Navigator/Browser Help for more information about this tool.

Hierarchy Navigator

The Hierarchy Navigator combines all the components of a multi-level design for viewing and analysis. You can traverse the full design, viewing each component in its full hierarchical context. See the Hierarchy Navigator/Browser Help for more information about this tool.

ispEXPLORER

The ispEXPLORER lets you run multiple passes of your design using different combinations of Fitter/Optimizer settings and critical timing constraints to achieve the best solution. Results are summarized in a single spreadsheet and detailed reports for each run are accessible. See the ispEXPLORER Help for more information about this tool.

Lattice Logic Simulator

Lattice Logic Simulator performs logic simulation on your design before you implement it into a Lattice device. You can observe not only the gate-level behavior at its inputs and outputs, but also the behavior of internal nodes. See the Lattice Logic Simulator Help for more information about this tool.

LeonardoSpectrum for Lattice

The LeonardoSpectrum™ synthesis environment from Mentor Graphics® lets you can create Lattice CPLD device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process. LeonardoSpectrum combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features. See the LeonardoSpectrum for Lattice documentation supplied by the manufacturer for more information about this tool.

Library Manager

The Library Manager manages libraries of symbols that are used in Schematic Editor. The Library Manager lets you browse through the libraries and lets you maintain the libraries by adding, deleting, copying, or renaming the symbol files in the libraries. See the Library Manager Help for more information about this tool.

ModelSim for Lattice

The ispLEVER software supports third-party HDL simulation and verification with ModelSim™ from Mentor Graphics®. Using this integrated package, you can simulate single Verilog or VHDL designs within one environment. See the ModelSim for Lattice documentation supplied by the manufacturer for more information about this tool.

Optimization Constraint Editor

The Optimization Constraint Editor lets you specify the global constraints used in optimization. It reads the constraint file and displays the constraint settings in the Opt Global Constraints sheet. You can directly modify the optimization constraints in the sheet. The Optimization Constraint Editor is opened prior to optimization, whereas the Constraint Editor is opened after optimization. See the Optimization Constraint Editor Help for more information about this tool.

Performance Analyst

The Performance Analyst analyzes the performance of your design after it has been optimized and implemented by the Fitter. See the Performance Analyst Help for more information about this tool.

Pin Migration Tool

The Pin Migration Tool is used to migrate pin assignment from an old device to a new one that uses the similar package. When you want to implement an old design onto a new device with the similar package, the tool will suggest alternative devices that are in similar pin packages, generate a new project targeted at the selected new device, and convert all compatible pin assignment of the old project to the new one. After the migration, all you have to do is to open the new project, re-assign any pins that are released, and re-compile the design. See the Pin Migration Tool Help for more information about this tool.

Project Navigator

The Project Navigator is the primary interface for ispLEVER and provides an integrated environment for managing the project elements and processes, as well as accessing all ispLEVER tools. See the Project Navigator Help for more information about this tool.

Report Viewer

You can use the Report Viewer to view, but not edit, the various report files generated by ispLEVER. See the Report Viewer Help for more information about this tool.

Schematic Editor

The Schematic Editor is the ispLEVER schematic entry tool. It lets you create schematic (`.sch`) files that can represent a complete design or any component of a hierarchical design. See the Schematic Editor Help for more information about this tool.

Symbol Editor

The ispLEVER software comes with a standard symbol library. Use the Symbol Editor to create symbols or primitive elements that represent an independent schematic module. You can also use the Symbol Editor to create decorative symbols, such as title blocks. See the Symbol Editor Help for more information about this tool.

Synplify for Lattice

Synplify® for Lattice is a logic synthesis tool for CPLD devices, developed by Synplicity®. Synplify starts with high-level designs written in Verilog or VHDL hardware description languages (HDLs). Synplify then converts the HDL into small, high-performance, design netlists that are optimized for Lattice devices. See the Synplify for Lattice documentation supplied by the manufacturer for more information about this tool.

Tcl Editor

The Tcl Editor is a design tool that allows you to create, edit, and run a series of Tool Control Language (TCL) commands. These commands invoke individual ispLEVER processes for generating a complete programmable logic solution. The default text editor permits you to edit TCL code, and it highlights key TCL language elements in different colors to clarify the nature and use of each language element. You can generate a Tcl script for a project in Project Navigator, open it in the Tcl Editor, edit the script, and run it. The Tcl Editor, together with the robust features of the language, gives you more control over the design environment. See the Tcl Editor Help for more information about this tool.

Text Editor

The Text Editor is the ispLEVER text entry tool. You use this tool to create and edit text-based files, such as ABEL-HDL files, test stimulus files, and project documentation files. See the Text Editor Help for more information about this tool.

Waveform Editor

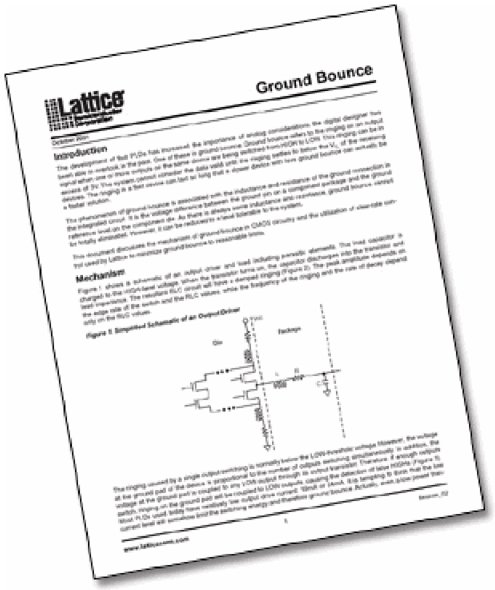
The Waveform Editor lets you create the stimulus graphically, by clicking and dragging with the mouse. You see exactly what each waveform will look like, as well as its timing relationship to all the other waveforms. See the Waveform Editor Help for more information about this tool.

Waveform Viewer

The Waveform Viewer lets you view the results of simulation. You can display the waveform of any net in your design. The Waveform Viewer is fully interactive with the Hierarchy Navigator; clicking on a net in the schematic automatically displays its waveform. See the Waveform Viewer Help for more information about this tool.

CPLD Application Notes

The Lattice Semiconductor web site lists several Application Notes for CPLD devices.



Ground Bounce

Introduction

Mechanism

Figure 1. Simplified Schematic of an Output Driver

www.latticesemi.com

Design Entry

ABEL-HDL Design Entry

ABEL-HDL is a hierarchical hardware description language that supports a variety of behavioral input forms, including high-level equations, state diagrams, and truth tables.

The ispLEVER ABEL-HDL compiler and supporting software functionally verify ABEL-HDL designs through simulation. The compilers then implement the designs in a programmable IC. ABEL-HDL designs can also be transferred to other design environments through standard format, design transfer files.

***Note**: ABEL-HDL design entry can be used for Lattice CPLD devices only.*

Using a Template to Create an ABEL-HDL Source

This procedure describes how to enter an ABEL-HDL design description using a template to create a single ABEL-HDL module. You can also use this procedure for creating an ABEL Test Vector file.

To create a template for an ABEL-HDL source file:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. Select **ABEL-HDL Module** and click **OK**. The Text Editor opens, and a dialog box prompts you for a module name, file name, and title.
3. Type a Module Name, which is the name of the MODULE statement. The MODULE statement is required. It defines the beginning of the module and must be paired with an END statement. The MODULE statement also indicates whether any module arguments are used.
4. Type a File Name. The file extension can be omitted.

   ***Note**: The module name and file name should have the same base name. (The base name is the name without the 3-character extension.) If the module and file names are different, some automatic functions in the Project Navigator might fail to run properly.*

5. Optionally, type a descriptive Title for the module.
6. When you have finished typing the information, click **OK**. You now have a template for an ABEL-HDL source file.

Entering Declarations

The Declarations section specifies the names and attributes of signals used in the design; defines constants, macros, and states; declares lower-level modules and schematics; and optionally declares a device. Each module must have at least one DECLARATIONS section, and declarations affect only the module in which they are defined. If a TITLE statement exists in the template file, enter these statements after the TITLE statement.

Using the andff module as an example, the following describes the DECLARATIONS statement for the three inputs (two AND gate inputs and the clock) and the output.

DECLARATIONS

These two statements declare four signals (`input_1`, `input_2`, `Clk`, and `output_q`).

```
input_1, input_2, Clk pin;
output_q pin istype 'reg';
```

***Note**: ABEL-HDL does not have an explicit declaration for inputs and outputs. Whether a given signal is an input or an output depends on how it is used in the design description that follows. The signal output_q is declared to be type 'reg', which implies that it is a registered output pin. The actual behavior of output_q, however, is specified using one or more equations.*

Entering Logic Descriptions

You can use Equations, a State diagram, or a Truth table to describe your logic design. The following EQUATIONS statement describes the actual behavior of the andff example module.

EQUATIONS

These two equations define the data to be loaded on the registered output, and define the clocking function for the output.

```
output_q := input_1 & input_2;
output_q.clk = Clk;
```

Entering Test Vectors

The traditional method for testing ABEL-HDL designs is to use test vectors. Test vectors are sets of input stimulus values and corresponding expected outputs that can be used with both Equation and JEDEC simulators.

You can specify test vectors in two ways: in the ABEL-HDL source file, or in an external ABEL Test Vector File (`.abv`). When you specify the test vectors in the ABEL-HDL source file, the ispLEVER software creates a “dummy” ABV file that points to the ABEL-HDL source containing the vectors. This file is necessary because an ABV file is required in order to have access to the Equation and JEDEC simulation processes.

To continue with the example andff module, the following describes the TEST_VECTORS statement.

TEST_VECTORS

```
([Clk, input_1 , input_2] -> output_q)
[ 0 , 0 , 0 ] -> 0;
[.C., 0 , 0 ]  -> 0;
[.C., 0 , 1 ]  -> 0;
[.C., 1 , 1 ] -> 1;
```

ABEL-HDL Source File Examples

The examples in this section are representative of programmable logic applications and serve to illustrate significant ABEL-HDL features. You can use these examples to get started creating your own source files.

All the examples in this section are installed with your software. You can use them without making any changes, or you can modify them in your designs. You will find the examples in your *<install_path>*\ispcpld\examples\cpld**abel** directory.

Equations

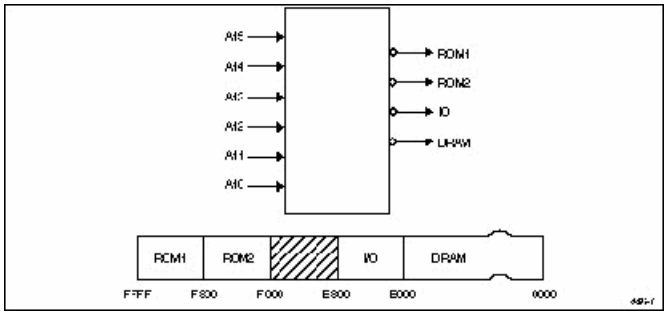
Memory Address Decoder

Address decoding is a typical application of programmable logic devices, and the following describes the ABEL-HDL implementation of such a design.

Design Specification

The following figure shows the block diagram for this design and a continuous block of memory divided into sections containing dynamic RAM (DRAM), I/O (IO), and two sections of ROM (ROM1 and ROM2). The purpose of this decoder is to monitor the 6 high-order bits (A15-A10) of a sixteen-bit address bus and select the correct section of memory based on the value of these address bits. To perform this function, a simple decoder with six inputs and four outputs is designed for implementation in a simple PLD.

Block Diagram: Memory Address Decoder



The address ranges associated with each section of memory are shown below. These address ranges can also be seen in the source file in *Simplified Block Diagram: Memory Address Decoder*, below.

Memory Section	Address Range (hex)
DRAM	0000-DFFF
I/O	E000-E7FF
ROM2	F000-F7FF
ROM1	F800-FFFF

Design Method

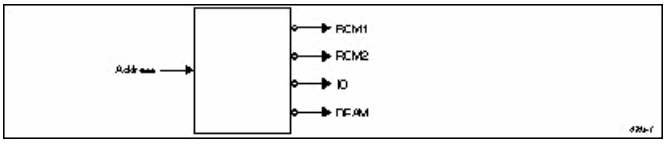
The following figure shows a simplified block diagram for the address decoder. The decoder is implemented with equations employing relational and logical operators, as shown in the Memory Address Decoder Source File example below.

Significant simplification is achieved by grouping the address bits into a set named Address. The ten address bits that are not used for the address decode are given no-connect values in the set. This indicates that the address in the overall design (that beyond the decoder) contains 16 bits, but that bits 0 to 9 do not affect the decode of that address and are not monitored. In contrast, defining the set as

```
Address = [A15,A14,A13,A12,A11,A10]
```

ignores the existence of the lower-order bits. Specifying all 16 address lines as members of the address set allows full 16-bit comparisons of the address value against the ranges shown above.

Simplified Block Diagram: Memory Address Decoder



Memory Address Decoder Source File

```
module decode
title 'memory decode'

A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
ROM1,IO,ROM2,DRAM pin 14,15,16,17 istype 'com';
H,L,X = 1,0,.X.;
Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];

equations
!DRAM = (Address <= ^hDFFF);
!IO = (Address >= ^hE000) & (Address <= ^hE7FF);
!ROM2 = (Address >= ^hF000) & (Address <= ^hF7FF);
!ROM1 = (Address >= ^hF800);

test_vectors
(Address -> [ROM1,ROM2,IO,DRAM])
^h0000 -> [ H, H, H, L ];
^h4000 -> [ H, H, H, L ];
^h8000 -> [ H, H, H, L ];
^hC000 -> [ H, H, H, L ];
^hE000 -> [ H, H, L, H ];
^hE800 -> [ H, H, H, H ];
^hF000 -> [ H, L, H, H ];
^hF800 -> [ L, H, H, H ];

end
```

Test Vectors

In this design, the test vectors are a straightforward listing of the values that must appear on the output lines for specific address values. The address values are specified in hexadecimal notation.

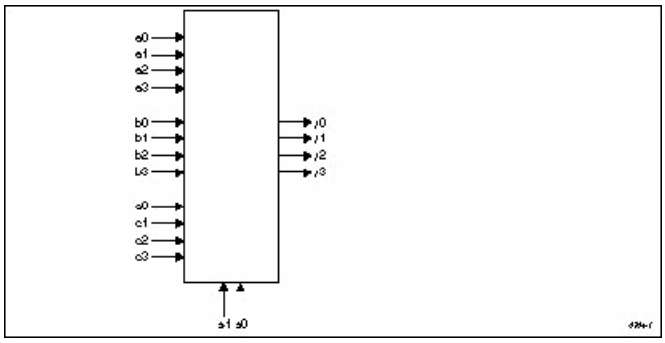
12-to-4 Multiplexer

The following describes the implementation of a 12-input to 4-output multiplexer using high level equations.

Design Specification

The figure below shows the block diagram for this design. The multiplexer selects one of the four inputs and routes that set to the output. The inputs are a0-a3, b0-b3, and c0-c3. The outputs are y0-y3. The routing of inputs to outputs is straightforward: a0 or b0 or c0 is routed to the output y0, a1 or b1 or c1 is routed to the output y1, and so on with the remaining outputs. The select lines, s0 and s1, control the decoding that determines which set is routed to the output.

Block Diagram: 12-to-4 Multiplexer



Design Method

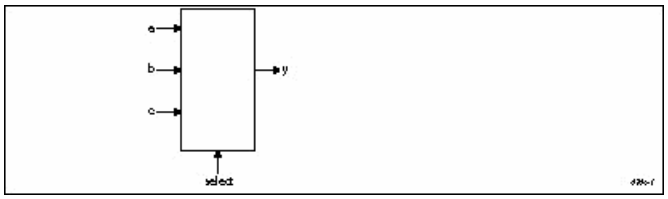
The figure below shows a block diagram for the same multiplexer after sets have been used to group the signals. All of the inputs have been grouped into the sets a, b, and c. The outputs and select lines are grouped into the sets, *y* and *select*, respectively. This grouping of signals into sets takes place in the declaration section of the source file listed in Example - 12-to-4 Multiplexer.

When the sets have been declared, specification of the design is made with the following four equations that use **When-then** statements.

```
when (select == 0) then y = a;
when (select == 1) then y = b;
when (select == 2) then y = c;
when (select == 3) then y = c;
```

The relational expression (==) inside the parentheses produces an expression that evaluates to true or false value, depending on the values of s0 and s1.

Simplified Block Diagram: 12-to-4 Multiplexer



In the first equation, this expression is then ANDed with the set *a*, which contains the four bits, a0-a3, and could be written as

```
y = (select == 0) & a
```

Assume select is equal to 0 (s1 = 0 and s0 = 0), so a true value is produced. The true is then ANDed with the set *a* on a bit by bit basis, which in effect sets the product term to *a*. If select were not equal to 0, the relational expression inside the parentheses would produce a false value. This value, when ANDed with anything, would give all zeroes.

The other product terms in the equation work in the same manner. Because select takes on only one value at a time, only one of the product terms passes the value of an input set along to the output set. The others contribute 0 bits to the ORs.

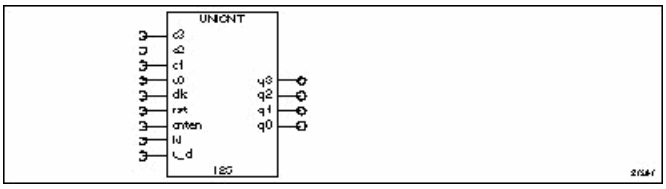
Test Vectors

The test vectors for this design are specified in terms of the input, output, and select sets. Note that the values for a set can be specified by decimal numbers and by other sets. The constants H and L, used in the test vectors, were declared as four-bit sets containing all ones or all zeroes.

4-Bit Universal Counter

The following design describes the implementation of a 4-bit up/down counter with parallel load and count enable. The design is described using high-level ABEL-HDL equations. The figure below shows a block diagram of the counter and its signals. Example – 4-bit Universal Counter shows the source file for this design.

Block Diagram: 4-bit Universal Counter



The outputs q3, q2, q1, and q0 contain the current count. The least significant bit (LSB) is q0, and the most significant bit (MSB) is q3.

Using Sets to Create Modes

The counter has four different modes of operation: Load Data From Inputs, Count Up, Count Down, and Hold Count. You select the modes by applying various combinations of values to the inputs **cnten**, **ld**, and **u_d**, as described below. The four modes have different priorities, which are defined in the ABEL-HDL description.

The Load mode has the highest priority. If the **ld** input is high, then the **q** outputs reflect the value on the **d** inputs after the next clock edge.

The Hold mode has the next highest priority. Provided **ld** is low, then when the **cnten** input is low, the **q** outputs maintain their current values upon subsequent clock edges, ignoring any other inputs.

The Up and Down modes have the same priority and by definition are mutually exclusive. Provided **cnten** is high and **ld** is low, then when **u_d** is high, the counter counts up; when **u_d** is low, the counter counts down.

Counter Reset

The counter is reset asynchronously by assertion of the input **rst**.

Using Range Operators

Because this design uses range operators and sets, you can modify the counter to be any width by making changes in the declarations section. You could create a 9-bit counter by changing the lines which read “d3..d0” and “q3..q0” to “d8..d0” and “q8..q0,” respectively. The range expressions are expanded and create register sets of corresponding width.

Design Description

Hierarchical Interface Declaration

Directly after the module name, the design contains a hierarchical interface declaration that is used by the ABEL-HDL compiler and linker if another ABEL-HDL source instantiates this source. The interface list shows all of the input, output, and bidirectional signals (if any) in the design.

Declarations

The declarations contain sections that make the design easier to interpret. The sections are as follows:

Constants	Constant values are defined.
Inputs	Design inputs are declared.
Outputs	The output pin list contains an **istype** declaration for retargetability.
Sets	The names data and count are defined as sets (groups) containing the inputs d3, d2, d1, and d0, and the outputs q3, q2, q1, and q0, respectively.
Modes	The "Mode equations" are actually more Constants declarations. First MODE is defined as the set containing **cnten**, **ld**, and **u_d**, in that order. Next, LOAD is defined as being true when the members of MODE are equal to X, 1, and X, respectively. HOLD, UP, and DOWN are defined similarly.

Equations

The design of the counter equations enables you to easily define modes and your actual register equations will be easily readable. The counter equation uses **When-then-else** syntax. The first line

```
when LOAD then count := data
```

uses the symbolic name LOAD, defined earlier in the source file as

```
LOAD = (MODE == [X, 1, X])
```

and MODE itself is a set of inputs in a particular order, defined previously as

```
MODE = [cnten, ld, u_d]
```

The first line of the equation could have been written as

```
when ((cnten == X) & (ld == 1) & (u_d == X)) then count := data
```

which is functionally the same, but the intermediate definitions used instead make the source file more readable and easier to modify.

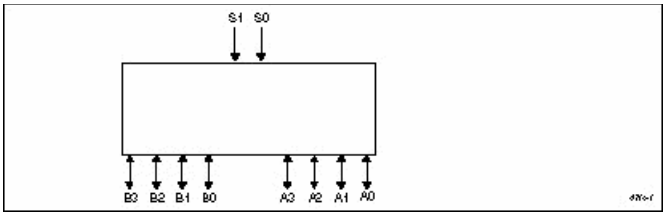
Bidirectional Three-state Buffer

A four-bit bidirectional buffer with tristate outputs is presented here. The design is implemented in an F153 FPLA with bidirectional inputs/outputs and programmable output polarity. Simple Boolean equations are used to describe the function.

Design Specification

The figure below shows a block diagram for this four-bit buffer. Signals A0-A3 and B0-B3 function both as inputs and outputs, depending on the value on the select lines, S0-S1. When the select value (the value on the select lines) is 1, A0-A3 are enabled as outputs. When the select value is 2, B0-B3 are enabled as outputs. (The choice of 1 and 2 for select values is arbitrary.) For any other values of the select lines, both the A and B outputs are at high impedance. Output polarity for this design is positive.

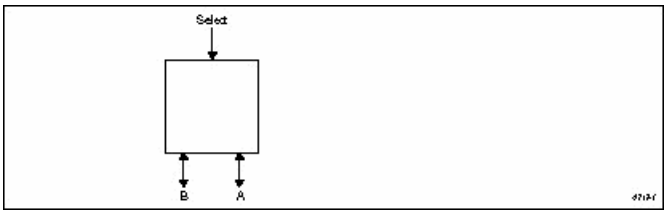
Block Diagram: Bidirectional Three-state Buffer



Design Method

A simplified block diagram for the buffer is shown in the figure below. The A and B inputs/outputs are grouped into two sets, A and B. The select lines are grouped into the select set. The *source file example: Bidirectional Three-state Buffer* below shows the source file that describes the design.

Simplified Block Diagram: Bidirectional Three-state Buffer



High-impedance and don’t-care values are declared to simplify notation in the source file. The equations section describes the full function of the design. What appear to be unresolvable equations are written for A and B, with both sets appearing as inputs and outputs. The enable equations, however, enable only one set at a time as outputs; the other set functions as inputs to the buffer.

Test vectors are written to test the buffer when either set is selected as the output set and for the case when neither is selected. The test vectors are written in terms of the previously declared sets so the element values do not need to be listed separately.

Source file example: Bidirectional Three-state Buffer

```
module tsbuffer
title 'bidirectional three state buffer'

S1,S0 Pin 1,2; Select = [S1,S0];
A3,A2,A1,A0 Pin 12,13,14,15; A = [A3,A2,A1,A0];
B3,B2,B1,B0 Pin 16,17,18,19; B = [B3,B2,B1,B0];

X,Z = .X., .Z.;

equations
A = B;
B = A;

A.oe = (Select == 1);
B.oe = (Select == 2);

test_vectors
([Select, A, B]-> [ A, B])
```

```
[ 0 , 0, 0]-> [ Z, Z];
[ 0 , 15, 15]-> [ Z, Z];

[ 1 , X, 5]-> [ 5, X];
[ 1 , X, 10]-> [ 10, X];

[ 2 , 5, X]-> [ X, 5];
[ 2 , 10, X]-> [ X, 10];
[ 3 , 0, 0]-> [ Z, Z];
[ 3 , 15, 15]-> [ Z, Z];

end
```

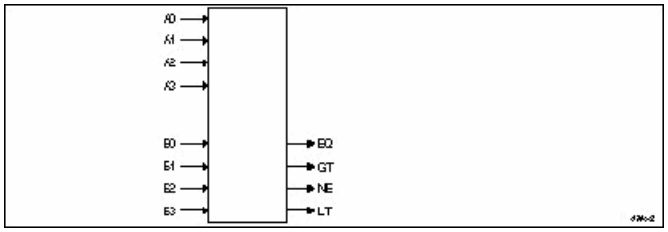
4-Bit Comparator

This is a design for a 4-bit comparator that provides an output for “equal to,” “less than,” “not equal to,” and “greater than” (as well as intermediate outputs). The design is implemented with high level equations.

Design Specification

The comparator, as shown the figure below, compares the values of two four-bit inputs (A0-A3 and B0-B3) and determines whether A is equal to, not equal to, less than, or greater than B. The result of the comparison is shown on the output lines, EQ, GT, NE, and LT.

Block Diagram: 4-bit Comparator



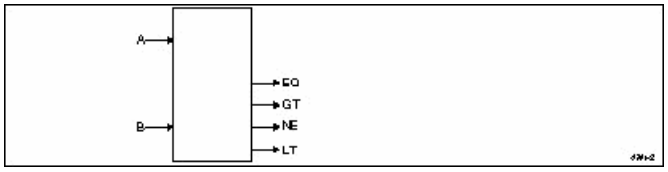
Design Method

The two figures below show the simplified block diagram and source file listing for the comparator. The inputs A0-A3 and B0-B3 are grouped into the sets A and B. YES and NO are defined as 1 and 0, to be used in the test vectors.

The equations section of the source file contains the following equations:

```
EQ = A == B;
NE = !(A == B);
GT = A > B;
LT = !((A > B) # (A == B));
```

Simplified Block Diagram: 4-bit Comparator



You could also use the following equations for the design of this comparator. However, many more product terms are used in the FPLA:

```
EQ = A == B;
NE = A != B;
GT = A > B;
LT = A < B;
```

The first set of equations takes advantage of product term sharing within the target FPLA, while the latter set requires a different set of product terms for each equation. For example, the equation

```
NE = !(A == B);
```

uses the same 16 product terms as the equation

```
EQ = A == B;
```

thereby reducing the number of product terms. In a similar manner, the equation

```
LT = !((A > B) # (A == B));
```

uses the same product terms as equations

```
EQ = A == B;
GT = A > B;
```

whereas the equation

```
LT = A < B;
```

(in the second set of equations) requires the use of additional product terms. Sharing product terms in devices that allow this type of design architecture can serve to fit designs into smaller and less expensive logic devices.

Source File: 4-bit Comparator

```
module comp4a
title '4-bit look-ahead comparator'

A3..A0 pin 1..4;
A = [A3..A0];
B3..B0 pin 5..8;
B = [B3..B0];

NE,EQ,GT,LT pin 16..19 istype 'com';

No,Yes = 0,1;

equations
EQ = A == B;
NE = !(A == B);
GT = A > B;
LT = !((A > B) # (A == B));
" test_vectors deleted...

end
```

Test Vectors

Three separate test vectors sections are written to test three of the four possible conditions. (The fourth and untested condition of NOT EQUAL TO is simply the inverse of EQUAL TO.) Each test vectors table includes a test vector message that helps make report output from the compiler and the simulators easier to read.

The three tested conditions are not mutually exclusive, so one or more of them can be met by a given A and B. In the test vectors table, the constants YES and NO (rather than 1 and 0) are used for ease of reading. YES and NO are declared in the declaration section of the source file.

Truth Table Examples

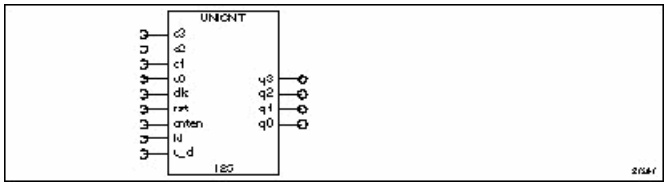
Seven-segment Display Decoder

This display decoder decodes a four-bit binary number to display the decimal equivalent on a seven-segment LED display. The design incorporates a truth table.

Design Specification

The figure below shows a block diagram for the design of a seven-segment display decoder and a drawing of the display with each of the seven segments labeled to correspond to the decoder outputs. To light a segment, the corresponding line must be driven low. Four input lines D0-D3 are decoded to drive the correct output lines. The outputs are named a, b, c, d, e, f, and g, corresponding to the display segments. All outputs are active low. An enable, ena, is provided. When ena is low, the decoder is enabled; when ena is high, all outputs are driven to high impedance.

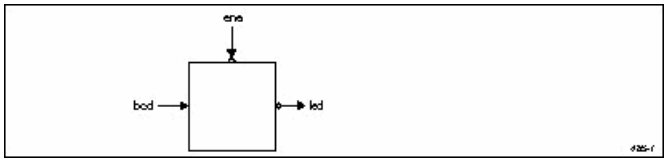
Block Diagram of Seven-segment Display Decoder



Design Method

The figure below and Example - 4-bit Counter with 2-input Mux show the simplified block diagram and the source file for the ABEL-HDL implementation of the display decoder. The binary inputs and the decoded outputs are grouped into the sets **bcd** and **led**. The constants ON and OFF are declared so that the design can be described in terms of turning a segment on or off. To turn a segment on, the appropriate line must be driven low; thus we declare ON as 0 and OFF as 1.

Simplified Block Diagram: Seven-segment Display Decoder



The design is described in two sections: an equations section and a truth table section. The decoding function is described with a truth table that specifies the outputs required for each combination of inputs. The truth table header names the inputs and outputs. In this example, the inputs are contained in the set named bcd and the outputs are in led. The body of the truth table defines the input to output function.

Because the design decodes a number to a seven segment display, values for bcd are expressed as decimal numbers, and values for led are expressed with the constants ON and OFF that were defined in the

declarations section of the source file. This makes the truth table easy to read and understand; the incoming value is a number, and the outputs are on and off signals to the LED.

The input and output values could just as easily have been described in another form. Take, for example, the line in the truth table:

```
5 -> [ ON, OFF, ON , ON, OFF, ON, ON]
```

This could have been written in the equivalent form:

```
[ 0, 1, 0, 1 ] -> 36
```

In this second form, 5 was simply expressed as a set containing binary values, and the LED set was converted to decimal. (Remember that ON was defined as 0 and OFF was defined as 1.) Either form is supported, but the first is more appropriate for this design. The first form can be read as, “the number five turns on the first segment, turns off the second, . . .” whereas the second form cannot be so easily translated into meaningful terms.

Test Vectors

The test vectors for this design test the decoder outputs for the ten valid combinations of input bits. The enable is also tested by setting ena high for the different combinations. All outputs should be at high impedance whenever ena is high.

State Diagram Examples

Three-state Sequencer

The following design is a simple sequencer that demonstrates the use of ABEL-HDL state diagrams. The design is implemented in a P16R4 device. The number of State Diagram states that can be processed depends on the number of transitions and the path of the transitions. For example, a 64-state counter uses fewer terms (and smaller equations) than a 63-state counter. For large counter designs, use the syntax CountA:= CountA + 1 to create a counter rather than using a state machine. See also example COUNT116.abl for further information on counter implementation.

Design Specification

The figure below shows the sequencer design with a state diagram that shows the transitions and desired outputs. The state machine starts in state A and remains in that state until the ‘start’ input becomes high. It then sequences from state A to state B, from state B to state C, and back to state A. It remains in state A until the 'start' input is high again. If the ‘reset’ input is high, the state machine returns to state A at the next clock cycle. If this reset to state A occurs during state B, a ‘halt’ synchronous output goes high, and remains high until the machine is again started.

During states B and C, asynchronous outputs ‘in_B’ and ‘in_C’ go high to indicate the current state. Activation of the ‘hold’ input will cause the machine to hold in state B or C until ‘hold’ is no longer high, or ‘reset’ goes high.

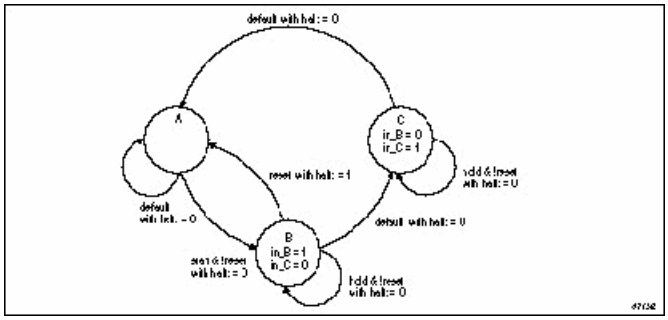
Design Method

The sequencer is described by using a STATE_DIAGRAM section in the source file. Example – Three-state sequencer shows the source file for the sequencer. In the source file, the design is given a title, the device type is specified, and pin declarations are made. Constants are declared to simplify the state diagram notation. The two state registers are grouped into a set called 'sreg' and the three states (A, B, and C) are declared, with appropriate values specified for each.

The state values chosen for this design allow the use of register preload to ensure that the machine starts in state A. For larger state machines with more state bits, careful numbering of states can dramatically reduce the logic required to implement the design. Using constant declarations to specify state values saves time when you make changes to these values.

The state diagram begins with the STATE_DIAGRAM statement that names the set of signals to use for the state register. In this example, 'sreg' is the set of signals to use.

State Diagram: Three-state Sequencer



Within the STATE_DIAGRAM, IF-THEN-ELSE statements are used to indicate the transitions between states and the input conditions that cause each transition. In addition, equations are written in each state that indicate the required outputs for each state or transition.

For example, state A reads:

```
State A:
in = 0;
in_C = 0;
if (start & !reset) then B with
halt := 0;
else A with halt := halt;
```

This means that if the machine is in state A, and **start** is high but **reset** is low, it advances to state **B**. In any other input condition, it remains in state **A**.

The equations for **in_B** and **in_C** indicate that those outputs should remain low while the machine is in state **A**. The equations for **halt,** specified with the **with** keyword, indicate that **halt** should go low if the machine transitions to state **B** but should remain at its previous value if the machine stays in state **A**.

Test Vectors

The specification of the test vectors for this design is similar to other synchronous designs. The first vector is a preload vector, to put the machine into a known state (state A), and the following vectors exercise the functions of the machine. The A, B, and C constants are used in the vectors to indicate the value of the current state, improving the readability of the vectors.

Combined Logic Descriptions

This section contains an advanced logic design and builds on examples and concepts presented in the earlier sections of this Help. This design, a blackjack machine, is the combination of more than one basic logic design. Design specification, methods, and complete source files are given for all parts of the blackjack machine example, which contains the following logic designs:

- Multiplexer
- 5-bit adder
- Binary to BCD converter
- State machine

This example is a classic blackjack machine based on C.R. Clare's design in *Designing Logic Systems Using State Machines* (McGraw Hill, 1972). The blackjack machine plays the dealer's hand, using typical dealer strategies to decide, after each round of play, whether to draw another card or to stand.

The blackjack machine consists of these functions:

- A card reader that reads each card as it is drawn
- Control logic that tells it how to play each hand (based on the total point value of the cards currently held)
- Display logic that displays scores and status on the machine's four LEDs.

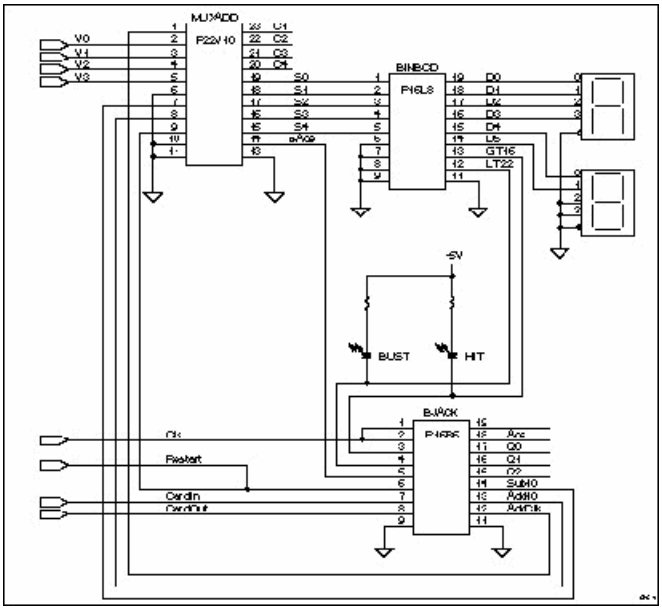
For this example, we are assuming that the two digital display devices used to display the score have built-in seven-segment decoders.

To operate the machine, insert the dealer's card into the card reader. The machine reads the value and, in the case of later card draws, adds it to the values of previously read cards for that hand. (Face cards are valued at 10 points, non-face cards are worth their face value, and aces are counted as either 1 or 11, whichever count yields the best hand.) If the point total is 16 or less, the GT16 line will be asserted (active low) and the Hit LED will light up. This indicates that the dealer should draw another card. If the point total is greater than 16 but less than 22, no LEDs will light up (indicating that the dealer should draw no new cards). If the point total is 22 or higher, LT22 will be asserted (active low) and the Bust LED will light (indicating that the dealer has lost the hand).

The figure below shows the blackjack machine is implemented in three PLDs:

- A multiplexer-adder-comparator, which adds the value of the newly drawn card to the existing hand (and indicates an ace to the state machine).
- A binary to binary-coded-decimal (BCD) converter, which takes in the five-bit binary score and converts it to two-digits of BCD for the digital display.
- The blackjack controller (a state machine that contains the game logic). This logic includes instructions that determine when to add a card value, when to count an ace as 1, and when to count an ace as 11.

Schematic of a Blackjack Machine Implemented in Three PLDs



Circuits that are a straightforward function of a set of inputs and outputs are often most easily expressed in equations; the adder is such a circuit. The PLD for the adder function, identified as MUXADD the above figure, includes three elements: a multiplexer, the adder itself, and a comparator.

The multiplexer selects either the value of the newly dealt card or one of the two fixed values used for the ace (ADD10 or SUB10). The adder adds the value selected by the multiplexer to the previous score when triggered by the clock signal, ADDCLK. The comparator detects when an ace is present and passes this information on to the blackjack controller, BJACK.

Outputs that do not follow a specific pattern are most easily expressed as truth tables. This is the case with the binary-to-BCD converter that is identified in the schematic above as BINBCD. This PLD converts five bits of binary input to BCD output for two digital display elements.

The following text describes the internal logic design necessary to keep the card count, to control the play sequence, and to show the count on the digital display (or the state on the Hit and Bust LEDs). Neither the card reader nor the physical design is discussed here. Assume that the card reader provides a binary value that is representative of the card read.

The design has eight inputs (four of which are the binary encoded card values, V0-V3). The remaining four inputs are signals that indicate the following:

- Restart (the machine is to be restarted)
- CardIn (a card is in the reader)
- CardOut (no card is in the reader)
- CLK (a clock signal to synchronize the design to the card reader)

CardIn, CardOut, and Clk are provided by the card reader. Restart is provided by a switch on the exterior of the machine.

Device	Function in the Blackjack Machine
P22V10	Multiplexer/Adder/Comparator
P16L8	Binary-BCD converter
P16R6	State machine

Design Specification - MUXADD

MUXADD consists of an input multiplexer, an adder, and a comparator. The multiplexer determines what value is added to the current score (by the adder) The added value consists of the contents of the external card reader (V0-V1 declared as Card), a numeric value of +10, or a numeric value of -10.

The inputs Add10 and Sub10 from the controller (state machine) BJACK determine which of the three values the multiplexer selects for application to the adder. Card is applied to the adder when Add10 and Sub10 are active high, as generated by the BJACK controller. When Add10 becomes active low, 10 is added to the current score (to count an ace as 11 instead of 1), and when Sub10 is active low, -10 is added to the current score (to count an ace as 1 instead of 11).

The adder provides an output named Score (S0-S4) which is the sum of the current adder contents and the value selected by the input multiplexer (the card reader contents, +10, or -10). The comparator monitors the contents of the external card reader (Card) and generates an output, is_Ace, to the BJACK controller that signifies that an ace is present in the card reader.

Design Method - MUXADD

MUXADD is implemented in a P22V10, and consists of a three-input multiplexer, a five-bit ripple adder, and a five-bit comparator. These circuit elements are defined in the equations shown in the figure *Schematic of a Blackjack Machine Implemented in Three PLDs* in the topic Combined Logic Descriptions. For the multiplexer inputs, a set named Card defines inputs V0 through V4 as the value of the card reader, while inputs Add10 and Sub10 are used directly in the following equations to define the multiplexer. The multiplexer output to the adder is named Data and is defined by the equations

```
Data = Add10 & Sub10 & Card
# !Add10 & Sub10 & ten
# Add10 & !Sub10 & minus_ten;
```

The adder (MUXADD) contained in the P22V10 is a five-bit binary ripple adder that adds the current input from the multiplexer to the current score, with carry. The adder is clocked by a signal (AddClk) from the BJACK controller and is described with the following equations.

```
Score := Data $ Score.FB $ CarryIn;
CarryOut = Data & Score.FB # (Data # Score.FB) & CarryIn;
Reset = !Clr;
```

In the above equations, Score is the sum of Data (the card reader output, value of ten, or value of minus ten), Score (the current or last calculated score), and CarryIn (the shifted value of CarryOut, described below). The new value of Score appears at the S0 through S4 outputs of MUXADD at the time of the AddClk pulse generated by the BJACK controller (state machine).

Before the occurrence of the AddClk clock pulse, an intermediate adder output appears at combinatorial outputs of the P22V10, labeled C0 through C4 and defined as the set named CarryOut (shown below). A second set named CarryIn defines the same combinatorial outputs as CarryOut, but the outputs are shifted one bit to the left, as shown below.

```
CarryIn = [C4..C1, 0];
CarryOut = [ X,C4..C1];
```

The set declarations define CarryIn as CarryOut with the required shift to the left for application back to adder input. At the time of the AddClk pulse from the BJACK controller, CarryIn is added to Score and Data by an exclusive-or operation.

The comparator portion of MUXADD is defined with

```
is_Ace = Card == 1;
```

which provides an input to the BJACK controller whenever the value provided by the card reader is 1.

Test Vectors - MUXADD

The test vectors shown in the source file example below verify operation of MUXADD by first clearing the adder (so Score is zero), then adding card reader values 7 and 10. The test vectors then input an ace (1) from the card reader (Card) to produce a Score of 1 and pull the is_Ace output high. Subsequent vectors verify the -10 function of the input multiplexer and adder. The **trace** statement lets you observe the carry signals in simulation.

Source File: Multiplexer / Adder / Comparator

```
module MuxAdd
title '5-bit ripple adder with input multiplex'

AddClk,Clr,Add10,Sub10,is_Ace pin 1,9,8,7,14;
V4..V0 pin 6..2;
```

```
S4..S0 pin 15..19;
C4..C1 pin 20..23;
X,C,L,H = .X., .C., 0, 1;
Card = [V4..V0];
Score = [S4..S0];
CarryIn = [C4..C1, 0];
CarryOut = [ X,C4..C1];
ten = [ 0, 1, 0, 1, 0];
minus_ten = [ 1, 0, 1, 1, 0];
S4..S0 istype 'reg' ;
" Input Multiplexer
Data = Add10 & Sub10 & Card
# !Add10 & Sub10 & ten
# Add10 & !Sub10 & minus_ten;
@DISP MARG = equations
Score := Data $ Score.FB $ CarryIn;
CarryOut = Data & Score.FB # (Data # Score.FB) & CarryIn;
Score.ar = !(Clr # Clr);
Score.clk = AddClk;
is_Ace = Card == 1;
" test_vectors edited...
end MuxAdd
```

Design Specification - BINBCD

To display the Score, appearing at the output of MUXADD, a binary to bcd converter is implemented in a P16L8. It is the function of the converter to accept the four lines of binary data generated by MUXADD and provide two sets of binary coded decimal outputs for two bcd display devices; one to display the units of the current score, and the other to display the tens. The four-bit output bcd1 (D0-D3) contains the units of the current score, and is connected to the high-order display digit. The two-bit output bcd2 (D4 and D5) contains the tens, and is fed to the low-order display digit.

BINBCD also provides a pair of outputs to light the Bust and Hit LEDs. Bust is lit whenever Score is 22 or greater; while Hit is lit whenever Score is 16 or less.

Design Method - BINBCD

The design of BINBCD is shown in the source file of Example - BINBCD. The design of the converter is easily expressed with a truth table that lists the value of Score (inputs S0 through S4 are declared as Score) for values of bcd1 and bcd2. bcd1 and bcd2 are sets that define the outputs that are fed to the two digital display devices. The truth table lists Score values up to decimal 31.

The truth table represents a method of expressing the design “manually.” You could use a macro to create a truth table in the following manner:

```
clear(binary);
@repeat 32 { binary - [binary/10,binary%10]; inc(binary);}
```

As indicated in Example - BINBCD (and described in Test Vectors -BINBCD), this macro is used to generate the test vectors for the converter. The generated *.lst file shows the truth table created from the macro.

The BINBCD design also provides the outputs LT22 and GT16 to control the Bust and Hit LEDs. A pair of equations generate an active-high LT22 signal to turn off the Bust LED when Score is less than 22, and an active-high GT16 signal to turn off the Hit LED when Score is greater than 16.

Test Vectors - BINBCD

The test vectors shown in Example - BINBCD verify operation of the LT22 and GT16 outputs of the converter by assigning various values for Score and checking for the corresponding outputs.

The test vectors for the binary to bcd converter are defined by means of the following macro:

```
test_vectors ( score - [bcd2,bcd1])
clear(binary);
@repeat 32 { binary - [binary/10,binary%10]; inc(binary);}
```

This macro generates a test vector with the variable binary set to 0 by the macro (a) {@const ?a=0}; (in the binbcd.abl source file shown in Example - BINBCD followed by 31 vectors provided by the @repeat directive. The 31 vectors are generated by incrementing the value of the variable binary by a factor of 1 for each vector. Refer to the inc macro (a) {@const ?a=?a+1;}; line in Example - BINBCD. On the output side of the test vectors, division is used to create the output for bcd2 (tens display digit), while the remainder (modulus) operator is used to create the output for bcd1 (units display digit).

Design Specification - BJACK

BJACK, the blackjack controller, is technically a state machine (a circuit capable of storing an internal state reflecting prior events). State machines use sequential logic, branching to new states and generating outputs on the basis of both the stored states and external inputs.

In the case of the controller, the state machine stores states that reflect the following blackjack machine conditions:

- The value of Score in one of the decimal value ranges (0 to 16, 17 to 21, or 22+).
- The status of the card reader (card in or card out).
- The presence of an ace in the card reader.

On the basis of these stored states (and input from each new card), the controller decides whether or not a +10 or -10 value is sent to the adder.

Design Method - BJACK

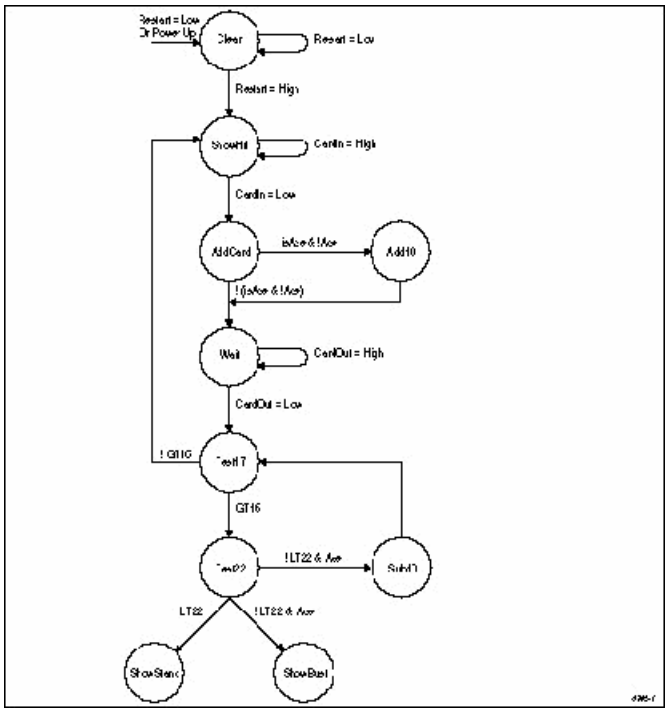
Developing a bubble diagram is the first step in describing a state machine. The pictoral state diagram, below, shows a bubble diagram (pictorial state diagram) for the controller. This bubble diagram indicates state transitions and the conditions that cause those transitions. Transitions are represented by arrows and the conditions causing the transitions are written alongside the arrow.

You must express the bubble diagram in the form shown in the state_diagram in Example – State Machine (Controller). There is a one-to-one correlation between the bubble diagram and the state diagram described in the source file Example – State Machine (Controller). The table below describes the state identifiers (state machine states) illustrated in the bubble diagram and listed in the source file.

State Identifier	Description
Clear	Clear the state machine, adder, and displays.
ShowHit	Indicate that another card is needed. Hit indicator is lit.
AddCard	Add the value at the adder input to the current count.
Add10	Add the fixed value 10 to the current count, effectively giving an ace a value of 11.
Wait	Wait until a card is taken out of the reader.

State Identifier	Description
Test17	Test the current count for a value less than 17.
Test22	Test the current count for a value less than 22.
Sub10	Add the fixed value -10 to the current count, effectively subtracting 10 and restoring an ace to 1.
ShowBust	Indicate that no more cards are needed. Bust indicator is lit.
ShowStand	Indicate that no more cards are needed. Neither Hit nor Bust indicators are lit.

Pictorial State Diagram: Blackjack Machine



Note that in Example – State Machine (Controller), each of the state identifiers (for example, ShowHit) is defined as sets having binary values. These values were chosen to minimize the number of product terms used in the P16R6.

Operation of the state machine proceeds as follows if no aces are drawn:

- If a card is needed from the reader, the state machine goes to state ShowHit.
- When CardIn goes low, meaning that a card has been read, a transition to state AddCard is made. The card value is added to the current score.
- The machine goes to Wait state until the card is withdrawn from the reader.
- The machine goes to Test17 state.
- If the score is less than 17, another card is drawn.
- If the score is greater than or equal to 17, the machine goes to state Test22.
- If the score is less than 22, the machine goes to the ShowStand state.
- If the score is 22 or greater, a transition is made to the ShowBust state.

- In either ShowStand or ShowBust state, a transition is made to Clear (to clear the state register and adder) when Restart goes low.
- When Restart goes back to high, the state machine returns to ShowHit and the cycle begins again.

Operation of the state machine when an ace is drawn is essentially the same. A card is drawn and the score is added. If the card is an ace and no ace has been drawn previously, the state machine goes to state Add10, and ten is added to the count (in effect making the ace an 11). Transitions to and from Test17 and Test22 proceed as before. However, if the score exceeds 21 and an ace has been set to 11, the state machine goes to state Sub10, 10 is subtracted from the score, and the state machine goes to state Test17.

Test Vectors - BJACK

Example – State Machine (Controller) shows three sets of test vectors; each set represents a different "hand" of play (as described above the set of vectors) and tests the different functions of the design. The Restart function is used to set the design to a known state between each hand, and the state identifiers are used instead of the binary values (which they represent).

Hierarchy Examples

Example – Module bjacktop shows how to combine the three blackjack examples into one top-level source for implementation in a larger device.

The three lower-level modules are unchanged from the non-hierarchical versions and still include their original device declarations. The device declarations in this example provide the ABEL-HDL compiler with information about device-specific requirements, such as implied logic and default signal attributes.

***Note**: To process this design, you must enable the "compatible" and "implied" properties of the Compile Logic process.*

The test vectors have been removed because this design is not targeted for a programmable device.

ABEL and ispLEVER Projects

Top-level ABEL-HDL Source instantiates variable instances of **prep6.abl** (below).

Lower-level Sources

The source example below shows the lower-level ABEL-HDL file instantiated by **p6top.abl**. This file does not contain an **interface** statement, which is optional in lower-level files.

Lower-level ABEL-HDL Source

```
module prep6

title '16-Bit Accumulator'

D15..D0 pin;
Q15..Q0 pin istype 'reg';
Clk, Rst pin;

Q = [Q15..Q0];
D = [D15..D0];

@carry 2;
equations
```

```
Q := D + Q;
Q.C = Clk;
Q.AR = Rst;
end
```

ABEL-HDL Language Reference

This online help provides detailed information about each of the language elements in ABEL-HDL. It assumes you are familiar with the basic syntax discussed in ABEL-HDL Design Entry. Each entry contains the following sections (if applicable):

- **Syntax** — is the required syntax for the element.
- **Purpose** — is a brief description of the intended use of the element.
- **Use** — is a discussion of the potential uses of the element, including any special considerations.
- **Examples** — are examples of the element as it is used in a design description.
- **See Also** — refers to other elements and discussions, and to design examples that demonstrate the use of an element.

Dot Extensions (.ext)

Syntax

signal_name.ext

Purpose

Dot extensions provide a way to refer specifically to internal signals and nodes that are associated with a primary signal in your design.

Use

Signal dot extensions describe more precisely the behavior of signals in a logic description and remove the ambiguities in equations.

You can use ABEL-HDL dot extensions in complex language constructs, such as nested sets or complex expressions.

Using Pin-to-Pin Vs. Detailed Dot Extensions

Dot extensions allow you to refer to various circuit elements (such as register clocks, presets, feedback and output enables) that are related to a primary signal.

Some dot extensions are general purpose and are intended for use with a wide variety of device architectures. These dot extensions are therefore referred to as pin-to-pin (or “architecture-independent”). Other dot extensions are intended for specific classes of device architectures, or require specific device configurations. These dot extensions are referred to as detailed (or “architecture-dependent” or “device-specific”) dot extensions.

Some of these extensions can be used in hardware architectures that do not support a true implementation. For example, if the logic is present to “mimic” a dot extension like a clock enable, the software is smart enough to create a functionally equivalent equation that will realize the effectiveness of a clock enable signal even though the register itself does not have a clock enable input.

In most cases, you can describe a circuit using either pin-to-pin or detailed dot extensions. Which form you use depends on the application and whether you want to implement the application in a variety of architectures. The advantages of each method are discussed later in this section.

ABEL-HDL Dot Extensions lists the ABEL-HDL dot extensions. Pin-to-pin dot extensions are indicated with a check in the **Pin-to-Pin** column.

Detailed Design Dot Extensions

Dot Extensions for Device-specific (detailed) Designs shows the dot extensions that are supported (and which of those are required) for different register types in detailed design descriptions. The required dot extensions are indicated with a check in the **Extension Required** column.

ABEL-HDL Dot Extensions

Dot Ext.	Pin-to-pin	Description
.ACLR[3,4]	?	A device-independent asynchronous register reset, equivalent to .AR with ISTYPE 'buffer' (or .AP with ISTYPE 'invert').
.AP		Asynchronous register preset
.AR		Asynchronous register reset
.ASET[2,3]	,	A device-independent asynchronous register preset, equivalent to .AP with ISTYPE 'buffer' (or .AR with ISTYPE 'invert').
.CE		Clock-enable input to a gated-clock flip-flop
.CLK[1]	,	Clock input to an edge-triggered flip-flop
.CLR[2,3]	,	A device-independent synchronous register reset, equivalent to .SR with ISTYPE 'buffer' (or .SP with ISTYPE 'invert').
.COM[3]	,	A combinational feedback from the flip-flop data input, normalized to the pin value and used to distinguish between pin (.PIN) and internal logic array (.COM) feedback.
.D[1]		When on the left side of an equation, .D is the data input to a D-type flip-flop; on the right side, .D is combinational feedback.
.FB	,	Register feedback
.FC		Flip-flop mode control
.J		J input to a JK-type flip-flop
.K		K input to a JK-type flip-flop
.LD		Register load input
.LE		Latch-enable input to a latch
.LH		Latch-enable (high) to a latch
.OE[1]	,	Output enable

Pin-to-Pin Design Dot Extensions

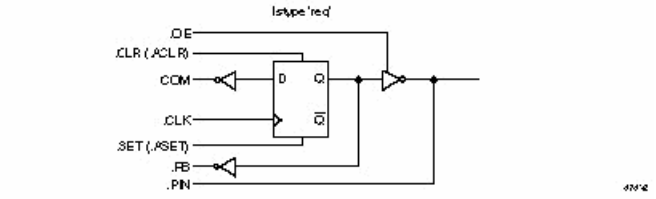
The table below shows the dot extensions that are allowable (and which of those are required) for pin-to-pin design descriptions. The required dot extensions are indicated with a check in the **Required** column.

Dot Extensions for Architecture-independent (pin-to-pin) Designs

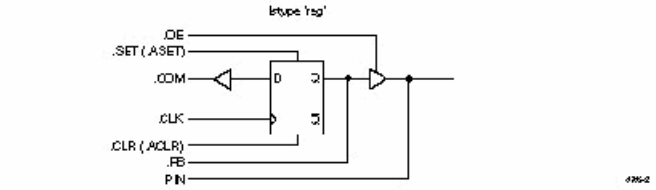
Register Type	Required	Allowable Extensions	Definition
combinational (no register)		none	output
		.oe	output enable
		.pin	pin feedback
registered logic		.clr	synchronous preset
		.aclr	asynchronous preset
		.set	synchronous set
		.aset	asynchronous set clock
	✓	.clk	combinational feedback
		.com	registered feedback
		.fb	pin feedback
		.pin	

The figures below show the effect of each dot extension. The actual source of the feedback may vary from that shown.

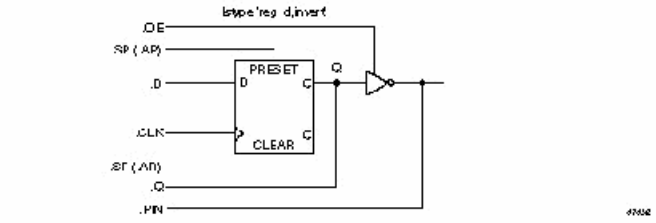
Pin-to-pin Dot Extensions in an Inverted Output Architecture



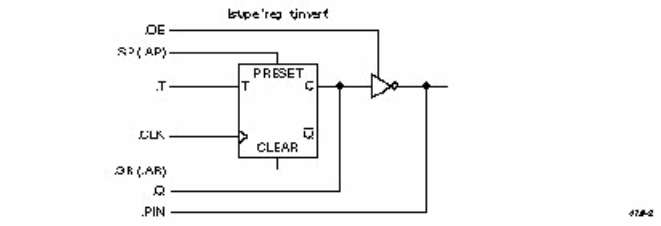
Pin-to-pin Dot Extensions in a Non-inverted Output Architecture



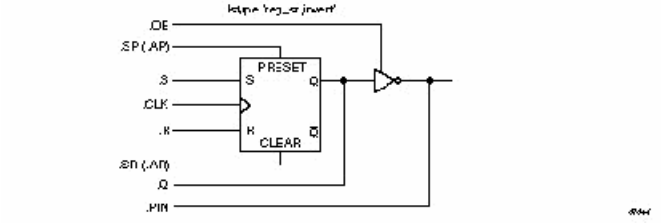
Detailed Dot Extensions for a D-type Flip-flop Architecture



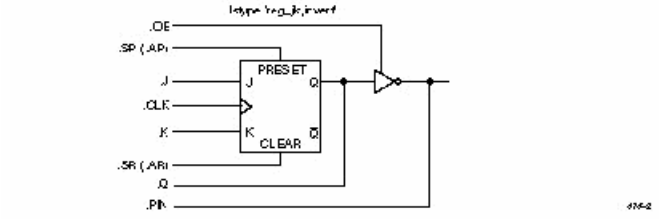
Detailed Dot Extensions for a T-type Flip-flop Architecture



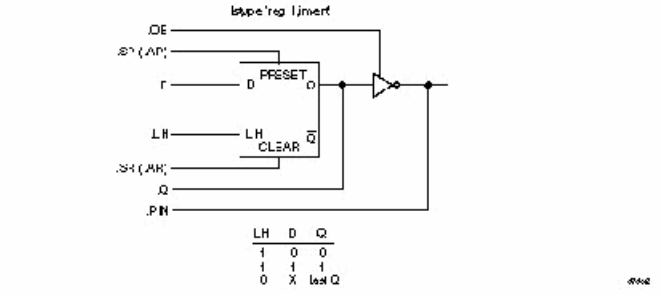
Detailed Dot Extensions for an RS-type Flip-flop Architecture



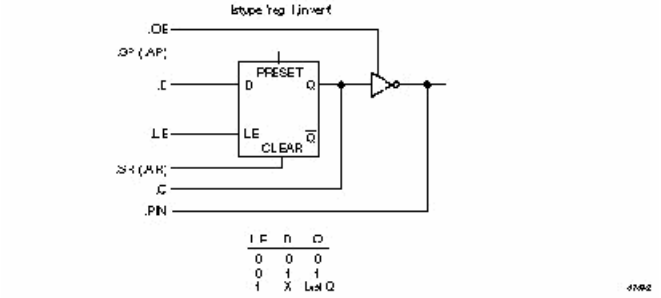
Detailed Dot Extensions for a JK-type Flip-flop Architecture



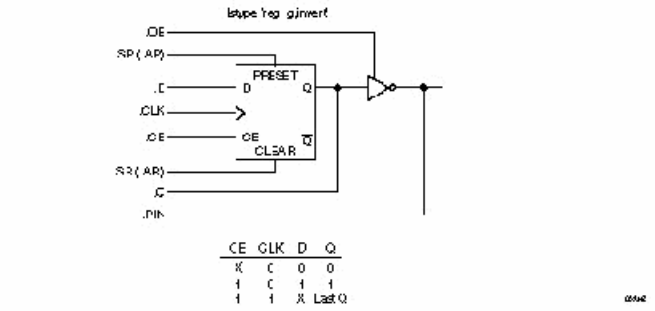
Detailed Dot extensions for a Latch with active High Latch Enable



Detailed Dot Extensions for a Latch with Active Low Latch Enable



Detailed Dot Extensions for a Gated-clock D Flip-flop



Examples

These equations precisely describe the desired circuit as a toggling D-type flip-flop that is clocked by the input Clock, assuming ISTYPE 'reg_D,buffer':

```
Q1.clk = Clock;
Q1.D = !Q1.Q # Preset;
```

Register preset:

```
Q2.PR = S & !T;
```

Register reset:

```
Q2.RE = R & !T;
```

The same circuit can be described without ISTYPE 'buffer' as:

```
Q1.clk = Clock;
Q1 := !Q1.FB # Preset;
Q2.SET = S & !T; Q2.CLR = R & !T;
```

Three-state Output Enables

Output enables are described in ABEL-HDL with the **.oe** dot extension applied to an output signal name. For example,

```
Q1.oe = !enab;
```

The equation specifies that the input signal **enab** controls the output enable for an output signal named **Q1**.

***Note**: If you explicitly state the value of a fixed output enable, you restrict the device fitters' ability to map the indicated signal to a simple input pin instead of a three-state I/O pin.*

Constant Declarations

Syntax

```
id [,id ]... = expr [,expr ]... ;
```

Purpose

A constant declaration that defines constants used in a module.

Use

id — An identifier naming a constant to be used within a module.

expr — An expression defining the constant value.

***Note**: The equal sign (=) used for constant declarations in the Declarations section is also used for equations in the Equations section.*

A constant is an identifier that retains a constant value throughout a module.

The identifiers on the left side of the equals sign are assigned the values listed on the right side. There is a one-to-one correspondence between the identifiers and the expressions listed. There must be one expression for each identifier.

The ending semicolon is required after each declaration.

Constants are helpful when you use a value many times in a module, especially when you may be changing the value during the design process. Constants allow you to change the value once in the declaration of the constant, rather than changing the value throughout the module.

Constant declarations may not be self-referencing. The following examples will cause errors:

```
X = X;
a = b;
b = a;
```

An include file, **constant.inc**, in the ABEL-HDL library, contains definitions for the most frequently used ABEL-HDL constants. To include this file, enter

```
Library 'constant' ;
```

Examples

```
ABC = 3 * 17; " ABC is assigned the value 51
Y = 'Bc' ; " Y = ^h4263 ;
X =.X.; " X means 'don't care'
ADDR = [1,0,15]; " ADDR is a set with 3 elements
A,B,C = 5,[1,0],6; " 3 constants declared here
D pin 6; " see next line
E = [5 * 7,D]; " signal names can be included
G = [1,2]+[3,4]; " set operations are legal
A = B & C; " operations on identifiers are valid
A = [!B,C]; " set and identifiers on right
```

Using Intermediate Expressions

You can use intermediate (constant) expressions in the declarations section to reduce the number of output pins required to implement multi-level functions. Intermediate expressions can be useful when a module has repeated expressions. In general, intermediate expressions

- decrease the number of output pins required, but
- increase the amount of logic required per output.

A constant expression is interpreted as a string of characters, not as a function to be implemented. For example, for the following Declarations and Equations

```
Declarations
TMP1 = [A3..A0] == [B3..B0];
TMP2 = [A7..A4] == [B7..B4];
Equations
F = TMP1 & TMP2;
```

the compiler substitutes the declarations into the equations, creating

```
F = (A7 !$ B7) & (A6 !$ B6) & (A5 !$ B5) & (A4 !$ B4) &(A3 !$ B3) & (A2 !$ B2)
& (A1 !$ B1) & (A0 !$ B0);
```

In contrast, if you move the constant declarations into the equations section

```
Declarations
TMP1,TMP2 pin 18,19
Equations
TMP1 = [A3..A0] == [B3..B0];
TMP2 = [A7..A4] == [B7..B4];
F = TMP1 & TMP2;
```

the compiler implements the equations as three discrete product terms, with the result

```
TMP1 =(A3 !$ B3) & (A2 !$ B2) & (A1 !$ B1) & (A0 !$ B0);
TMP2 =(A7 !$ B7) & (A6 !$ B6) & (A5 !$ B5) & (A4 !$ B4);
F = TMP1 & TMP2;
```

The first example (using intermediate expressions) requires one output with 16 product terms, and the second example (using equations) requires three outputs with less than 8 product terms per output. In some cases, the number of product terms required for both methods can be reduced during optimization.

***Note**: As an alternate method for specifying multi-level circuits such as this, you can use the @CARRY directive. See "@directive" later in this chapter.*

Signal Attributes (attr)

See lstype.

Directives (@directive)

Purpose

Directives control the contents or processing of a source file. You can use directives to conditionally include sections of ABEL-HDL source code, bring code in from another file, and print messages during processing. The available directives are given on the following pages.

Use

Some of the directives use arguments to determine how the directive is processed. The arguments can be actual arguments, or dummy arguments preceded by question marks (?).

@Alternate - Alternate Operator Set

Syntax

```
@alternate
```

Use

@Alternate enables an alternate set of operators. If you are more familiar with the alternate set, you may want to use this directive.

The alternate operators remain in effect until the @Standard directive is used or the end of the module is reached.

Using the alternate operator set precludes use of the ABEL-HDL addition, multiplication, and division operators because they represent the OR, AND, and NOT logical operators in the alternate set. The standard operators !, &, #, $, and !$ still work when @Alternate is in effect.

The alternate operator set is listed in the following table.

Alternate Operator Set

ABEL-HDL Operator	Alternate Operator	Description
!	/	NOT
&	*	AND
#	+	OR
$	: + :	XOR
!$	: * :	XNOR

@Carry - Maximum Bit-width for Arithmetic Functions

Syntax

```
@carry expression ;
```

Use

expression — A numeric expression.

The @Carry directive allows you to reduce the amount of logic required for processing large arithmetic functions by specifying how adders, counters, and comparators are generated. The number generated by the expression indicates the maximum bit-width to use when performing arithmetic functions.

For example, for an 8-bit adder, a @Carry statement with an expression which results in 2 would divide the 8-bit adder into four 2-bit adders, creating intermediate nodes. This would reduce the amount of logic generated.

The statement:

```
@carry 1;
```

generates chains of one-bit adders and comparators for all subsequent adder and comparator equations (instead of the full look-ahead carry equations normally generated).

This directive automatically generates additional combinational nodes. Use different values for the @Carry statement to specify the types of adders and comparators required for the design.

Examples

```
@carry 2; "generate adder chain
[s8..s0] = [.x.,a7..a0]+[.x.,b7..b0]
```

@Const - Constant Declarations

Syntax

```
@const id = expression ;
```

Use

id — An identifier.

expression — An expression.

@Const allows new constant declarations to be made in a source file outside the normal (and required) declarations section.

The @Const directive defines internal constants inside macros. Constants defined with @Const override previous constant declarations. You cannot use @Const to redefine an identifier that was used earlier in the source file as something other than a constant (for example, a macro or pin).

Examples

```
@CONST count = count + 1;
```

@Dcset - Don't Care Set

Syntax

```
@dcset
```

Use

ABEL-HDL uses don't-care conditions to help optimize partially-specified logic functions. Partially-specified logic functions are those that have less than 2*n* significant input conditions, where *n* is the number of input signals. The @Dcset directive allows the optimization to use either 1 or 0 for don't cares to optimize these functions.

***Caution**: The @Dcset directive overrides Istype attributes 'dc', 'neg' and 'pos'.*

@Dcstate - State Output Don't Cares

Syntax

```
@dcstate
```

Use

When @dcstate is specified, all unspecified state diagram states and transitions are applied to the design outputs as don't cares. You must use this option in combination with @dcset or with the 'dc' attribute.

@Exit - Exit Directive

Syntax

```
@exit
```

Use

The **@Exit** directive stops processing of the source file with error bits set. (Error bits allow the operating system to determine that a processing error has occurred.)

@Expr - Expression Directive

Syntax

```
@expr [ {block} ] expression ;
```

Use

block — A block.

expression — An expression.

@Expr evaluates the given expression and converts it to a string of digits in the default base numbering system. This string and the block are then inserted into the source file at the point where the @Expr directive occurs. The expression must produce a number.

@Expr can contain variable values, and you can use it in loops with @Repeat.

Examples

```
@expr {ABC} ^B11 ;
```

Assuming that the default base is base ten, this example causes the text ABC3 to be inserted into the source file.

@If - If Directive

Syntax

```
@if expression {block }
```

Use

expression — An expression.

block — A block of text.

@IF includes or excludes sections of code based on the value of an expression. If the expression is non-zero (logical true), the block of code is included.

Dummy argument substitution is supported in the expression.

Examples

```
@if (A > 17) { C = D $ F ; }
```

@Ifb - If Blank Directive

Syntax

```
@IFB (arg) {block }
```

Use

arg — An actual argument, or a dummy argument preceded by a "?"

block — A block of text.

@IFB includes the text contained within the block if the argument is blank (if it contains 0 characters).

Examples

```
@IFB ()
{text here is included with the rest of the source file.}
@IFB ( hello ) { this text is not included }
@IFB (?A) {this text is included if no value is substituted for A. }
```

@Ifdef - If Defined Directive

Syntax

```
@ifdef id {block }
```

Use

id — An identifier.

block — A block of text.

@IFDEF includes the text contained within the block, if the identifier is defined.

Examples

```
A pin 5 ;
@ifdef A { Base = ^hE000 ; }
"the above assignment is made because A was defined
```

@Ifiden - If Identical Directive

Syntax

```
@ifiden (arg1,arg2 ) {block }
```

Use

arg1,2 — Actual arguments, or dummy argument names preceded by a "?"

block — A block of text.

The text in the block is included if arg1 and arg2 are identical.

Examples

```
@ifiden (?A,abcd) { ?A device 'P16R4'; }
```

A device declaration for a P16R4 is made if the actual argument substituted for A is identical to abcd.

@Ifnb - If Not Blank Directive

Syntax

```
@ifnb (arg) {block }
```

Use

arg — An actual argument, or a dummy argument name preceded by a "?"

block — A block of text.

@IFNB includes the text contained within the block if the argument is not blank (if it contains more than 0 characters).

Examples

```
@IFNB () { ABEL-HDL source here is not included with the rest of the source
file. }
@IFNB ( hello ) { this text is included }
@IFNB (?A) {this text is included if a value is substituted for A}
```

@Ifndef - If Not Defined Directive

Syntax

```
@ifndef id {block }
```

Use

id — An identifier.

block — A block of text.

@IFNDEF includes the text contained within the block, if the identifier is undefined. Thus, if no declaration (pin, node, device, macro, or constant) has been made for the identifier, the text in the block is inserted into the source file.

Examples

```
@ifndef A{Base=^hE000;}
"if A is not defined, the block is inserted in the text
```

@Ifniden - If Not Identical Directive

Syntax

```
@ifniden (arg1,arg2 ) {block }
```

Use

arg1,2 — Actual arguments, or dummy argument names preceded by a "?"

block — A block of text.

The text in the block is included in the source file if arg1 and arg2 are not identical.

Examples

```
@ifniden (?A,abcd) { ?A device 'P16R8'; }
```

A device declaration for a P16R8 is made if the actual argument substituted for A is not identical to abcd.

@Include - Include Directive

Syntax

```
@include filespec
```

Use

filespec — A string specifying the name of a file.

@INCLUDE causes the contents of the specified file to be placed in the ABEL-HDL source file. The inclusion begins at the location of the @INCLUDE directive. The file specification can include an explicit drive or path specification that indicates where the file is found. If no drive or path specification is given, the default drive or path is used.

Examples

```
@INCLUDE 'macros.abl' "file specification
@INCLUDE '\\incs\\macros.inc' "DOS paths require 2 slashes
```

@Irp - Indefinite Repeat Directive

Syntax

```
@irp dummy_arg ( arg [,arg ]... ) {block }
```

Use

dummu_arg — A dummy argument.

arg — An actual argument, or a dummy argument name preceded by a "?"

block — A block of text.

@IRP causes the block to be repeated in the source file n times, where n equals the number of arguments contained in the parentheses. Each time the block is repeated, the dummy argument takes on the value of the next successive argument.

Examples

```
@IRP A (1, ^H0A, 0)
{B = ?A ; }
```

results in

```
B = 1 ;
B = ^H0A ;
B = 0 ;
```

which is inserted into the source file at the location of the @IRP directive. Note that multiple assignments to the same identifier result in an implicit OR.

Note that if the directive is specified like this

```
@IRP A (1, ^H0A,0)
{B = ?A ; }
```

the resulting text would be

```
B = 1 ; B = ^H0A ; B = 0 ;
```

The text appears all on one line because the block in the @IRP definition contains no end-of-lines. Remember that end-of-lines and spaces are significant in blocks.

@Irpc - Indefinite Repeat, Character Directive

Syntax

```
@irpc dummy_arg (arg) {block }
```

Use

dummy_arg — A dummy argument.

arg — An actual argument, or a dummy argument name preceded by a "?"

block — A block.

@IRPC causes the block to be repeated in the source file *n* times, where *n* equals the number of characters contained in *arg*. Each time the block is repeated, the dummy argument takes on the value of the next character.

Examples

```
@IRPC A (Cat)
{B = ?A ;
}
```

results in

```
B = C ;
B = a ;
B = t ;
```

which is inserted into the source file at the location of the @IRPC directive.

@Message - Message Directive

Syntax

```
@message 'string'
```

Use

string — Any string.

@Message sends the message specified in *string* to your monitor. You can use this directive to monitor the progress of the parsing step of the compiler, or as an aid to debugging complex sequences of directives.

Examples

```
@message 'Includes completed'
```

@Onset - No Don't Care's

Syntax

```
@onset
```

Use

The **@onset** directive disables the use of don't care input conditions for optimization.

@page - Page Directive

Syntax

```
@page
```

Use

Send a form feed to the listing file. If no listing is being created, **@page** has no effect.

@Radix - Default Base Numbering Directive

Syntax

```
@radix expr ;
```

Use

expr — An expression that produces the number 2, 8, 10 or 16 to indicate a new default base numbering.

The **@Radix** directive changes the default base. The default is base 10 (decimal). This directive is useful when you need to specify many numbers in a base other than 10. All numbers that do not have their base explicitly stated are assumed to be in the new base. (See *Numbers* in Chapter 4, "ABEL-HDL Design".)

The newly-specified default base stays in effect until another **@radix** directive is issued or until the end of the module is reached. Note that when a new @radix is issued, the specification of the new base must be in the current base format.

When the default base is set to 16, all numbers in that base that begin with an alphabetic character must begin with leading zeroes.

Examples

```
@radix 2 ; "change default base to binary
@radix 1010 ; "change from binary to decimal
```

@Repeat - Repeat Directive

Syntax

```
@repeat expr {block }
```

Use

expr — A numeric expression.

block — A block.

@REPEAT causes the block to be repeated *n* times, where *n* is specified by the constant expression.

Examples

The following use of the repeat directive,

```
@repeat 5 {H,}
```

results in the insertion of the text "H,H,H,H,H," into the source file. The @REPEAT directive is useful in generating long truth tables and sets of test vectors. Examples of @REPEAT can be found in the example files.

@Setsize - Set Indexing

Syntax

```
@setsize [expression];
```

Purpose

The **@setsize** directive generates a number corresponding to the number of elements in the expression, which must be a set. This directive is useful for set indexing operations.

Example

```
@SETSIZE [a,b,c]
```

generates the number 3.

For set indexing, you can use the @SETSIZE directive in macros in the following manner:

```
high macro (s) {?S[@SETSIZE(?S);-1..@SETSIZE(?S);/2-1]};
```

The **high** macro returns the upper half of a set of any size (the high 4 bits of an 8-bit set, for example).

***Note**: The terminating semicolons are required.*

@Standard - Standard Operators Directive

Syntax

```
@standard
```

Use

The **@standard** option resets the operators to the ABEL-HDL standard. The alternate set is chosen with the **@alternate** directive.

Async_reset and Sync_reset

Syntax

```
SYNC_RESET symbolic_state_id : input_expression ;
ASYNC_RESET symbolic_state_id : input_expression ;
```

Purpose

In symbolic state descriptions, the **SYNC_RESET** and **ASYNC_RESET** statements specify synchronous or asynchronous state machine reset logic in terms of symbolic states.

Use

symbolic_state_id — An identifier used for reference to a symbolic state.

input_expression — Any expression.

Examples

```
ASYNC_RESET Start : Reset ;
SYNC_RESET Start : Reset ;
```

Case

Syntax

```
CASE expression : state_exp;
[ expression : state_exp; ] ...
ENDCASE ;
```

Purpose

Use the CASE statement in a **State_diagram** to indicate transitions of a state machine when multiple conditions affect the state transitions.

Use

expression — An expression.

state_exp — An expression identifying the next state, optionally followed by WITH transition equations.

You can nest CASE statements with If-Then-Else, GOTO, and other CASE statements, and you can use equation blocks.

***Note**: Equation blocks used within a conditional expression such as IF-THEN, CASE, or WHEN-THEN result in logic functions that are logically ANDed with the conditional expression that is in effect.*

The state machine advances to the state indicated by *state_exp* (following the expression that produces a true value). If no expression is true, the result is undefined, and the resulting action depends on the device being used. (For devices with D flip-flops, the next state is the cleared register state.) For this reason, you should be sure to cover all possible conditions in the CASE statement expressions. If the expression produces a numeric rather than a logical value, 0 is false and any non-zero value is true. The expressions contained within the Case-endcase keywords must be mutually exclusive (only one of the expressions can be true at any given time). If two or more expressions within the same Case statement are true, the resulting equations are undefined.

Examples

```
"Mutually exclusive Case statement
case a == 0 : 1 ;
a == 1 : 2 ;
a == 2 : 3 ;
a == 3 : 0 ;
endcase ;

"Not mutually exclusive Case statement
case (a == 0) : 1 ;
(a == 0) & (B == 0) : 0 ;
endcase ;
```

Cycle

This keyword specifies the signal to repeat until the end of simulation. The default time unit is nano-second. The syntax for CYCLE is:

```
CYCLE signal_name (logic_value, integer) (logic_value, integer) [(logic_value,
integer)];
```

Example

```
CYCLE clk1 (0,3) (1,5);
```

In this example, signal `clkl` is low (`0`) for 3 cycles and then high (`1`) for 5 cycles.

***Note**: The Wait keyword is only defined in the Lattice Logic Simulator. It cannot be used with all of the supported special constants, except .R. See Lattice Logic Simulator Help – "Special Constants" for information.*

Declarations

Syntax

```
Declarations declarations
```

Purpose

The declarations keyword allows you to declare declarations (such as sets or other constants) in any part of the ABEL-HDL source file.

Use

declarations — You can use any declarations after the **Declarations** keyword.

The Declarations keyword is not necessary for declarations immediately following the module and/or title statement(s).

Examples

An example of declared equations is shown below:

```
module castle
moat device 'P16V8C'; "declarations implied
A,B pin 1,2;
Out1 pin 15 istype 'com';
```

```
Equations
Out1 = A & B;

Declarations "declarations keyword required
C,D,E,F pin 3,4,5,6;
Out2 pin 16 istype 'com';
Temp1 = C & D;
Temp2 = E & F;

Equations
Out2 = Temp1 # Temp2;
end;
```

Device

Syntax

device_id **DEVICE** *real_device* ;

Purpose

The device declaration statement associates the device name used in a module with an actual programmable logic device on which designs are implemented.

Use

device_id — An identifier used for the programmer to load filenames.

real_device — A string describing the architecture name of the real device represented by device_id.

The device declaration is optional.

You should give device identifiers, which are used in device declarations, valid filenames since JEDEC files are created by appending the extension `.jed` to the identifier. The architecture name of the programmable logic device is indicated by the string `real_device`.

The ending semicolon is required.

Examples

```
D1 DEVICE 'P16R4' ;
```

End

Syntax

end *module_name*

Purpose

The **end** statement denotes the end of the module.

Use

The end statement can be followed by the module name. For multi-module source files, the module name is required.

Equations

Syntax

```
EQUATIONS
 element [?]= condition ;
 element [?]:= condition ;
 [ WHEN condition THEN ] [ ! ] element=expression;
  [ ELSE equation ];
```

or

```
[ WHEN condition THEN ] equation; [ ELSE equation];
<inst_name> <macro_name> ([<mcr_port>]{,[<mcr_port>]}*);
```

Purpose

The equations statement defines the beginning of a group of equations associated with a device.

Use

condition — An expression.

element — An identifier naming a signal, set of signals, or actual set to which the value of the expression is assigned.

expression — An expression.

=, :=, ?= and ?:= — Combinational and registered (pin-to-pin) on-set and dc-set assignment operators.

when-then-else — When-then-else statements.

inst_name — Identifier.

macro_name — Identifier; Lattice PLL macro name.

mcr_port — Signal(s), signal set, or Boolean expressions.

Equations specify logic functions with an extended form of Boolean algebra. A semicolon is required after each equation. The equations following the equation statement are equations as described in Chapter 1, “Language Structure” of the *ABEL-HDL Reference Manual*.

The equations following the equation statement are equations as described in Chapter 4, “ABEL-HDL Design.”

The syntax <*inst_name*> <*macro_name*> ([<*mcr_port*>]{,[<*mcr_port*>]}*); is enhanced to support Lattice PLL macros in ABEL-HDL. When the Lattice PLL macros are being called, the user-defined macro port numbers must match the macro need. The <*mcr_port*> is the signal(s) in the current design. Two consecutive commas mean a macro port is not connected. The output port of a macro can be dangled, while the input ports or bidirectional ports cannot be dangled.

Examples

A sample equations section follows:

```
equations
A = B & C # A ;
[W,Y] = 3 ;
!F = (B == C) ;
Output.D = In1 # In2
```

Functional_block

Syntax

```
DECLARATIONS
instance_name FUNCTIONAL_BLOCK source_name ;

EQUATIONS
instance_name.port_name = signal_name;
```

Purpose

You can use a **functional_block** declaration in an upper-level ABEL-HDL source to instantiate a declared lower-level module and make the ports of the lower-level module accessible in the upper-level source. You must declare modules with an **interface** declaration before you can instantiate them with a **functional_block** statement.

Use

instance_name — A unique identifier for this instance of the functional block in the current source.

source_name — The name of the lower-level module that is being instantiated.

***Note**: When a module is instanced by an upper-level source, any signal attributes (explicit or implied) are inherited by the upper-level source signals. Therefore, you do not need to specify ISTYPEs in higher-level sources for instantiated signals.*

Creating Multiple Instances

You can use the range operator (..) to instantiate multiple instance names of the module. For example,

```
CNT0..CNT3 functional_block cnt4 ;
```

creates 4 instances of the lower-level module cnt4.

Mapping Ports to Signals

Signal names are mapped to port names, using equations (similar to wiring the signals on a schematic). You need to specify only the signals used in the upper-level source, if default values have been specified in the lower-level module interface statement. See “Interface (lower-level)” in this chapter for more information on setting default values.

To specify the signal wiring, map signal names to the lower-level module port names with dot extension notation. There are three kinds of wire: input, output, and interconnect.

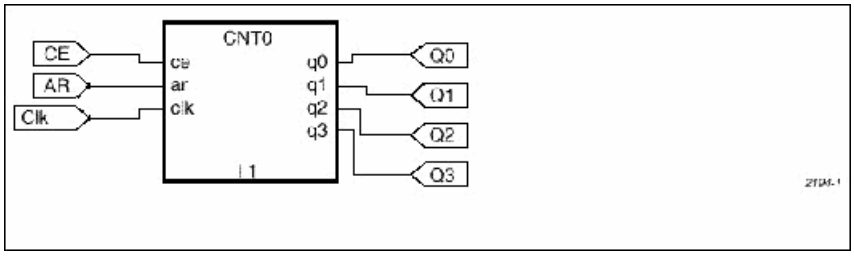
Input Wire	Connects lower-level module inputs to upper-level source inputs. Instance.port = input
Output Wire	Connects upper-level source outputs to lower-level module outputs. output = instance.port
Interconnect Wire	Connects the outputs of one instance of a lower-level module to another instance’s inputs. Instance0.port = instance1.port

Examples

```
module counter;

cnt4 interface (ce, ar, clk, [q0..q3]); // cnt4's top-level interface
declaration.
CNT0..CNT3 functional_block cnt4; // Four instances of cnt4.

Clk, AR, CE pin;
Q0..Q3 pin;

equations
CNT0.[clk, ar, ce] = [Clk; AR, CE]; // Connecting to Clk, AR, and CE inputs.
CNT0.[q0..q3] = [Q0..Q3]; // Connecting to Q0..Q3 outputs.
End
```

The figure below shows how the above ABEL-HDL file wires the upper-level source's signals to the lower-level module's ports. Note that the above file instantiates four instances of **cnt4**, but only one (CNT0) is wired.

Wiring of CNT0



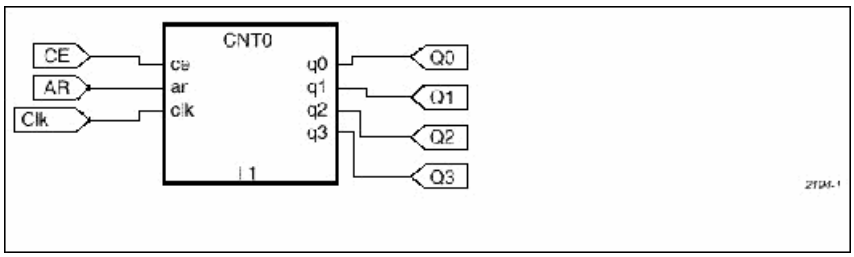
Overriding Default Values

You can override the default values given in a lower-level module's interface statement by specifying default equations in the higher-level source. For example, if you have specified a default value of 1 for the signal **ce** in interface **cnt4** (but in instance **CNT0,** you want **ce** to be 0), you would write:

```
CNT0.ce = 0;
```

This equation overrides the 1 with a 0. If you override the default values, you may want to re-optimize the post-linked design.

Wiring of CNT0



Unused Outputs (no connects)

If you do not want to use a lower-level module's outputs, specify them as No Connects (NC) by not wiring up the port to a physical pin. For example, to make a 3-bit counter out of a 4-bit counter in the upper-level source, you might use the following wiring equations:

```
q2..q0 pin ; "upper-level signals
Equations
[q2..q0] = CNT_A.[q2..q0]
```

Fuses

Syntax

```
FUSES
fuse_number = fuse_value ;
```

or

```
FUSES
[ fuse_number_set ] = fuse_value ;
```

Purpose

The **fuses** section explicitly declares the state of any fuse in the targeted device.

Use

fuse_number — The fuse number obtained from the logic diagram of the device.

fuse_number_set — The set of fuse numbers (contained in square brackets).

fuse_value — The number indicating the state of fuse(s).

The **fuses** statement provides device-specific information, and precludes changing devices without editing the statement in the source file.

Fuse values that appear on the right side of the = symbol can be any number. If a single fuse number is specified on the left side of the = symbol, the least significant bit (LSB) of the fuse value is assigned to the fuse. A 0 indicates an intact fuse and a 1 indicates a blown fuse. In the case of multiple fuse numbers, the fuse value is expanded to a binary number and truncated or given leading zeros to obtain fuse values for each fuse number.

***Caution**: When fuse states are specified using the FUSES section, the resulting fuse values supersede the fuse values obtained through the use of equations, truth tables and state diagrams, and affect device simulation accordingly.*

ABEL-HDL has a limit of 128 fuses per statement, due to the set size limitations.

Examples

```
FUSES
3552 = 1 ;
[3478...3491] = ^Hff;
```

Goto

Syntax

```
GOTO state_exp ;
```

Purpose

The **GOTO** statement is used in the **State_diagram** section to cause an unconditional transition to the state indicated by *state_exp*.

Use

state_exp — An expression identifying the next state, optionally followed by WITH transition equations.

GOTO statements can be nested with If-Then-Else and CASE statements.

Examples

```
GOTO 0 ; "goto state 0
GOTO x+y ; "goto the state x + y
```

If-Then-Else

Syntax

```
IF exp THEN state_exp
[ ELSE state_exp ] ;
```

Chained IF-THEN-ELSE:

```
IF expr THEN state_exp
ELSE IF exp THEN state_exp
ELSE state_exp ;
```

Nested IF-THEN-ELSE:

```
IF exp THEN state_exp
ELSE IF exp THEN
IF exp THEN state_exp
ELSE state_exp
ELSE state_exp ;
```

Nested IF-THEN-ELSE with Blocks:

```
IF exp THEN
{ IF exp THEN state_exp
IF exp THEN state_exp
}
ELSE state_exp ;
```

Purpose

The **If-then-else** statements are used in the **State_diagram** section to describe the progression from one state to another.

Use

exp — An expression.

state_exp — An expression or block identifying the next state, optionally followed by WITH transition equations.

***Caution**: If-Then-Else is only supported within a state_diagram description. Use When-Then-Else for equations.*

***Note**: Equation Blocks used within a conditional expression (such as If-then, Case, or When-then) result in logic functions that are logically ANDed with the conditional expression that is in effect.*

The expression following the **If** keyword is evaluated, and if the result is true, the machine goes to the state indicated by the *state_exp*, following the **Then** keyword. If the result of the expression is false, the machine jumps to the state indicated by the **Else** keyword.

Any number of **If** statements may be used in a given state, and the **Else** clause is optional. The indenting and formatting of an **If-then-else** statement is not significant: breaking a complex transition statement across many lines (and indenting) improves readability.

If-then-else statements can be nested with **Goto**, **Case**, and **With** statements. **If-then-else** and **Case** statements can also be combined and nested.

Chained IF-THEN-ELSE Statements

Any number of **If-then-else** statements can be chained, but the final statement must end with a semicolon. The chained **If-then-else** statement is intended for situations where the conditions are not mutually exclusive. The **Case** statement more clearly expresses the same function as chained mutually-exclusive **If-then-else** statements.

Chained **If-then-else** statements can provide multiway branching transition logic. For example, multiple **If-then-else** statements can be chained to describe a three-way branch in the following manner:

```
STATE S0:
IF (address < ^h0400)
THEN S0
ELSE
IF (address <= ^hE100)
THEN S2
ELSE
S1;
```

Examples

```
if A==B then 2 ; "if A equals B goto state 2
if x-y then j else k; "if x-y is not 0 goto j, else goto k
if A then b*c; "if A is true (non-zero) goto state b*c
```

Chained IF-THEN-ELSE

```
if a then 1
else
if b then 2
else
if c then 3
else 0 ;
```

Nested IF-THEN-ELSE with Blocks

```
IF (Hold) THEN
{ IF (!RESET) THEN State1 ;
IF (Error) THEN State2 ;
}
ELSE State3 ;
```

Interface (top-level)

Syntax

```
source_name INTERFACE (input/set[=value] -> output/set :> bidir/set);
```

Purpose

The **interface** keyword declares lower-level modules and their ports (signals) that are used in the current source. This declaration is used in conjunction with a **functional_block** declaration for each instantiation of the module.

Use

module_name — The name of the module being declared.

inputs->outputs:>bidirs — A list of signals in the lower-level module used in the current source. Signal names are separated by commas. Use -> and :> to indicate the direction of each port of a functional block.

value — An optional default value for an input that overrides defaults in the lower-level module.

If the lower-level module uses the **interface** keyword to declare signals, the upper-level source interface statement must exactly match the signal listing.

***Caution**: Interface declarations cannot contain dot extensions. If you need a specific dot extension across a source boundary (to resolve feedback ambiguities, for example), you must introduce an intermediate signal into the lower-level module to provide the connection to the higher-level source. All dot extension equations for a given output signal must be located in the ABEL-HDL module in which the signal is defined. No references to the signal's dot extensions can be made outside of the ABEL-HDL module.*

***Note**: When you instantiate a lower-level module in an upper-level source, any signal attributes (either explicit or implicit) are inherited by the higher-level source signals that map to the lower-level signals. Do not specify ISTYPEs for instantiated signals.*

Examples

```
module top;
cnt4 interface (ce,ar,clk -> [q3..q0])
```

Map port names to signal names with equations.

Interface (lower-level)

Syntax

```
MODULE module_name
INTERFACE (input/set[=port_value] -> output/set [:> bidir/set]);
```

Purpose

The **interface** declaration is optional for lower-level modules. Use the **interface** declaration in lower-level modules to assign a default port list and input values for the module when instantiated in higher-level ABEL-HDL sources. If you use the interface statement in an instantiated module, you must declare the signals and sets in the upper-level source in the same order and grouping as given in the **interface** statement in the lower-level module.

Declaring signals in the lower-level module, although optional, does allow the compiler to check for signal declaration mismatches and therefore reduces the possibility of wiring errors.

Use

module_name — The standard module statement.

signal/set — Signals or sets in the lower-level module used as ports to higher-level sources. Use -> and :> to indicate the direction of each port of a functional block. Use commas to separate groups of signals

port_value — The default value for the port for input signals only. Default values do not apply to output and bidirectional signals.

Declared Signals

Declared signals can be a list of lower-level pins, sets, or a combination of both. The following constraints apply to the different signal types:

Signal Type	Constraints
Input	Default values must be binary if applied to an individual bit, or any positive integer applied to a set. All inputs must be listed.
Output	Unlisted outputs are interpreted as No connects (NC). Unlisted, fed-back outputs are interpreted as nodes in the upper-level source, following the naming convention *instance_name/node_name*
Bidirectional	Listing bidirectional signals is optional, except for those with output enable (OE). If you specify bidirectional signals, the compiler checks for invalid wire connections.

***Caution**: Interface declarations cannot contain dot extensions. If you need a specific dot extension across a source boundary (to resolve feedback ambiguities, for example), you must introduce an intermediate signal into the lower-level module to provide the connection to the higher-level source. All dot extension equations for a given output signal must be located in the ABEL-HDL module in which the signal is defined. No references to the signal's dot extensions can be made outside of the ABEL-HDL module.*

***Note**: When you instantiate a lower-level module in a higher-level source, any signal attributes (explicit or implicit) are inherited by the higher-level source signals that map to the lower-level signals. Do not specify ISTYPEs for instantiated signals.*

Unlisted Signals

If you do not list some signals of the lower-level module in the interface statement, the following rules apply:

Unlisted Pins Are:	The Compiler Interprets Them As:
Inputs or Bi-directionals with OE	Errors
Outputs	No Connects (NC), and they can be removed
Feedback outputs	Nodes in the upper-level source, following the naming convention: `instance_name/node_name`

Examples

The following interface statement declares inputs ce, ar, and clk (giving default values for two of them) and outputs q3 through q0.

```
module cnt4 interface (ce=1,ar=1,clk -> [q3..q0]) ;
```

Specifying default values allows you to instantiate **cnt4** without declaring the **ce** and **ar** inputs in the upper-level source. If you do not declare these inputs, they are replaced with the constants 1 and 0, respectively. Since these constants may affect optimization, you may need to re-optimize the lower-level module with the constants.

***Note**: Supported default values are 1, 0, or X (don't care). You can give default values for a set with a positive integer, and each digit of the integer's binary form supplies the default value for the corresponding signal in the set.*

lstype_Attribute Declarations

Syntax

```
signals [PIN | NODE] numbers ISTYPE 'attr';
```

signal — A pin or node identifier.

number — Optional pin or node numbers.

attr — A string that specifies attributes for the signals, separated by commas.

Use

The ISTYPE statement defines attributes (characteristics) of signals (pins and nodes). You should use signal attributes to remove ambiguities in architecture-independent designs. Even when you have specified a device, using attributes ensures that the design operates consistently if the device is changed later.

You can combine a pin or node declaration with an ISTYPE statement in a single declaration. Supported attributes are listed below.

Examples

```
F0, A pin istype 'invert, reg' ;
```

This declaration statement defines F0 and A as inverted D-type flip-flops. The following signal declarations are all supported:

```
q3,q2,q1,q0 NODE ISTYPE 'reg_SR';Clk,a,b,c PIN 1,2,3,4;reset PIN;reset ISTYPE
'com';Output PIN 15 ISTYPE 'reg,invert';
```

'buffer'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'buffer' ;
```

Use

Signal attribute 'buffer,' (along with its counterpart, 'invert,' to control output inversion) enforces the existence or non-existence of a hardware inverter at the device pin associated with the specified output signal. This is important because the state of the inverter affects a register's reset, preset, preload, and power up behavior (as observed on the associated output pin).

The 'buffer' attribute indicates that the target architecture does not have an inverter between the associated flip-flop (if any) and the actual output pin.

Examples

```
q0,q1,q2 pin istype 'buffer';a0,a2 istype 'buffer';
```

'collapse'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'collapse' ;
```

Use

Collapse (remove) this combinational node. If neither 'keep' nor 'collapse' is specified, the optimization and fitter programs will keep or collapse the node for best optimization. The opposite attribute is ISTYPE 'keep'.

Example

In the following example, signal **b** is given the 'collapse' attribute:

```
module coll_b
a,c,d,e pin ;

b node istype 'collapse'

equations
a = b & e;
b = c & d;
end
```

The resulting equation collapses b out of the equations:

```
a = c & d & e ;
```

'com'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'com' ;
```

Use

Signal attribute 'com' is used to specify that a signal has no register element associated with it or that any register should be bypassed.

'dc'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'com' ;
```

Use

Signal attribute 'dc' is used to specify that unspecified logic should be given a value of don't care.

The 'dc' attribute is equivalent to the @DCSET directive, except that it operates on individual signals instead of on an entire section of a design.

***Note**: 'dc,' 'neg,' and 'pos' are mutually exclusive.*

***Caution**: The @DCSET directive overrides 'dc,' 'neg,' and 'pos'.*

'invert'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'invert' ;
```

Use

The target architecture has an inverter between the associated flip-flop (if any) and the actual output pin.

Control of output inversion in devices is accomplished through the use of the 'invert' or 'buffer' attributes. These attributes enforce the existence ('invert') or non-existence ('buffer') of a hardware inverter at the device pin associated with the output signal specified.

In registered devices, the 'invert' attribute ensures that an inverter is located between the output pin and its associated register output.

***Note**: Ensuring an inverter is important for both pin-to-pin and detailed design descriptions because the location of the inverter affects a register's reset, preset, preload, and powerup behavior as observed on the associated output pin.*

'keep'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'keep' ;
```

Use

Signal attribute 'keep' indicates that the combinational node should not be collapsed (removed).

'neg'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'neg' ;
```

Use

Signal attribute 'neg' indicates that unspecified logic should be given the value of 1. The 'neg' or 'pos' attribute is implied if a device is specified. For example, 'neg' is implied if the device output is inverted (for example, with a 16L8).

***Caution**: The @DCSET directive overrides 'dc,' 'neg,' and 'pos'.*

***Note**: 'dc,' 'neg,' and 'pos' are mutually exclusive.*

'pos'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'pos' ;
```

Use

Signal attribute 'pos' indicates that unspecified logic should be given the value of 0.

***Note**: The 'neg' or 'pos' attribute is implied if a device is specified. For example, 'pos' is implied if the device output is not inverted.*

***Caution**: The @DCSET directive overrides 'dc,' 'neg,' and 'pos'.*

***Note**: 'dc,' 'neg,' and 'pos' are mutually exclusive.*

'reg'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'reg' ;
```

Use

Signal attribute 'reg' indicates that the associated signal has a D-type flip-flop as its memory element. Since flip-flop types are normally determined by the use of ':=' or by the use of dot extensions (.D, .T, etc.) in equations, the 'reg' attribute is only significant when state diagrams are used. When 'reg' is specified, the equations resulting from an ABEL-HDL state diagram will be pin-to-pin registered equations (the same as if you had written equations using ':='). You do not need to worry about the existence of inverted output pins.

'reg_d'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'reg_d' ;
```

Use

Signal attribute 'reg_d' indicates that the associated signal has a D-type flip-flop as its memory element. When 'reg_d' is specified, equations generated from an ABEL-HDL state diagram will assume a D-type register. If this attribute is specified; however, you will need to specify the 'invert' or 'buffer' attribute to ensure consistent operation in different architectures. To eliminate the need for the 'invert' or 'buffer' attributes, use the 'reg' attribute instead of 'reg_d'.

'reg_g'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'reg_g' ;
```

Use

Signal attribute 'reg_g' indicates that the associated signal has a D-type flip-flop with gated clock as its memory element. Equations generated from an ABEL-HDL state diagram will assume this register type if the 'reg_g' attribute is specified; however, you will need to specify the 'invert' or 'buffer' attribute to ensure consistent operation in different architectures.

'reg_jk'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'reg_jk' ;
```

Use

Signal attribute 'reg_jk' indicates that the associated signal has a JK-type flip-flop as its memory element. Equations generated from an ABEL-HDL state diagram will assume this register type if the 'reg_jk' attribute is specified; however, you will need to specify the 'invert' or 'buffer' attribute to ensure consistent operation in different architectures.

'reg_sr'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'reg_sr' ;
```

Use

Signal attribute 'reg_sr' indicates that the associated signal has an SR-type flip-flop as its memory element. Equations generated from an ABEL-HDL state diagram will assume this register type if the 'reg_sr' attribute is specified; however, you will need to specify the 'invert' or 'buffer' attribute to ensure consistent operation in different architectures.

'reg_t'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'reg_t' ;
```

Use

Signal attribute 'reg_t' indicates that the associated signal has a T-type flip-flop as its memory element. Equations generated from an ABEL-HDL state diagram will assume this register type if the 'reg_t' attribute is specified; however, you will need to specify the 'invert' or 'buffer' attribute to ensure consistent operation in different architectures.

'retain'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'retain' ;
```

Use

Signal attribute 'retain' indicates that redundant product terms for this signal should be retained. The 'retain' attribute only controls optimization performed in the compiler. To preserve redundant product terms, you must also specify no reduction for optimization and for the device fitting software.

'xor'

Syntax

```
signal_name [,signal_name] [PIN|NODE] ISTYPE 'xor' ;
```

Use

Signal attribute 'xor' indicates that the target architecture has an XOR gate, so one top-level exclusive-OR operator is retained in the design equations.

Library

Syntax

```
LIBRARY 'name' ;
```

Purpose

The LIBRARY statement causes the contents of the indicated file to be inserted in the ABEL-HDL source file. The insertion begins at the LIBRARY statement.

Use

name — A string that specifies the name of the library file, excluding the file extension.

The file extension of '.inc' is appended to the name specified, and the resulting filename is searched for. If no file is found, the abel5lib.inc library file is searched.

Macro

Syntax

```
macro_id MACRO [ (dummy_arg [,dummy_arg ]... ) ] {block } ;
```

Purpose

The macro declaration statement defines a macro. Macros are used to include ABEL-HDL code in a source file without typing or copying the code everywhere it is needed.

Use

macro_id — An identifier naming the macro

dummy_arg — A dummy argument

block — A block

A macro is defined once in the declarations section of a module and then used anywhere within the module as frequently as needed. Macros can be used only within the module in which they are declared.

Wherever the macro_id occurs, the text in the block associated with that macro is substituted. With the exception of dummy arguments, all text in the block (including spaces and end-of-lines) is substituted exactly as it appears in the block.

When debugging your source file, you can use the **-list expand** option to examine macro statements. The **-list expand** option causes the parsed and expanded source code (and the macros and directives that caused code to be added to the source) to be written to the listing file.

Macros and Declared Equations

Use declared equations for constant expressions (instead of macros) for faster processing. The file, **mac.abl**, in Example – Differences Between MACRO and Declared Equations demonstrates the difference:

Examples

The dummy arguments used in the macro declaration allow different actual arguments to be used each time the macro is referenced. Dummy arguments are preceded by a “?” to indicate that an actual argument is substituted for the dummy by the compiler.

The equation,

```
NAND3 MACRO (A,B,C) { !(?A & ?B & ?C) } ;
```

declares a macro named NAND3 with the dummy arguments A, B, and C. The macro defines a three-input NAND gate. When the macro identifier occurs in the source, actual arguments for A, B, and C are supplied.

For example, the equation

```
D = NAND3 (Clock,Hello,Busy) ;
```

brings the text in the block associated with NAND3 into the code, with Clock substituted for ?A, Hello for ?B, and Busy for ?C.

This results in:

```
D = !( Clock & Hello & Busy ) ;
```

which is the three-input NAND.

The macro NAND3 has been specified by a Boolean equation, but it could have been specified using another ABEL-HDL construct, such as the truth table shown here:

```
NAND3 MACRO (A,B,C,Y)
{ TRUTH_TABLE ( [?A ,?B ,?C ] -> ?Y )
[ 0 ,.X.,.X.] -> 1 ;
[.X., 0 ,.X.] -> 1 ;
[.X.,.X., 0 ] -> 1 ;
[ 1 , 1 , 1 ] -> 0 ; } ;
```

In this case, the line,

```
NAND3 (Clock,Hello,Busy,D)
```

causes the text,

```
TRUTH_TABLE ( [Clock,Hello,Busy] -> D )
[ 0 , .X. ,.X. ] -> 1 ;
[ .X. , 0 ,.X. ] -> 1 ;
[ .X. , .X. , 0 ] -> 1 ;
[ 1 , 1 , 1 ] -> 0 ;
```

to be substituted into the code. This text is a truth table definition of D, specified as the function of three inputs, Clock, Hello, and Busy. This is the same function as that given by the Boolean equation above. The truth table format is discussed under Truth_table.

Other examples of macros:

```
"macro with no dummy arguments
nodum macro { W = S1 & S2 & S3 ; } ;
onedum MACRO (d) { !?d } ; "macro with 1 dummy argument
```

and when macros are called in logic descriptions:

```
nodum
X = W + onedum(inp) ;
Y = W + onedum( )C ; "note the blank actual argument
```

resulting in:

```
"note leading space from block in nodum
W = S1 & S2 & S3 ;
X = W + ! inp ;
Y = W + ! C ;
```

Recursive macro references (when a macro definition refers to itself) are not supported, and the compiler halts abnormally. If errors appear after the first use of a macro, and the errors cannot be easily explained otherwise, check for a recursive macro reference by examining the listing file.

Module

Syntax

```
MODULE modname [ ( dummy_arg [,dummy_arg ] ... ) ]
```

Purpose

The module statement defines the beginning of a module and must be paired with an END statement that defines the module's end.

Use

modname — An identifier naming the module.

dummy_arg — Dummy arguments.

The optional dummy arguments allow actual arguments to be passed to the module when it is processed. The dummy argument provides a name to refer to within the module. Anywhere in the module where a dummy argument is found preceded by a "?", the actual argument value is substituted.

Examples

```
MODULE MY_EXAMPLE (A,B)
:
C = ?B + ?A
```

In the module named MY_EXAMPLE, C takes on the value of "A + B" where A and B contain actual arguments passed to the module when the language processor is invoked.

Node

Syntax

```
[!]node_id [,[!]node_id...] NODE [node# [,node# ]] [ISTYPE 'attributes' ];
```

Purpose

The NODE keyword declares signals assigned to buried nodes.

Use

node_id — An identifier used for reference to a node in a logic design.

node# — The node number on the real device.

attributes — A string that specifies node attributes for devices with programmable nodes. Any number of attributes can be listed, separated by commas.

***Note**: Using the NODE keyword does not restrict a signal to a buried node. A signal declared with NODE can be assigned to a device I/O pin by a device fitter.*

You can use the range operator (..) to declare sets of nodes. The ending semicolon is required after each declaration.

When lists of *node_id* and *node #* are used in one node declaration, there is a one-to-one correspondence between the identifiers and numbers.

The following example declares three nodes A, B, and C.

```
A, B, C NODE ;
```

The node attribute string, **Istype '*attributes*,'** should be used to specify node attributes. Since a node declaration is only required in a detailed description, use detailed attributes, not pin-to-pin attributes. The ISTYPE statement and attributes are discussed under Istype.

The node declaration,

```
B NODE istype 'reg' ;
```

specifies that node B is a buried flip-flop.

Example

```
a0..a3 node 22..25;
```

assigns a0, a1, a2 and a3 to nodes 22, 23, 24 and 25, respectively.

Pin

Syntax

```
[!]pin_id [,[!]pin_id...] PIN  [pin# [, pin# ]] [ISTYPE 'attr' ];
```

Purpose

The PIN keyword declares input and output signals that must be available on a device I/O pin.

Use

pin_id — An identifier that refers to a pin in a module.

pin# — The pin number on the physical device.

attr — A string that specifies pin attributes for devices with programmable pins. Attributes are listed in ISTYPE.

When lists of *pin_id*s and *pin#*s are used in a pin declaration statement, there is a one-to-one correspondence between the identifiers and numbers given. There must be one pin number associated with each identifier listed.

You can use the range operator (..) to declare sets of pins. The ending semicolon is required after each declaration.

***Note**: Assigning pin numbers defines the particular pin-outs necessary for the design. Pin numbers only limit the device selection to a minimum number of input and output pins. Pin number assignments can be changed later by a fitter.*

The **!** operator in pin declarations indicates that the pin is active-low, and is automatically negated when the source file is compiled.

The pin attribute string, **Istype '*attributes*,'** should be used to specify pin attributes. The ISTYPE statement and attributes are discussed under Istype. Istype attribute statements are recommended for all pins.

Examples

```
Clock, !Reset, S1 PIN 1,15,3;
```

Clock is assigned to pin 1, Reset to pin 15, and S1 to pin 3.

```
a0..a3 PIN 2..5 istype 'reg,buffer';
```

Assigns a0, a1, a2 and a3 to pins 2, 3, 4 and 5, respectively.

Property

Syntax

```
property_id PROPERTY 'string' ;
```

Purpose

The **property** declaration statement allows you to specify additional design information associated with an external processing module (such as a device kit).

The format of the string depends on the fitter to which the property is being passed. See your device kit user manuals for syntax descriptions.

***Note**: You can specify properties for any number of fitters in your design, since all fitters process only properties with their property ID and ignore all other properties.*

Use

property_id — Identifies properties relevant to specific external modules, such as fitters.

string — Argument containing the actual property data.

***Caution1**: Property IDs and strings can be case-sensitive. Check your vendor's fitter documentation.*

***Caution2**: Property Information will not be present in pre-route/functional simulation. Consider using schematics to access property features that affect simulation*

Example

```
AMDMACH property 'GROUP A Q7 Q6 Q5 Q4 Q3 Q2 Q1 Q0';.
```

State (Declarations)

Syntax

```
state_id [, state_id ...] STATE [IN statereg_id] ;
```

Purpose

The State declaration is made to declare a symbolic state name and, optionally, associate it with a state register.

Use

state_id — A symbolic state name to be referenced in a symbolic state description.

statereg_id — An identifier for a state register.

If your design includes more than one symbolic state machine, use the IN keyword to associate each state with the corresponding state register.

Each state you declare corresponds to one flip-flop in a one-hot machine.

State (in State_diagram)

Syntax

```
[STATE state_exp : [equation ]
[equation ]
:
:
:
trans_stmt ; ...]
```

Purpose

The state keyword and the associated section describes one state of a state diagram. It includes a state value (or a symbolic state name), a state transition statement, and optional state output equations.

Use

state_exp — An expression, value, or symbolic state name giving the current state.

equation — An equation that defines the state machine outputs.

trans_stmt — IF-THEN-ELSE, CASE, or GOTO statements, optionally followed by WITH transition equations.

The specification of a state description requires the use of the **State_diagram** syntax (which defines the state machine) and the **If-Then-Else, Case, Goto**, and **With** statements (which determine the operation of the state machine). Symbolic state machines (machines for which the actual state registers and state values are unspecified) require additional declarations for the symbolic state register and state names.

A semicolon is required after each transition statement.

State_diagram

Syntax

```
State_diagram state_reg
 [-> state_out ]
[STATE state_exp : [equation ]
 [equation ]
:
trans_stmt ; ...]
```

Purpose

The state description describes the operation of a sequential state machine implemented with programmable logic.

Use

state_reg — An identifier or set of identifiers specifying the signals that determine the current state of the machine. For symbolic state diagrams, this identifier is a symbolic state register name that has been declared with a State_register declaration.

state_out — An identifier or set of identifiers that determines the next state of the machine (for designs with external registers).

state_exp — An expression or symbolic state name giving the current state.

Equation — An equation that defines the state machine outputs.

trans_stmt — IF-THEN-ELSE, CASE, or GOTO statements, optionally followed by WITH transition equations.

A semicolon is required after each transition statement.

Use **State_diagram** syntax to define a state machine, and the **If-Then-Else, Case, Goto,** and **With** statements to determine the operation of the state machine. Symbolic state machines (machines for which the actual state registers and state values are unspecified) require additional declarations for the symbolic state register and state names (see *Symbolic State Declarations* in Chapter 4, "ABEL-HDL Design").

The syntax for the IF-THEN-ELSE, CASE, GOTO, WITH, SYNC_RESET, and ASYNC_RESETstatements are presented here briefly, and are discussed further in their respective sections.

A state machine starts in one of the states defined by *state_exp*. The equations listed after that state are evaluated, and the transition statement (*trans_stmt*) is evaluated after the next clock, causing the machine to advance to the next state.

Equations associated with a state are optional; however, each state must have a transition statement. If none of the transition conditions for a state is met, the next state is undefined. (For some devices, undefined state transitions cause a transition to the cleared register state.)

Transitions Statements

Transition statements describe the conditions that cause transition from one state to the next. Each state in a state diagram must contain at least one transition statement. Transition statements can consist of GOTO statements, IF-THEN-ELSE conditional statements, CASE statements, or combinations of these different statements.

GOTO Syntax

```
GOTO state_exp ;
```

The GOTO statement unconditionally jumps to a different state. When GOTO is used, it is the only transition for the current state. Example:

```
STATE S0:
GOTO S1; "unconditional branch to state S1
```

CASE Syntax

```
CASE expression : state_exp ;
[ expression : state_exp ; ] ...
ENDCASE ;
```

The CASE statement is used to list a sequence of mutually exclusive transition conditions and corresponding next states. Example:

```
STATE S0:
CASE (sel == 0): S0 ;
(sel == 1): S1 ;
ENDCASE
```

CASE statement conditions must be mutually exclusive. No two transition conditions can be true at the same time, or the resulting next state is unpredictable.

IF-THEN-ELSE Syntax

```
IF expression THEN state_exp
[ ELSE state_exp ] ;
```

IF-THEN-ELSE statements specify mutually exclusive transition conditions. Example:

```
STATE S0:
IF (address > ^hE100) THEN S1 ELSE S2;
```

You can use blocks in IF-THEN-ELSE statements, for example,

```
IF (Hold) THEN State1 WITH {o1 := o1.fb; o2 := o2.fb;}
ELSE State2;
```

The ELSE clause is optional. A sequence of IF-THEN statements with no ELSE clauses is equivalent to a sequence of CASE statements. IF-THEN-ELSE statements can be chained and nested. See IF-THEN-ELSE for more information.

WITH Syntax

```
state_exp WITH equation ;
[equation ; ]
```

You can use the WITH statement in any of the above transition statements (the GOTO, IF-THEN-ELSE, or CASE statements) in place of a simple state expression. For example, to specify that a set of registered outputs are to contain a specific value after one particular transition, specify the equation using a WITH statement similar to the one shown below:

```
STATE S0:
IF (reset)
THEN S9 WITH {
ErrorFlag := 1;
ErrorAddress := address;
}
ELSE
IF (address <= ^hE100)
THEN S2
ELSE
S0;
```

The WITH statement is also useful when you describe output behavior for registered outputs (since registered outputs written only for a current state would lag by one clock cycle).

SYNC_RESET and ASYNC_RESET Syntax

In symbolic state descriptions the SYNC_RESET and ASYNC_RESET statements are used to specify synchronous or asynchronous state machine reset logic in terms of symbolic states. For example, to specify that a state machine must asynchronously reset to state Start when the Reset input is true, you would write

```
ASYNC_RESET Start : Reset ;
```

State Descriptions and Pin-to-Pin Descriptions

Sequential circuits described with ABEL-HDL's state diagram language are normally written with a pin-to-pin behavior in mind, regardless of the flip-flop type specified.

The architecture-independent State Machine operates the same (in terms of the behavior seen on its outputs) no matter what type of register is substituted for 'reg' in the signal declarations. To allow this flexibility, the

specification of 'buffer' or 'invert' is required when a state diagram is written for a register type other than 'reg.

State_register

Syntax

```
statereg_id STATE_REGISTER ;
```

Purpose

For symbolic state diagrams, the State_register is made to declare a symbolic state machine name.

Use

statereg_id — An identifier naming the state register.

Sync_reset

See Async_reset.

Test_Vectors

Syntax

```
Test_vectors [note ]
(input [,input ]... -> output[,output ]...)

[invalues -> outvalues ; ]
:
```

Purpose

Test vectors specify the expected functional operation of a logic device by explicitly defining the device outputs as functions of the inputs.

***Note**: Test_vectors are only used with PLD JEDEC file simulation. For functional simulation with the ABEL Simulator, use a test stimulus file.*

Use

note — An optional string that describes the test vectors.

inputs — An identifier or set of identifiers specifying the names of the input signals, or feedback output signals.

outputs — An identifier or set of identifiers specifying the output signals.

invalues — An input value or set of input values.

outvalues — A pin-to-pin output value or set of output values resulting from the given inputs.

Test vectors are used for simulation of an internal model of the device and functional testing of the design and device. The number of test vectors is unlimited.

The format of the test vectors is determined by the header. Each vector is specified in the format described within the parentheses in the header statement. An optional note string can be specified in the header to describe what the vectors test, and is included as output in the simulation output file, the document output file, and the JEDEC programmer load file.

The table lists input combinations and their resulting outputs. All or some of the possible input combinations can be listed. All values specified in the table must be constants, either declared, numeric, or a special

constant (for example, .X. and .C.). Each line of the table (each input/output listing) must end with a semicolon. Test vector output values always represent the pin-to-pin value for the output signals.

Test vectors must be sequential for state machines. Test vectors must go through valid state transitions.

The **Trace** keyword can be used to control simulator output from within the source file.

Functional testing of the physical device is performed by a logic programmer after a device has been programmed. The test vectors become part of the programmer load file.

Examples

Following is a simple test vectors section:

```
TEST_VECTORS
( [A,B] -> [C, D] )

[0,0] -> [1,1] ;
[0,1] -> [1,0] ;
[1,0] -> [0,1] ;
[1,1] -> [0,0] ;
```

The following test vectors are equivalent to those specified above because values for sets can be specified with numeric constants.

```
TEST_VECTORS
( [A,B] -> [C, D] )

0 -> 3 ;
1 -> 2 ;
2 -> 1 ;
3 -> 0 ;
```

If the signal identifiers in the test vector header are declared as active-low in the declaration section, then constant values specified in the test vectors are inverted accordingly (interpreted pin-to-pin).

Title

Syntax

```
title 'string'
```

Purpose

The title statement gives a module a title that appears as a header in both the programmer load file and documentation file created by the language processor.

Use

The title is specified in the string following the keyword, **title**. The string is opened and closed by an apostrophe and is limited to 324 characters.

The title statement is optional.

Asterisks in the title string do not appear in the programmer load file header in order to conform to the JEDEC standard.

Examples

An example of a title statement that spans three lines and describes the logic design is shown below:

```
module m6809a
title '6809 memory decode
Jean Designer
Data I/O Corp Redmond WA'
```

Truth_table

Syntax

```
TRUTH_TABLE ( in_ids -> out_ids )
inputs -> outputs ;
```

or

```
TRUTH_TABLE ( in_ids :> reg_ids )
inputs :> reg_outs ;
```

or

```
TRUTH_TABLE
( in_ids :> reg_ids -> out_ids )
inputs :> reg_outs -> outputs ;
```

Purpose

Truth tables specify outputs as functions of input combinations, in a tabular form.

Use

in_ids — Input signal identifiers.

out_ids — Output signal identifiers.

reg_ids — Registered signal identifiers.

Inputs — The inputs to the logic function.

outputs — The outputs from the logic function.

reg_outs — The registered (clocked) outputs.

-> :> — Indicates the input to output function for combinational (->) and registered (:>) outputs.

Truth tables are another way to describe logic designs with ABEL-HDL and may be used in lieu of (or in addition to) equations and state diagrams. A truth table is specified with a header describing the format of the table and with the table itself.

A semicolon is required after each line in the truth table.

The truth table header can have one of the three forms shown above, depending on whether the device has registered or combinational outputs or both.

The inputs and outputs (both registered and combinational) of a truth table are either single signals, or (more frequently) sets of signals. If only one signal is used as either the input or output, its name is specified. Sets of signals used as inputs or outputs are specified in the normal set notation with the signals surrounded by brackets and separated by commas.

The syntax shown in the first form defines the format of a truth table with simple combinational outputs. The values of the inputs determine the values of the outputs.

The second form describes a format for a truth table with registered outputs. The symbol ":" preceding the outputs distinguishes these outputs from the combinational outputs. Again the values of the inputs determine the values of the outputs, but now the outputs are registered or clocked: they will contain the new value (as determined by the inputs) after the next clock pulse.

The third form is more complex, defining a table with both combinational and registered outputs. It is important in this format to make sure that the different specification characters "-" and ":" are used for the different types of outputs.

Truth Table Format

The truth table is specified according to the form described within the parentheses in the header. The truth table is a list of input combinations and resulting outputs. All or some of the possible input combinations may be listed.

All values specified in the table must be constants, either declared, numeric, or the special constant .X. Each line of the table (each input/output listing) must end with a semicolon.

The header defines the names of the inputs and outputs. The table defines the values of inputs and the resulting output values.

Examples

This example shows a truth table description of a simple state machine with four states and one output. The current state is described by signals A and B, which are put into a set. The next state is described by the registered outputs C and D, which are also collected into a set. The single combinational output is signal E. The machine simply counts through the different states, driving the output E low when A equals 1 and B equals 0.

```
TRUTH_TABLE ( [A,B] :> [C,D] -> E )
0 :> 1 -> 1 ;
1 :> 2 -> 0 ;
2 :> 3 -> 1 ;
3 :> 0 -> 1 ;
```

Note that the input and output combinations are specified by a single constant value rather than by set notation. This is equivalent to:

```
[0,0] :> [0,1] -> 1 ;
[0,1] :> [1,0] -> 0 ;
[1,0] :> [1,1] -> 1 ;
[1,1] :> [0,0] -> 1 ;
```

When writing truth tables in ABEL-HDL (particularly when describing registered circuits), follow the same rules for dot extensions, attributes, and pin-to-pin/detailed descriptions described earlier for writing equations. The only difference between equations and truth tables is the ordering of the inputs and outputs.

The following two fragments of source code, for example, are functionally equivalent:

Fragment 1:

```
equations

q := a & load # !q.fb & !load;
```

Fragment 2:

```
truth_table ( [a ,q.fb,load] :> q)
[0 , 0 , 0 ] :> 1;
[0 , 1 , 0 ] :> 0;
[1 , 0 , 0 ] :> 1;
[1 , 1 , 0 ] :> 0;
[0 , 0 , 1 ] :> 0;
[1 , 0 , 1 ] :> 1;
[0 , 1 , 1 ] :> 0;
[1 , 1 , 1 ] :> 1;
```

As an example, the following truth table defines an exclusive-OR function with two inputs (A and B), one enable (en), and one output (C):

```
TRUTH_TABLE ( [en, A , B ] -> C )
[ 0,.X.,.X.] -> .X.;" don't care w/enab off
[ 1, 0 , 0 ] -> 0 ;
[ 1, 0 , 1 ] -> 1 ;
[ 1, 1 , 0 ] -> 1 ;
[ 1, 1 , 1 ] -> 0 ;
```

Wait

This keyword represents the relative time delay which effects the test vectors immediately following the WAIT statement. It can be used in the MACRO block only when MACRO is used in the TEST_VECTORS block. The default time unit is nano-second. The syntax for WAIT is:

```
WAIT < integer>;
```

Example

```
test_vectors MACRO {
WAIT 10;
1 -> [0,1];
WAIT 24;
6 -> [0,1];
WAIT 34;
8 -> [1.0];
};
```

***Note**: The Wait keyword is only defined in the Lattice Logic Simulator. They cannot be used with all of the supported special constants, except .R. See Lattice Logic Simulator Help – "Special Constants" for information.*

When-Then-Else

Syntax

```
[ WHEN condition THEN ] [ ! ] element=expression;
[ ELSE equation ];
```

or

```
[ WHEN condition THEN ] equation; [ ELSE equation];
```

Purpose

The **When-then-else** statement is used in equations to describe a logic function.

Use

condition — Any valid expression.

element — An identifier naming a signal or set of signals, or an actual set, to which the value of the expression is assigned.

expression — Any valid expression.

=, :=, ?= and ?:= — Combinational and registered (pin-to-pin) assignment operators.

Equations use the assignment operators = and ?= (combinational), and := and ?:= (registered).

The complement operator, "!", expresses negative logic. The complement operator precedes the signal name and implies that the expression on the right of the equation is to be complemented before it is assigned to the signal. Using the complement operator on the left side of equations is also supported; equations for negative logic parts can just as easily be expressed by complementing the expression on the right side of the equation.

***Caution**: When-Then-Else is only supported in equations. Use If-Then-Else in state_diagram descriptions.*

***Note**: Equation blocks in conditional expressions such as WHEN-THEN result in logic functions that are logically ANDed with the conditional expression that is in effect.*

Examples

```
WHEN (Mode == S_Data) THEN { Out_data := S_in;
S_Valid := 1; }
ELSE WHEN (Mode == T_Data) THEN { Out_data := T_in;
T_Valid := 1; }
```

With

Syntax

```
trans_stmt state_exp WITH equation
[equation ]..;
```

Purpose

The WITH statement is used in the **State_diagram** section. When used in conjunction with the IF-THEN or CASE statement, it allows output equations to be written in terms of transitions.

Use

trans_stmt — The IF-THEN-ELSE, GOTO, or CASE statement.

state_exp — The next state.

equation — An equation for state machine outputs.

You can use the WITH statement in any transition statement, in place of a simple state expression.

The WITH statement is also useful when you are describing output behavior for registered outputs, since registered outputs written only for a current state would lag by one clock cycle.

To specify that a set of registered outputs should contain a specific value after one particular transition, specify the equation using a WITH statement similar to the one shown below:

```
STATE S0:
IF (reset) THEN S9 WITH { ErrorFlag := 1;
ErrorAddress := address;}
ELSE IF (address <= ^hE100)
THEN S2
ELSE S0;
```

Examples

```
State 5 :
IF a == 1 then 1 WITH { x := 1 ;
y := 0 ;}
ELSE 2 WITH { x := 0 ;
y := 1 ;}
```

XOR_factors

Syntax

```
XOR_Factors
signal name = xor_factors ;
```

Purpose

Use XOR_factors to specify a Boolean expression to be factored out of (and XORed with) the sum-of-products reduced equations. Factors can dramatically reduce the reduced equations if you use a device featuring XOR gates.

Use

XOR_factors converts a sum of products (SOP) equation into an exclusive OR (XOR) equation. The resulting equation contains the sum of product functions that, when exclusive ORed together, have the same function as the original. The XOR_Factors equation is divided into the original equation, with the factor (or its complement) on one side of the XOR and the remainder on the other.

After deciding the best XOR_Factors, remember to revise the source file to use an XOR device for the final design.

***Note**: The assignment operator you use in XOR_Factors equations must match the assignment operator in the Equations section.*

Examples

```
!Q16 = A & B & !D
# A & B & !C
# !B & C & D
# !A & C & D;
```

Reordering the product terms indicates that (A & B) and (C & D) are good candidate factors, as shown below:

```
!Q16 = A & B & (!C # !D)
# (!A # !B) & C & D;
```

If we process the following source file, the program reduces the equations according to the XOR_Factors, A & B.

```
module xorfact
xorfact device 'P20X10';
Clk,OE pin 1,13;
A,B,C,D pin 2,3,4,5;
Q16 pin 16 istype 'reg,xor';
XOR_Factors
Q16 := A & B;
equations
!Q16 := A & B & !D
# !B & C & D
# !A & C & D
# A & B & !C;
end
```

Using A & B as the XOR_Factors, the reduced equations are

```
!Q16 := ((A & B) $ (C & D));
```

Example 2

The example octalf.abl uses a more complex high-level equation:

```
module OCTALF
title 'Octal counter with xor factoring'

octalf device 'P20X8';
D0..D7 pin 3..10;
Q7..Q0 pin 15..22 istype 'reg,xor';
CLK,I0,I1,OC,,CarryIn pin 1,2,11,13,23;
CarryOut pin 14 istype 'com';
H,L,X,Z,C = 1, 0, .X., .Z., .C.;

Data = [D7..D0];
Count = [Q7..Q0];

Mode = [I1,I0];
Clear = [ 0, 0];
Hold = [ 0, 1];
Load = [ 1, 0];
Inc = [ 1, 1];

xor_factor
Count.FB := Count & I0;

" ..comments removed..

equations
Count := (Count.FB + 1) & (Mode == Inc) & !CarryIn
# (Count.FB ) & (Mode == Inc) & CarryIn
```

```
# (Count.FB ) & (Mode == Hold)
# (Data ) & (Mode == Load)
# (0 ) & (Mode == Clear);

!CarryOut = !CarryIn & (Count.FB == ^hFF);

Count.C = CLK;
Count.OE = !OC;
"..test vectors removed..
"..comments removed..
end OCTALF;
```

ABEL-HDL Language Structure

Basic Structure

ABEL-HDL source files can contain independent modules. Each module contains a complete logic description of a circuit or sub-circuit. Any number of modules can be combined into one source file and processed at the same time.

This section covers the basic elements that make up an ABEL-HDL source file module. A module can be divided into five sections:

- Header
- Declarations
- Logic Description
- Test Vectors
- End

Structure Rules

The following rules apply to module structure:

- A module must contain only one header (composed of the Module statement and optional Title and Options statements).
- All other sections of a source file can be repeated in any order. Declarations must immediately follow either the header or the Declarations keyword.
- No symbol (identifier) can be referenced before it is declared.

Header Section

The Header section can consist of the following elements:

- Module (required)
- Interface (lower level, optional)
- Title

Module

Syntax

```
MODULE modname [ ( dummy_arg [,dummy_arg ] ... ) ]
```

Purpose

The module statement defines the beginning of a module and must be paired with an END statement that defines the module's end.

Use

modname — An identifier naming the module.

dummy_arg — Dummy arguments.

The optional dummy arguments allow actual arguments to be passed to the module when it is processed. The dummy argument provides a name to refer to within the module. Anywhere in the module where a dummy argument is found preceded by a "?", the actual argument value is substituted.

Examples

```
MODULE MY_EXAMPLE (A,B)
:
C = ?B + ?A
```

In the module named MY_EXAMPLE, C takes on the value of "A + B" where A and B contain actual arguments passed to the module when the language processor is invoked.

Interface

Keyword: interface

The interface statement is used in lower-level sources to indicate signals used in upper-level files. The interface statement is optional.

Title

Syntax

```
title 'string'
```

Purpose

The title statement gives a module a title that appears as a header in both the programmer load file and documentation file created by the language processor.

Use

The title is specified in the string following the keyword, **title**. The string is opened and closed by an apostrophe and is limited to 324 characters.

The title statement is optional.

Asterisks in the title string do not appear in the programmer load file header in order to conform to the JEDEC standard.

Examples

An example of a title statement that spans three lines and describes the logic design is shown below:

```
module m6809a
title '6809 memory decode
Jean Designer
```

```
Data I/O Corp Redmond WA'
```

Declarations Section

The declarations section of a module specifies the names and attributes of signals used in the design; defines constants, macros, and states; declares lower-level modules and schematics; and optionally declares a device. Each module must have at least one declarations section, and declarations affect only the module in which they are defined.

The Declarations section can consist of the following elements:

- Declarations Keyword
- Device Declaration
- Hierarchy Declarations
- Signal Declarations
- Constant Declarations
- Symbolic State Declarations
- Macro Declarations
- Library Declarations

Declarations Keyword

Keyword: declarations

This keyword allows declarations (such as sets or other constants) in any part of the source file.

Device Declaration

Keyword: device

```
device_id DEVICE real_device ;
```

The Device declaration is optional, and only one can be made per module. It associates a device identifier with a specific programmable logic device.

Hierarchy (Interface and Functional Block) Declarations

Top-level Interface Declarations

Lower-level Interface Declarations

Functional Block Statement

Example of Functional Block Instantiation

Signal Declarations

The **Pin** and **Node** declarations are made to declare signals used in the design, and optionally to associate pin and/or node numbers with those signals. Actual pin and node numbers do not have to be assigned until you want to map the design into a device. Attributes can be assigned to signals within pin and node declarations with the **Istype** statement. Dot extensions can also be used in equations to precisely describe the signals.

***Note**: Assigning pin numbers defines the particular pin-outs necessary for the design. Pin numbers only limit the device selection to a minimum number of input and output pins. Pin number assignments can be changed later by a fitter.*

Constant Declarations

Syntax

id [,*id*]... = *expr* [,*expr*]... ;

Purpose

A constant declaration that defines constants used in a module.

Use

id — An identifier naming a constant to be used within a module.

expr — An expression defining the constant value.

***Note**: The equal sign (=) used for constant declarations in the Declarations section is also used for equations in the Equations section.*

A constant is an identifier that retains a constant value throughout a module.

The identifiers on the left side of the equals sign are assigned the values listed on the right side. There is a one-to-one correspondence between the identifiers and the expressions listed. There must be one expression for each identifier.

The ending semicolon is required after each declaration.

Constants are helpful when you use a value many times in a module, especially when you may be changing the value during the design process. Constants allow you to change the value once in the declaration of the constant, rather than changing the value throughout the module.

Constant declarations may not be self-referencing. The following examples will cause errors:

```
X = X;
a = b;
b = a;
```

An include file, **constant.inc**, in the ABEL-HDL library, contains definitions for the most frequently used ABEL-HDL constants. To include this file, enter

```
Library 'constant' ;
```

Examples

```
ABC = 3 * 17; " ABC is assigned the value 51
Y = 'Bc' ; " Y = ^h4263 ;
X =.X.; " X means 'don't care'
ADDR = [1,0,15]; " ADDR is a set with 3 elements
A,B,C = 5,[1,0],6; " 3 constants declared here
D pin 6; " see next line
E = [5 * 7,D]; " signal names can be included
G = [1,2]+[3,4]; " set operations are legal
A = B & C; " operations on identifiers are valid
A = [!B,C]; " set and identifiers on right
```

Symbolic State Declarations

The State_register and State declarations are made to declare a symbolic state machine name and to declare symbolic state names.

Macro Declarations

Keyword: macro

```
macro_id MACRO [(dummy_arg [,dummy_arg]... )] {block} ;
```

The macro declaration statement defines a macro. Use macros to include functions in a source file without repeating the code.

Library Declaration

Keyword: library

```
LIBRARY 'name' ;
```

The LIBRARY statement extracts the contents of the indicated file from the ABEL-HDL library and inserts it into your file.

Logic Description Section

One or more of the following elements can be used to describe your design.

- Dot Extensions
- Equations
- Truth Tables
- State Descriptions
- Fuses Declarations
- XOR Factors

In addition, dot extensions (like ISTYPE attributes in the Declarations section) enable you to more precisely describe the behavior of a circuit in a logic description that may be targeted to a variety of different devices.

Dot Extensions

Dot extensions can be specific for certain devices (device-specific) or generalized for all devices (architecture-independent). Device-specific dot extensions are used with detailed syntax; architecture-independent dot extensions are used with pin-to-pin syntax. Detailed and pin-to-pin syntax is described in more detail in “Design Considerations”. Dot extensions can be applied in complex language constructs such as nested sets or complex expressions.

Dot Extension	Description
Pin-to-Pin Syntax, Architecture-independent	
.ACLR	*Asynchronous clear
.ASET	*Asynchronous set
.CLK	Clock input to an edge-triggered flip-flop
.CLR	*Synchronous clear
.COM	*Combinational feedback normalized to the pin value
.FB	Register feedback
.OE	Output enable
.PIN	Pin feedback

Equations

Syntax

```
EQUATIONS
 element [?]= condition ;
 element [?]:= condition ;
 [ WHEN condition THEN ] [ ! ] element=expression;
  [ ELSE equation ];
```

or

```
[ WHEN condition THEN ] equation; [ ELSE equation];
<inst_name> <macro_name> ([<mcr_port>]{,[<mcr_port>]}*);
```

Purpose

The equations statement defines the beginning of a group of equations associated with a device.

Use

condition — An expression.

element — An identifier naming a signal, set of signals, or actual set to which the value of the expression is assigned.

expression — An expression.

=, :=, ?= and ?:= — Combinational and registered (pin-to-pin) on-set and dc-set assignment operators.

when-then-else — When-then-else statements.

inst_name — Identifier.

macro_name — Identifier; Lattice PLL macro name.

mcr_port — Signal(s), signal set, or Boolean expressions.

Equations specify logic functions with an extended form of Boolean algebra. A semicolon is required after each equation. The equations following the equation statement are equations as described in Chapter 1, “Language Structure” of the *ABEL-HDL Reference Manual*.

The equations following the equation statement are equations as described in Chapter 4, “ABEL-HDL Design.”

The syntax <*inst_name*> <*macro_name*> ([<*mcr_port*>]{,[<*mcr_port*>]}*); is enhanced to support Lattice PLL macros in ABEL-HDL. When the Lattice PLL macros are being called, the user-defined macro port numbers must match the macro need. The <*mcr_port*> is the signal(s) in the current design. Two consecutive commas mean a macro port is not connected. The output port of a macro can be dangled, while the input ports or bidirectional ports cannot be dangled.

Examples

A sample equations section follows:

```
equations
A = B & C # A ;
[W,Y] = 3 ;
!F = (B == C) ;
Output.D = In1 # In2
```

Truth Tables

Keyword: truth_table

```
TRUTH_TABLE (inputs -> outputs )
inputs -> outputs ;
:
```

or

```
TRUTH_TABLE (inputs [:> registered outputs] [-> outputs ] )
```

Truth tables specify outputs as functions of input combinations in tabular form.

State Descriptions

Keyword: state_diagram

```
STATE_DIAGRAM state_reg
 [-> state_out]
[STATE state_exp : [equation]
 [equation]
:
:
:
trans_stmt ...]
```

The **State_Diagram** section contains state descriptions that describe the logic design.

The specification of a state description requires the use of the **State_diagram** syntax, which defines the state machine, and the **If-Then-Else, Case,** and **Goto** statements that determine the operation of the state machine.

Fuse Declarations

Keyword: fuses

```
FUSES
fuse_number = fuse value ;
```

or

```
fuse_number_set = fuse value ;
```

The FUSES section explicitly declares the state of fuses in the associated device. A device must be declared before a fuses declaration.

XOR_Factors

Syntax

```
XOR_Factors
signal name = xor_factors ;
```

Purpose

Use XOR_factors to specify a Boolean expression to be factored out of (and XORed with) the sum-of-products reduced equations. Factors can dramatically reduce the reduced equations if you use a device featuring XOR gates.

Use

XOR_factors converts a sum of products (SOP) equation into an exclusive OR (XOR) equation. The resulting equation contains the sum of product functions that, when exclusive ORed together, have the same function as the original. The XOR_Factors equation is divided into the original equation, with the factor (or its complement) on one side of the XOR and the remainder on the other.

After deciding the best XOR_Factors, remember to revise the source file to use an XOR device for the final design.

***Note**: The assignment operator you use in XOR_Factors equations must match the assignment operator in the Equations section.*

Examples

```
!Q16 = A & B & !D
# A & B & !C
# !B & C & D
# !A & C & D;
```

Reordering the product terms indicates that (A & B) and (C & D) are good candidate factors, as shown below:

```
!Q16 = A & B & (!C # !D)
# (!A # !B) & C & D;
```

If we process the following source file, the program reduces the equations according to the XOR_Factors, A & B.

```
module xorfact
xorfact device 'P20X10';
Clk,OE pin 1,13;
A,B,C,D pin 2,3,4,5;
Q16 pin 16 istype 'reg,xor';
XOR_Factors
Q16 := A & B;
equations
!Q16 := A & B & !D
# !B & C & D
# !A & C & D
# A & B & !C;
end
```

Using A & B as the XOR_Factors, the reduced equations are

```
!Q16 := ((A & B) $ (C & D));
```

Example 2

The example octalf.abl uses a more complex high-level equation:

```
module OCTALF
title 'Octal counter with xor factoring'

octalf device 'P20X8';
```

```
D0..D7 pin 3..10;
Q7..Q0 pin 15..22 istype 'reg,xor';
CLK,I0,I1,OC,,CarryIn pin 1,2,11,13,23;
CarryOut pin 14 istype 'com';
H,L,X,Z,C = 1, 0, .X., .Z., .C.;

Data = [D7..D0];
Count = [Q7..Q0];

Mode = [I1,I0];
Clear = [ 0, 0];
Hold = [ 0, 1];
Load = [ 1, 0];
Inc = [ 1, 1];

xor_factor
Count.FB := Count & I0;

" ..comments removed..
equations
Count := (Count.FB + 1) & (Mode == Inc) & !CarryIn
# (Count.FB ) & (Mode == Inc) & CarryIn
# (Count.FB ) & (Mode == Hold)
# (Data ) & (Mode == Load)
# (0 ) & (Mode == Clear);

!CarryOut = !CarryIn & (Count.FB == ^hFF);

Count.C = CLK;
Count.OE = !OC;
"..test vectors removed..
"..comments removed..
end OCTALF;
```

Test Vectors Section

Test vectors are only used for Equation or JEDEC Simulation. A Test Vectors section can consist of the following elements:

- Test Vectors
- Trace Statement

Test Vectors

Keyword: test_vector

```
Test_vectors [note ]
(inputs -> outputs)
[invalues -> outvalues ; ] ...
```

Test vectors specify the expected operation of a logic device by defining its outputs as a function of its inputs.

Trace Statement

Keyword: trace

```
trace (inputs -> outputs) ;
```

The Trace statement limits which inputs and outputs are displayed in the simulation report.

End Statement

Keyword: end

```
end module_name
```

The **End** statement ends the module, and is required.

Other Elements

Directives

Keyword: *@directive*

```
@directive [options ]
```

Directives provide options that control the contents or processing of a source file. Sections of ABEL-HDL source code can be included conditionally, code can be brought in from another file, and messages can be printed during processing.

Some directives take arguments that determine how the directive is processed. These arguments can be actual arguments or dummy arguments preceded by a question mark. The rules applying to actual and dummy arguments are presented under “Arguments and Argument Substitution” earlier in this chapter.

Available directives are listed below. See “Directives” in *ABEL-HDL Language Reference* for complete information.

@ALTERNATE	@IFNDEF
@CARRY	@IFNIDEN
@CONST	@INCLUDE
@DCSET	@IRP
@DCSTATE	@IRPC
@EXPR	@MESSAGE
@EXIT	@ONSET
@IF	@PAGE
@IFB	@RADIX
@IFDEF	@REPEAT
@IFIDEN	@SETSIZE
@IFNB	@STANDARD

Basic Syntax

The basic syntax of an ABEL-HDL source file includes the following:

- Supported ASCII characters
- Identifiers and keywords
- Constants
- Blocks
- Comments

- Numbers
- Strings
- Operators, expressions, and equations
  - Logical operators
  - Arithmetic operators
  - Relational operators
  - Assignment operators
  - Expressions
  - Equations
- Sets and set operation
- Arguments and argument substitution

Syntax Rules

Each line in an ABEL-HDL source file must conform to the following syntax rules and restrictions:

- A line can be up to 150 characters long.
- Lines are ended by a line feed (hex 0A), by a vertical tab (hex 0B), or by a form feed (hex 0C). Carriage returns in a line are ignored, so common end-of-line sequences, such as carriage return/line feed, are interpreted as line feeds. In most cases, you can end a line by pressing RETURN.
- Keywords, identifiers, and numbers must be separated by at least one space. Exceptions to this rule are lists of identifiers separated by commas, expressions where identifiers or numbers are separated by operators, or where parentheses provide the separation.
- Neither spaces nor periods can be imbedded in the middle of keywords, numbers, operators, or identifiers. Spaces can appear in strings, comments, blocks, and actual arguments. For example, if the keyword MODULE is entered as MOD ULE, it is interpreted as two identifiers, MOD and ULE. Similarly, if you enter 102 05 (instead of 10205), it is interpreted as two numbers, 102 and 5.
- Keywords can be uppercase, lowercase or mixed-case.
- Identifiers (user-supplied names and labels) can be uppercase, lowercase or mixed-case, but are case sensitive: the identifier, **output**, typed in all lowercase letters, is not the same as the identifier, **Output**.

Supported ASCII Characters

All uppercase and lowercase alphabetic characters and most other characters on common keyboards are supported. Valid characters are listed or shown below.

```
a - z (lowercase alphabet)
A - Z (uppercase alphabet)
0 - 9 (digits)
<space>
<tab>
! @ # $ ? + & * ( ) -

_ = + [ ] { } ; : ' "

` \ | , < > . / ^ %
```

Identifiers

Identifiers are names that identify the following items:

- Devices
- Device pins or nodes
- Functional blocks
- Sets
- Input or output signals
- Constants
- Macros
- Dummy arguments

The rules and restrictions for identifiers are the same regardless of what the identifier describes.

Identifier Rules

The rules governing identifiers are listed below:

- Identifiers can be up to 31 characters. Longer names are flagged as an error.
- Identifiers must begin with an alphabetic character or with an underscore.
- Other than the first character, identifiers can contain upper- and lowercase characters, digits, tildes (~), and underscores.
- You cannot use spaces in an identifier. Use underscores or uppercase letters to separate words.
- Except for Reserved Identifiers (Keywords), identifiers are case sensitive: uppercase letters and lowercase letters are not the same.
- You cannot use periods in an identifier, except with a supported dot extension.

Supported Identifiers

Some supported Identifiers are listed below:

```
HELLO
hello
_K5input
P_h
This_is_a_long_identifier
AnotherLongIdentifier
```

Some unsupported identifiers are listed below:

```
7_ Does not begin with a letter or underscore
$4 Does not begin with a letter or underscore
HEL.LO Contains a period (.LO is not a valid dot extension)
b6 kj Contains a space
```

The last of these identifiers is interpreted as two identifiers, b6 and kj.

Reserved Identifiers (Keywords)

The keywords listed below are reserved identifiers. Keywords cannot be used to name devices, pins, nodes, constants, sets, macros, or signals. If a keyword is used in the wrong context, an error is flagged.

async_reset	goto	state
case	if	state_diagram
cycle	in	state_register
declarations	interface	sync_reset
device	istype	test_vectors
else	Lattice	then
end	library	title
endcase	macro	trace
endwith	module	truth_table
equations	node	when
external	options	with
functional_block	pin	xreset
fuses	property	xtest_oe

Choosing Identifiers

Choosing the right identifiers can make a source file easy to read and understand. The following suggestions can help make your logic descriptions self-explanatory, eliminating the need for extensive documentation.

- Choose identifiers that match their function. For example, the pin you're going to use as the carry-in on an adder could be named Carry_In. For a simple OR gate, the two input pins might be given the identifiers IN1 and IN2, and the output might be named OR.
- Avoid large numbers of similar identifiers. For example, do not name the outputs of a 16-bit adder: ADDER_OUTPUT_BIT_1 ADDER_OUTPUT_BIT_2 and so on.
- Use underscores or mixed-case characters to separate words in your identifier.

```
THIS_IS_AN_IDENTIFIER
ThisIsAnIdentifier
```

is much easier to read than

```
THISISANIDENTIFIER
```

Constants

You can use constant values in assignment statements, truth tables, and test vectors. You can assign a constant to an identifier, and then use the identifier to specify that value throughout a module. Constant values can be either numeric or one of the non-numeric special constant values. The special constant values are listed in the table below.

Special constants

Constant	Description
.C.	Clocked input (low-high-low transition)
.D.	Clock down edge (high-low transition)
.F.	Floating input or output signal
.K.	Clocked input (high-low-high transition)
.P.	Register preload
.SVn.	n = 2 through 9. Drive the input to super voltage 2 through 9.

Constant	Description
.U.	Clock up edge (low-high transition)
.X.	Don’t care condition.
.Z.	Tristate value

When you use a special constant, it must be entered as shown in the above table. Without the periods, .C. is an identifier named C. You can enter special constants in upper- or lowercase.

Blocks

Blocks are sections of text enclosed in braces, { and }. Blocks are used in equations, state diagrams, macros, and directives. The text in a block can be on one line or it can span many lines. Some examples of blocks are shown below.

```
{ this is a block }
{ this is also a block, and it
spans more than one line. }

{ A = B # C;
D = [0, 1] + [1, 0];
}
```

Blocks can be nested within other blocks, as shown below, where the block { D = A } is nested within a larger block:

```
{ A = B $ C;
{ D = A; }
E = C; }
```

Blocks and nested blocks can be useful in macros and when used with directives.

If you need a brace as a character in a block, precede it with a backslash. For example, to specify a block containing the characters { }, write:

```
{ \{ \} }
```

Using Blocks in Logic Descriptions

Using blocks can simplify the description of output logic in equations and state diagrams and allow more complex functions than possible without blocks. Blocks can also improve the readability of your design.

Blocks are supported anywhere a single equation is supported. You can use blocks in simple equations, **When-Then-Else**, **If-Then-Else**, **Case**, and **With** statements

When you use equation blocks within a conditional expression (such as **If-Then**, **Case,** or **When-Then**), the logic functions are logically ANDed with the conditional expression.

Blocks in Equations

The following expressions, written without blocks, are limited by the inability to specify more than one output in a **When-Then** expression without using set notation:

Example 1: Without blocks

```
WHEN (Mode == S_Data) THEN Out_data := S_in;
ELSE WHEN (Mode == T_Data) THEN Out_data := T_in;
WHEN (Mode == S_Data) THEN S_Valid := 1;
ELSE WHEN (Mode == T_Data) THEN T_Valid := 1;
```

With blocks (delimited with braces { }), the syntax above can be simplified. The logic specified for Out_data is logically ANDed with the WHEN clause:

Example 2: With blocks

```
WHEN (Mode == S_Data) THEN { Out_data := S_in;
S_Valid := 1; }
ELSE WHEN (Mode == T_Data) THEN { Out_data := T_in;
T_Valid := 1; }
```

Blocks in State Diagrams

Blocks also provide a simpler way to write state diagram output equations. For example, the following two state transition statements are equivalent.

Example 3: Without Blocks

```
IF (Hold) THEN State1 WITH o1 := o1.fb; o2 := o2.fb;
ENDWITH
ELSE State2;
```

Example 4: With Blocks

```
IF (Hold) THEN State1 WITH {o1 := o1.fb; o2 := o2.fb;}
ELSE State2;
```

Using Blocks for State Diagram Transitions

Blocks can be used to nest **If-Then** and **If-Then-Else** statements in state diagram descriptions, simplifying the description of complex transition logic.

Blocks for Transition Logic

Example 5: Without Blocks

```
IF (Hold & !Reset) THEN State1;
If (Hold & Error) THEN State2;
If (!Hold) THEN State3;
```

Example 6: With Blocks

```
If (Hold) THEN
{ IF (!Reset) THEN State1;
IF (Error) THEN State2; }
ELSE State3;
```

Comments

Comments provide another way to make a source file easy to understand. Comments explain what is not readily apparent from the source code itself and do not affect the code. Comments cannot be imbedded within keywords.

You can enter comments two ways:

- Begin with a double quotation mark (") and end with either another double quotation mark or the end of line.
- Begin with a double forward slash (//) and end with the end of the line. This is useful for commenting out lines of ABEL source that contain quote-delineated comments.

Examples of Comments

Examples of comments are shown in boldface below:

```
MODULE Basic_Logic; "gives the module a name
TITLE 'ABEL-HDL design example: simple gates'; "title
"declaration section"
IC4 device 'P10L8'; "declare IC4 to be a P10L8
IC5 "decoder PAL" device 'P10H8';
//IC5 "decoder PAL" device 'p10h8';
```

The information inside single quotation marks (apostrophes) is required and is part of the statement; it is not a comment.

Numbers

All numeric operations in ABEL-HDL are performed to 128-bit accuracy, which means the supported numeric values are in the range 0 to 2^{128} minus 1. Numbers are represented in any of five forms. The four most common forms represent numbers in different bases. The fifth form uses alphabetic characters to represent a numeric value.

When one of the four bases other than the default base is chosen to represent a number, the base used is indicated by a symbol preceding the number. The following table lists the four bases supported by ABEL-HDL and their accompanying symbols. The base symbols can be upper- or lowercase

Number representation in different bases

Base Name	Base	Symbol
Binary	2	^b
Octal	8	^o
Decimal	10	^d (default)
Hexadecimal	16	^h

When a number is specified and is not preceded by a base symbol, it is assumed to be in the default base numbering system. The normal default base is base 10. Therefore, numbers are represented in decimal form unless they are preceded by a symbol indicating that another base is to be used.

You can change the default number base. Examples of supported number specifications are shown below in the table below.

Strings

Strings are series of ASCII characters, including spaces, enclosed by apostrophes. Strings are used in the TITLE, MODULE, and OPTIONS statements, and in pin, node, and attribute declarations, as shown below:

```
'Hello'
' Text with a space in front'
' '
'The preceding line is an empty string'
'Punctuation? is allowed !!'
```

You can include a single quote in a string by preceding the quote with a backslash, (\).

```
'It\'s easy to use ABEL and ispLEVER'
```

You can include backslashes in a string by using two of them in succession.

```
'He\\she can use backslashes in a string'
```

***Note**: The grave accent (`) is also accepted as a string delimiter and can be used interchangeably with the apostrophe (').*

Operators, Expressions, and Equations

Items such as constants and signal names can be brought together in expressions. Expressions combine, compare, or perform operations on the items they include to produce a single result. The operations to be performed (addition and logical AND are two examples) are indicated by operators within the expression.

You can use the set operator (..) in expressions and equations.

ABEL-HDL operators are divided into four basic types: *logical, arithmetic, relational,* and *assignment*. Each of these types is discussed separately in the following topics, including a description of how they are combined into expressions. Following the descriptions is a summary of all the operators and the rules governing them and an explanation of how equations use expressions

Logical Operators

Logical operators are used in expressions. ABEL-HDL incorporates the standard logical operators listed in the table below. Logical operations are performed bit by bit.

Logical Operators

Operator	Description
!	NOT: ones complement
&	AND
#	OR
$	XOR: exclusive OR
!$	XNOR: exclusive NOR

Arithmetic Operators

Arithmetic operators define arithmetic relationships between items in an expression. The shift operators are included in this class because each left shift of one bit is equivalent to multiplication by 2 and a right shift of one bit is the same as division by 2. The following table lists the arithmetic operators.

Arithmetic Operators

Operator	Example	Description
-	-A	twos complement (negation)
-	A-B	subtraction
+	A+B	addition
Not Supported for Sets:		
*	A*B	multiplication
/	A/B	unsigned integer division
%	A%B	modulus: remainder from /
<<	A<<B	shift A left by B bits
>>	A>>B	shift A right by B bits

***Note**: A minus sign has a different significance, depending on its usage. When used with one operand, it indicates that the twos complement of the operand is to be formed. When the minus sign is found between two operands, the twos complements of the second operand are added to the first.*

Division is unsigned integer division: the result of division is a positive integer. Use the modulus operator (%) to get the remainder of a division. The shift operators perform logical unsigned shifts. Zeros are shifted in from the left during right shifts and in from the right during left shifts.

Relational Operators

Relational operators compare two items in an expression. Expressions formed with relational operators produce a Boolean true or false value.

Relational Operators

Operator	Description
==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

All relational operations are unsigned. For example, the expression !0 > 4 is true since the complement of !0 is 1111 (assuming 4 bits of data), which is 15 in unsigned binary, and 15 is greater than 4. In this example, a four-bit representation was assumed; in actual use, !0, the complement of 0, is 128 bits all set to 1.

Some examples of relational operators in expressions

Expression	Value
2 == 3	False
2 != 3	True
3 < 5	True
-1 > 2	True
	False

The logical values true and false are represented by numbers. Logical true is -1 in twos complement, so all 128 bits are set to 1. Logical false is 0 in twos complement, so all 128 bits are set to 0. This means that an expression producing a true or false value (a relational expression) can be used anywhere a number or numeric expression could be used and -1 or 0 will be substituted in the expression depending on the logical result.

For example,

```
A = D $ (B == C);
```

means that:

- A equals the complement of D if B equals C
- A equals D if B does not equal C.

When using relational operators, always use parentheses to ensure the expression is evaluated in the order you expect. The logical operators & and # have a higher priority than the relational operators (see the priority table later in this chapter).

The following equation:

```
Select = [A15..A0] == ^hD000 # [A15..A0] == ^h1000;
```

needs parentheses to obtain the desired result:

```
Select = ([A15..A0] == ^hD000) # ([A15..A0] == ^h1000);
```

Without the parentheses, the equation would have the default grouping

```
Select = [A15..A0] == (^hD000 # [A15..A0]) == ^h1000;
```

which is not the intended equation.

Assignment Operators

Assignment operators are used in equations rather than in expressions. Equations assign the value of an expression to output signals.

There are four assignment operators (two combinational and two registered). Combinational or immediate assignment occurs without any delay, as soon as the equation is evaluated. Registered assignment occurs at the next clock pulse from the clock associated with the output.

Assignment Operators

Operator	Set	Description
=	ON (1)	Combinational or detailed assignment
:=	ON (1)	Implied registered assignment
?=	DC (X)	Combinational or detailed assignment
?:=	DC (X)	Implied registered assignment

***Caution**: The := and ?:= assignment operators are used only when writing pin-to-pin registered equations. Use the = and ?= assignment operators for registered equations using detailed dot extensions.*

These assignment operators allow you to fully specify outputs in equations. For example, in the following truth table, the output F is fully specified:

```
TRUTH_TABLE ([A,B]->[F]);
[1,1]-> 0 ; "off-set
[1,0]-> 1 ; "on-set
[0,1]-> 1 ; "on-set
```

The equivalent functionality can be expressed in equations:

```
@DCSET
F = A & !B # !A & B; "on-set
F ?= !A & !B; "dc-set
```

***Note**: Specifying both the on-set and the don't-care set conditions enhances optimization.*

***Caution**: With equations, @DCSET or ISTYPE 'dc' must be specified or the ?= equations are ignored.*

Expressions

Expressions are combinations of identifiers and operators that produce one result when evaluated. Any logical, arithmetic, or relational operators may be used in expressions.

Expressions are evaluated according to the particular operators involved. Some operators take precedence over others, and their operation is performed first. Each operator has been assigned a priority that determines the order of evaluation. Priority 1 is the highest priority, and priority 4 is the lowest. The following table summarizes the logical, arithmetic, and relational operators, presented in groups according to their priority.

Operator Priority

Priority	Operator	Description
1	-	negate
1	!	NOT
2	&	AND
2	<<	shift left
2	>>	shift right
2	*	multiply
2	/	unsigned division
2	%	modulus
3	++	add
3	-	subtract
3	#	OR
3	$	XOR: exclusive OR
3	!$	XNOR: exclusive NOR
4	==	equal
4	!=	not equal
4	<	less than
4	< =	less than or equal
4	>	greater than
4	> =	greater than or equal

Operations of the same priority are performed from left to right. Use parentheses to change the order in which operations are performed. The operation in the innermost set of parentheses is performed first. The following examples of supported expressions in the table show how the order of operations and the use of parentheses affect the evaluated result.

How parentheses affect evaluated work

Expression	Result	Comments
2 * 3/2	3	operators with same priority
2 * 3 / 2	3	spaces are OK
2 * (3/2)	2	fraction is truncated
2 + 3 * 4	14	multiply first
(2 + 3) * 4	20	add first
2#4$2	4	OR first

Expression	Result	Comments
2#(4$2)	6	XOR first
2 == ^hA	0	
14 == ^hE	-1	

Equations

Equations assign the value of an expression to a signal or set of signals in a logic description. The identifier and expression must follow the rules for those elements.

Equations use the assignment operators =, ?= (combinational) and := ?:=, (registered) described above.

You can use the complement operator (!) to express negative logic. The complement operator precedes the signal name and implies that the expression on the right of the equation is to be complemented before it is assigned to the signal. Use of the complement operator on the left side of equations is provided as an option; equations for negative logic parts can just as easily be expressed by complementing the expression on the right side of the equation.

Equation Blocks

Equation blocks let you specify more complex functions and improve the readability of your equations. An equation block is enclosed in braces { }, and is supported wherever a single equation is supported. When used within a conditional expression, such as **If-Then**, **Case**, or **When-Then**, the logic functions are logically ANDed with the conditional expression that is in effect.

Multiple Assignments to the Same Identifier

When an identifier appears on the left side of more than one equation, the expressions assigned to the identifier are first ORed together, and then the assignment is made. If the identifier on the left side of the equation is complemented, the complement is performed after all the expressions have been ORed.

Equations Found	Equivalent Equation
A = B; A = C;	A = B # C;
A = B; A = C & D;	A = B # (C & D);
A = !B; A = !C;	A = !B # !C;
!A = B; !A = C;	A = !(B # C);
!A = B; A = !C;	A = !C #!B;
!A = B; !A = C; A = !D; A = !E;	A = !D # !E # !(B # C);

***Note**: When the complement operator appears on the left side of multiple assignment equations, the right sides are ORed first, and then the complement is applied.*

Sets

A set is a collection of signals and constants. Any operation applied to a set is applied to each element in the set. Sets simplify ABEL-HDL logic descriptions and test vectors by allowing groups of signals to be referenced with one name.

For example, you could collect the outputs (B0-B7) of an eight-bit multiplexer into a set named MULTOUT, and the three selection lines into a set named SELECT. You could then define the multiplexer in terms of MULTOUT and SELECT rather than individual input and output bits.

A set is represented by a list of constants and signals separated by commas or the range operator (..) and surrounded by brackets. The sets MULTOUT and SELECT would be defined as follows

```
MULTOUT = [B0,B1,B2,B3,B4,B5,B6,B7]
SELECT = [S2,S1,S0]
```

The above sets could also be expressed by using the range operator; for example:

```
MULTOUT = [B0..B7]
SELECT = [S2..S0]
```

Identifiers used to delimit a range must have compatible names: they must begin with the same alphabetical prefix and have a numerical suffix. Range identifiers can also delimit a decrementing range or a range which appears as one element of a larger set as shown below:

```
[A7..A0] "decrementing range
[Q1,Q2,.X.,A10..A7] "range within a larger set
```

The brackets are required to delimit the set. ABEL-HDL source file sets are not mathematical sets.

Set Indexing

Set indexing allows you to access elements within a set. The following example uses set indexing to assign four elements of a 16-bit set to a smaller set.

```
declarations
Set1 = [f15..f0];
Set2 = [q3..q0];
```

equations

```
Set2 := Set1[7..4];
```

The numeric values used for defining a set index refer to the bit positions of the set, with 0 being the least significant (left-most) element in the set. So Set1[7..4] is Set1, values f8 to f11.

If you are indexing into a set to access a single element, then you can use the following syntax:

```
declarations
out1 pin istype 'com';
Set1 = [f15..f0];
```

equations

```
out1 = Set1[4] == 1;
```

In this example, a comparator operator (==) was used to convert the single-element set (Set1[4]) into a bit value (equivalent to f4).

Set Operations

Most operators can be applied to sets, with the operation performed on each element of the set, sometimes individually and sometimes according to the rules of Boolean algebra.

Two-set Operations

For operations involving two or more sets, the sets must have the same number of elements. The expression "[a,b]+[c,d,e]" is not supported because the sets have different numbers of elements.

For example, the Boolean equation:

```
Chip_Sel = A15 & !A14 & A13;
```

represents an address decoder where A15, A14 and A13 are the three high-order bits of a 16-bit address. The decoder can easily be implemented with set operations. First, a constant set that holds the address lines is defined so the set can be referenced by name. This definition is done in the constant declaration section of a module.

The declaration is:

```
Addr = [A15,A14,A13];
```

which declares the constant set Addr. The equation

```
Chip_Sel = Addr == [1,0,1];
```

is functionally equivalent to

```
Chip_Sel = A15 & !A14 & A13;
```

If Addr is equal to [1,0,1], meaning that A15 = 1, A14 = 0 and A13 = 1, then Chip_Sel is set to true. The set equation could also have been written as

```
Chip_Sel = Addr == 5;
```

because 101 binary equals 5 decimal.

In the example above, a special set with the high-order bits of the 16-bit address was declared and used in the set operation. The full address could be used and the same function arrived at in other ways, as shown below:

Example 7:

```
" declare some constants in declaration section
Addr = [a15..a0];
X = .X.; "simplify notation for don't care constant
Chip_Sel = Addr == [1,0,1,X,X,X,X,X,X,X,X,X,X,X,X];
```

Example 8:

```
" declare some constants in declaration section
Addr = [a15..a0];
X =.X.;
Chip_Sel = (Addr >= ^HA000) & (Addr <= ^HBFFF);
```

Both solutions presented in these two examples are functionally equivalent to the original Boolean equation and to the first solution in which only the high order bits are specified as elements of the set:

```
(Addr = [a15, a14, a13])
```

Set Assignment and Comparison

Values and sets of values can be assigned and compared to a set. For example,

```
sigset = [1,1,0] & [0,1,1];
```

results in sigset being assigned the value, [0,1,0]. The set assignment

```
[a,b] = c & d;
```

is the same as the two assignments

```
a = c & d;
b = c & d;
```

Numbers in any representation can be assigned or compared to a set. The preceding set equation could have been written as

```
sigset = 6 & 3;
```

When numbers are used for set assignment or comparison, the number is converted to its binary representation and the following rules apply:

- If the number of significant bits in the binary representation of a number is greater than the number of elements in a set, the bits are truncated on the left.
- If the number of significant bits in the binary representation of a number is less than the number of elements in a set, the number is padded on the left with leading zeroes.

Thus, the following two assignments are equivalent:

```
[a,b] = ^B101011; "bits truncated to the left
[a,b] = ^B11;
```

And so are these two:

```
[d,c] = ^B01;
[d,c] = ^B1; "compiler will add leading zero
```

Supported Set Operations

Operator	Example	Description
=	A = 5	combinational assignment
:=	A := [1,0,1]	registered assignment
!	!A	NOT: ones complement
&	A & B	AND
#	A # B	OR
$	A $ B	XOR: exclusive OR
!$	A!$ B	XNOR: exclusive NOR
-	-A	negate
-	A - B	subtraction
++	A + B	addition
==	A == B	equal
!=	A != B	not equal
<	A < B	less than
<=	A <= B	less than or equal

Operator	Example	Description
>	A > B	greater than
>=	A >= B	greater than or equal

Set Evaluation

How an operator is performed with a set may depend on the types of arguments the operator uses. When a set is written `[a , b , c , d ]`, **a** is the MOST significant bit and **d** is the LEAST significant bit.

The result, when most operators are applied to a set, is another set. The result of the relational operators (==, !=, >, >=, <, <=) is a value: TRUE (all ones) or FALSE (all zeros), which is truncated or padded to as many bits as needed. The width of the result is determined by the context of the relational operator, not by the width of the arguments.

The different contexts of the AND (&) operator and the semantics of each usage are described below.

signal & signal a & b	This is the most straightforward use. The expression is TRUE if both signals are TRUE.
signal & number a & 4	The number is converted to binary and the least significant bit is used. The expression becomes a & 0 and then is reduced to 0 (FALSE).
signal & set a & [x, y, z]	The signal is distributed over the elements of the set to become [a & x, a & y, a & z]
set & set [a, b] & [x, y]	The sets are ANDed bit-wise resulting in: [a & x, b & y]. An error is displayed if the set widths do not match.
set & number [a, b, c] & 5	The number is converted to binary and truncated or padded with zeros to match the width of the set. The sequence of transformations is `[a, b, c] & [1, 0, 1]` `[a & 1, b & 0, c & 1]` `[a, 0, c]`
number & number 9 & 5	The numbers are converted to binary, ANDed together, and then truncated or padded.

Example Equations

```
select = [a15..a0] == ^H80FF
```

select (signal) is TRUE when the 16-bit address bus has the hex value 80FF. Relational operators always result in a single bit.

```
[sel1, sel0] = [a3..a0] > 2
```

The width of **sel** and **a** are different, so the 2 is expanded to four bits (of binary) to match the size of the **a** set. Both **sel1** and s**el2** are true when the value of the four **a** lines (taken as a binary number) is greater than 2.

The result of the comparison is a single-bit result that is distributed to both members of the set on the output side of the equation.

```
[out3..out0] = [in3..in0] & enab
```

If **enab** is TRUE, then the values on **in0** through **in3** are seen on the **out0** through **out3** outputs. If **enab** is FALSE, then the outputs are all FALSE.

Set Operation Rules

Set operations are applied according to Boolean algebra rules. Uppercase letters are set names, and lowercase letters are elements of a set. The letters k and n are subscripts to the elements and to the sets. A subscript following a set name (uppercase letter) indicates how many elements the set contains.

So A_k indicates that set A contains k elements. a_{k-1} is the (k-1)th element of set A. a_1 is the first element of set A.

Expression	Is Evaluated As...
$!A_k$	$[!a_k, !a_{k-1}, ..., !a_1]$
$-A_k$	$!A_k + 1$
$A_k.OE$	$[a_k.OE, a_{k-1}.OE, ..., a_1.OE]$
A_k & B_k	$[a_k \& b_k, a_{k-1} \& b_{k-1}, ..., a_1 \& b_1]$
A_k # B_k	$[a_k \# b_k, a_{k-1} \# b_{k-1}, ..., a_1 \# b_1]$
A_k $ B_k	$[a_k \$ b_k, a_{k-1} \$ b_{k-1}, ..., a_1 \$ b_1]$
A_k !$ B_k	$[a_k !\$ b_k, a_{k-1} !\$ b_{k-1}, ..., a_1 !\$ b_1]$
$A_k == B_k$	$(a_k == b_k) \& (a_{k-1} == b_{k-1}) \& ... \& (a_1 == b_1)$
$A_k != B_k$	$(a_k != b_k) \# (a_{k-1} != b_{k-1}) \# ... \# (a_1 != b_1)$
$A_k + B_k$	D_k where: d_n is evaluated as $a_n \$ b_n \$ c_{n-1}$ c_n is evaluated as $(a_n \$ b_n) \# (a_n \& c_{n-1}) \# (b_n \& c_{n-1})$ c0 is evaluated as 0
$A_k - B_k$	$A_k + (-B_k)$
$A_k < B_k$	c_k where: c_n is evaluated as (!an & (bn # cn-1) # an & bn & cn-1) != 0 c0 is evaluated as 0

Limitations/ Restrictions on Sets

If you have a set assigned to a single value, the value will be padded with 0s and then applied to the set. For example,

```
[A1,A2,A3] = 1
```

is equivalent to

```
A1 = 0
A2 = 0
A3 = 1
```

which may not be the intended result. If you want 1 assigned to each member of the set, you'd need binary 111 or decimal 7.

The results of using an operator depend on the sequence of evaluation. Without parentheses, operations are performed from left to right. Consider the following two equations. In the first, the constant 1 is converted to a set; in the second, the 1 is treated as a single bit.

Example: Constant 1 is converted to a set

The first operation is **[a, b] & 1**, so 1 is converted to a set [0, 1].

```
[x1, y1] = [a, b] & 1 & d
= ([a, b] & 1 ) & d
= ([a, b] & [0, 1]) & d
= ([a & 0, b & 1]) & d
= [ 0 , b ] & d
= [0 & d, b & d]
= [0, b & d]
x1 = 0
y1 = b & d
```

Example: The 1 is treated as a single bit

The first operation is 1 & d, so 1 is treated as a single bit.

```
[x2,y2] = 1 & d & [a, b]
= (1 & d) & [a, b]
= d & [a, b]
= [d & a, d & b]
x2 = a & d
y2 = b & d
```

If you are unsure about the interpretation of an equation, try the following:

- Fully parenthesize your equation. Errors can occur if you are not familiar with the precedence rules.
- Write out numbers as sets of 1s and 0s instead of as decimal numbers. If the width is not what you expected, you will get an error message.

Arguments and Argument Substitution

Variable values can be used in macros, modules, and directives. These values are called the arguments of the construct that uses them. In ABEL-HDL, a distinction must be made between two types of arguments: *actual* and *dummy.* Their definitions are given here.

Dummy argument — An identifier used to indicate where an actual argument is to be substituted in the macro, module, or directive.

Actual argument — The argument (value) used in the macro, directive, or module. The actual argument is substituted for the dummy argument. An actual argument can be any text, including identifiers, numbers, strings, operators, sets, or any other element of ABEL-HDL.

Dummy arguments are specified in macro declarations and in the bodies of macros, modules, and directives. The dummy argument is preceded by a question mark in the places where an actual argument is to be substituted. The question mark distinguishes the dummy arguments from other ABEL-HDL identifiers occurring in the source file.

Take for example, the following macro declaration arguments:

```
OR_EM MACRO (a,b,c) { ?a # ?b # ?c };
```

This defines a macro named OR_EM that is the logical OR of three arguments. These arguments are represented in the definition of the macro by the dummy arguments, a, b, and c. In the body of the macro, which is surrounded by braces, the dummy arguments are preceded by question marks to indicate that an actual argument is substituted.

The equation

```
D = OR_EM (x,y,z&1);
```

invokes the OR_EM macro with the actual arguments, x, y, and z&1. This results in the equation:

```
D = x # y # z&1;
```

Arguments are substituted into the source file before checking syntax and logic, so if an actual argument contains unsupported syntax or logic, the compiler detects and reports the error only after the substitution.

Spaces in Arguments

Actual arguments are substituted exactly as they appear, so any spaces (blanks) in actual arguments are passed to the expression. In most cases, spaces do not affect the interpretation of the macro. The exception is in functions that compare character strings, such as @IFIDEN and IFNIDEN. For example, the macro

```
iden macro(a,b) {@ifiden(?a,?b)
{@message 'they are the same';};};
```

compares the actual arguments and prints the message if they are identical. If you enter the macro with spaces in the actual arguments:

```
iden(Q1, Q1);
```

The value is false because the space is passed to the macro.

Argument Guidelines

- Dummy arguments are placeholders for actual arguments.
- A question mark preceding the dummy argument indicates that an actual argument is to be substituted.
- Actual arguments replace dummy arguments before the source file is checked for correctness.
- Spaces in actual arguments are retained.

ABEL Design Considerations

Hierarchy in ABEL-HDL

You use hierarchy declarations in an upper-level ABEL-HDL source to refer to (instantiate) an ABEL-HDL module. To instantiate an ABEL-HDL module:

In the lower-level module: (optional)

- Identify lower-level I/O Ports (signals) with an **Interface** statement.

In the top-level source:

- Declare the lower-level module with an **Interface** declaration.
- Instantiate the lower-level module with **Functional_block** declarations.

***Note**: For instructions on instantiating lower-level modules in schematics, refer to your schematic reference.*

Instantiating a Lower-level Module in an ABEL-HDL Source

Identifying I/O Ports in the Lower-level Module

The way to identify an ABEL-HDL module's input and output ports is to place an **Interface** statement immediately following the **Module** statement. The **Interface** statement defines the ports in the lower-level module that are used by the top-level source.

You must declare all input pins in the ABEL-HDL module as ports, and you can specify default values of 0, 1, or Don't-care.

You do not have to declare all output pins as ports. Any undeclared outputs become No Connects or redundant nodes. Redundant nodes can later be removed from the designs during post-link optimization.

The following source fragment is an example of a lower-level **interface** statement.

```
module lower
interface (a=0, [d3..d0]=7 -> [z0..z7]) ;
title 'example of lower-level interface statement ' ...
```

This statement identifies input **a**, **d3**, **d2**, **d1** and **d0** with default values, and outputs **z0** through **z7**.

Specifying Signal Attributes

Attributes specified for pins in a lower-level module are propagated to the higher-level source. For example, a lower-level pin with an 'invert' attribute affects the higher-level signal wired to that pin (it affects the pin's preset, reset, preload, and power-up value).

Output Enables (OE)

Connecting a lower-level tri-state output to a higher-level pin results in the output enable being specified for the higher-level pin. If another OE is specified for the higher-level pin, it is flagged as an error. Since most tristate outputs are used as bi-directionals, it might be important to keep the lower-level OE.

Buried Nodes

Buried nodes in lower-level sources are handled as follows:

Dangling Nodes	Lower-level nodes that do not fan out are propagated to the higher-level module and become dangling nodes. Optimization may remove dangling nodes.
Combinational nodes	Combinational nodes in a lower-level module become collapsible nodes in the higher-level module.
Registered nodes	Registered nodes are preserved with hierarchical names assigned to them.

Declaring Lower-level Modules in the Top-level Source

To declare a lower-level module, you match the lower-level module's interface statement with an interface declaration. For example, to declare the lower-level module given above, you would add the following declaration to your upper-level source declarations:

```
lower interface (a, [d3..d0] -> [z0..z7]) ;
```

You could specify different default values if you want to override the values given in the instantiated module; otherwise, the instantiated module must exactly match the lower-level interface statement.

Instantiating Lower-level Modules in a Top-level Source

Use a **functional_block** declaration in a top-level ABEL-HDL source to instantiate a declared lower-level module and make the ports of the lower-level module accessible in the upper-level source. You must declare sources with an **interface** declaration before you instantiate them.

To instantiate the module declared above, add an interface declaration and signal declarations to your top-level declarations and add port connection equations to your top-level equations, as shown in the source fragment below:

```
DECLARATIONS
low1 FUNCTIONAL_BLOCK lower ;
zed0..zed7 pin ; "upper-level inputs
atop pin istype 'reg,buffer'; "upper-level output
d3..d0 pin istype 'reg,buffer'; "upper-level ouputs
EQUATIONS
atop = low1.a; "wire this source's outputs
[d3..d0] = low1.[d3..d0] ; " to lower-level inputs
low1.[z0..z7] = [zed0..zed7]; "wire this source's inputs to
```

```
" lower-level outputs
```

Hierarchy and Retargeting and Fitting

Redundant Nodes

When you link multiple sources, some unreferenced nodes may be generated. These nodes usually originate from lower-level outputs that are not being used in the top-level source: for example, when you use a 4-bit counter as a 3-bit counter. The most significant bit of the counter is unused and can be removed from the design to save device resources. This step also removes trivial connections. In the following example, if out1 is a pin and t1 is a node:

```
out1 = t1;
t1 = a86;
```

would be mapped to

```
out1 = a86;
```

Merging Feedbacks

Linking multiple modules can produce signals with one or more feedback types, such as .FB and .Q. You can tell the optimizer to combine these feedbacks to help the fitting process.

Post-linked Optimization

If your design has a constant tied to an input, you can re-optimize the design. Re-optimizing may further reduce the product terms count.

For example, if you have the equation

```
out = i0 & i1 || !i0 & i2;
```

and i0 is tied to 1, the resulting equation would be simplified to

```
out = i1;
```

Hierarchy and Test Vectors

Hierarchy and Test Vectors (PLD JEDEC Simulation)

If you are targeting a PLD device and want to do JEDEC simulation of your project, you must specify your test vectors in the top-level source. If you have existing test vectors in lower-level sources, you can merge the inputs stimulus of blocks that are connected to the top-level pins with the expected values of blocks that are connected to the top-level outputs. The test vectors in the lower-level modules can still be used for individual JEDEC simulation.

Node Collapsing

All combinational nodes are collapsible by default. Nodes that are to be collapsed (or nodes that are to be preserved) are flagged through the use of signal attributes in the language. The signal attributes are:

Istype 'keep' — Do not collapse this node.

'collapse' — Collapse this node.

Collapsing provides multi-level optimization for combinational logic. Designs with arithmetic and comparator circuits generally generate a large number of product terms that will not fit to any programmable logic device. Node collapsing allows you to describe equations in terms of multi-level combinational nodes, then collapse the nodes into the output until it reaches the product term you specify. The result is an equation that is optimized to fit the device constraints.

Selective Collapsing

In some instances you may want to prevent the collapsing of certain nodes. For example, some nodes may help in the simulation process. You can specify nodes you do not want collapsed as Istype 'keep', and the optimizer will not collapse them.

Pin-to-pin Language Features

ABEL-HDL is a device-independent language. You do not have to declare a device or assign pin numbers to your signals until you are ready to implement the design into a device. However, when you do not specify a device or pin numbers, you need to specify pin-to-pin attributes for declared signals.

Because the language is device-independent, the ABEL-HDL compiler does not have predetermined device attributes to imply signal attributes. If you do not specify signal attributes or other information (such as the dot extensions, which are described later), your design might not operate consistently if you later transfer it to a different target device.

Device-independence vs. Architecture-independence

The requirement for signal attributes does not mean that a complex design must always be specified with a particular device in mind. You may still have to understand the differences between, for example, a P22V10 PAL and a MACH211 device, but you do not have to specify a particular device when describing your design.

Attributes and dot extensions help you refine your design to work consistently when moving from one class of device architecture to another: for example, from devices having inverted outputs to those with a particular kind of reset/preset circuitry. However, the more you refine your design, using these language features, the more restrictive your design becomes in terms of the number of device architectures for which it is appropriate.

Signal Attributes

Signal attributes remove ambiguities that occur when no specific device architecture is declared. If your design does not use device-related attributes (either implied by a DEVICE statement or expressed in an ISTYPE statement), it may not operate the same way when targeted to different device architectures.

Signal Dot Extensions

Signal dot extensions, like attributes, enable you to more precisely describe the behavior of a circuit that may be targeted to different architectures. Dot extensions remove the ambiguities in equations.

Pin-to-pin vs. Detailed Description Methods for Registered Designs

You can use ABEL-HDL assignment operators when you write high-level equations. The **=** operator specifies a combinational assignment, where the design is written with only the circuit's inputs and outputs in mind. The **:=** assignment operator specifies a registered assignment, where you must consider the internal circuit elements (such as output inverters, presets and resets) related to the memory elements (typically flip-flops). The semantics of these two assignment operators are discussed below.

Using := for Pin-to-pin Descriptions

The **:=** implies that a memory element is associated with the output defined by the equation. For example, the equation;

```
Q1 := !Q1 # Preset;
```

implies that **Q1** will hold its current value until the memory element associated with that signal is clocked (or unlatched, depending on the register type). This equation is a pin-to-pin description of the output signal **Q1**. The equation describes the signal's behavior in terms of desired output pin values for various input conditions. Pin-to-pin descriptions are useful when describing a circuit that is completely architecture-independent.

Language elements that are useful for pin-to-pin descriptions are the ":=" assignment operator, and the **.CLK**, **.OE**, **.FB**, **.CLR**, **.ACLR**, **.SET**, **.ASET** and **.COM** dot extensions. These dot extensions help resolve circuit ambiguities when describing architecture-independent circuits.

Resolving Ambiguities

In the equation above (Q1 := !Q1 # Preset;), there is an ambiguous feedback condition. The signal **Q1** appears on the right side of the equation, but there is no indication of whether that fed-back signal should originate at the register, come directly from the combinational logic that forms the input to the register, or come from the I/O pin associated with Q1. There is also no indication of what type of register should be used (although register synthesis algorithms could, theoretically, map this equation into virtually any register type). The equation could be more completely specified in the following manner.

```
Q1.CLK = Clock; "Register clocked from input
Q1 := !Q1.FB # Preset; "Reg. feedback normalized to pin value
```

This set of equations describes the circuit completely and specifies enough information that the circuit will operate identically in virtually any device in which you can fit it. The feedback path is specified to be from the register itself, and the **.CLK** equation specifies that the memory element is clocked, rather than latched.

Detailed Circuit Descriptions

In contrast to a pin-to-pin description, the same circuit can be specified in a detailed form of design description in the following manner:

```
Q1.CLK = Clock; "Register clocked from input
Q1.D = !Q1.Q # Preset; "D-type f/f used for register
```

In this form of the design, specifying the D input to a D-type flip-flop and specifying feedback directly from the register restricts the device architectures in which the design can be implemented. Furthermore, the equations describe only the inputs to and the feedback from the flip-flop and do not provide any information regarding the configuration of the actual output pin. This means the design will operate quite differently when implemented in a device with inverted outputs (a simple P16R4 PAL device, for example), versus a device with non-inverting outputs (such as an M4-32).

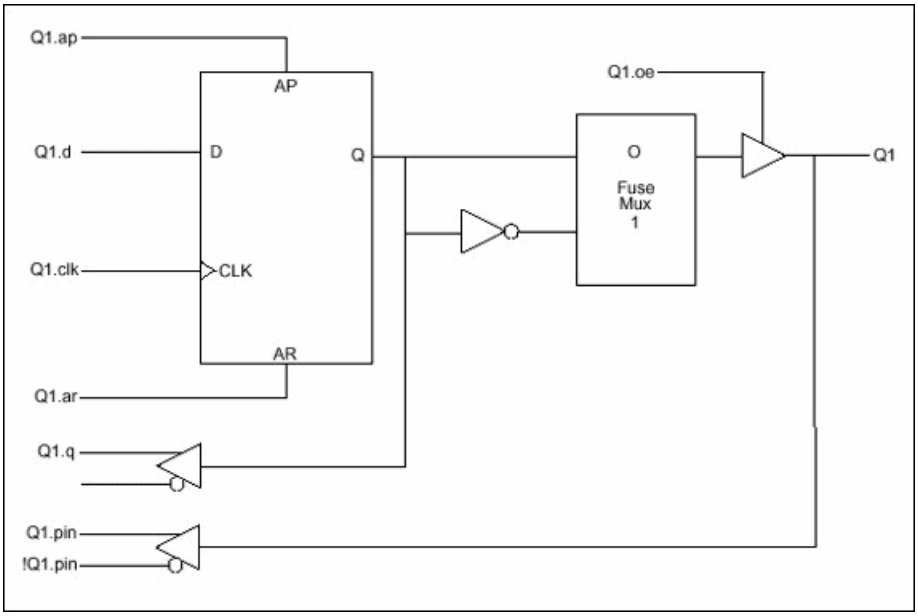
To maintain the correct pin behavior, using detailed equations, one additional language element is required: a 'buffer' attribute (or its complement, an 'invert' attribute). The 'buffer' attribute ensures that the final implementation in a device has no inversion between the specified D-type flip-flop and the output pin associated with **Q1**. For example, add the following to the Declarations section:

Q1 pin istype 'buffer';

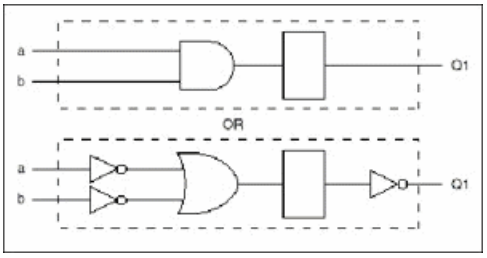
Detailed Descriptions: Designing for Macrocells

One way to understand the difference between pin-to-pin and detailed description methods is to think of detailed descriptions as macrocell specifications. A macrocell is a block of circuitry that is normally, but not always, associated with a device's I/O pin.

Detailed descriptions are written for the various input ports of the macrocell, shown in the figure below with dot extension labels. Note that the macrocell features a configurable inversion between the Q output of the flip-flop and the output pin labeled **Q1**. If you use this inverter, or select a device that features a fixed inversion, the behavior you observe on the **Q1** output pin will be inverted from the logic applied to, or observed on, the various macrocell ports, including the feedback port **Q1.q**.



Pin-to-pin descriptions, on the other hand, allow you to describe your circuit in terms of the expected behavior on an actual output pin, regardless of the architecture of the underlying macrocell. When pin-to-pin descriptions are written in ABEL-HDL, the "generic macrocell," shown below, is synthesized from whatever type of macrocell actually exists in the target device.



Examples of Pin-to-Pin and Detailed Descriptions

Pin-to-Pin Module Description

```
module Q1_1
Q1 pin istype 'reg';
Clock,Preset pin;

equations
Q1.clk = Clock;
Q1 := !Q1.fb # Preset;

test_vectors ([Clock,Preset] -> Q1)
[ .c. , 1 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 0 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 1 ] -> 1;
```

```
[ .c. , 1 ] -> 1;
end
```

Detailed Module Description

```
module Q1_2
Q1 pin istype 'reg_D,buffer';
Clock,Preset pin;

equations
Q1.CLK = Clock;
Q1.D = !Q1.Q # Preset;

test_vectors ([Clock,Preset] -> Q1)
[ .c. , 1 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 0 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 1 ] -> 1;
[ .c. , 1 ] -> 1;
end
```

The first description can be targeted into virtually any device (if register synthesis and device fitting features are available), while the second description can be targeted only to devices featuring D-type flip-flops and non-inverting outputs.

To implement the second (detailed) module in a device with inverting outputs, the source file would need to be modified as shown in the following section.

Detailed Module with Inverted Outputs

```
module Q1_3
Q1 pin istype 'reg_D,invert';
Clock,Preset pin;

equations
Q1.CLK = Clock;
!Q1.D = Q1.Q # Preset;

test_vectors ([Clock,Preset] -> Q1)
[ .c. , 1 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 0 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 1 ] -> 1;
[ .c. , 1 ] -> 1;
end
```

In this version of the module, the existence of an inverter between the output of the D-type flip-flop and the output pin (specified with the 'invert' attribute) has necessitated a change in the equation for **Q1.D**.

As this example shows, device-independence and pin-to-pin description methods are preferable, since you can describe a circuit completely for any implementation. Using pin-to-pin descriptions and generalized dot extensions (such as **.FB**, **.CLK** and **.OE**) as much as possible allows you to implement your ABEL-HDL module into any one of a particular class of devices; for example, any device that features enough flip-flops and appropriately configured I/O resources. However, the need for particular types of device features, such as register preset or reset, might limit your ability to describe your design in a completely architecture-independent way.

If, for example, a built-in register preset feature is used in a simple design, the target architectures are limited. Consider this version of the design.

```
module Q1_5l
Q1 pin istype 'reg,buffer';
Clock,Preset pin;

equations
Q1.CLK = Clock;
Q1.AP = Preset;
Q1 := !Q1.fb ;

test_vectors ([Clock,Preset] -> Q1)
[ .c. , 1 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 0 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 1 ] -> 1;
[ .c. , 1 ] -> 1;
end
```

The equation for **Q1** still uses the **:=** assignment operator and **.FB** for a pin-to-pin description of Q1's behavior. But the use of **.AP** to describe the reset function requires consideration of different device architectures. The **.AP** extension, like the **.D** and **.Q** extensions, is associated with a flip-flop input, not with a device output pin. If the target device has inverted outputs, the design will not reset properly, so this ambiguous reset behavior is removed by using the 'buffer' attribute, which reduces the range of target devices to those with non-inverted outputs.

Using .ASET instead of .AP can solve this problem if the fitter being used supports the .ASET dot extension.

Versions 5 and 7 of the design above and below are unambiguous, but each is restricted to certain device classes.

```
module Q1_7l
Q1 pin istype 'reg,invert';
Clock,Preset pin;

equations
Q1.CLK = Clock;
Q1.AR = Preset;
Q1 := !Q1.fb ;

test_vectors ([Clock,Preset] -> Q1)
```

```
[ .c. , 1 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 0 ] -> 1;
[ .c. , 0 ] -> 0;
[ .c. , 1 ] -> 1;
[ .c. , 1 ] -> 1;
end
```

When to Use Detailed Descriptions

Although the pin-to-pin description is preferable, there will frequently be situations when you must use a more detailed description. If you are unsure about which method to use for various parts of your design, examine the design's requirements. If your design requires specific features of a device (such as register preset or unusual flip-flop configurations), detailed descriptions are probably necessary. If your design is a simple combinational function, or if it matches the generic macrocell in its requirements, you can probably use simple pin-to-pin descriptions.

Using Alternative Flip-flop Types

In ABEL-HDL you can specify a variety of flip-flop types using attributes such as istype 'reg_D' and 'reg_JK'. However, these attributes do not enforce the use of a specific type of flip-flop when a device is selected, and they do not affect the meaning of the **:=** assignment operator.

You can think of the **:=** assignment operator as a memory operator. The type of register that most closely matches the **:=** assignment operator's behavior is the D-type flip-flop.

The primary use for attributes such as istype 'reg_D', 'reg_JK' and 'reg_SR' is to control the generation of logic. Specifying one of the 'reg_' attributes (for example, istype 'reg_D') instructs the AHDL compiler to generate equations using The .D extension regardless of whether the design was written using .D, := or some other method (for example, state diagrams).

***Note**: You also need to specify istype 'invert' or 'buffer' when you use detailed syntax.*

Using **:=** for flip-flop types other than D-type is only possible if register synthesis features are available to convert the generated equations into equations appropriate for the alternative flip-flop type specified. Since the use of register synthesis to convert D-type flip-flop stimulus into JK or SR-type stimulus usually results in inefficient circuitry, the use of **:=** for these flip-flop types is discouraged. Instead, you should use the .J and .K extensions (for JK-type flip-flops) or the .S and .R extensions (for SR-type flip-flops) and use a detailed description method (including 'invert' or 'buffer' attributes) to describe designs for these register types.

There is no provision in the language for directly writing pin-to-pin equations for registers other than D-type. State diagrams, however, may be used to describe pin-to-pin behavior for any register type.

Using Active-low Declarations

In ABEL-HDL you can write pin-to-pin design descriptions using implied active-low signals. Active-low signals are declared with a '!' operator, as shown below.

```
!Q1 pin istype 'reg';
```

If a signal is declared active-low, it is automatically complemented when you use it in the subsequent design description. This complementing is performed for any use of the signal itself, including as an input, as an output, and in test vectors. Complementing is also performed if you use the **.fb** dot extension on an active-low signal.

The following three designs, for example, operate identically.

Design 1: Implied Pin-to-Pin Active-low

```
module act_low2
!q0,!q1 pin istype 'reg';
clock pin;
reset pin;

equations
[q1,q0].clk = clock;
[q1,q0] := ([q1,q0].FB + 1) & !reset;

test_vectors ([clock,reset] -> [ q1, q0])
[ .c. , 1 ] -> [ 0 , 0 ];
[ .c. , 0 ] -> [ 0 , 1 ];
[ .c. , 0 ] -> [ 1 , 0 ];
[ .c. , 0 ] -> [ 1 , 1 ];
[ .c. , 0 ] -> [ 0 , 0 ];
[ .c. , 0 ] -> [ 0 , 1 ];
[ .c. , 1 ] -> [ 0 , 0 ];
end
```

Design 2: Explicit Pin-to-Pin Active-low

```
module act_low1
q0,q1 pin istype 'reg';
clock pin;
reset pin;

equations
[q1,q0].clk = clock;
![q1,q0] := (![q1,q0].FB + 1) & !reset;

test_vectors ([clock,reset] -> [!q1,!q0])
[ .c. , 1 ] -> [ 0 , 0 ];
[ .c. , 0 ] -> [ 0 , 1 ];
[ .c. , 0 ] -> [ 1 , 0 ];
[ .c. , 0 ] -> [ 1 , 1 ];
[ .c. , 0 ] -> [ 0 , 0 ];
[ .c. , 0 ] -> [ 0 , 1 ];
[ .c. , 1 ] -> [ 0 , 0 ];
end
```

Design 3: Explicit Detailed Active-low

```
module act_low3
q0,q1 pin istype 'reg_d,buffer';
clock pin;
```

```
reset pin;

equations
[q1,q0].clk = clock;
   ![q1,q0].D = (![q1,q0].Q + 1) & !reset;

test_vectors ([clock,reset] -> [!q1,!q0])
[ .c. , 1 ] -> [ 0 , 0 ];
[ .c. , 0 ] -> [ 0 , 1 ];
[ .c. , 0 ] -> [ 1 , 0 ];
[ .c. , 0 ] -> [ 1 , 1 ];
[ .c. , 0 ] -> [ 0 , 0 ];
[ .c. , 0 ] -> [ 0 , 1 ];
[ .c. , 1 ] -> [ 0 , 0 ];
end
```

Both of these designs describe an up counter with active-low outputs. The first example inverts the signals explicitly (in the equations and in the test vector header), while the second example uses an active-low declaration to accomplish the same thing.

Polarity Control

Automatic polarity control is a powerful feature in ABEL-HDL where a logic function is converted for both non-inverting and inverting devices.

A single logic function may be expressed with many different equations. For example, all three equations below for F1 are equivalent.

```
(1)  F1 = (A & B);
(2)  !F1 = !(A & B);
!F1 = !A # !B;
```

In the example above, equation (3) uses two product terms, while equation (1) requires only one. This logic function will use fewer product terms in a non-inverting device such as the P10H8 than in an inverting device such as the P10L8. The logic function performed from input pins to output pins will be the same for both polarities.

Not all logic functions are best optimized to positive polarity. For example, the inverted form of F2, equation (3), uses fewer product terms than equation (2).

```
(1)  F2 = (A # B) & (C # D);
(2)  F2 = (A & C) # (A & D) # (B & C) # (B & D);
!F2 = (!A & !B) # (!C & !D);
```

Programmable polarity devices are popular because they can provide a mix of non-inverting and inverting outputs to achieve the best fit.

Polarity Control with Istype

In ABEL-HDL, you control the polarity of the design equations and target device (in the case of programmable polarity devices) in two ways:

- Using Istype *neg, pos,* and *dc*
- Using Istype *invert* and *buffer*

Using Istype neg, pos, and dc to Control Equation and Device Polarity

The 'neg', 'pos', and 'dc' attributes specify types of optimization for the polarity as follows:

neg — Istype neg optimizes the circuit for negative polarity. Unspecified logic in truth tables and state diagrams becomes a 0.

pos — Istype pos optimizes the circuit for positive polarity. Unspecified logic in truth tables and state diagrams becomes a 1.

dc — Istype dc uses polarity for best optimization. Unspecified logic in truth tables and state diagrams becomes don't care (X).

Using invert and buffer to Control Programmable Inversion

An optional method for specifying the desired state of a programmable polarity output is to use the 'invert' or 'buffer' attributes. These attributes ensure that an inverter gate either does or does not exist between the output of a flip-flop and its corresponding output pin. When you use the 'invert' and 'buffer' attributes, you can still use automatic polarity selection if the target architecture features programmable inverters located before the associated flip-flop.

These attributes are particularly useful for devices such as the P22V10, where the reset and preset behavior is affected by the programmable inverter.

***Note**: The 'invert' and 'buffer' attributes do not actually control device or equation polarity — they only enforce the existence or nonexistence of an inverter between a flip-flop and its output pin.*

The polarity of devices that feature a fixed inverter in this location and a programmable inverter before the register cannot be specified using 'invert' and 'buffer'.

Flip-flop Equations

Pin-to-pin equations (using the **:=** assignment operator) are only supported for D flip-flops. ABEL-HDL does not support the **:=** assignment operator for T, SR or JK flip-flops and has no provision for specifying a particular output pin value for these types.

If you write an equation of the form:

```
Q1 := 1;
```

and the output, **Q1**, has been declared as a T-type flip-flop, the ABEL-HDL compiler will give a warning and convert the equation to

```
Q1.T = 1;
```

Because the T input to a T-type flip-flop does not directly correspond to the value you observed on the associated output pin, this equation will not result in the pin-to-pin behavior you want.

To produce specific pin-to-pin behavior for alternate flip-flop types, you must consider the behavior of the flip-flop you used and write detailed equations that stimulate the inputs of that flip-flop. A detailed equation to set and hold a T-type flip-flop is shown below.

```
Q1.T = !Q1.Q;
```

Feedback Considerations - Dot Extensions

The source of feedback is normally set by the architecture of the target device. If you don't specify a particular feedback path, the design may operate differently in different device types. Specifying feedback paths (with the .FB, .Q or .PIN dot extensions) eliminates architectural ambiguities. Specifying feedback paths also allows you to use architecture-independent simulation.

The following rules should be kept in mind when you are using feedback:

- **No Dot Extension** — A feedback signal with no dot extension (for example, count := count+1;) results in pin feedback if it exists in the target device. If there is no pin feedback, register feedback is used, with the value of the register contents complemented (normalized) if needed to match the value observed on the pin.
- **.FB Extension** — A signal specified with the .FB extension (for example, count := count.fb+1;) results in register feedback normalized to the pin value if a register feedback path exists. If no register feedback is available, pin feedback is used, and the fuse mapper checks that the output enable does not conflict with the pin feedback path. If there is a conflict, an error is generated if the output enable is not constantly enabled.
- **.COM Extension** — A signal specified with the .COM extension (for example, count := count.com+1;) results in OR-array (pre-register) feedback, normalized to the pin value if an OR-array feedback path exists. If no OR-array feedback is available, pin feedback is used and the fuse mapper checks that the output enable does not conflict with the pin feedback path. If there is a conflict, an error is generated if the output enable is not constantly enabled.
- **.PIN Extension** — If a signal is specified with the .PIN extension (for example, count := count.pin+1;), the pin feedback path will be used. If the specified device does not feature pin feedback, an error will be generated. Output enables frequently affect the operation of fed-back signals that originate at a pin.
- **.Q Extension** — Signals specified with the .Q extension (for example, count.d = count.q + 1;) will originate at the Q output of the associated flip-flop. The fed-back value may or may not correspond to the value you observe on the associated output pin. If an inverter is located between the Q output of the flip-flop and the output pin (as is the case in most registered PAL-type devices), the value of the fed-back signal will be the complement of the value you observe on the pin.
- **.D Extension** — Some devices, such as the MACH210, allow feedback of the input to the register. To select this feedback, use the .D extension.

Dot Extensions and Architecture-Independence

For your design to be architecture-independent, you must write it in terms of its pin-to-pin behavior rather than in terms of specific device features (such as flip-flop configurations or output inversions).

The following simple ABEL-HDL design describes this simple one-bit synchronous circuit. The design description uses architecture-independent dot extensions to describe the circuit in terms of its behavior, as observed on the output pin of the target device. Since this design is architecture-independent, it will operate the same (disregarding initial power-up state), irrespective of the device type.

Pin-to-pin One-bit Synchronous Circuit module pin2pin.

```
Clk pin 1;
Toggle pin 2;
Ena pin 11;
Qout pin 19 istype 'reg';

equations
Qout := !Qout.FB & Toggle;
Qout.CLK = Clk;
Qout.OE = !Ena;

test_vectors([Clk,Ena,Toggle] -> [Qout])
[.c., 0 , 0 ] -> 0;
[.c., 0 , 1 ] -> 1;
[.c., 0 , 1 ] -> 0;
```

```
[.c., 0 , 1 ] -> 1;
[.c., 0 , 1 ] -> 0;
[.c., 1 , 1 ] -> .Z.;
[ 0 , 0 , 1 ] -> 1;
[.c., 1 , 1 ] -> .Z.;
[ 0 , 0 , 1 ] -> 0;
end
```

Dot Extensions and Detail Design Descriptions

You may need to be more specific about how you implement a circuit in a target device. More complex device architectures have many configurable features, and you may want to use these features in a particular way. You may want a precise power-up and preset operation, or, in some cases, you may need to control internal elements.

The circuit previously described (using architecture-independent dot extensions) could be described, for example, using detailed dot extensions in the following ABEL-HDL source file.

Detailed One-bit Synchronous Circuit with Inverted Qout

```
module detail1
d1 device 'P16R8';
Clk pin 1;
Toggle pin 2;
Ena pin 11;
Qout pin 19 istype 'reg_D';

equations
!Qout.D = Qout.Q & Toggle;
Qout.CLK = Clk;
Qout.OE = !Ena;

test_vectors([Clk,Ena,Toggle] -> [Qout])
[.c., 0 , 0 ] -> 0;
[.c., 0 , 1 ] -> 1;
[.c., 0 , 1 ] -> 0;
[.c., 0 , 1 ] -> 1;
[.c., 0 , 1 ] -> 0;
[.c., 1 , 1 ] -> .Z.;
[ 0 , 0 , 1 ] -> 1;
[.c., 1 , 1 ] -> .Z.;
[ 0 , 0 , 1 ] -> 0;
end
```

This version of the design will result in exactly the same fuse pattern as indicated in the preceding figure for Circuit 2. As written, this design assumes the existence of an inverted output for the signal Qout. This is why the Qout.D and Qout.Q signals are reversed from the architecture-independent version of the design presented earlier.

***Note**: The inversion operator applied to Qout.D does not correspond directly to the inversion found on each output of a P16R8. The equation for Qout.D actually refers to the D input of one of the P16R8's flip-flops; the output inversion found in a P16R8 is located after the register and is assumed rather than specified.*

Using Don't Care Optimization

Use Don't Care optimization to reduce the amount of logic required for an incompletely specified function. The @DCSET directive (used for logic description sections) and ISTYPE attribute 'dc' (used for signals) specify don't care values for unspecified logic.

Consider the following ABEL-HDL truth table:

```
truth_table ([i3,i2,i1,i0]->[f3,f2,f1,f0])
[ 0, 0, 0, 0]->[ 0, 0, 0, 1];
[ 0, 0, 0, 1]->[ 0, 0, 1, 1];
[ 0, 0, 1, 1]->[ 0, 1, 1, 1];
[ 0, 1, 1, 1]->[ 1, 1, 1, 1];
[ 1, 1, 1, 1]->[ 1, 1, 1, 0];
[ 1, 1, 1, 0]->[ 1, 1, 0, 0];
[ 1, 1, 0, 0]->[ 1, 0, 0, 0];
[ 1, 0, 0, 0]->[ 0, 0, 0, 0];
```

This truth table has four inputs, and therefore sixteen (24) possible input combinations. The function specified, however, only indicates eight significant input combinations. For each of the design outputs (f3 through f0) the truth table specifies whether the resulting value should be 1 or 0. For each output, then, each of the eight individual truth table entries can be either a member of a set of true functions called the on-set, or a set of false functions called the off-set.

Using output f3, for example, the eight input conditions can be listed as on-sets and off-sets as follows (maintaining the ordering of inputs as specified in the truth table above).

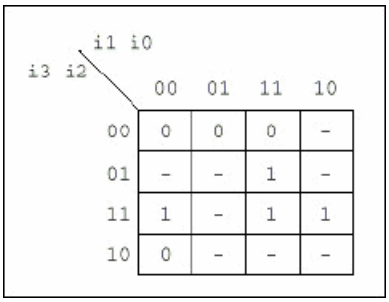
```
on-set of f3 off-set of f3
0 1 1 1 0 0 0 0
1 1 1 1 0 0 0 1
1 1 1 0 0 0 1 1
1 1 0 0 1 0 0 0
```

The remaining eight input conditions that do not appear in either the on-set or off-set are said to be members of the dc-set, as follows for f3.

```
dc-set of f3
0 0 1 0
0 1 0 0
0 1 0 1
0 1 1 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 1
```

Expressed as a Karnaugh map, the on-set, off-set and dc-set would appear as follows, with ones indicating the on-set, zeroes indicating the off-set, and dashes indicating the dc-set.

Truth table expressed as a Karnaugh map



If the don't-care entries in the Karnaugh map are used for optimization, the function for f3 can be reduced to a single product term (f3 = i2) instead of the two (f3 = i3 & i2 & !i0 # i2 & i1 & i0) otherwise required.

The ABEL-HDL compiler uses this level of optimization if the @DCSET directive or ISTYPE 'dc' is included in the ABEL-HDL source file, as shown below.

Source File Showing Don't Care Optimization

```
module dc.
i3,i2,i1,i0 pin;
f3,f2,f1,f0 pin istype 'dc,com';

truth_table ([i3,i2,i1,i0]->[f3,f2,f1,f0])
[ 0, 0, 0, 0]->[ 0, 0, 0, 1];
[ 0, 0, 0, 1]->[ 0, 0, 1, 1];
[ 0, 0, 1, 1]->[ 0, 1, 1, 1];
[ 0, 1, 1, 1]->[ 1, 1, 1, 1];
[ 1, 1, 1, 1]->[ 1, 1, 1, 0];
[ 1, 1, 1, 0]->[ 1, 1, 0, 0];
[ 1, 1, 0, 0]->[ 1, 0, 0, 0];
[ 1, 0, 0, 0]->[ 0, 0, 0, 0];
end
```

This example results in a total of four single-literal product terms, one for each output. The same example (with no istype 'dc') results in a total of twelve product terms.

For truth tables, Don't Care optimization is almost always the best method. For state machines, however, you may not want undefined transition conditions to result in unknown states, or you may want to use a default state (determined by the type of flip-flops used for the state register) for state diagram simplification.

When using don't care optimization, be careful not to specify overlapping conditions (specifying both the on-set and dc-set for the same conditions) in your truth tables and state diagrams. Overlapping conditions result in an error message.

For state diagrams, you can perform additional optimization for design outputs if you specify the @dcstate attribute. If you enter @dcstate in the source file, all state diagram transition conditions are collected during state diagram processing. These transitions are then complemented and applied to the design outputs as don't-cares. You must use @dcstate in combination with @dcset or the 'dc' attribute.

Exclusive OR Equations

Designs written for exclusive-OR (XOR) devices should contain the 'xor' attribute for architecture-independence.

Optimizing XOR Devices

You can use XOR gates directly by writing equations that include XOR operators, or you can use implied XOR gates. XOR gates can minimize the total number of product terms required for an output or they can emulate alternate flip-flop types.

Using XOR Operators in Equations

If you want to write design equations that include XOR operators, you must either specify a device that features XOR gates in your ABEL-HDL source file, or specify the 'xor' attribute for all output signals that will be implemented with XOR gates. This preserves one top-level XOR operator for each design output. For example:

```
module X1
Q1 pin istype 'com,xor';
a,b,c pin;
equations
Q1 = a $ b & c;
end
```

Also, when writing equations for XOR PALs, you should use parentheses to group those parts of the equation that go on either side of the XOR. This is because the XOR operator ($) and the OR operator (#) have the same priority in ABEL-HDL.

Using Implied XORs in Equations

High-level operators in equations often result in the generation of XOR operators. If you specify the 'XOR' attribute, these implied XORs are preserved, decreasing the number of product terms required. For example,

```
module X2
q3,q2,q1,q0 pin istype 'reg,xor';
clock pin;
count = [q3..q0];
equations
count.clk = clock;
count := count.FB + 1;
end
```

This design describes a simple four-bit counter. Since the addition operator results in XOR operators for the four outputs, the 'xor' attribute can reduce the amount of circuitry generated.

***Note**: The high-level operator that generates the XOR operators must be the top-level (lowest priority) operation in the equation. An equation such as count := (count.FB + 1) & !reset does not result in the preservation of top-level XOR operators, since the **&** operator is the top-level operator.*

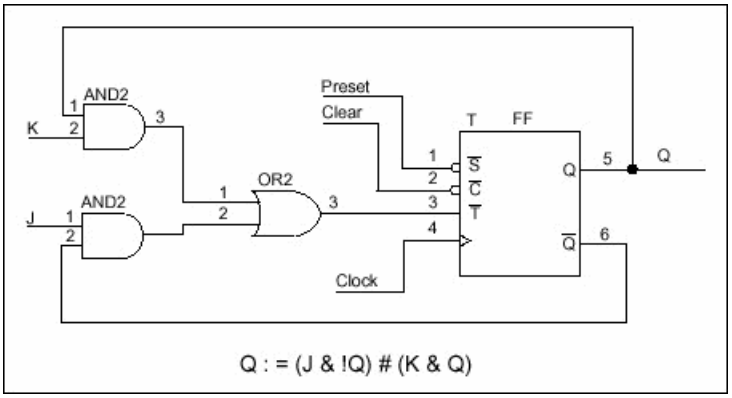
Using XORs for Flip-flop Emulation

Another way to use XOR gates is for flip-flop emulation. If you are using an XOR device that has outputs featuring an XOR gate and D-type flip-flops, you can write your design as if you were going to be implementing it in a device with T-type flip-flops. The XOR gates and D-type flip-flops emulate the specified T-type flip-flops. When using XORs in this way, you should not use the ‘xor’ attribute for output signals unless the target device has XOR gates.

JK Flip-Flop Emulation

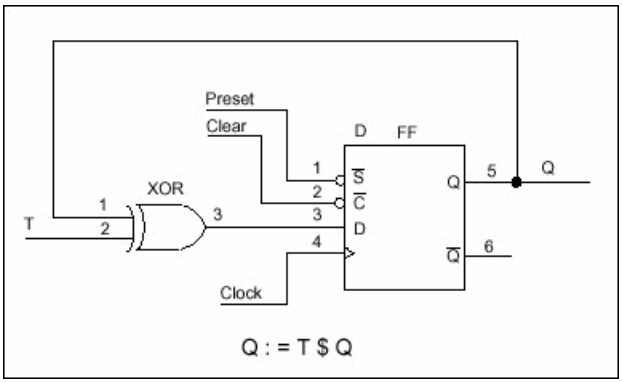
You can emulate JK flip-flops using a variety of circuitry found in programmable devices. When a T-type flip-flop is available, you can emulate JK flip-flops by ANDing the Q output of the flip-flop with the K input. The !Q output is then ANDed with the J input. The figure below illustrates the circuitry and the Boolean expression.

Flip-flop Emulation Using T Flip-flop



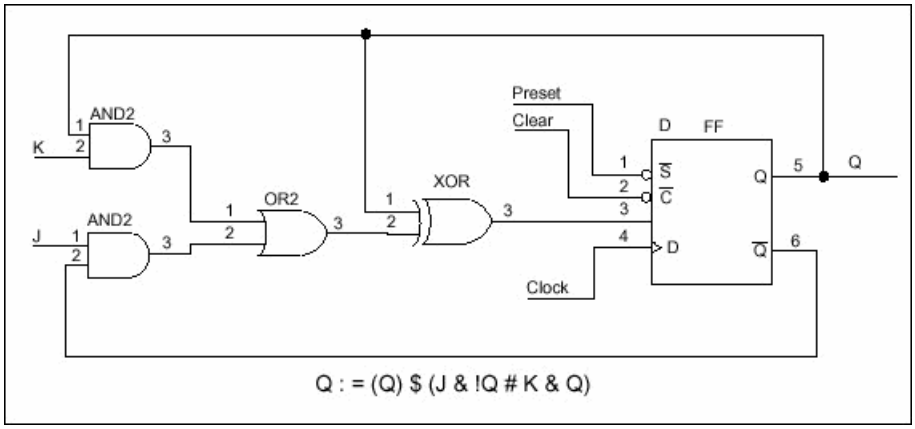
You can emulate a JK flip-flop with a D flip-flop and an XOR gate. This technique is useful in devices such as the GAL20VP8. The circuitry and Boolean expression is shown in the figure below.

T Flip-flop Emulation Using D Flip-flop



Finally, you can also emulate a JK flip-flop by combining the D flip-flop emulation of aT flip-flop with the circuitry of the above figure. The following figure illustrates this concept.

JK Flip-flop Emulation, D Flip-flop with XOR



State Machines

A state machine is a digital device that traverses a predetermined sequence of states. State-machines are typically used for sequential control logic. In each state, the circuit stores its past history and uses that history to determine what to do next.

This section provides some guidelines to help you make state diagrams easy to read and maintain and to help you avoid problems. State machines often have many different states and complex state transitions that contribute to the most common problem, which is too many product terms being created for the chosen device. The topics discussed in the following subsections help you avoid this problem by reducing the number of required product terms.

Use Identifiers Rather than Numbers for States

A state machine has different “states” that describe the outputs and transitions of the machine at any given point. Typically, each state is given a name, and the state machine is described in terms of transitions from one state to another. In a real device, such a state machine is implemented with registers that contain enough bits to assign a unique number to each state. The states are actually bit values in the register, and these bit values are used along with other signals to determine state transitions.

As you develop a state diagram, you need to label the various states and state transitions. If you label the states with identifiers that have been assigned constant values, rather than labeling the states directly with numbers, you can easily change the state transitions or register values associated with each state.

When you write a state diagram, you should first describe the state machine with names for the states, and then assign state register bit values to the state names.

For an example, see the following source file for a state machine named “sequence.” (This state machine is also discussed in the design examples.) Identifiers (A, B, and C) specify the states. These identifiers are assigned a constant decimal value in the declaration section that identifies the bit values in the state register for each state. A, B, and C are only identifiers: they do not indicate the bit pattern of the state machine. Their declared values define the value of the state register (sreg) for each state. The declared values are 0, 1, and 2.

Using identifiers for states

```
module Sequence
title 'State machine example';


q1,q0 pin 14,15 istype 'reg';
clock,enab,start,hold,reset pin 1,11,4,2,3;
```

```
halt pin 17 istype 'reg';
in_B,in_C pin 12,13 istype 'com';
sreg = [q1,q0];
"State Values...
A = 0; B = 1; C = 2;

equations
[q1,q0,halt].clk = clock;
[q1,q0,halt].oe = !enab;
state_diagram sreg;
State A: " Hold in state A until start is active.
in_B = 0;
in_C = 0;
IF (start & !reset) THEN B WITH halt := 0;
ELSE A WITH halt := halt.fb;

State B: " Advance to state C unless reset is active
in_B = 1; " or hold is active. Turn on halt indicator
in_C = 0; " if reset.
IF (reset) THEN A WITH halt := 1;
ELSE IF (hold) THEN B WITH halt := 0;
ELSE C WITH halt := 0;

State C: " Go back to A unless hold is active
in_B = 0; " Reset overrides hold.
in_C = 1;
IF (hold & !reset) THEN C WITH halt := 0;
ELSE A WITH halt := 0;
test_vectors([clock,enab,start,reset,hold]->[sreg,halt,in_B,in_C])
[ .p. , 0 , 0 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
[ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ C , 0 , 0 , 1 ];

[ .c. , 0 , 1 , 0 , 0 ]->[ A , 0 , 0 , 0 ];
[ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 1 , 0 ]->[ A , 1 , 0 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ A , 1 , 0 , 0 ];

[ .c. , 0 , 1 , 0 , 0 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 1 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 1 ]->[ B , 0 , 1 , 0 ];
[ .c. , 0 , 0 , 0 , 0 ]->[ C , 0 , 0 , 1 ];
end
```

Power-up Register States

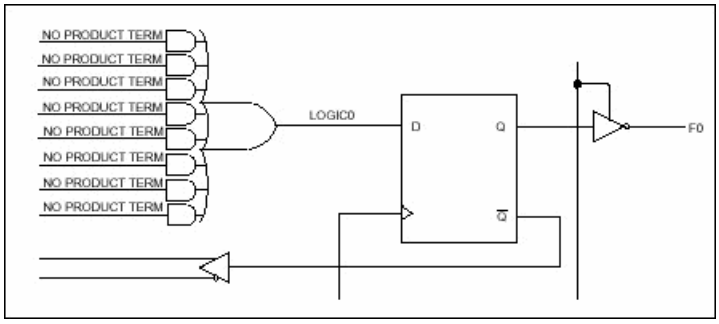
If a state machine has to have a specific starting state, you must define the register power-up state in the state diagram description or make sure that your design goes to a known state at power-up. Otherwise, the next state is undefined.

Unsatisfied Transition Conditions

D-Type Flip-Flops

For each state described in a state diagram, you specify the transitions to the next state and the conditions that determine those transitions. For devices with D-type flip-flops, if none of the stated conditions are met, the state register, shown in the following figure, is cleared to all 0s on the next clock pulse. This action causes the state machine to go to the state that corresponds to the cleared state register. This can either cause problems or you can use it to your advantage, depending on your design.

D-type register with false inputs



You can use the clearing behavior of D-type flip-flops to eliminate some conditions in your state diagram, and some product terms in the converted design, by leaving the cleared-register state transition implicit. If no specified transition condition is met, the machine goes to the cleared-register state. This behavior can also cause problems if the cleared-register state is undefined in the state diagram, because if the transition conditions are not met for any state, the machine goes to an undefined state and stays there.

To avoid problems caused by this clearing behavior, always have a state assigned to the cleared-register state. Or, if you don't assign a state to the cleared-register state, define every possible condition so that some condition is always met for each state. You can also use the automatic transition to the cleared-register state by eliminating product terms and explicit definitions of transitions. You can also use the cleared-register state to satisfy illegal conditions.

Other Flip-flops

If none of the state conditions are met in a state machine that employs JK, RS, and T-type flip-flops, the state machine does not advance to the next state but holds its present state, due to the low input to the register from the OR array output. In such a case, the state machine can get stuck in a state. You can use this holding behavior to your advantage in some designs.

If you want to prevent the hold, you can use the complement array provided in some devices (such as the F105) to detect a “no conditions met” situation and reset the state machine to a known state.

Precautions for Using Don’t Care Optimization

When you use don't care optimization, you need to avoid certain design practices. The most common design technique that conflicts with this optimization is mixing equations and state diagrams to describe default transitions. For example, consider the design shown below.

State machine description with conflicting logic

```
module TRAFFIC
title 'Traffic Signal Controller'
Clk,SenA,SenB pin 1, 8, 7;
PR pin 16; "Preset control
GA,YA,RA pin 15..13;
GB,YB,RB pin 11..9;

"Node numbers are not required if fitter is used
S3..S0 node 31..34 istype 'reg_sr,buffer';
COMP node 43;

H,L,Ck,X = 1, 0, .C., .X.;
Count = [S3..S0];

"Define Set and Reset inputs to traffic light flip-flops
GreenA = [GA.S,GA.R];
YellowA = [YA.S,YA.R];
RedA = [RA.S,RA.R];
GreenB = [GB.S,GB.R];
YellowB = [YB.S,YB.R];
RedB = [RB.S,RB.R];
On = [ 1 , 0 ];
Off = [ 0 , 1 ];
" test_vectors edited
equations
[GB,YB,RB].AP = PR;
[GA,YA,RA].AP = PR;
[GB,YB,RB].CLK = Clk;
[GA,YA,RA].CLK = Clk;
[S3..S0].AP = PR;
[S3..S0].CLK = Clk;

"Use Complement Array to initialize or restart
[S3..S0].R = (!COMP & [1,1,1,1]);
[GreenA,YellowA,RedA] = (!COMP & [On ,Off,Off]);
[GreenB,YellowB,RedB] = (!COMP & [Off,Off,On ]);

state_diagram Count
State 0: if ( SenA & !SenB ) then 0 with COMP = 1;
if (!SenA & SenB ) then 4 with COMP = 1;
if ( SenA == SenB ) then 1 with COMP = 1;
State 1: goto 2 with COMP = 1;
State 2: goto 3 with COMP = 1;
State 3: goto 4 with COMP = 1;
```

```
State 4: GreenA = Off;
YellowA = On ;
goto 5 with COMP = 1;
State 5: YellowA = Off;
RedA = On ;
RedB = Off;
GreenB = On ;
goto 8 with COMP = 1;
State 8: if (!SenA & SenB ) then 8 with COMP = 1;
if ( SenA & !SenB ) then 12 with COMP = 1;
if ( SenA == SenB ) then 9 with COMP = 1;
State 9: goto 10 with COMP = 1;
State 10: goto 11 with COMP = 1;
State 11: goto 12 with COMP = 1;
State 12: GreenB = Off;
YellowB = On ;
goto 13 with COMP = 1;
State 13: YellowB = Off;
RedB = On ;
RedA = Off;
GreenA = On ;
goto 0 with COMP = 1;
end
```

This design uses the complement array feature of the Signetics FPLA devices to perform an unconditional jump to state [0,0,0,0]. If you use the **@DCSET** directive, the equation that specifies this transition

```
[S3,S2,S1,S0].R = (!COMP & [1,1,1,1]);
```

will conflict with the dc-set generated by the state diagram for S3.R, S2.R, S1.R, and S0.R. If equations are defined for state bits, the **@DCSET** directive is incompatible. This conflict would result in an error and failure when the logic for this design is optimized.

To correct the problem, you must remove the **@DCSET** directive so that the implied dc-set equations are folded into the off-set for the resulting logic function. Another option is to rewrite the module as shown below.

@DCSET-compatible state machine description

```
module TRAFFIC1
title 'Traffic Signal Controller'

Clk,SenA,SenB pin 1, 8, 7;
PR pin 16; "Preset control
GA,YA,RA pin 15..13;
GB,YB,RB pin 11..9;

S3..S0 node 31..34 istype 'reg_sr,buffer';
H,L,Ck,X = 1, 0, .C., .X.;
Count = [S3..S0];
```

```
"Define Set and Reset inputs to traffic light flip flops
GreenA = [GA.S,GA.R];
YellowA = [YA.S,YA.R];
RedA = [RA.S,RA.R];
GreenB = [GB.S,GB.R];
YellowB = [YB.S,YB.R];
RedB = [RB.S,RB.R];
On = [ 1 , 0 ];
Off = [ 0 , 1 ];
" test_vectors edited
equations
[GB,YB,RB].AP = PR;
[GA,YA,RA].AP = PR;
[GB,YB,RB].CLK = Clk;
[GA,YA,RA].CLK = Clk;
[S3..S0].AP = PR;
[S3..S0].CLK = Clk;
@DCSET
state_diagram Count
State 0: if ( SenA & !SenB ) then 0;
if (!SenA & SenB ) then 4;
if ( SenA == SenB ) then 1;
State 1: goto 2;
State 2: goto 3;
State 3: goto 4;

State 4: GreenA = Off;
YellowA = On ;
goto 5;
State 5: YellowA = Off;
RedA = On ;
RedB = Off;
GreenB = On ;
goto 8;
State 6: goto 0;
State 7: goto 0;

State 8: if (!SenA & SenB ) then 8;
if ( SenA & !SenB ) then 12;
if ( SenA == SenB ) then 9;
State 9: goto 10;
State 10: goto 11;
State 11: goto 12;
State 12: GreenB = Off;
```

```
YellowB = On ;
goto 13;
State 13: YellowB = Off;
RedB = On ;
RedA = Off;
GreenA = On ;
goto 0;
State 14: goto 0;
State 15: "Power up and preset state
RedA = Off;
YellowA = Off;
GreenA = On ;
RedB = On ;
YellowB = Off;
GreenB = Off;
goto 0;
end
```

Number Adjacent States for One-bit Change

You can reduce the number of product terms produced by a state diagram by carefully choosing state register bit values. Your state machine should be described with symbolic names for the states, as described above. Then, if you assign the numeric constants to these names so that the state register bits change by only one bit at a time as the state machine goes from state to state, you will reduce the number of product terms required to describe the state transitions.

As an example, take the states A, B, C, and D, which go from one state to the other in alphabetical order. The simplest choice of bit values for the state register is a numeric sequence, but this is not the most efficient method. To see why, examine the following bit value assignments. The preferred bit values cause a one-bit change as the machine moves from state B to C, whereas the simple bit values cause a change in both bit values for the same transition. The preferred bit values produce fewer product terms.

Simple vs. preferred bit values

State	Simple Bit Values	Preferred Bit Values
A	00	00
B	01	01
C	10	11
D	11	10

If one of your state register bits uses too many product terms, try reorganizing the bit values so that state register bit changes in value as few times as possible as the state machine moves from state to state.

Obviously, the choice of optimum bit values for specific states can require some tradeoffs; you may have to optimize for one bit and, in the process, increase the value changes for another. The object should be to eliminate as many product terms as necessary to fit the design into the device.

Use State Register Outputs to Identify States

Sometimes it is necessary to identify specific states of a state machine and signal an output that the machine is in one of these states. Fewer equations and outputs are needed if you organize the state register bit values so that one bit in the state register determines if the machine is in a state of interest. Take, for example, the following sequence of states in which identification of the C*n* states is required.

State register bit values

State Name	Q3	Q2	Q1
A	0	0	0
B	0	0	1
C1	1	0	1
C2	1	1	1
C3	1	1	0
D	0	1	0

This choice of state register bit values allows you to use Q3 as a flag to indicate when the machine is in any of the C*n* states. When Q3 is high, the machine is in one of the C*n* states. Q3 can be assigned directly to an output pin on the device. Notice also that these bit values change by only one bit as the machine cycles through the states, as is recommended in the section above.

Using Symbolic State Descriptions

Symbolic state descriptions describe a state machine without having to specify actual state values. A symbolic state description is shown below.

Symbolic state description

```
module SM
a,b,clock pin; " inputs
a_reset,s_reset pin; " reset inputs
x,y pin istype 'com'; " simple outputs

sreg1 state_register;
S0..S3 state;

equations
sreg1.clk = clock;

state_diagram sreg1
state S0:
goto S1 with {x = a & b;
y = 0; }
state S1: if (a & b)
then S2 with {x = 0;
y = 1; }
state S2: x = a & b;
y = 1;
if (a) then S1 else S2;
```

```
state S3:
goto S0 with {x = 1;
y = 0; }

async_reset S0: a_reset;
sync_reset S0: s_reset;
end
```

Symbolic state descriptions use the same syntax as non-symbolic state descriptions; the only difference is the addition of the **State_register** and **State** declarations and the addition of symbolic synchronous and asynchronous reset statements.

Symbolic Reset Statements

In symbolic state descriptions, the **Sync_Reset** and **Async_Reset** statements specify synchronous or asynchronous state machine reset logic. For example, to specify that a state machine must asynchronously reset to state Start when the Reset input is true, you write:

ASYNC_RESET Start : (Reset) ;

Symbolic Test Vectors

You can also write test vectors to refer to symbolic state values by entering the symbolic state register name in the test vector header (in the output sections) and the symbolic state names in the test vectors as output values.

Using Complement Arrays

The complement array is a unique feature found in some logic sequencers. This topic shows a typical use ending counter sequence.

You can use transition equations to express the design of counters and state machines in some devices with JK or SR flip-flops. A transition equation expresses a state of the circuit as a variation of, or adjustment to, the previous state. This type of equation eliminates the need to specify every node of the circuit; you can specify only those that require a transition to the opposite state.

An example of transition equations is shown in the example below, a source file for a decade counter having a single (clock) input and a single latched output. This counter divides the clock input by a factor of ten and generates a 50% duty-cycle squarewave output. In addition to its registered outputs, this device contains a set of “buried” (or feedback) registers whose outputs are fed back to the product term inputs. These nodes must be declared, and can be given any names.

Node 49, the complement array feedback, is declared (as COMP) so that it can be entered into each of the equations. In this design, the complement array feedback is used to wrap the counter back around to zero from state nine, and also to reset it to zero if an illegal counter state is encountered. Any illegal state (and also state 9) will result in the absence of an active product term to hold node 49 at a logic low. When node 49 is low, product term 9 resets each of the feedback registers so that the counter is set to state zero. (To simplify the following description of the equations below, node 49 and the complement array feedback are temporarily ignored.)

The first equation states that the F0 (output) register is set (to provide the counter output) and the P0 register is set when registers P0, P1, P2, and P3 are all reset (counter at state zero) and the clear input is low. The complemented outputs of the registers (with the clear input low) form product term 0. Product term 0 sets register P0 to increment the decade counter to state 1, and it sets register F0 to provide an output at pin 18.

Example

Transition Equations for a Decade Counter

```
module DECADE
title 'Decade Counter Uses Complement Array
Michael Holley Data I/O Corp'

decade device 'F105';

Clk,Clr,F0,PR pin 1,8,18,19;
P3..P0 node 40..37;
COMP node 49;

F0,P3..P0 istype 'reg_sr,buffer';
_State = [P3,P2,P1,P0];
H,L,Ck,X = 1, 0, .C., .X.;

Equations

[P3,P2,P1,P0,F0].ap = PR;
[F0,P3,P2,P1,P0].clk = Clk;
"Output Next State Present State Input
[F0.S, COMP,  P0.S] = !P3.Q & !P2.Q & !P1.Q & !P0.Q & !Clr; "0 to 1
[ COMP, P1.S,P0.R] = !P3.Q & !P2.Q & !P1.Q & P0.Q & !Clr; "1 to 2
[ COMP, P0.S] = !P3.Q & !P2.Q & P1.Q & !P0.Q & !Clr; "2 to 3
[ COMP, P2.S,P1.R,P0.R] = !P3.Q & !P2.Q & P1.Q & P0.Q & !Clr; "3 to 4
[ COMP, P0.S] = !P3.Q & P2.Q & !P1.Q & !P0.Q & !Clr; "4 to 5
[F0.R, COMP, P1.S,P0.R] = !P3.Q & P2.Q & !P1.Q & P0.Q & !Clr; "5 to 6
[ COMP, P0.S] = !P3.Q & P2.Q & P1.Q & !P0.Q & !Clr; "6 to 7
[ COMP,P3.S,P2.R,P1.R,P0.R] = !P3.Q & P2.Q & P1.Q & P0.Q & !Clr; "7 to 8
[ COMP P0.S] = P3.Q & !P2.Q & !P1.Q & !P0.Q & !Clr; "8 to 9
[ P3.R,P2.R,P1.R,P0.R] = !COMP; "Clear
"After Preset, clocking is inhibited until High-to-Low clock transition.
test_vectors  ([Clk,PR,Clr] -> [_State,F0 ])
  [ 0 , 0, 0 ] -> [ X , X];
  [ 1 , 1, 0 ] -> [^b1111, H]; " Preset high
  [ 1 , 0, 0 ] -> [^b1111, H]; " Preset low
  [ Ck, 0, 0 ] -> [ 0 , H]; " COMP forces to State 0
  [ Ck, 0, 0 ] -> [ 1 , H];
  " ..vectors edited...
  [ Ck, 0, 1 ] -> [ 0 , H]; " Clear
end
```

The second equation performs a transition from state 1 to state 2 by setting the P1 register and resetting the P0 register. (The .R dot extension is used to define the reset input of the registers.) In state 2, the F0 register remains set, maintaining the high output. The third equation again sets the P0 register to achieve state 3 (P0 and P1 both set), while the fourth equation resets P0 and P1, and sets P2 for state 4, and so on.

Wraparound of the counter from state 9 to state 0 is achieved by means of the complement array node (node 49). The last equation defines state 0 (P3, P2, P1, and P0 all reset) as equal to !COMP, that is, node 49, at a logic low. When this equation is processed, the fuses are blown. As a result, the !COMP signal is true to generate product term 9 and reset all the “buried” registers to zero.

ABEL-HDL and Truth Tables

Truth Tables in ABEL-HDL represent a very easy and straightforward description method, well suited in a number of situations involving combinational logic.

The principle of the Truth Table is to build an exhaustive list of the input combinations (referred to as the ON set) for which the outputs become active.

The following list summarizes design considerations for Truth Tables. Following the list are more detailed examples.

- The **OFF-set** lines in a Truth Table are necessary when more than one output is assigned in the Truth Table. In this case, not all Outputs are fired under the same conditions, and therefore OFF-set conditions do exist.
- OFF-set lines are ignored because they represent the default situation, unless the output variable is declared **dc**. In this case, a third set is built, the DC-set, and the Output inside it is assigned proper values to achieve the best logic reduction possible.
- If output type dc (or @dcset) is not used and multiple outputs are specified in a Truth table, consider the outputs one by one and ignore the lines where the selected output is not set.
- Don't Cares (.X.) used on the right side of a Truth Table have no optimization effect.
- When dealing with multiple outputs of different kinds, avoid general settings like @DCSET, which will affect all your outputs. Use istype ‘.....DC’ on outputs for which this reduction may apply.
- Beware of Outputs for which the ON-set might be empty.
- As a general guideline, it is important not to rely on first impression or simple intuition to understand Truth tables. The way they are understood by the compiler is the only possible interpretation. This means that Truth Tables should be presented in a clear and understandable format, should avoid side effects, and should be properly documented (commented).

Basic Syntax and Simple Examples

In the following example, the lines commented as L1 and L2 are the **ON-set**. Lines **L3 and L4 are ignored** because Out is type *default* (meaning ‘0’ for unspecified combinations). The resulting equation does confirm this.

```
MODULE DEMO1
TITLE 'Module 1'
" Inputs
 A, B, C pin;
"Output
 Out pin istype 'com';
Truth_Table
( [A, B, C] -> Out )
 [0, 1, 0] -> 1; // L1
 [1, 1, 1] -> 1; // L2
 [0, 0, 1] -> 0; // L3
 [1, 0, 0] -> 0; // L4
END
```

```
// Resulting Reduced Equation :
// Out = (!A & B & !C) # (A & B & C);
```

Module 2 (below) differs from Module 1 (above) because Out is now type 'COM, **DC**'. (optimizable don't care)

In this case, the lines commented as L1 and L2 are the ON-set, L3 and L4 are the *OFF-set*, and other combinations become *don't care* (DC-set) meaning 0 or 1 to produce the best logic reduction. As a result in this example, the equation is very simple.

@DCSET instruction would have produced the same result as to declare Out of type **dc**. But @DCSET must be used with care when multiple outputs are defined: they all become dc.

```
MODULE DEMO1
TITLE 'Module 2'
" Inputs
 A, B, C pin;
"Output
 Out pin istype 'com, dc';
Truth_Table
( [A, B, C] -> Out )
 [0, 1, 0] -> 1; // L1
 [1, 1, 1] -> 1; // L2
 [0, 0, 1] -> 0; // L3
 [1, 0, 0] -> 0; // L4
END
// Resulting Reduced Equation :
// Out = (B);
```

Influence of Signal Polarity

We'll see now with Module 3 how the polarity of the signal may influence the truth table:

In this example, **Out1** and **Out2** are strictly equivalent. For !Out1, note that the ON set is the 0 values. *The third line L3 is ignored*.

```
MODULE DEMO2
TITLE 'Module 3'
" Inputs
 A, B, C pin;
"Output
 Out1 pin istype 'com, neg';
 Out2 pin istype 'com, neg';
 Out3 pin istype 'com, neg'; // BEWARE

Truth_Table
( [A, B, C] -> [!Out1, Out2, Out3] )
 [0, 0, 1] -> [ 0, 1, 0 ];//L1
 [0, 1, 1] -> [ 0, 1, 0 ];//L2
 [1, 1, 0] -> [ 1, 0, 1 ];//L3
END
// Resulting Equations :
```

```
// !Out1 = !Out2 = (A # !C);
// or: Out1 = Out2 = (!A & C);
// BUT: Out3 = (A & B & !C); <<what you wanted ?
```

For active-low outputs, one must be careful to specify **1** for the active state if the Output appears without the exclamation point (!).

0 must be used when **!**output is defined in the table header.

We recommend the style used for Out1.

For Out3, line used is L3, L1 and L2 are ignored.

Using .X. in Truth Tables Conditions

Don't Care used on the left side in Truth tables have no optimization purpose; they only serve as a shortcut to write several conditions in one single line.

Be careful when using .X. in conditions. This can lead to overlapping conditions, which look inconsistent (see Module 4 below). *Due to the way the compiler works, this type of inconsistency is not checked or reported.* In fact, only the ON-set condition is taken into account, the OFF-set condition is ignored. The following example illustrates this:

```
MODULE DEMO3
TITLE 'Module 4'
"Inputs
 A, B, C pin;
"Output
 Out pin istype 'com';
"Equivalence
 X = .X.;
Truth_Table
```

([A, B, C] -> Out)

[0, 0, 1] -> 0; //L1 ignored in fact

```
 [0, 1, 0] -> 1; //L2
 [1, X, X] -> 1; //L3
 [0, 0, 1] -> 1; //L4 incompatible
 [1, 1, 0] -> 0; //L5 incompatible
END
// Result : Out = A # (B & !C) # (!B & C)
```

L1 is in fact ignored. Out is active high, therefore only line L4 is taken into account.

Likewise, L5 intersects L3, but is ignored since it is not in the ON-set for Out.

Globally, only L2, L3 and L4 are taken into account, as we can check in the resulting equation, without any error reported.

Using .X. on the right side

The syntax allows you to use .X. as a target value for an output. In this case, the condition is simply **ignored**.

***Note**: This is **not** the method to specify optimizable don't care states.*

Module 5 shows that**-> .X.** states **are not optimized** if DC type or @DCSET are not used.

These lines are ALWAYS ignored.

```
MODULE DEMO6
TITLE 'Module 5'
"Inputs
 A, B, C pin;
"Output
 Out pin istype 'com';
"Equivalence
 X = .X.;
Truth_Table
( [A, B, C] -> Out )
 [0, 0, 0] -> 0;
 [0, 0, 1] -> X;
 [0, 1, 0] -> 1;
 [0, 1, 1] -> X;
 [1, X, X] => X;
END
// As is : Out = (!A & B & !C);
// With istype 'com,DC' : Out = (B);
```

They are in fact of no use, except maybe as a way to document that output does not matter.

Special Case: Empty ON-set

This is a special case that is unlikely, but may sometimes occur. Consider this example:

```
MODULE DEMO5
TITLE 'Module 6'
"Inputs
 A, B, C pin;
"Output
 Out pin istype 'com, pos';
Truth_Table
( [A, B, C] -> Out )
 [0, 0, 1] -> 0;
 [0, 1, 0] -> 0;
 [1, 0, 0] -> 0;
// [0, 0, 0] -> 1;//changes everything!
END
// Without the last line L4 :
// !Out=(A & !B & !C)# (!A & B & !C)# (!A & !B & C);
// WITH L4 : Out = (!A & !B & !C);
```

What we obtain is slightly unexpected. This table should produce **Out=0;** as the result. (We enumerated only OFF conditions, and the polarity is POS (or default), so unlisted cases should **also** turn into zeroes.)

One reason to build such a table could be when multiple outputs are defined and when Out needs to be shut off for whatever reason.

In the absence of the line L4, the **result is not intuitive.** The output is **0 only** for the listed cases (L1, L2, L3), and is **1 for *all* other cases**, even if dc or pos is used.

When line L4 is restored, then the output equation becomes Out = (!A & !B & !C); because we fall in the general situation where the ON-set is not empty.

Registered Logic in Truth tables

Truth Tables can specify registered outputs. In this case, the assignment becomes **:>** (instead of **->**).

Designing with CPLDs

ABEL-HDL allows you to generate source files with efficient logic for CPLDs, including ispLSI devices. This section of ispLEVER Help describes designing with CPLDs.

CPLD Design Strategies

The following design strategies are helpful when designing for CPLDs:

- Define external and internal signals with PIN and NODE statements, respectively.
- For state machines and truth tables, include @Dcset (or 'dc' attributes) if possible, since it usually reduces logic.
- Use only dot extensions that are appropriate for CPLD designs.
- Use intermediate signals to create multi-level logic to match CPLD architectures.

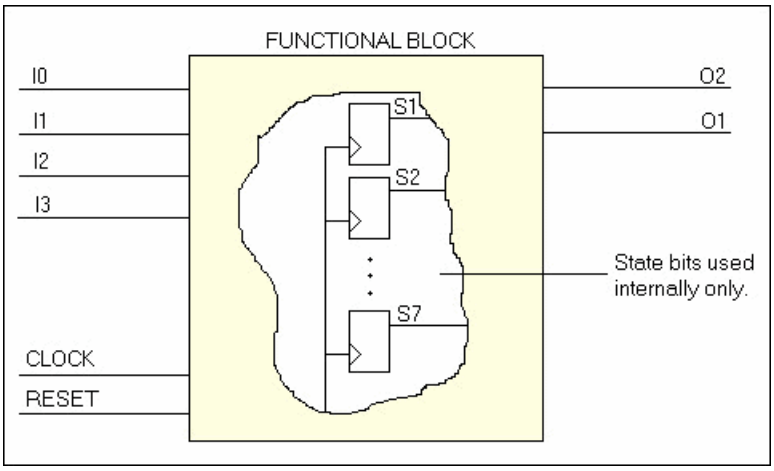
Declaring Signals

The first step in creating a logic module for a CPLD is to declare the signals in your design. In ABEL-HDL, you do this with PIN and NODE statements.

Pin	PIN statements indicate external signals (used as inputs and outputs to the functional block). Pin numbers are optional in ABEL-HDL and are not recommended for CPLDs, since pin statements do not actually generate pins on the device package. If you declare an external signal as a node instead of a pin, the compiler may interpret the signal incorrectly and delete it.
Node	NODE statements indicate internal signals (not accessible by circuitry outside the functional block). Signals declared as nodes are expected to have a source and loads.

The following figure shows a state machine as a functional block. State bits S1 through S7 are completely internal; all other signals are external.

Hypothetical State Machine as a Functional Block



The code example below shows the corresponding signal declarations. The CLOCK, RESET, input,and output signals must connect with circuitry outside the functional block, so they are declared as pins. The state bits are not used outside the functional block, so they are declared as nodes.

Signal Declarations

```
CLOCK, RESET    Pin;
I0,I1,I2,I3    Pin;
O1,O2     Pin;
S7,S6,S5,S4,S3,S2,S1  Node;
```

Using Intermediate Signals

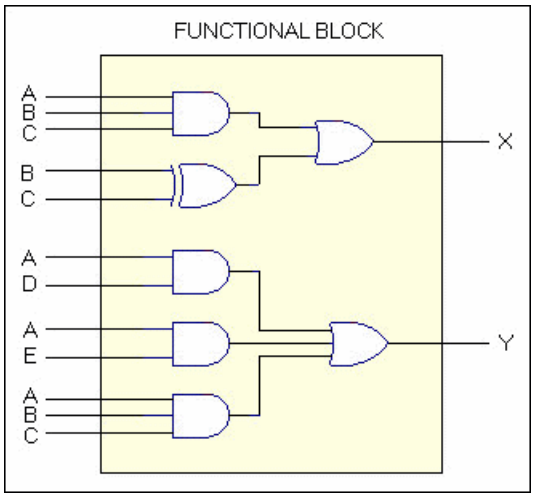
An intermediate signal is a combinatorial signal that is declared as a node and used as a component of other more complex signals in a design. Intermediate signals minimize logic by forcing it to be factored. Creating intermediate signals in an ABEL-HDL logic description has the following benefits:

- Reduces the amount of optimization a compiler needs to perform
- Increases the chances of a fit
- Simplifies the ABEL-HDL source file

Example Without Intermediate Signal

The following figure shows a schematic of combinational logic. Signals A, B, C, D, and E are inputs; X and Y are outputs. There are no intermediate signals; every declared signal is an input or an output to the subcircuit.

Schematic Without Intermediate Signal



The following example shows the ABEL-HDL declarations and equations that would generate the logic in the above figure.

Example

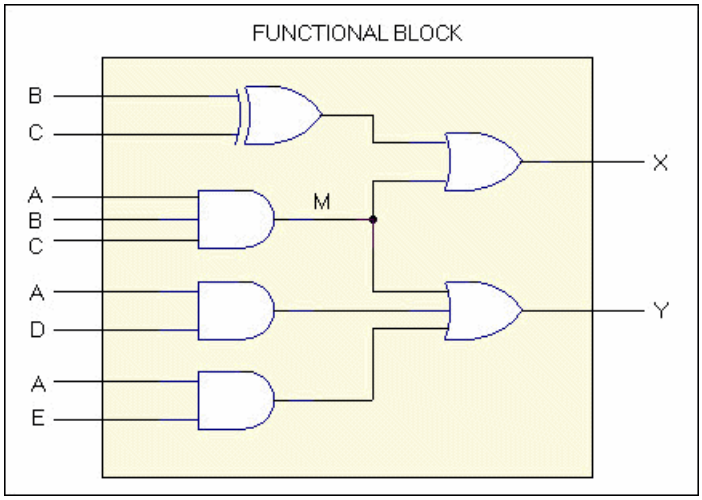
Declarations and Equations

```
"declarations
A, B, C, D, E pin;
X, Y pin;
equations
X = (A&B&C) # (B$C);
Y = (A&D) # (A&E) # (A&B&C);
```

Example With Intermediate Signal

The schematic example below shows logic using an intermediate signal, M, which is declared as a node and named, but is used only inside the subcircuit as a component of other, more complex signals.

Schematic with Intermediate Signal M



The following example shows the declarations and equations that would generate the logic.

Example

Declarations and Equations

```
"declarations
A, B, C, D, E  pin;
X, Y   pin;
M    node;

equations
"intermediate signal equations
M = A&B&C;
X = M # (B$C);
Y = (A&D) # (A&E) # M;
```

Design descriptions that do not use intermediate signals, and those that use intermediate signals, are functionally the same.

Without the intermediate signal, compilation generates the AND gate associated with A&B&C twice, and the device compiler must filter out the common term.

With the intermediate signal, this sub-signal is generated only once as the intermediate signal, M, and the compiler has less to do.

Example of CPLD with Intermediate Signal

Using intermediate signals in a large design, targeted for a complex PLD, can save compiler optimization effort and time. It also makes the design description easier to interpret. Compare the following two state machine descriptions. Note that the first description is easier to read.

Example

State Machine Description without Intermediate Signals

```
CASE
which_code_enter==from_disarmed_ready:
CASE
(sens_code==sens_off) & (key_code!=key_pound)
    & (key_code!=key_star)
    & (key_code!=key_none):
code_entry_?X  WITH {
   which_code_enter := which_code_enter; }
(key_code==sens_off) & (key_code==key_none):
code_entry_?Y WITH {
  which_code_enter := which_code_enter; }
(key_code==key_pound)# (key_code==key_star):
error;
(sens_code!=sens_off):
error;
ENDCASE

which_code_enter==from_armed:
```

```
CASE
(key_code!=key_pound)
& (key_code!=key_star)
& (key_code!=key_none):
code_entry_?X WITH {
   which_code_enter := which_code_enter;
((key_code==key_pound) # (key_code==key_star)):
armed   WITH {
   which_code_enter := which_code_enter; }
(key_code==key_none):
code_entry_?Y WITH {
   which_code_enter := which_code_enter; }
ENDCASE
ENDCASE
```

Example

State Machine Description with Intermediate Signals

```
CASE

enter_from_disarmed_ready:

CASE

sensors_off & key_numeric:
code_entry_?X  WITH {
 which_code_enter := which_code_enter; }
sensors_off & key_none:
code_entry_?Y  WITH {
  which_code_enter := which_code_enter; }
key_pound_star:
error;

!sensors_off:
error;

ENDCASE

enter_from_armed:

CASE

key_numeric:
code_entry_?X  WITH {
   which_code_enter := which_code_enter; }
```

```
key_pound_star:
armed   WITH {
   which_code_enter := which_code_enter; }

key_none:
code_entry_?Y  WITH {
  which_code_enter := which_code_enter; }
ENDCASE

ENDCASE
```

The declarations and equations required to create the intermediate signals used in the above example are shown below.

Example

Intermediate Signal Declarations and Equations

```
"pin and node declarations
sens_code_0, sens_code_1,
sens_code_2, sens_code_3  pin;

key_code_0, key_code_1,
key_code_2, key_code_3  pin;

which_code_enter_0,
which_code_enter_1,
which_code_enter_2   node istype 'reg';

"set declarations
which_code_enter = which_code_enter_0..which_code_enter_2];
sens_code = [sens_code_0..sens_code_3];
key_code = [key_code_0 ..key_code_3];

"code-entry sub-states
from_disarmed_ready = [1, 0, 0];
from_armed = [0, 0, 0];
sens_off = [0, 0, 0, 0];
"key encoding
key_pnd = [1, 1, 0, 0];
key_str = [1, 0, 1, 1];
key_non = [0, 0, 0, 0];

"intermediate signals
enter_from_disarmed_ready node;
enter_from_armed node;
sensors_off node;
key_numeric node;
```

```
key_none node;
key_pound_star node;
equations
"intermediate equations
enter_from_disarmed_ready =
(which_code_enter==from_disarmed_ready);
enter_from_armed  = (which_code_enter==from_armed);
sensors_off  = (sens_code==sens_off);
key_numeric  = (key_code!=key_pnd)
   & (key_code!=key_str)
   & (key_code!=key_non);
key_none =   (key_code==key_non);

key_pound_star = (key_code==key_pnd)
   # (key_code==key_str);
```

Rules for Using Intermediate Signals

For large designs, using intermediate signals can be essential. An expression such as

```
IF (input==code_1) . . .
```

generates a product term (AND gate). If the input is 8 bits wide, so is the AND gate. If the expression above is used 10 times, the amount of logic generated will cause long run times during compilation and fitting or may cause fitting to fail.

If you write the expression as an intermediate equation,

```
code_1_found node;

equations
code_1_found = (input==code_1);
```

you can use the intermediate signal many times without creating an excessive amount of circuitry.

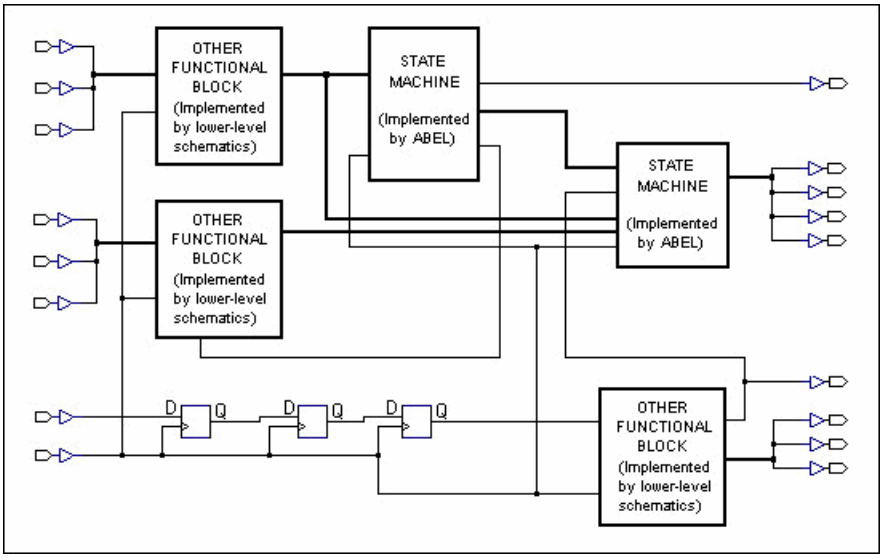
```
IF code_1_found . . .
```

Another way to create intermediate equations is to use the @Carry directive. The @Carry directive causes comparators and adders to be generated using intermediate equations for carry logic. This results in an efficient multilevel implementation.

Typical CPLD Design

The following figure shows a typical CPLD design.

Typical CPLD Design



Verilog HDL Design Entry

The ispLEVER software supports Verilog HDL, a hardware description language used to design and document electronic systems. Verilog HDL allows designers to design at various levels of abstraction.

All Lattice devices support Verilog HDL design.

Adding a Verilog HDL Module to Your Design

To add a Verilog HDL module to a design, you can either import a `.v` file, or create a new Verilog HDL module file with the Text Editor.

Creating a New Verilog HDL Module

You can use the Text Editor to create a new Verilog HDL module.

To create a new Verilog HDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. Select **Verilog Module** and click **OK**. The Text Editor window appears together with the New Verilog Module dialog box.
3. In the dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new Verilog HDL file appears in the Text Editor window.
5. Use the commands on the Edit menu to Cut, Copy, Paste, Find, or Replace text.

Synthesizing Your Verilog HDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor Graphics LeonardoSpectrum. You can synthesize your Verilog HDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

VHDL Design Entry

VHDL is a language for describing the structure and function of integrated circuits. VHDL allows you to:

- Describe the hierarchical structure and interconnect of a design.
- Specify the function of designs using familiar programming language forms.
- Simulate the design before being manufactured, so that design alternatives can be quickly compared and tested.

All Lattice devices support VHDL design.

Adding a VHDL Module to Your Design

To add a VHDL module to a design, you can either import a `.vhd` file, or create a new VHDL module file with the Text Editor.

Creating a New VHDL Module

You can use the Text Editor to create a new VHDL module.

To create a new VHDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. In the dialog box, select VHDL Module and click **OK**. The Text Editor window appears together with the New VHDL Source dialog box.
3. In the New VHDL Source dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new VHDL file appears in the Text Editor window.
5. Use the commands in the Edit menu to Cut, Copy, Paste, or Replace text.

Synthesizing Your VHDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor graphics LeonardoSpectrum. You can synthesize your VHDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

EDIF Design Entry

The Electronic Design Interchange Format (EDIF) is a format used to exchange design data between different ECAD systems.

The EDIF format is designed to be written and read by computer programs that are constituent parts of EDA systems or tools. Its syntax has been designed for easy machine parsing and is similar to LISP.

The ispLEVER software supports EDIF Version 2 0 0.

All Lattice devices support EDIF design entry.

Importing an EDIF Netlist

You can import a design netlist description into the ispLEVER software from a third-party synthesis or schematic tool if the design file is formatted as EDIF 2 0 0.

***Note**: The project that you are importing the netlist into must be an EDIF project.*

To import an EDIF netlist into your project:

1. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
2. Change Files of type to **EDIF Netlist** (`*.ed*`), and then select the EDIF file that you want to import.
3. Click **Open** to open the Import EDIF dialog box.
4. The default setting for power and ground in the ispLEVER software are the VCC and GND symbols. If you know that the EDIF generated by other tools uses a different convention, you can change it in the window. Select **Custom**. Select either **Symbol** or **Net** representation. Then type the new names for VCC and GND.
5. If you are following the recommendation from Lattice for generating the EDIF file from the supported third party design kit, select **CAE Vendors**. From the list, choose the vendor that generated the EDIF file: Mentor, Synopsys, Synplicity, or Viewlogic.
6. Click **OK**. The software adds the selected EDIF file to the project sources.

Translating EDIF Properties

By default, the ispLEVER software ignores EDIF properties. If you want the ispLEVER software to translate EDIF properties to design constraints for the Fitter, do the following:

1. In Project Navigator, choose **Tools > Import Source Constraint Option** to open the dialog box. The Import Source Constraints Option dialog box lets you import constraints, such as Location (pin/node) Assignments, Group Assignments, and Output Slew Rate, from source files (ABEL, schematic, or EDIF).
2. In the dialog box, select **Auto Import Source Constraints**.
3. Click **OK**.

When you select this option, the ispLEVER software displays a confirmation dialog box prior to implementing the function. This confirmation dialog box appears every time you run the Fit Design process, unless you select the Do Not Import Source Constraints option.

On the warning message dialog box, if you click **Yes**, the constraints from the source files are written into the project constraint file.

***Important**: Constraints from source files and existing constraints in the project constraint file are not merged; existing constraints are overridden by the new constraints.*

Existing constraints (only Location Assignments, Group Assignments, and Output Slew Rate are affected) in the project are cleared, regardless of constraints that might exist in the source file. If there are constraints in the source file, the new constraints are written into the project constraint file. If there are no constraints in the source file, no constraints are written into the file.

EDIF Properties

The ispLEVER software will take design-specific constraints from the properties in the EDIF netlist. The following is the list of properties that the Fitter supports.

PIN LOCATION Property

Name: LOC

Value: {PIN # }

Example: LOC = P20

Scope: IO PORT, net connect to the IO port.

GROUPING Property

Name: GROUPING

Value: GROUP NAME

Example: Use the following command to assign signal locations in your design. In this case, you have a list of internal nodes: a, b, and c, and you want to assign them into a group “mg.” The location of this group needs to be Block “A”, Segment “2”:

On net a, grouping = mg

On net a, loc = "A, 2"

On net b, grouping = mg

On net b, loc = "A, 2"

On net c, grouping = mg

On net c, loc = "A, 2"

OUTPUT SLEW Property

Name: SLEW

Value: {Fast, Slow}

Example: To set port A to high slew, put the following property on the net or port: SLEW=Fast

Scope: OUTPUT PORT/NET

SIGNAL OPTIMIZATION Property

Name: OPT

Value: {KEEP, COLLAPSE}

Scope: On any net of the design.

OPEN DRAIN Property

Name: OPENDRAIN

Value: {On/Off}

Example: To set port A to an open drain, put the following property on the net or port: OPENDRAIN=on

Scope: OUTPUT PORT/NET

PULL Property

Name: PULL

Value: {On/Off/Hold}

Example: To set port A to pull up, put the following property on the net or port: PULL=on

Scope: OUTPUT PORT/NET.

OUTPUT VOLTAGE Property

Name: VOLTAGE

Value: {VCC/VCCIO}

Example: To set port A to output voltage at VCCIO level, put the following property on the net or port: VOLTAGE=VCCIO

Scope: OUTPUT PORT/NET.

Schematic Design Entry

The schematic design entry environment is a set of tools that allows you to capture the structure of a design as either a flat description or a hierarchical set of components, and the connectivity between these components. Then you can use this description to drive the Fitter and verification tools. Designs can be single-level (flat) or multi-level (hierarchical). Schematics can be drawn on multiple “sheets” and be of any size.

The Schematic Editor works in conjunction with the Hierarchy Navigator, Symbol Editor, and Library Manager programs.

Devices that support schematic entry include:

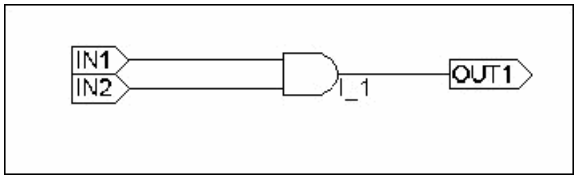
- GAL
- ispLSI 1K, 2K, and 8K
- ispLSI 5000
- ispMACH 4000
- ispMACH 5000
- MACH 4 and MACH 5
- ispXPLD 5000MX

Schematic Overview

Unless you're using them just for documentation, schematics are actually the starting point of the development process, not the goal. The schematic will eventually be used to analyze the device's behavior using the Functional Simulator and the Waveform Viewer.

The ispLEVER software includes a Schematic Editor for creating and editing schematic sources. The figure below is an example schematic that represents an AND gate.

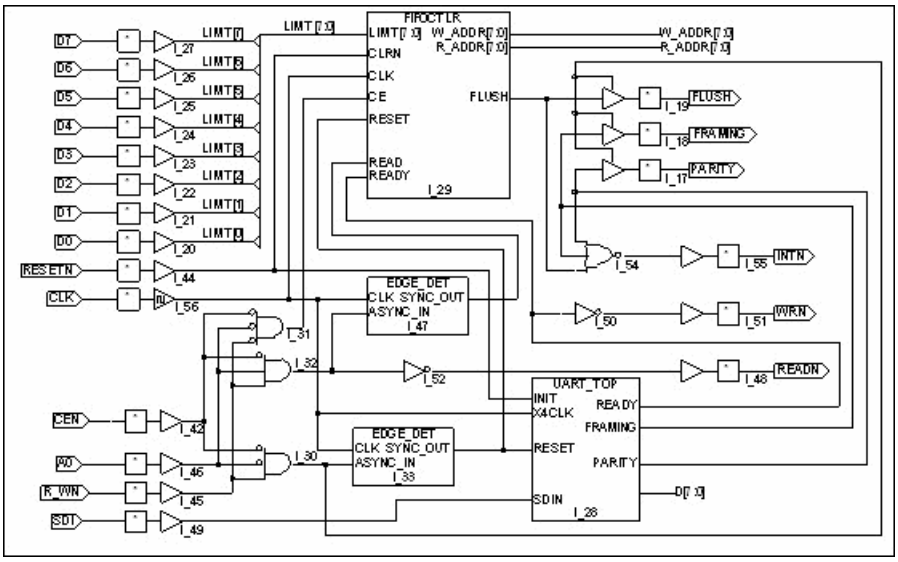
Schematic of an AND gate



In the above schematic there are two inputs (labeled **IN1** and **IN2**) and one output (**OUT1**). The function of **I_1** is to combine the signals **IN1** and **IN2**.

The following figure is a much more complex schematic that represents a programmable IC.

Complex schematic of a programmable IC



What is a Schematic?

The schematic file (saved as a `.sch` file) describes your circuit in terms of the components used and how they connect to each other. The schematic can be used to create netlists for the ispLEVER Fitter.

A schematic can represent a simple logic process (such as an AND gate) or a more complex component in your design (such as a Register). It can also represent the top-level of your design.

What do Schematics Consist of?

A schematic is composed of the following items:

Symbols — These can be symbols from the standard Symbol libraries, symbols representing other schematics you have drawn (Block symbols), or symbols you have created from scratch.

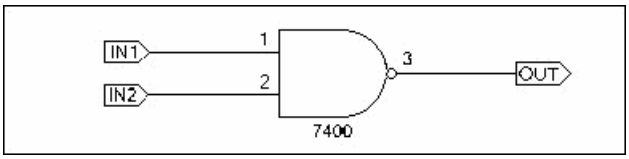
Wires — Wires connect the symbols. They can be single-signal (nets) or multiple-signal (buses).

I/O Markers — I/O markers show where signals enter or exit the schematic and the direction (polarity) of the signal. That is, whether it is an input, output, or bi-directional signal.

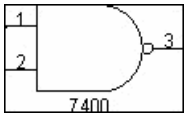
Graphics & Text — Graphics and text are usually added to display explanatory data. They are optional and have no electrical meaning.

A valid schematic must contain at least the first three components—*symbols, wires,* and *I/O markers*. For instance, a single, isolated component symbol cannot be the only element in a schematic. The schematic must include I/O markers for the external connections to the schematic, and these markers must be connected to the symbol with wires.

A Valid Schematic



Invalid Schematic (no wires or I/O markers)



Symbols

Symbols are graphic representations of components. The term “symbol” usually refers to an electrical symbol, such as a gate or a sub-circuit. You can draw graphic-only symbols (such as title blocks) with the Symbol Editor, but these have no electrical meaning.

Symbols are the most basic elements of a schematic. Symbols represent primitive design elements, whether those elements are complete gates or a complex macro. A symbol can also be the hierarchical representation of a sub-circuit, or a “Block” symbol.

Symbol Information

Each schematic symbol is a file ending with a `.sym` extension, and may be included in a library file with a `.lib` extension. The symbol file contains four types of information: *graphics*, *text*, *pins*, and *attributes*.

Graphics and Text

Graphics are pictures of the symbols. Symbol graphics have no electrical meaning and show only the position of the component in the circuit. A symbol’s attributes and pins, not the graphics that represent it, define the electrical behavior of a symbol. Explanatory or descriptive text displayed with a symbol is also considered “graphic” information without electrical meaning.

Pins

Symbol pins are the connecting points between the symbol and the schematic wiring. If the symbol represents an individual component, the symbol pin represents the physical pin where a conductor can be attached. If the symbol represents a subcircuit (block symbol), the symbol pin represents a connection to an internal net of the subcircuit.

Attributes

Each symbol has a number of predefined attributes that describe its component type and other unchanging characteristics. (These were discussed briefly in the preceding section, “Symbols.”). Other attributes can be given values after the symbol is placed in the schematic.

Wires

Wires are the lines that electrically connect the symbol pins. Symbol pins are the only connection points for wires. You cannot connect wires to the symbol body itself.

There are two types of wires: single-wire *nets* and multiple-wire *buses*. Buses allow more than one signal to be routed as single line.

Wires (both nets and buses) are added to schematics using the **Add > Wire** or the **Add > Net Name** commands of the Schematic Editor.

***Note**: There is only one kind of wire you can add (using the **Add > Wire** command). Whether it is a net or a bus depends on how you name the wire (using the **Add > Net Name** command). For instance, Buses are named as* `busname[numberlist]` *where* busname *is the name of the bus and* numberlist *is a list of numbers separated by commas representing each net in the bus. You can also draw and name a net or bus (you do not have to use the **Add > Wire** command) by choosing **Add > Net Name** and dragging on the schematic to draw the wire.*

Wire Names

Wires have names. These names identify the wires to the Schematic Editor and netlister programs.

You would normally name all wires that connect to inputs or outputs and any "internal" nets with signals you want to view during simulation. You can use any name you like, but you usually choose a name that suggests the name or function of the signal carried by that wire. If you don't give a wire a name, the Schematic Editor automatically supplies one, of the form N_*n* (where *n* is an integer).

Giving a single wire a compound name creates multi-wire buses. You can then tap off any signal you want anywhere along the bus.

Buses are most often used to group related signals, such as a 16-bit data path. However, a bus can be any combination of signals, related or not. Buses are especially useful when you need to route a large number of signals from one side of the schematic to the other.

Buses also make it possible for a single I/O marker to connect more than one signal to a Block symbol. The signal names don't have to match, but both pins must carry the same number of signals.

Net Attributes

Like symbols and symbol pins, nets (the wiring that connects symbols to each other and makes external connections) can also have attributes. These attributes include the net's name, as assigned in the schematic. There are also net attributes that pass parameters to other programs, such as simulators.

I/O Markers

I/O markers mark the points at which signals leave or enter the schematic. They are required. Any unconnected wire without an I/O marker will eventually be flagged as an error when you try to create a netlist, run the simulator, or load the design into the Hierarchy Navigator.

I/O markers are added to schematics using the **Add > I/O Marker** command in the Schematic Editor.

The I/O marker automatically takes the name of the wire it is attached to. If the wire is a bus, the marker will have the same compound name as the bus.

When a Block symbol and its matching schematic are created, the I/O markers for the signals that enter and leave the schematic must have the same names as the corresponding pins on the Block symbol. The matching names identify which signal attaches to which pin.

Graphics

Although symbols, wires, and I/O markers are visible, graphical items, they also have a functional or electrical meaning. In this context, "graphics" refers to the non-functional graphical parts of the schematic.

For example, you might add graphics showing the expected waveforms at different points in the circuit. Or, you could draw the company's logo and add it to each schematic for identification.

The most common use of graphics is to create a title block. The block shows the name and address of your company, and can include the company logo and blank spaces for the project name, schematic sheet number, and so on.

The title block is a symbol (usually called **Title** and located in the project directory, in the `..\isptools\ispcpld\sym_libs\`**`title.lib`** library, or as the `..\isptools\ispcpld\generic\generic\misc\`**`title.sym`** file.). You can modify this symbol in the Symbol Editor and save the file (with the same name) in the project directory, or you can use the Library Manager to add the revised symbol to the `misc.lib` symbol library file.

Text

Text, like graphics, can provide additional information about the schematic or its project. Text can be placed anywhere on a schematic, even if it overlaps symbols or wires.

Text is added to schematics using the Schematic Editor: Add > Text command.

Naming Schematic and Symbol Files

When you name schematic or symbol files, observe the following rules:

- Observe DOS file naming conventions. The dollar sign (**$**) cannot be the first letter of a file name, however.
- Don't type a file name extension; the Schematic Editor will add it automatically. If you enter a non-standard extension, the Schematic Editor will replace it with the correct extension (`.sch` for schematics, `.sym` for symbols, and `.lib` for symbol libraries).

Schematic Attributes

You use attributes to describe the characteristics or properties belonging to, or associated with, a **symbol**, **pin**, or **net**. Attributes only apply to describing characteristics in schematics. ABEL-HDL source files have their own syntax for describing characteristics.

***Note**: See each device family application note for a list of supported attributes.*

Attribute Use

Attributes are primarily used to control certain aspects of design optimization and signal placement. You can also use attributes to control how the Fitter implements your design.

Attribute Types

There are two types of attributes used in the Schematic Editor: *symbol* and *net*.

- **Symbol** attributes describe features related to a whole symbol. Symbol attributes usually apply only to the symbol on which they appear.
- **Net** attributes describe characteristics associated with nets.

Attribute Components

Every attribute consists of four components*: name, value, modifier,* and *window*. The following is a brief discussion of attribute components as they apply to programmable IC design.

Attribute Name

An attribute's name identifies the attribute to the user. Width, Length, Refname, and PinNumber are examples of attribute names. Consult your **Vendor Kit** (Device Kit or Interface Kit) documentation for the names and descriptions of any attributes you need to use.

Attribute Value

An attribute can be assigned a *value*. A value is usually a number or a text string.

Attribute Modifier

An attribute modifier specifies the conditions under which an attribute's value can be modified. The attribute modifiers are grouped based on where you can edit their values:

- Anywhere in Design (<blank>)
- Not Editable (!)
- Symbol Only (-)
- Symbol or Schematic ($)
- Derived (*)

Attribute Window

Attribute values are displayed in attribute windows. Attribute values cannot be displayed unless the symbol has at least one attribute window.

You add attribute windows to a symbol when you define the symbol. Each window is assigned a unique number and the default attribute that will be displayed in that window. (The window number *does not* have to match the number of the assigned attribute.) When the symbol is placed in a schematic, the value of the assigned attribute appears in the window.

You can temporarily change which attribute is displayed in an attribute window, using the Attribute Display command from the Options menu. This is useful when you need to view attributes that are not currently displayed.

Attribute windows in schematics can be repositioned, one at a time, with the Attribute Location command on the Edit menu. Repositioning can make a crowded schematic more readable.

***Note**: Attribute windows do not have visible outlines. Rather, they are predefined areas on or near the symbol.*

Setting Attribute Values

The Schematic Editor passes attribute information to the Fitter. For those attributes that can be edited, you can set or override the values as follows:

- You can set **symbol** and **pin attribute** values for all occurrences of a symbol in the Symbol Editor.
- You can override the attribute values in any schematic where the symbol appears by using the Schematic Editor.
- You can override the attribute values in any design where the symbol appears by using the Hierarchy Navigator.

Default Values

Attribute values in a symbol definition become the default values for each symbol instance. These values are frequently overridden in the completed design, usually to optimize its performance.

When you are in the Symbol Editor, any attribute values you set in the Symbol Editor will be used.

When you are in the Schematic Editor, any attribute values you set in the Schematic Editor will be used. If you did not specify attributes for the symbol in the Schematic Editor, the Schematic Editor will use the values set for the symbol in the Symbol Editor.

When you are in the Hierarchy Navigator, any attribute values you set in the Hierarchy Navigator will be used. If you did not specify attributes for the symbol in the Hierarchy Navigator, the Hierarchy Navigator will use the values set for the symbol in the Schematic Editor. If you did not specify attributes for the symbol in the Hierarchy Navigator or the Schematic Editor, the Hierarchy Navigator will use the values set for the symbol in the Symbol Editor. (Note that most netlisters use the values you set in the Hierarchy Navigator.)

Attributes that apply to all instances of a symbol (such as the vendor part number and the pin polarity) are generally assigned values when the symbol is created. Attributes that apply to a single instance (such as the instance name) are assigned after a symbol has been placed in the design.

The symbol libraries supplied with the ispLEVER software have predefined values for all the attributes required by most simulators and netlisters.

Displaying Attribute Values on a Schematic

Attribute values are displayed in attribute windows. Before an attribute can be displayed on a schematic, an attribute window number must be assigned to the attribute in the Symbol Editor.

You can add an attribute window number to any value to display it. The attribute window number does not have to match the attribute number.

You can assign one attribute window number to several attributes. The attribute with the lowest attribute number *that has a value assigned* is displayed.

Mixed-Mode Design Entry

The ispLEVER software supports mixed-mode design entry: a design with at least one schematic module as the top project source, and one or more sources of the same language. The language sources are mutually exclusive, so you must choose one of the three types when you begin a new project. For example, a schematic and an ABEL-HDL source, a Verilog HDL source, or a VHDL source.

Hierarchical Design

What is a Hierarchical Design?

A design with more than one level is called hierarchical. A single-level design is referred to as being flat. Converting a section of circuitry to a block makes a flat design hierarchical. This is commonly referred to as “hierarchical” design.

The ispLEVER software supports full hierarchical design. Hierarchical structuring permits a design to be broken into multiple levels, either to clarify its function or to permit the reuse of functional blocks. For instance, a large, complex design does not have to be created as a single module. By using a hierarchical design, each component or piece of a complex design can be created as a separate module.

A design is hierarchical when it is broken up into functional blocks, or modules. For example, you could create a top-level schematic describing an integrated circuit. In the schematic, you could place a Block symbol (a Block symbol represents a functional block) that provides a specific function of the chip. You can then elaborate the underlying logic for the Block symbol as a separate schematic or as a separate HDL module.

The module represented by the Block symbol is said to be at one level below the schematic in which the symbol appears. Or, the schematic is at one level above the Block’s module. Regardless of how you refer to the levels, any design with more than one level is called a hierarchical design.

Advantages of Hierarchical Design

The most obvious advantage of hierarchical design is that it encourages modularity. A careful choice of the circuitry that composes your module will give you a Block symbol that can be reused.

Another advantage of hierarchical design is the way it lets you organize your design into useful levels of abstraction and detail. For example, you can begin a project by drawing a top-level schematic that consists of nothing but Block symbols and their interconnections. This schematic shows how the project is organized but does not display the details of the modules (Block symbols).

You then draw the schematic for each Block symbol. These schematics can also contain Block symbols for which you have not yet drawn schematics. This process of decomposition can be repeated as often as required until all components of the design have been fully described as schematics.

Breaking the schematic into modules adds a level of abstraction that lets you focus on the functions (and their interaction) rather than on the device that implements them. At the same time, you are free to view or modify an individual module.

Although there are many ways of "breaking apart" a complex design, some may be better than others. In general:

- Each module should have a clearly defined purpose or function and a well-defined interface.
- Look for functions or component groupings that can be reused in other projects.
- The way in which a design is divided into modules should clarify the structure of the project, not obscure it.

Hierarchy vs. Sheets

Creating a hierarchical design is not the same as creating a schematic with multiple sheets. In a schematic, you can add as many sheets as desired to extend beyond the original sheet. However, regardless of how many sheets you add, all the components of the design are still at a single level; all sheets are still contained in the same module.

Approaches to Hierarchical Design

Hierarchical designs consist of one top-level module. This module can be of any format, such as ABEL-HDL, VHDL, Verilog HDL, schematic, or EDIF netlist. Lower-level modules can be of any supported sources and are represented in the top-level module by functional blocks or other "place-holders."

Following are some rules you need to follow when creating a hierarchical design in ispLEVER.

- The top-level source can be of any format, such as ABEL-HDL, VHDL, Verilog HDL, schematic, or EDIF netlist.
- For hierarchical Schematic/ABEL designs:
  - If the upper-level source is a schematic file, the lower-level source can be either a schematic or an ABEL-HDL file.
  - If the upper-level source is an ABEL-HDL file, the lower-level source can be either a schematic or an ABEL-HDL file.
- For hierarchical Schematic/VHDL designs:
  - If the upper-level source is a schematic file, the lower-level source can be either a VHDL file or a schematic file.
  - If the upper-level source is a VHDL file, the lower-level source can only be a VHDL file.

- For hierarchical Schematic/Verilog HDL designs:
  - If the upper-level source is a schematic file, the lower-level source can be either a Verilog HDL file or a schematic file.
  - If the upper-level source is a Verilog HDL file, the lower-level source can only be a Verilog HDL file.
- For EDIF designs:
  - Hierarchical EDIF design is not allowed.

You can create the top-level module first, or create it after creating the lower-level modules. For example, in the Schematic Editor you can create schematic project components in any order and then combine them into a complete design. You can draw a schematic first and create a Block symbol for it afterwards; or you can specify the Block first and create the schematic for it later.

Hierarchical ABEL-HDL Design

You can use ispLEVER to specify a lower-level block symbol in an ABEL-HDL design. Also, you can instantiate a lower-level schematic Block symbol in an ABEL module.

Hierarchical Schematic Design

You can use the ispLEVER software to specify a lower-level block symbol in a schematic, or you can instantiate a lower-level ABEL-HDL block symbol in a schematic.

Hierarchical Verilog HDL Design

You can use the ispLEVER software to specify a lower-level schematic block symbol in a Verilog HDL module, or you can instantiate a lower-level Verilog HDL block symbol in an upper-level Verilog HDL.

Hierarchical VHDL Design

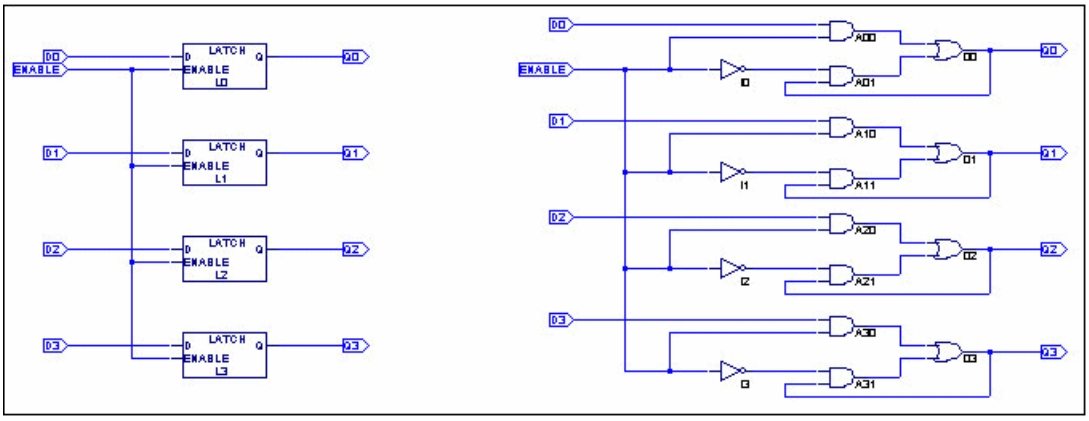
You can use the ispLEVER software to specify a lower-level schematic block symbol in a VHDL module, or you can instantiate a lower-level VHDL block symbol in an upper-level VHDL.

Hierarchical Design Considerations

Apply the following considerations when using hierarchical design techniques. We take a hierarchical schematic design as an example (shown below). The rules implied in the example are also applicable to other types of hierarchical designs.

- Hierarchical Design Structure
- Hierarchical Naming
- Nets in the Hierarchy
- Automatic Aliasing of Nets

Example: REG4 and its Equivalent Circuit



Hierarchical Design Structure

When a symbol is placed in a schematic, the component or subcircuit that the symbol represents is added to the circuit. For example, when you place a latch symbol, you are actually including the OR gate, inverter, and two AND gates from the latch's schematic.

The example shows a 4-bit register (`REG4`) constructed from four `latch` symbols (`latch.sym`). The right side of the figure shows the underlying components. The four latch symbols represent a total of eight AND gates, four OR gates, and four inverters.

This hierarchical building process could be repeated by using the Schematic Editor's **File > Matching Symbol** command or **File > Generate Symbol** command (if the corresponding `.naf` file has been generated) to create a symbol for schematic `reg4`, and then placing the `reg4` symbol in a higher-level schematic. If you created a schematic for a 16-bit register, `reg16`, by placing four copies of symbol `reg4`, you would be defining a circuit with a total of 64 gates. But instead of having to view 64 gates on a single level, you can work with symbols that represent gates, at the appropriate level of detail.

Hierarchical Naming

In the `latch` schematic example, the inverter has the instance name `I1`. In schematic `reg4`, four copies of the symbol `latch` are placed and assigned instance names `L1` through `L4`. Schematic `reg4`, therefore, contains four copies of inverter `I1`.

The Hierarchy Navigator distinguishes among these otherwise identical inverters by combining the inverter's instance name with the instance name of the latch containing it. The four inverters are therefore named (in the Hierarchy Navigator):

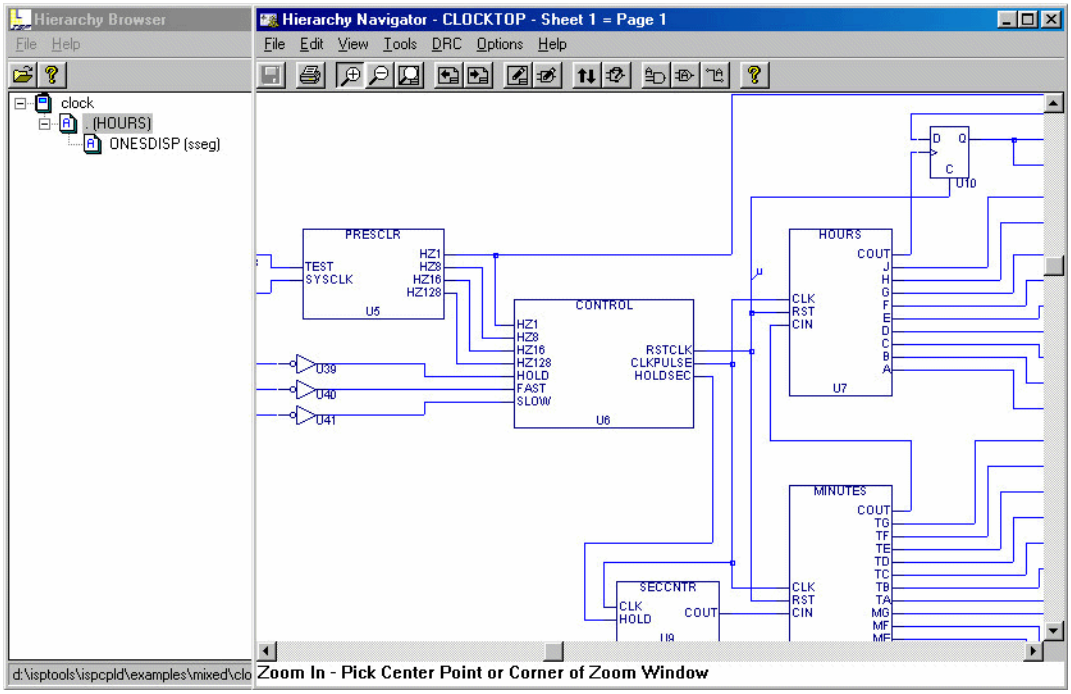
```
L1.I1
L2.I1
L3.I1
L4.I1
```

If we created a 16-bit register by combining four `reg4` symbols, the resulting schematic would represent a new hierarchical level containing four copies of reg4 (named `R1` through `R4`). Each copy of `reg4` contains the four inverters as named above. The Hierarchy Navigator would then name the 16 inverters by combining the instance names of the four `reg4` symbols with each of the four instance names of the inverters as follows:

```
R1.L1.I1, R1.L2.I1, R1.L3.I1, R1.L4.I1
R2.L1.I1, R2.L2.I1, R2.L3.I1, R2.L4.I1
```

```
R3.L1.I1, R3.L2.I1, R3.L3.I1, R3.L4.I1
R4.L1.I1, R4.L2.I1, R4.L3.I1, R4.L4.I1
```

When you view an individual latch schematic in the Schematic Editor, you see the instance names of the gates, without the hierarchical context. When the schematic becomes part of a larger design and is viewed in the Hierarchy Navigator, the instance names include the hierarchical path (as shown above) to assure their uniqueness.



Nets in the Hierarchy

The schematic definition for the latch circuit contains both local and external nets. The output of the inverter is connected to the AND gate with a local net. Two other local nets connect the outputs of the AND gates to the inputs of the OR gate. Assume these nets have been named `N1`, `N2`, and `N3`. When16 copies of this circuit are combined in `reg16`, 16 copies of these local nets are created.

The 16 local nets named `N1` are individual nets, not branches of the same net, so the Hierarchy Navigator creates a unique name for each. The local net name (`N1`) is prefixed with the instance name of the schematic where the net is defined. A dash separates the net and instance names. The 16 `N1`s then become:

```
R1.L1-N1, R1.L2-N1... R4.L3-N1, R4.L4-N1
```

The `latch` schematic contains three external nets, `D`, `ENABLE`, and `Q`. The symbol pins on the latch connect these nets to the hierarchical level mentioned above.

Automatic Aliasing of Nets

When a design is loaded into the Hierarchy Navigator, nets take the name of the highest (top-level) net in the design. That is, the name of top-level net propagates downward through the hierarchy to override the local name. By forcing all nets to the same name, this aliasing feature greatly speeds signal tracing in a multi-level design.

In the preceding example, the net name `D` from the latch is overridden by the higher-level external reference to become `D1`, `D2`, `D3`.... This override becomes the reference at all levels of the hierarchy. If, in the

suggested 16-bit register, the D0, D1, D2... inputs were connected to wires named and marked Bit0, Bit1, ... Bit15, these new names would take precedence and the D0, D1, D2... names would no longer be accessible at any level of the hierarchy.

Hierarchical Design Examples

ABEL-HDL Hierarchy Example

The first example below shows an upper-level ABEL-HDL module (top.abl) that references a lower-level ABEL-HDL module (add.abl). Following that, the example shows a lower-level module implemented as an ABEL-HDL block, while the figure shows the lower-level module implemented as a schematic block (add.sch). Both add.abl and add.sch can be instantiated in the upper-level source top.abl.

Top-level ABEL-HDL Module (top.abl)

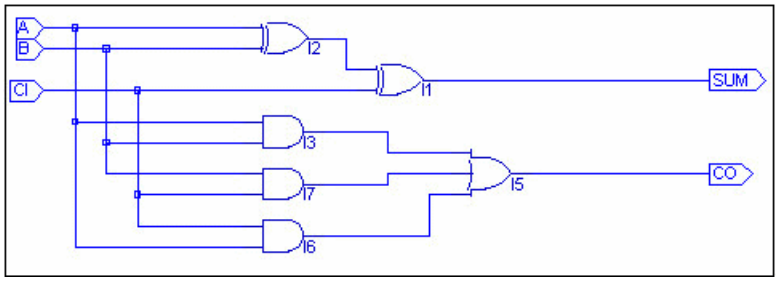
```
MODULE top

"inputs
AIN,BIN,CARRYIN pin;

"outputs
CARRYOUT,SUMOUT pin;

add INTERFACE(A,B,CI -> SUM,CO);

my_add functional_block add;

EQUATIONS
my_add.A = AIN;
my_add.B = BIN;
my_add.CI = CARRYIN;
SUMOUT = my_add.SUM;
CARRYOUT = my_add.CO;

END
```

Lower-level ABEL-HDL module (add.abl)

```
MODULE add

"inputs
A,B,CI pin;

"outputs
CO,SUM pin;

EQUATIONS
SUM = A&B&CI
+!A&!B&CI
+!A&B&!CI
+A&!B&!CI;
```

```
CO = A&B
+A&CI
+B&CI;

END
```

Lower-level Schematic block (add.sch)



***Note**: If you are in a lower-level schematic, you can choose **Add > New Block Symbol** and then click **Use Data From This Block** on the dialog box to automatically create a functional block symbol for the current schematic.*

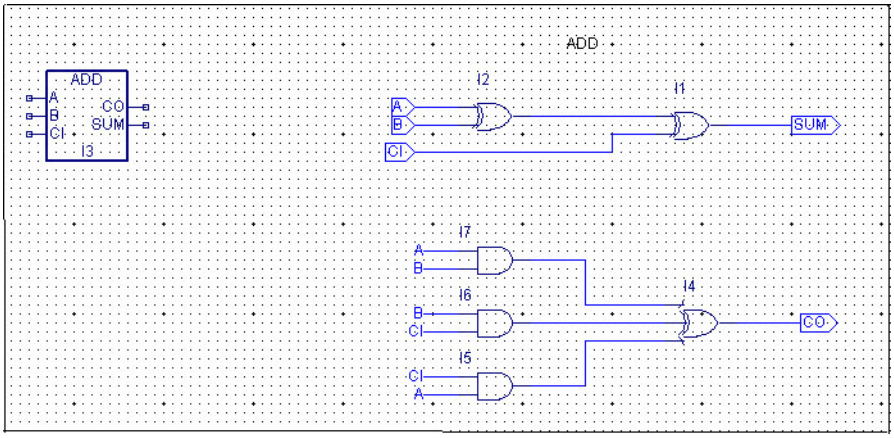
The name of the lower-level schematic must match the block name (schematic) or the interface name (ABEL-HDL) in the upper-level module. This associates the lower-level module with the symbol representing it. For example, the schematic in the Figure must be named `add.sch`.

The net name in the lower-level schematic corresponds to the pin names in the upper-level module that can be either schematic or ABEL-HDL.

Schematic Hierarchy Example

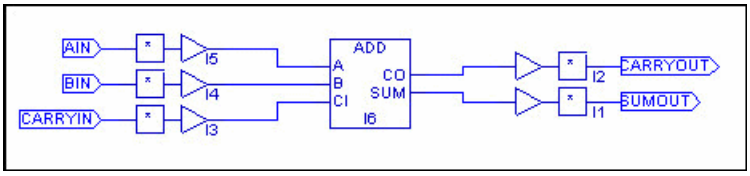
The figure below shows an example of how the new symbol corresponds to an underlying schematic. In this figure, pin A on the Block symbol corresponds to the net in the schematic, which is also named A. The other pins, B, CI (Carry In), CO (Carry Out) and SUM, also correspond to named nets in the schematic.

A Block Symbol and its Underlying Schematic



This following figure shows one top-level schematic and different ways to implement the lower-level modules.

Top-level Schematic for Top (top.sch)



***Note**: If you are in a lower-level schematic, you can choose **Add > New Block Symbol** and then click **Use Data From This Block** on the dialog box to automatically create a functional block symbol for the current schematic.*

The name of the lower-level schematic must match the block name (schematic) or the interface name (ABEL-HDL) in the upper-level module. This associates the lower-level module with the symbol representing it. The above schematic must be named `add.sch`.

The net name in the lower-level schematic corresponds to the pin names in the upper-level module that can be either schematic or ABEL-HDL.

The symbol should be a Block symbol.

Lower-level ABEL-HDL Module for Add Block Symbol

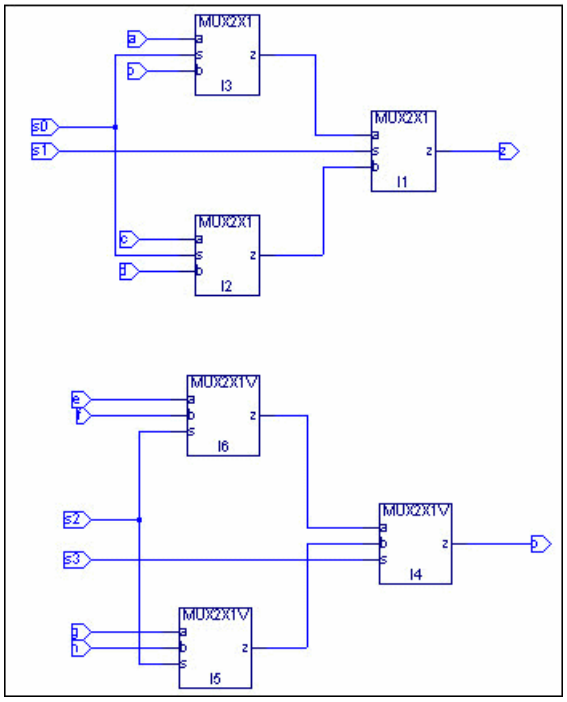
```
MODULE add

"inputs
A,B,CI pin;

"outputs
CO,SUM pin;

EQUATIONS
SUM = A&B&CI
+!A&!B&CI
+!A&B&!CI
+A&!B&!CI;

CO = A&B
+A&CI
+B&CI;
END
```

***Note**: It is best to create the lowest-level sources first and then import or create the higher-level sources.*

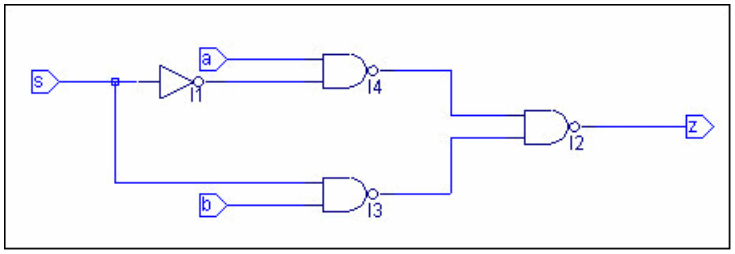
Schematic/Verilog HDL Hierarchy Example

The first figure shows the upper-level schematic `mux4x1.sch` that references a lower-level schematic and a lower-level Verilog HDL module. The second figure shows the lower-level schematic `mux2x1.sch`, and the file after the second figure shows the lower-level Verilog HDL module `mux2x1v.v`.

Top-level Schematic (mux4x1.sch)



Lower-level Schematic (mux2x1.sch)



Lower-level Verilog HDL (mux2x1v.v)

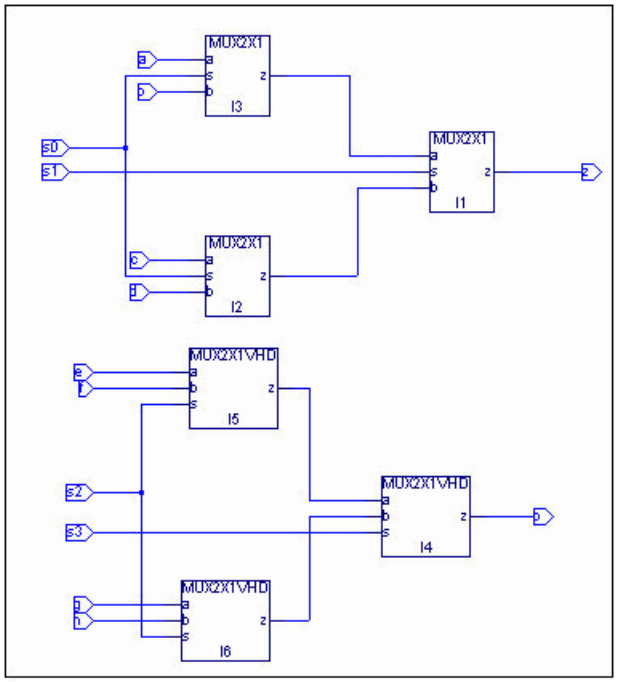
```
module mux2x1v(a,b,s, z);

output z;
input a, b, s;

reg z;

always @(a or b or s)
begin
case (s)
1'b1: z = b;
1'b0: z = a;
default: z = 'bx;
endcase
```

```
end

endmodule
```

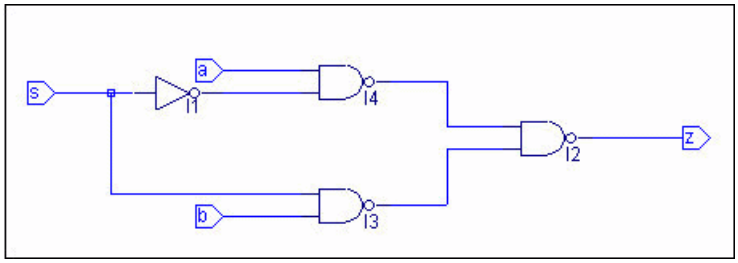
Schematic/VHDL Hierarchy Example

The first figure shows the upper-level schematic `mux4x1.sch` that references a lower-level schematic and a lower-level VHDL module. The second figure shows the lower-level schematic `mux2x1.sch`, and the file after the second figure shows the lower-level VHDL module `mux2x1vhd.vhd`.

Top-level Schematic (mux4x1.sch)



Lower-level Schematic (mux2x1.sch)



Lower-level VHDL Module (mux2x1vhd.vhd)

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

entity mux2x1vhd is
 port ( z: out std_logic;
 a, b, s: in std_logic );
end;
```

```
architecture mux2x1_arch of mux2x1vhd is
begin
process (s, a, b)
begin
case s is
when '0' =>
z <= a;
when '1' =>
z <= b;
when others =>
z <= 'X';
end case;
end process;
end mux2x1_arch;
```

Design Simulation

Running a functional simulation after a design description is complete allows you to verify that the description is functionally correct. Also, by simulating the functionality of your design *before* synthesis, you can find and correct basic design errors sooner. While functional simulation will verify your Boolean equations, it does not indicate timing problems.

The ispLEVER software supports functional simulation for Lattice Semiconductor CPLD and FPGA devices using the Lattice Logic Simulator or *ModelSim* from Mentor Graphics. The RTL design can be simulated for functionality before synthesis using the VHDL or Verilog design description and an input stimulus file.

Simulation Environments

The functional simulators operate in both integrated and stand-alone environments.

Integrated Simulation — To simulate a design inside the current project, the ispLEVER software provides integrated simulation. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the simulation source file, the processes in the tables below are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

For ModelSim, ispLEVER includes scripts that run functional simulation automatically using Lattice-defined settings and preferences. You can customize the run by creating a different script files (DO file), which is a simple script that contain commands that are equivalent to the ModelSim GUI commands. This macro is automatically called when you run ModelSim.

For information about creating your own ModelSim macros, see the *ModelSim User's Manual*, *Chapter 11 Tcl and Macros*, provided with your ispLEVER software (Third-Party Manuals).

CPLD and ispGDX Project Navigator Processes	Simulation Tool Invoked
Functional Simulation	Lattice Logic Simulator
Verilog Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Functional Simulation	ModelSim
Verilog Post-Route Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim
VHDL Post-Route Functional Simulation	ModelSim

Stand-alone Simulation — The ispLEVER software supports stand-alone functional simulation. This provides an easy entry if you need to simulate a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator **Tools** menu.

Design File Descriptions

Lattice Logic Simulator and ModelSim enable you to simulate the operation of your design in the following design entry formats:

- ABEL-HDL format (*design*.abl) — a hierarchical logic description language that supports a variety of behavioral input forms, including high-level equations, state diagrams, and truth tables. (CPLD designs only)
- Schematic format (*design*.sch) — describes your circuit in terms of the components used and how they connect to each other. (CPLD designs only)
- VHDL format (*design*.vhd) — Very High Speed IC Hardware Description Language format.
- Verilog HDL format (*design*.v) — an industry-standard hardware description language used to describe the behavior of hardware that can be implemented directly by logic synthesis tools.

The Lattice Logic Simulator also supports mixed design entry as follows:

- Schematic and ABEL-HDL (CPLD designs only)
- Schematic and VHDL
- Schematic and Verilog HDL

CPLD Test Stimulus Files

Once you have completed your design (or a module of the design), you can test it to confirm that it behaves the way you expect it to. Simulation requires a test stimulus file that specifies the input waveforms.

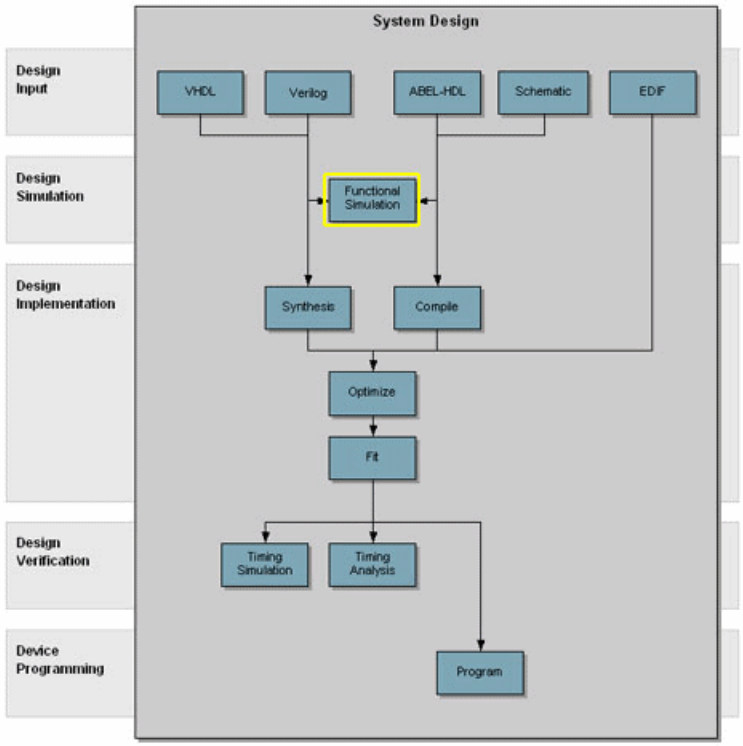
The Lattice Logic Simulator supports CPLD functional simulation for all design entry methods and requires at least one graphic waveform (.wdl) or test vector (.abv) stimulus file.

The ModelSim simulator supports functional simulation for all design entry methods and requires at least one Verilog test fixture (.tf) or VHDL test bench (.vhd) stimulus file.

	Lattice Logic Simulator		ModelSim	
	*.wdl	*.abv	*.tf	*.vhd
ABEL	X	X	X	
Schematic	X	X	X	
Verilog	X	X	X	
VHDL	X	X		X

CPLD Simulation Process Flow

The figure below shows functional simulation within the CPLD process flow.



Creating a Waveform Stimulus File using the Waveform Editor

The Waveform Editor lets you graphically create a test stimulus file by clicking and dragging with the mouse. You see exactly what each waveform will look like, as well as its timing relationship to all the other waveforms. The output file is a waveform display file (`.wdl`). This file can be imported into any CPLD project as a waveform stimulus file and used by the Lattice Logic Simulator for simulation.

See the Waveform Editor Help for specific information about creating waveform stimuilus files

Creating ABEL Test Vectors from a Template

***Note**: ispGDX devices do not accept ABEL source files but will accept an ABEL test vector file.*

ABEL test vectors can be specified either in a top-level ABEL-HDL source or in a separate test vector (`.abv`) file. The ABV file is considered a text document and is kept above the device level in the Sources window. Whether the test vectors are part of a top-level ABEL-HDL source (`.abl`) or are in a separate file, they will be compiled and passed to the simulator.

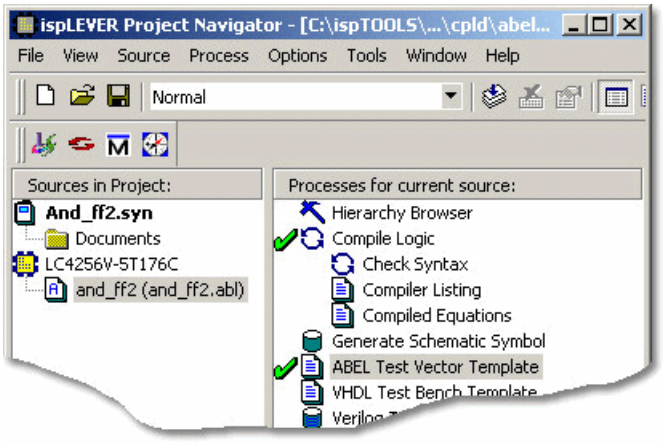
The easiest way to automatically create an ABEL Test Vector template is using the Project Navigator ABEL Test Vector Template process. After the test vector template file (`.abt`) is created, you must add your test vectors and rename it with the extension `.abv` before importing it into your design.

To automatically generate the Verilog test fixture template file and import it into your design:

1. Open your ABEL-HDL design in the Project Navigator.
2. In the Sources window, select the top-level ABEL design source (`*.abl`) file.

   ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

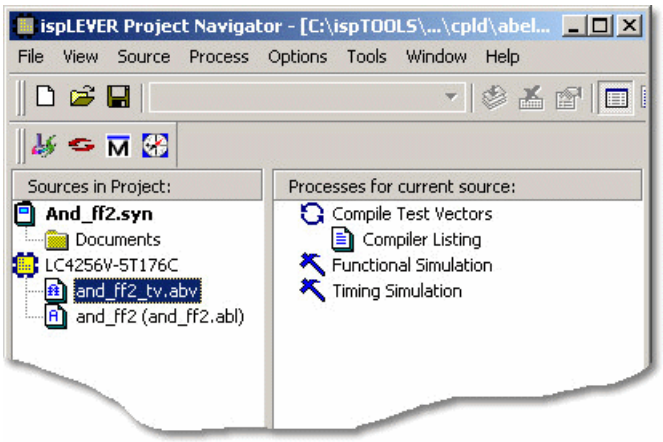
3. In the Processes window, double-click the **Verilog Test Fixture Declarations** process.



   This process creates a template file for an ABEL Test Vector file (`<abel_sourcefile_name>.abt`). However, in order to use this file as a test vector in your design, you must edit it and rename it with the extension `.abv`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `and_ff2_TV.abv`. Add the "`TV`" to the name so that the file will not be overwritten. Change the file extension to "`.abv`" so that it can be imported into the project as a test vector source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test vector file. Click **Open**.

   The file is imported into the project as an ABEL Test Vector. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Creating a Verilog Test Fixture from a Template

Verilog test stimulus can be specified either in the top-level HDL source or in a separate test fixture (`.tf`) file. You can create the test fixture manually using a text editor or use a Verilog Test Fixture template (`.tfi`) file.

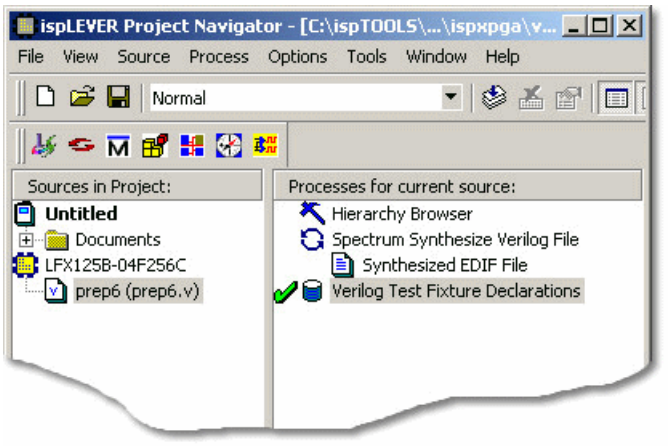
The easiest way to automatically create a Verilog Test Fixture template is using the Project Navigator Verilog Test Fixture Declarations process. After the test fixture template file (`.tfi`) is created, you must add your test vectors and rename it with the extension `.tf` before importing it into your design.

To automatically generate the Verilog test fixture template file and import it into your design:

1. Open your Verilog design in the Project Navigator.
2. In the Sources window, select the top-level Verilog design source (`*.v`) file.

   ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

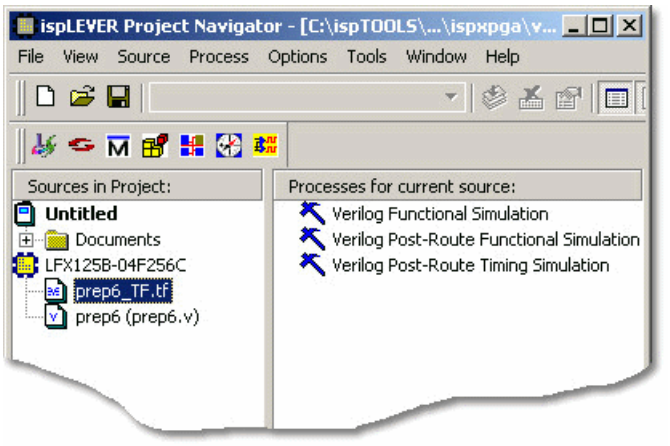
3. In the Processes window, double-click the **Verilog Test Fixture Declarations** process.



   This process creates a template file for a Verilog Test fixture (`<verilog_sourcefile_name>.tfi`). However, in order to use this file as a test fixture in your design, you must edit it and rename it with the extension `.tf`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `prep6_TF.tf`. Add the "`TF`" to the name so that the file will not be overwritten. Change the file extension to "`.tf`" so that it can be imported into the project as a test fixture source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test fixture file. Click **Open**.
7. In the Associate Verilog Test Fixture dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a Verilog Test Fixture. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Creating a VHDL Test Bench from a Template

VHDL test stimulus can be specified either in the top-level HDL source or in a separate test bench (`.vhd`) file. You can create the test bench manually using a text editor or use a VHDL Test Bench template (`.vht`) file.

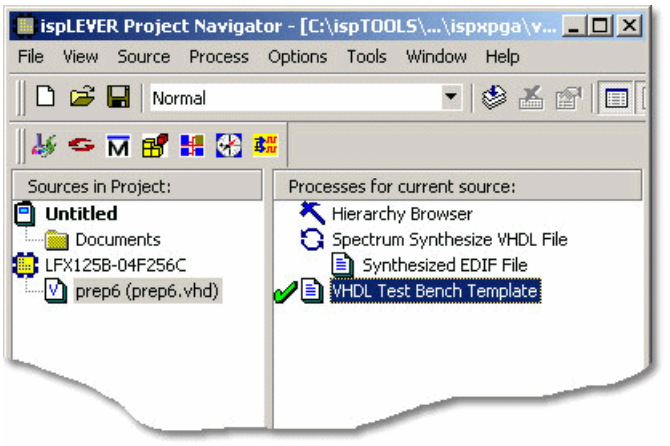
The easiest way to automatically create a VHDL Test Bench template is using the Project Navigator VHDL Test Bench Template process. After the test bench template file is created, you must add your test stimulus and rename it with the extension `.vhd` before importing it into your design.

To generate the VHDL test bench template and import it into your design:

1. Open your VHDL design in the Project Navigator.
2. In the Sources window, select the top-level VHDL design source (`*.vhd`) file.

   ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

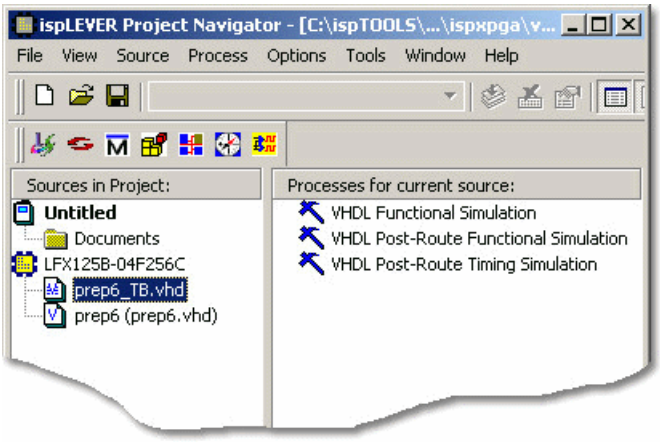
3. In the Processes window, double-click the **VHDL Test Bench Template** process.



This process creates a template file for a VHDL Test Bench (<*vhdl_sourcefile_name*>.vht). However, to use this file as a test bench in your design, you must edit it and rename it with the extension .vhd.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example prep6_TB.vhd. Add the "TB" to the name so that the file will not be overwritten. Change the file extension to ".vhd" so that it can be imported into the project as a test bench source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test bench file. Click **Open**.
7. In the Import Source Type dialog box, select **VHDL Test Bench** and click **OK**.
8. In the Associate VHDL Test Bench dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a VHDL Test Bench. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Interfacing with ModelSim

ModelSim functional simulation generates several batch files, such as .fdo, .udo, and .tdo. ModelSim users will frequently take advantage of customizing batch files to control their simulation, for example specifying signals to display, run time, and waveform display options.

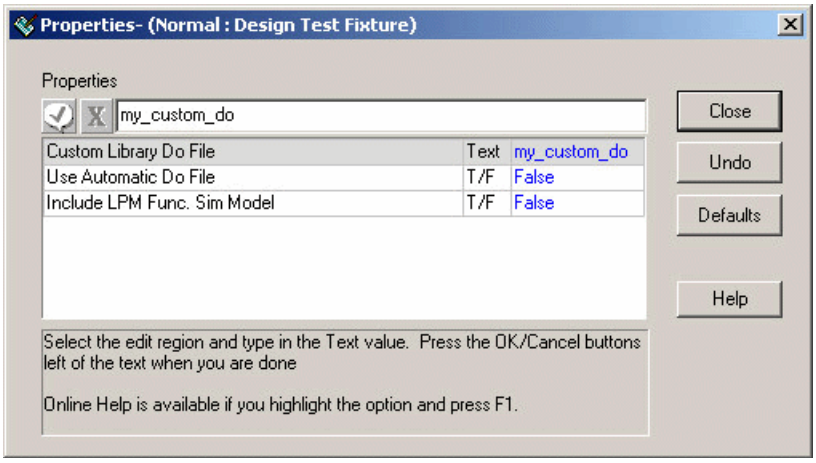
There are two options for creating custom DO files:

Option 1

- In the project folder, edit the *.udo file. The Project Navigator will not overwrite this user DO file.

Option 2

1. In the Project Navigator Sources window, select the test bench source.
2. In the Processes window, right-click the functional simulation process to open the Properties dialog box.
3. In the dialog:
   - Type a name for the file in the Custom Library Do File field and click the checkmark icon
   - Set Use Automatic Do File to **False**.
   - Click **Close**.



4. The software will automatically create this file when functional simulation is run. Edit this file as needed.

Design Implementation

Synthesizing

For Verilog and VHDL designs, the ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity *Synplify* and Mentor Graphics *LeonardoSpectrum*. You can synthesize your Verilog or VHDL design as a stand-alone process by choosing the synthesis tool from the Lattice Semiconductor program group in your Start menu, or you can synthesize automatically and seamlessly within the Project Navigator.

For Verilog and VHDL mixed-mode designs, the HDL portions are synthesized using the selected synthesis tool. For ABEL mixed-mode designs, both the ABEL and schematic portions are compiled. You can only do mixed-mode design using the Project Navigator integrated flow.

The ispLEVER Tutorials contain synthesis design flow tutorials and is the best place to start if you want to get some hands-on experience. By working through the tutorial lessons, you'll learn how to create sample design projects with some of ispLEVER's most useful and powerful features.

For additional information about synthesis using LeonardoSpectrum or Synplify, see the ispLEVER Third-Party Manuals.

Synthesis Design Flows

You can run the synthesis tools from within the integrated ispLEVER environment, or as a stand-alone process. Whether using LeonardoSpectrum or Synplify, the high-level design flows are basically the same.

Integrated Flow

This approach lets you create, synthesize, import, and implement a design targeting one of the Lattice devices completely from within the ispLEVER Project Navigator environment.

1. Using the Project Navigator, create a new HDL project.
2. Target a device.
3. Using the Text Editor, create the HDL modules.
4. For mixed-mode designs, use the Schematic Editor to create the schematic files.
5. Using the Project Navigator, import the source files.
6. Select a synthesis tool.
7. Fit (Place and Route) the design.

Stand-alone Flow

The stand-alone approach requires you to create or load a VHDL or Verilog HDL design into the synthesis tool environment. Then you synthesize the design and generate an EDIF netlist that you imported into ispLEVER for implementing into a Lattice device.

1. Using your synthesis tool, create a project.
2. Target a device.
3. Load the source files.
4. Synthesize the design to create an EDIF file.
5. Using the ispLEVER Project Navigator, create an EDIF project.
6. Target a device (same as step 2).
7. Import the EDIF source file.

8. Fit (Place and Route) the design.

Integrated Third-Party Tools

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment.

LeonardoSpectrum

Within the LeonardoSpectrum synthesis environment, you can create Lattice device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process.

LeonardoSpectrum from Mentor Graphics combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features.

Synplify

The Synplify solution from Synplicity is a high-performance, sophisticated logic synthesis engine that utilizes proprietary technology to deliver fast, highly efficient FPGA and CPLD designs. Synplify uses Verilog and VHDL Hardware Description Languages as input, and outputs an optimized netlist for the Lattice device.

Selecting the Synthesis Tool

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment. This integrated approach lets you create, synthesize, import, and implement a design targeting a Lattice device completely from within the ispLEVER Project Navigator environment.

To specify the synthesis tool that the ispLEVER software will use:

1. In the Project Navigator, choose **Options > Select RTL Synthesis** to open the dialog box.
2. Select the synthesis tool that you want to use. This tool will be associated with all devices in the current device family. You can also make a synthesis tool the default for all device families.

Setting Constraints

Setting and Editing Constraints

For ispGDX, CPLD, ispXPLD, and ispXPGA devices, the ispLEVER software supports setting and editing of constraints in these ways:

Constraint Editor

Many ispGDX, CPLD, ispXPLD, and ispXPGA constraints can be edited within the Constraint Editor. You can specify pin and node assignments, group assignments, resource reservations, power level settings, output slew-rates, and nodal constraints, as well as PLL and HSI attributes. Modifications to the constraint file are made via the function dialog boxes or directly in the appropriate spreadsheets.

To run the Constraint Editor:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window, double-click the **Constraint Editor** process.

   Any invalid attribute or incorrect assignment is displayed in red in the Constraint Editor constraint sheet as well as in all the dialog boxes of the Constraint Editor. All the default values are displayed in blue. However, you can change the system default colors by choosing View > Set Colors.

Optimization Constraint Editor

For most CPLD and ispXPLD devices, the Optimization Constraint Editor lets you specify the global constraints used in optimization. It reads the constraint file and displays the constraint settings in the Opt Global Constraints sheet. You can directly modify the optimization constraints in the sheet.

To edit global optimization constraints using the Optimization Constraint Editor:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window, double-click the **Optimization Constraint** process. The software runs the process and opens the Optimization Constraint Editor.
3. In the Opt Global Constraints sheet, double-click a table cell in the **Constraint Value** column.
4. Select a desired option from the list, or directly type your setting in the edit box.
5. Choose **File > Save** to save the edits to the project constraint file (`.lci`).

ispXPGA Floorplanner

For ispXPGA designs using the ispXPGA Floorplanner, you have the option of saving one or a combination of constraint types—PFU, group assignments, pin assignments, region assignments—to the design's constraint file (`.lct`) using teh File > Save Constraints command. The software overwrites the current LCT file with the new constraints and clears the checkmarks from the processes displayed in the Project Navigator Processes window. The software will apply the changes to the physical design file (`.ld2` or `.ld3`) the next time you run Pack & Place or Route.

To save constraints in the ispFloorplanner:

1. Choose **File > Save Constraints**.
2. In the Save Constraints dialog box, select **Region**, **Pin Assignments**, and/or **PFU Packing/Placement**, depending on the types of changes you have made.
3. In the File name box, accept the default name and file type and click **Save**.
4. Click **Yes** to confirm that you want to replace the current LCT file. The Floorplanner saves the changes to the constraint file.

Compiling

The ispLEVER software accepts several design entry formats. With the exception of EDIF, all CPLD designs must be either synthesized or compiled before going to the Fitter.

For ABEL-HDL and Schematic designs, the compilation process is an integrated part of the ispLEVER process flow. When you compile a design, you are changing your design entry format into Boolean equations, which serve as input to simulation and device implementation programs. In general, compiling a design involves running every process after design entry. These processes include compiling and optimizing steps that can be performed on a single source or on the entire design.

Keeping Track of Processes

The Project Navigator automatically keeps track of your design's processes for you. For example, it knows which processes should be run for a targeted device, a selected source, or for the entire design. Also, you can choose to run any process step and the Project Navigator will run all other processes required to complete that selected step, but not run further, unnecessary steps.

The Project Navigator lists all processes for a selected source in the Processes window. Device-related processes, such as fitting the design, are shown in the Processes window after you select a target device, and highlight it in the Sources window.

Understanding the Compilation Process

The Project Navigator processes each logic module, schematic file, or EDIF netlist to obtain an intermediate file that can later be linked together before fitting the design into a Lattice device.

There are more processes required to compile a logic source than a schematic source, primarily because logic designs are language-based and are stored in ASCII format. This means that the ispLEVER software must check the language syntax and process equation statements that are within the logic file.

The processing steps required to compile a design are listed below in the order in which they run:

- Compile (for logic, schematic, EDIF, or test vector files)
- Check Syntax (for logic files)
- Compiler Listing (for logic files)
- Compiled Equations (for logic, schematic, and EDIF files)
- Signal Cross Reference (for EDIF files)

Compiling Logic, Schematic, or EDIF

This comprehensive process compiles a logic module, a schematic design file, or an EDIF netlist. Design compilation steps differ between source types, as described below:

Compile Logic (for logic sources)

- Checks for and flags syntax errors
- Converts state diagrams and truth tables into equations
- Expands macros
- Converts equations with sets to equations without sets
- Replaces all operators with equivalent operations using only NOTs, ANDs, ORs and XORs
- ORs together equations that cause multiple assignments to the same identifier
- Performs simple logic reduction
- Translates the equations into the OPEN-ABEL-2.0 file format

Compile Schematic (for schematic sources)

- Compiles the schematic to produce a BLIF format file, including any attributes or properties specified. Schematics compiled this way should use only the Device-Independent symbol library.

***Note**: You should run design rule checking (Schematic Editor: DRC > Consistency Check) before compiling the schematic.*

Compile EDIF (EDIF)

- Compiles the EDIF file to produce a BLIF format file, including any attributes or properties specified.

Check Syntax

Checks the syntax of a logic module. No compilation is run. If there are syntax errors, the errors can be viewed in the Process Log File. If you want to see the errors in a compiler listing report format, use the Compiler Listing process.

Compiler Listing

The Compiler Listing process gives a record of the compilation of your source file. The report shows your logic file by line numbers, with errors and warnings below the line on which they occurred.

Compiled Equations

This process shows the Boolean equations produced by the compiler. The equations are shown in sum-of-products form. Positive and reverse polarity equations are displayed, along with product term and fan-in/fan-out summaries for each signal.

Signal Cross-Reference (EDIF)

This option displays a cross-reference of the old names to the new names (converting long and hierarchical to mangled names). Names that are more than 32 characters long or that contain one or more of the characters '/', '>', or '@' will not be displayed properly.

Compiling Source Files

The Project Navigator Auto-Update feature reprocesses sources when they are needed to perform the process you request.

However, you can compile individual source files by selecting the file in the Sources window, and then double-clicking **Compile Logic** in the Processes window. Alternatively, you can double-click a report in the Processes window, and the software will compile the source automatically.

Optimizing

The default options in the ispLEVER software are set up to achieve the highest possible performance in the smallest possible device, for most designs. You can choose to maximize design flexibility by spreading out logic or exercise tighter control over the fitting process to achieve your design goals.

Each clock signal is evaluated and classified as a global clock or a non-global clock. The Fitter attempts to place all global clock signals at global clock pins (check the log file for the status of all clock signals after optimization). The Fitter assigns all other clock signals to I/O pins and implements them as Product Term clocks, if the architecture supports Product Term clocks. Input pins and nodes that are defined but not referenced (not used by another equation) are discarded from the design during optimization (warning messages are generated).

Design Resources Check

Information about the internal architecture of the specified device is loaded and resource checks are performed on the design. Errors are reported if the design exceeds the device's product term, macrocell, pin, clock, set, reset, or output enable control resources.

Logic Synthesis Options

Logic Synthesis options allow you to control how logic functions are optimized before partitioning takes place.

Boolean Logic Reduction

This option removes redundant product terms from each equation. Unless your equations have redundant logic to prevent problems (for example, in combinatorial functions), you should always leave this option selected.

D/T Synthesis

This option lets the optimizer automatically choose between a D-Type or T-Type register, thereby reducing the product term requirements. In some cases, the speed of the design may improve if only D-Type registers are used in an M4A device. This option should be selected for most designs.

Input Register Optimization

This option allows the Fitter to automatically place single-variable registered functions in input pad registers. This option should be selected for M4A devices unless you are trying to prevent the use of input registers.

XOR Synthesis

This option enables or disables exclusive OR synthesis. When this option is selected, the optimizer synthesizes XOR equations, if this can be achieved in the design. When this option is cleared, the sum-of-product equations will be generated. This option is device-dependent. Default state = Enabled.

Node Collapsing

This option allows the optimizer to collapse intermediate combinatorial nodes into registers and output pins, thus speeding up the design. Unless you have handcrafted each equation in your design, you should leave this option selected. This option should always be selected for designs that have been synthesized or described in low-level combinatorial gates.

Speed

This option collapses all nodes up to the set Product Term limit, globally optimized, without regard for the path.

Area

This option collapses all nodes up to the set Product Term limit, without increasing area cost.

Fmax

This option causes the Logic Optimizer to automatically identify all critical paths between any pair of registers, from clock-pin of one register to data-pin of the other register (or the same register). The Logic Optimizer then attempts to collapse/combine the logic nodes along the critical paths, reduce the logic level, and allow the chip to run at a higher frequency.

Collapsing Max. Product Term

This option lets you control the Fitter optimization process by setting a maximum limit on the number of Product Terms (PT) in each equation. In other words, the Optimizer shapes the equations relative to the set number of PT. For example, if the value is set to 35, the Optimizer stops collapsing equations when it exceeds 35 PT.

This option works the opposite of Splitting Max. Product Term.

Collapsing Max. Input

This option lets you control the Fitter optimization process by setting a maximum limit on the number of inputs in each equation. For example, if the value is set to 32, the Optimizer stops collapsing inputs when it exceeds 32 inputs.

Splitting Max. Product Term

This option lets you control the Fitter optimization process by setting a maximum limit on the number of Product Terms (PT) in each equation. In other words, the Optimizer shapes the equations relative to the set number of PT. For example, if the value is set to 35, the Optimizer splits equations if it has more than 35 PT.

This option works the opposite of Collapsing Max. Product Term.

Example

An M4A-32 design consists of six equations having 12 product terms each, and one equation having 21 product terms. (An M4A-32 macrocell can implement up to 20 product terms without equation splitting.) The Fitter can implement each of the six smaller equations as single-macrocell equations, but the one larger equation must be implemented using two macrocells. In its default mode, the optimizer will split the 21-product term equation into one equation of 20 product terms and one equation of 2 product terms (the extra product term is required to accept feedback from the second macrocell).

Reducing the equation-splitting threshold to 12 will result in less of an imbalance in the number of product terms placed at each macrocell. Each of the original 12 product term equations remains at a single macrocell, while the 21 product term equations is split into two macrocells: one with 12 product terms and one with 10 product terms. Thus, none of the equations are using the maximum capacity of its macrocell, which improves the odds of fitting the design and makes it easier to add logic to the design later.

***Note**: Do not reduce the equation splitting threshold if doing so will cause many equations to be split. If, for instance, the preceding example's six smaller equations had contained 15 product terms each, setting the gate-splitting threshold to 12 would have caused all seven equations to be split, resulting in 16 under-utilized macrocells.*

Example

Consider the following:

- A synchronous registered equation with 22 product terms
- Splitting Max. Product Term field set to 20

The equation will be split into two equations, one with 20 product terms and one with 3 product terms. It will take two passes through the array to implement the new equations.

Setting Logic Synthesis Options

You can set logic synthesis options using the Project Navigator Help: Global Constraints Logic Synthesis Tab.

Utilization Options

Utilization options let you specify the percentage of device resources available during each fitter run. You can choose to reduce the device resources available during the initial fitter run, back annotate the pins, and then increase the available device resources when making design changes.

Reducing available device resources during the fitter run may increase the fitter runtime. For example, when the maximum number of block inputs is set too low (<60%). This condition may cause the Fitter to take a long time grouping (i.e. partitioning) logic equations into blocks because the blocks have fewer available resources.

Logic Grouping

You can use logic grouping to exercise manual control over the Partitioner. Logic grouping lets you manually pack selected portions of your design into the same block while setting up the global optimization options to spread out the rest of the logic.

Logic grouping can also be used to group selected inputs, outputs, and buried logic functions into the same block or segment to achieve performance goals.

In cases where the Partitioner is unable to find a solution, manually grouping a small portion of the design may aid in the fitting process.

If you do attempt manual grouping, try to place logic with common inputs and feedback in the same block. This minimizes the number of signals crossing between blocks, which results in a lower demand for interconnection resources and an increased likelihood of a successful fit.

Fitting

The ispLEVER software has a single user interface with all options preset to deliver the highest possible push-button performance. At the end of a successful fitter run, the ispLEVER software generates a JEDEC file, as well as a fitter report, so that you can see how the ispLEVER software has routed the design and utilized resources on the part.

Performing Multiple Runs

For CPLD, ispXPLD, and ispXPGA devices, you can use the ispEXPLORER to try many combinations of constraints to achieve a fit. This is especially useful for designs that the software cannot fit because of the constraints. The ispEXPLORER produces a spreadsheet summary of results and settings for each run, making it easy to compare one group of settings with another.

The Fitting Process

After you have entered your design, you are ready to run the Fitter. The fitting process consists of four phases. An understanding of each phase can help you choose the best corrective action if the design does not fit, or your performance criteria are not met.

- Initialization
- Optimization
- Partitioning
- Fitting (Placement and Routing)

Initialization

At the beginning of a new project, the ispLEVER software automatically copies a default constraint file from the ispLEVER directory into your project directory. For first-time users, the default settings allow most designs to achieve a First-Time Fit (FTF). For users requiring more control, the default settings can be easily changed to achieve better fitting density or performance.

You can control the contents of the constraint file using the Global Constraints dialog box and the Location Assignments dialog box.

Using the Global Constraints Dialog Box to Control Optimization

Using the Global Constraints dialog box, you can pack your design, spread your design, or use other advanced options such as specifying device utilization levels.

Using the Location Assignments Dialog Box to Pre-assign Pins and Nodes

The Location Assignments dialog box in the Constraint Editor lets you specify pin and block locations, group signals in specific blocks, or even reserve pins for later use.

Assigning Pin and Node Locations

The ispLEVER software lets you pre-assign pin and node locations. You can use the Location Assignment dialog box in the Constraint Editor to assign input and output pins and buried nodes. The Macrocell, Block, and Segment list boxes are context sensitive to the selected device; only applicable features are available.

You can also use the drag and drop feature in the Package View of the Constraint Editor to assign input, output and bi-directional pins.

Pin and Node Pre-Assignment

Pre-assigning pins lets you lay out your board at the same time as you are doing logic design, thus shortening the design cycle. Pre-assigning nodes is usually not required and is not recommended.

Pin Assignment Guidelines

If you want to pre-place signals (not recommended unless pinout configuration is important), follow these guidelines:

- Do not place large equations to macrocells or pins at the beginning or end of a block.
- Signals that share many common inputs should generally be grouped in the same block (the Partitioner does this automatically). Signals that do not share many common inputs should generally be distributed across several blocks to avoid overburdening the switch matrix for a single block.

Large Functions at the End of a Block

The macrocells at the end of a block have access to fewer product terms than other macrocells.

- Cell number 0, the first cell in all devices, can access the product term clusters from adjacent, higher-numbered cells, but it cannot access any lower-numbered cells (cell 0 being the lowest-numbered cell in the block).
- The last cell in a block can access the product term cluster from the adjacent lower-numbered cell, but it cannot access any higher-numbered cells.

If signals have not been assigned to macrocells, the Fitter will find a macrocell replacement for all the signals that satisfy their product term requirements.

Adjacent Macrocell Use

In MACH devices, adjacent macrocells can share clusters. Therefore, with designs having equations that use a high number of product terms, it is a good idea not to place them in adjacent macrocells.

Modifying Assignments

You can use the Location Assignment dialog box of the Constraint Editor to modify the current location assignment. Modifications can also be made directly in the Pin Attributes sheet of the Constraint Editor.

Deleting Assignments

You can delete project assignments via the Constraint Editor. To do this, select the entire row whose existing assignment(s) you want to delete. From the Edit menu, select **Delete Row(s)**. You may also right-click and select **Delete Row(s)**. In cases where you no longer want any of the current assignments, you can delete all of them at the same time.

Ignoring Assignments

There may be times when you want to ignore, but not delete, the current assignments. For example, after you complete a design, you may want to try fitting it into a different device. In this case, the current pin assignments may not be valid for the new device. The ispLEVER software lets you ignore current constraints for the next Fitter run.

Power Control

Using the ispLEVER software, you can control power settings for your device. By default, the device is always set to high power, high speed. However, you can set the device or blocks of the device to low power mode. This setting results in slightly decreased speed, but increased power savings. This is useful for handheld and battery-operated devices.

Slew Rate Control

For the majority of Lattice devices, you can set the slew rate to either Slow or Fast. By default, the slew rate is set to Fast. However, changing it to Slow can result in less board noise.

Optimization

If your design failed to fit, or it did not meet your performance criteria, you can apply optimization procedures your design again.

Partitioning

After optimization, the design is partitioned into individual blocks on the specified device. Partitioning is achieved by assigning logic to specific blocks, based on the following considerations:

- Individual signal pre-placements and Grouping assignments
- A block’s available internal resources (free macro cells, product terms, clock signals, and so forth)
- The switch-matrix interconnect resources available to the block

The Partitioner considers commonality of signals, macro cell requirements, Set/Reset requirements, product-term requirements, and other factors to determine which partition is most likely to succeed in fitting the design. Only partitions that are likely to succeed (according to the Partitioner's rules) are attempted.

Balanced Partitioning

Controlling how the Partitioner works can be very important. There is one important strategy for partitioning and that is called Balanced Partitioning. By selecting the Balanced Partitioning option in the Global Constraints dialog box, you are telling the Partitioner to spread all of the signals among all the blocks in the device, rather than trying to fill a few blocks to their maximum potential.

There are advantages to either side of the strategy. If you turn balanced partitioning on, you can save room in the device for any future functionality you might want to add to existing logic. However, turning balanced partitioning off lets you “pack” as much logic into the minimum number of blocks in the device as possible, leaving some free blocks for future design enhancements.

Place and Route (Fitting)

Placement is the assignment of physical block resources such as I/O pins, XORs, registers, and product-term clusters to logic equations. *Routing* is the assignment of switch-matrix interconnect resources to logic equations, after the logic equations are placed.

Placement

In the placement phase of the fitting process, individual equations are assigned to physical resources, as follows:

- Logic equations that have been pre-assigned to pins are assigned first.
- Buried logic functions are placed in the remaining unused macrocells.
- Inputs are assigned to any available pin. These pins can be dedicated inputs pins, clock/input pins, or I/O pins that correspond to macrocells that are either unused or used to implement buried logic functions.
- Outputs can be assigned to any unused I/O pin.

Spread Placement

Controlling how the Placer works is also important. When you select the Spread Placement option, you are telling the Placer to spread the signals in the block as far out as possible.

Routing

In the routing phase, the Fitter attempts to route input, output, and feedback signals to and from the physical resources assigned in the placement phase. If the Fitter fails to route all signals, it tries another placement. The Fitter continues trying different placements, and different routing attempts within each placement, until a successful fit is found or the time allotted for fitting is exceeded.

Fitter Options

The Global Constraints dialog box lets you set options for the Fitter. Using the Global Constraints dialog box, you can tell the Fitter to pack as much logic into the device as possible, spread the logic across the entire device, or use other advanced options such as specifying device utilization. The following sections describe these options.

Pack Design

The Pack Design option lets you pack as much logic into the device as possible. This option allows you to achieve the highest possible performance in the smallest possible device, for most designs. Each block may be completely filled, leaving less room for any design changes or logic additions.

Spread Design

The Spread Design option spreads all of the logic across the entire device rather than trying to fill each block to its maximum potential. This option allows you to achieve the highest possible performance, while leaving room for any additional functionality that you may want to add in the future. The fitter leaves room to accommodate design changes to existing logic. Because each block may be incompletely filled, the design may *or may not* require a larger device to achieve a successful fit.

Advanced Options

The advanced options let you individually control the partitioning and placement algorithms.

Balance Partitioning

The Balance Partitioning advanced option partitions the design evenly among all the blocks in the device, so each block should have the same amount of resources used. When this option is cleared, the software partitions the design block-by-block, filling up one block at a time. This means that some blocks may be filled up completely, while others may be unused.

Spread Placement

The Spread Placement advanced option places the signals evenly, or spreads them out, among macrocells in the block. Spreading out the placement lets you make minor changes to the existing output and node signals in the block. When this option is cleared, the software assigns design signals to the first available macrocell, making it easier to add new outputs or nodes to a block.

Fitter Effort

The Fitter Effort option is used to instruct the Fitter how much effort to apply to a fit. The Low option enables a faster fitting process, but will be more likely to result in failures to fit when the utilization gets higher. The High option provides the most exhaustive search of the solution space, but takes more time.

Fitter Report Formats

Two Fitter Report formats are available in the ispLEVER software, text and HTML.

- If you select the Fitter Report process associated with the target device, the Fitter Report is opened in the Output Panel of the Project Navigator or in the Report Viewer.

  ***Note**: By default, the ispLEVER software opens Fitter Report in the Output Panel of the Project Navigator. If you want it opened in the Report Viewer, select **Using Report Viewer** in the Log tab of the Environment Options dialog box (Project Navigator: Options > Environment).*

- If you select the HTML Fitter Report process associated with the target device, the Fitter Report is opened with your local Internet Browser.

Formatting the Fitter Report

For CPLD devices, you can select various options that determine the information in the Fitter report using the Fitter Report Options dialog box.

Understanding the Fitter Report

The Fitter Report displays statistics and information on the fitting process of your design, including utilization numbers, pin assignments, etc. The Fitter Report is also written into HTML format to allow user to browse through the report easily.

The Fitter Report is divided into several sections, each briefly described below.

Project Summary

As the name implies, this section summarizes the design. It reports the name and location of the project, and the date it was fitted. This section also reports the targeted device and package, as well as the design source format.

Compilation Times

This section tells you how long it took the Fitter to fit the design in the specified device. The name for each process step is listed, as well as the total elapsed time. Prefit Time consists mainly of run-times of the design compilation and optimization phases. Total Fit Time is the total run-time of the design compilation, optimization, partition, placement and routing phases.

Design Summary

This section reports statistical information about the design, such as the number of Inputs, Outputs, Bidir Signals, Flip-flops, Registered Functions, Product Terms and Reserved Pins. It also points out the number of unique control signals in the design.

Device Resource Summary

This section lists all of the resources available within the device and how much of each resource has been used by the design. It also reports how much of each resource is still available.

GLB Resource Summary

This section lists various GLB (and segment) level resource counts, such as fan-in (or array inputs), I/O pins, input registers, macrocells, logic product terms and product term clusters.

GLB Control Summary

This section lists the totals for all control signals, and how much of each is utilized by individual GLBs.

Optimizer and Fitter Options

This section displays all of the settings that were used to fit and optimize the design. These include things such as Ignoring Constraints to the type of flip-flop synthesis you have chosen. The information in this section is set with the Constraints Options dialog box.

Pinout Listing

This section lists the I/Os and control signals on the device, and how they are assigned.

(Input, Output, Bidir, Buried) Signal List

This section reports information on individual I/Os, such as I/O type, location assignment, the fan-out, and other signal attributes.

Signals Fan-out List

This section lists signal resources and the functions they fan-out to.

GLB (GLB name) Cluster Steering Tables

This section shows information about how functions and inputs are placed in a GLB. It shows how product terms are steered to a macrocell on which a function has been placed. It also contains information about what type of control signal has been used.

GLB (GLB name) Logic Array Fan-in

This section shows how design signals are mapped to individual GLB block inputs.

Product Term Histogram

This section lists and sorts the equations according to the number of product terms they use (in the logic only).

GLB Input Histogram

This section lists and sorts the equations according to their number of inputs, which includes the logic and the ctrl signals, but not the global signals (dedicated routing).

Post-Fit Equations

This section reports the equations in your design, after fitting. It begins with a product term histogram and a GLB input histogram.

Back Annotating Assignments

You can back annotate (write) assignments from the Fitter output to the project constraint file using the Back Annotation tab on the Constraints Options dialog box. This feature lets you retain the assignments made by the Fitter so that they can be used in a future fitting process.

You can back annotate the following location and constraint assignment options:

- Pin Assignments — Only pin assignments are back annotated to the project constraint file. This option lets you retain the fitter pin assignments. All buried nodes assignments are not retained. Existing buried node assignments are removed from the project constraint file.
- Pin and GLB Assignments — Only pin and GLB assignments are back annotated to the project constraint file. The GLB assignments for the back annotated buried nodes are retained, but the associated macrocell assignments are removed. This allows the buried nodes to be placed in the same blocks, but it does not force the Fitter to use the same macrocell assignments.
- Pin, GLB and Macrocell Assignments — The pin, GLB, and macrocell assignments for the design are back annotated. This option retains all the pin and macrocell assignments.
- IO Types – You can also back annotate IO Types constraint settings.

You can only back annotate project assignments after the Fit Design process has been successfully completed. An error message appears if the ispLEVER software detects that this process did not complete successfully.

Design Verification

Verifying Designs

The ispLEVER software supports two types of timing verification: *static timing analysis* and *dynamic timing simulation*. Both of these methods support all Lattice devices.

Static Timing Analysis

Static timing analysis (timing analysis) is the process of verifying 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/hold time requirements, and the maximum frequency. Lattice has two static timing analysis tools, Performance Analyst and TRACE (for FPGAs).

The primary advantage of timing analysis is that it can be run at any time and requires no input test vectors, which can be very time consuming and tedious to create. Another major advantage of static timing analysis is that it exhaustively checks every possible input-to-output path. One shortcoming of all static timing analysis tools is that they detect false paths that will never be exercised during the course of normal operation of a circuit so that you could spend a lot time instructing the analyzer to ignore those paths. In this process, you could accidentally ignore a real issue. Although timing analysis does not give you a complete timing picture, it is an excellent way to quickly verify the speed of critical paths and identify performance bottlenecks.

Dynamic Timing Simulation

This type of analysis is based on an event-driven simulator and requires you to specify a test vector (waveform). Whereas timing analysis returns partial timing information, dynamic timing simulation (timing simulation) will give you detailed information about gate delays and worst-case circuit conditions. Because total delay of a complete circuit will depend on the number of gates the signal sees and on the way the gates have been placed in the device, timing simulation can only be run after the design has been implemented. Timing simulation also requires several input files to run.

There are two basic types of dynamic timing simulation tools, logic simulators (e.g, Lattice's Logic Simulator) and dynamic simulation analyzers (e.g., MTI's ModelSim™). Logic simulators function in a single delay mode, whereas dynamic simulation analyzers will simulate the ambiguity in delay pairs. Dynamic simulation analyzers typically take longer to process simulation results than logic simulators and considerably longer than static timing analysis tools.

Timing Verification Tools

The ispLEVER software offers timing verification with the following tools:

- Performance Analyst — Static timing analysis tool that runs timing analysis (All CPLD, ispXPLD, ispXPGA, and ispGDX2 devices except ispLSI 1K and 2K).
- Lattice Logic Simulator — Logic simulator that runs timing simulation (ispGDX and CPLD devices only).
- ModelSim for Lattice — Dynamic timing analyzer that runs timing simulation (all devices).
- TRACE — The *Timing Reporter and Circuit Evaluator* (TRACE) is an integrated flow tool that provides static timing analysis based on timing preferences (for FPGA devices only).

Additionally, timing simulation for all devices is supported with these tools:

- Text Editor — Used to create test stimulus files
- Waveform Editor — Used to create test stimulus files graphically

- Waveform Viewer — Used to view the results of simulation

Verification Environments

The timing verification tools operate in both integrated and stand-alone environments.

Integrated Timing Analysis

The Performance Analyst is a static timing analysis tool that lets you quickly determine the performance of designs implemented in any Lattice Semiconductor device. To run timing analysis, launch the Performance Analysis from the Project Navigator. The Performance Analyst traces each logical path in the design and calculates the path delays using the device's timing model and worst-case AC specs supplied in the device data sheet.

The timing analysis results are displayed in a graphical spreadsheet with source signals displayed on the vertical axis and destination signals displayed on the horizontal axis. The worst-case delay value is displayed in a spreadsheet cell if there is at least one delay path between the source and destination. To more easily identify performance bottlenecks, you can double-click a cell to view the path delay details.

Integrated Timing Simulation

To verify a design inside the current project, the ispLEVER software provides integrated verification with the Lattice Logic Simulator and ModelSim™ for Lattice from Mentor Graphics®. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the verification test bench, the following processes are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

CPLD and GDX Project Navigator Process	Simulation Tool Invoked
Timing Simulation	Lattice Logic Simulator
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

Stand-alone Simulation

The ispLEVER software supports stand-alone timing simulation. This provides an easy entry if you need to verify a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator Tools menu.

Required Files for Verification

The Lattice Logic Simulator and ModelSim support Lattice ispGDX and CPLD device timing simulation. In addition to your design files, you will need at least one test stimulus file, a netlist file, and a timing delay file.

	Lattice Logic Simulator	ModelSim
Test Stimulus File Formats		
• **.wdl** (Graphic waveform)	X	
• **.abv/.abl** (Test vector)	X	
• **.tf** (Verilog test fixture)		X
• **.vhd** (VHDL test bench)		X
Netlist File Formats		
• **.vo** (Verilog netlist)		X
• **.vho** (VHDL netlist)		X
• **.edo/.sim** (EDIF/simulation)	X	
Delay File Formats		
• **.sdf** (Standard Delay File)	X	X

Verification File Descriptions

Test Stimulus Files

- Graphic Waveforms (`.wdl`) — A file created by the Waveform Editor file that graphically represents a waveform as a sequence of signal states separated by time intervals.
- Test Vectors (`.abv/.abl`) — Test vectors are sets of input stimulus values and corresponding expected outputs that can be used with both functional and timing simulators. Test vectors can be specified either in a top-level ABEL-HDL source or in a separate ABEL-HDL test vector format (`.abv`) file. The ABV file is considered a text document and is kept above the device level in the Sources window. Whether the test vectors are part of a top-level ABEL-HDL source (`.abl`) or are in a separate file, they will be compiled and passed to the simulator.
- Verilog Test Fixtures (`.tf`) — A Verilog test stimulus file that specifies the input waveforms for simulation in ASCII format.
- VHDL Testbench (`.vhd`) — A VHDL test stimulus file that specifies the input waveforms for simulation in ASCII format.

Netlist Files

- Verilog Netlist (`.vo`) — For Verilog designs, the back-annotated timing simulation netlist is named `<design_name>.vo`.
- VHDL Netlist (`.vho`) — For VHDL designs, the back-annotated timing simulation netlist is named `<design_name>.vho`.
- EDIF/Simulation Netlist (`.edo/.sim`) — For EDIF designs, the back-annotated timing simulation netlist is named `<design_name>.edo`.

Timing Delay Files

- Standard Delay Format (`.sdf`) — A file containing delay and timing constraint data for cell instances named `<design_name>.sdf`.

Generating Timing Simulation Files

After you have fit the design, the ispLEVER software lets you export the netlist and delays for timing simulation. For netlist files, ispLEVER supports VHDL, EDIF, and Verilog formats. For timing delay files, ispLEVER supports the standard SDF and Viewlogic DTB timing formats.

To choose a simulation file format:

1. In the Project Navigator, choose **Tools > Generate Timing Simulation Options** to open the dialog box.
2. Select the format options that you want.
   - For netlist formats, you have a choice of Verilog, VHDL, or EDIF (Version 2.0.0). If you choose the EDIF format, you can customize the Power/Ground representation by selecting either Cell or Net.
   - For timing format, SDF (version 2.1) and Viewlogic DTB format are supported.
3. Click **OK** to close the dialog.
4. When you run the **Generate Timing Simulation Files** process, ispLEVER generates the files in the specified formats.

Viewing the Simulation Input Files

You can use the Report Viewer to view simulation input files.

1. In the Project Navigator Sources window, select the target device.
2. In the Project Navigator Processes window, double-click **Report File**. Notice the two output files (Netlist and Delay) listed at the top of the report. The ispLEVER software generates these files and places them in the project directory.
3. To view these files, choose **File** > **View** and select the file that you want to view.

***Note**: You cannot modify files using the Report Viewer. You can only view them.*

CPLD Verification Summary

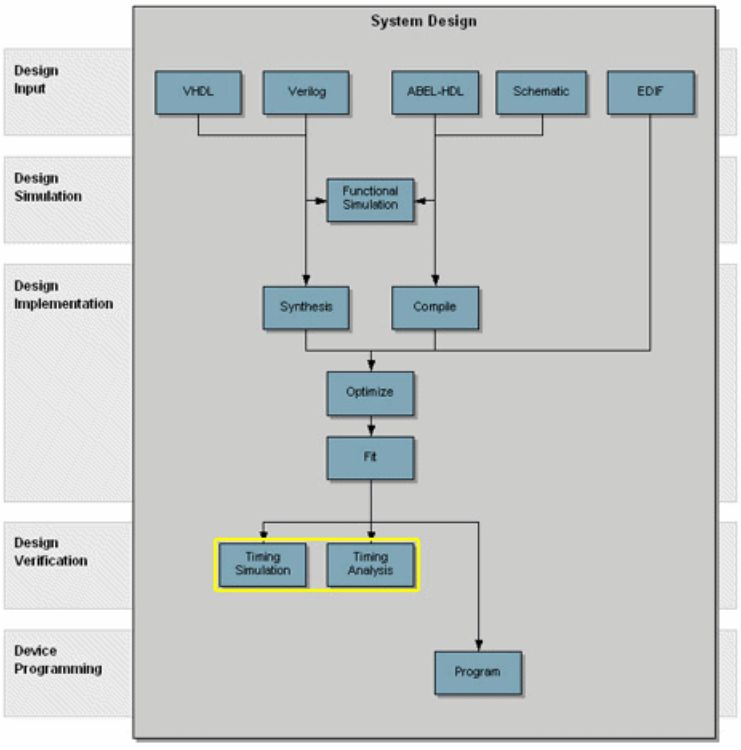
The Lattice Logic Simulator supports CPLD timing simulation for all design entry methods and requires at least one graphic waveform (`.wdl`) or test vector (`.abv`) stimulus file.

The ModelSim simulator supports timing simulation for all design entry methods and requires at least one Verilog test fixture (`.tf`) or VHDL testbench (`.vhd`) stimulus file. Additionally, ModelSim requires a netlist file (`.vo` or `.vho`) and a timing delay file (`.sdf`).

	Lattice Logic Simulator		ModelSim	
	.wdl	.abv	.tf	.vhd
			.vo + .sdf	.vho + .sdf
ABEL	X	X	X	
Schematic	X	X	X	
EDIF	X	X	X	X
Verilog	X	X	X	
VHDL	X	X		X

CPLD Verification Process Flow

The figure below shows timing analysis and simulation within the CPLD process flow.



Stamp Model

Introduction to Static Timing Analysis Tool

In the era of high-performance electronics, timing continues to be the number one issue in CPLD design. The designers are spending an increasingly large percentage of their time addressing CPLD performance.

Static Timing Analysis Tool provides solutions to the complexity, the performance, and even the time-to-market crises. Static Timing Analysis separates performance analysis from functional verification. It assumes that the design is functionally correct and analyzes performance only.

Functionality of Static Timing Analysis Tool

Static Timing Analysis Tool can be used as the board-level static timing analyzer. You can obtain results of static timing analysis and eliminate the timing errors in significantly less time for a board-level design. Its modes are mainly chip-level.

Static Timing Analysis Tool can also be integrated with synthesis tools. After the integration, it can be used for timing optimization of synthesis. There are four modeling levels: black box, gray box, gate-level, and transistor-level.

Moreover, Static Timing Analysis Tool is used as a gate-level timing analyzer, which enables you to get some timing information such as critical path, maximum operating frequency, setup and hold time for flip-flop, delay of combination circuit, pin-to-pin delay. The Performance Analyst is used for this purpose.

Stamp Modeling Language

Stamp is a new modeling language developed to enable the accurate, concise, and complete specification of timing models for large cores and blocks.

Stamp Model Generator Command Line

```
stamppar -i input_file [-o output_file][-tech tech_file][-vl][-gui][-log
automake.err][-help]
```

Input Files (produced by the Performance Analyst):

`design_name.trp` (Produced by the Performance Analyst and contain all information for stamp model generator)

Output Files

`design_name.mod` (Stamp model file of a design)

`design_name.data` (Stamp model data file of a design)

Technology Files

Technology file allows you to specify technology-dependent constraint values for some Stamp timing arcs.

Report files

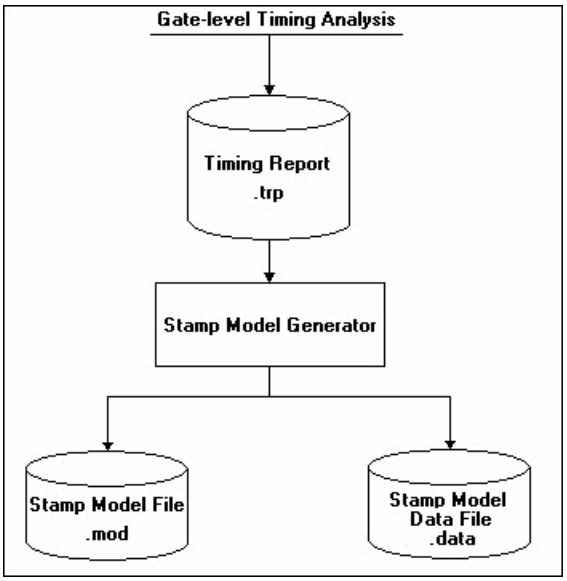
`automake.log` (-gui mode, integration in ispLEVER)

`stamppar.log` (in stand-alone mode)

Design Flow

Lattice Stamp Model Generator is used to build board-level (or chip-level) Stamp Models of a design with any Lattice Semiconductor device for board-level static timing analysis tools. With board-level Stamp Models for PMS (Processor, Memories, Switch, and peripheral circuit) parts such as CPU, memories, ispMACH chip of a design, a board-level static timing analysis tool enables you to manage large, high performance designs while minimizing development time. With the Stamp models, you accelerate board-level design.

Design Flow of Lattice Timing Model Generator



Running the Lattice Stamp Model Generator

You can run Lattice Stamp Model Generator either in its integration mode or in its stand-alone mode.

Integration Mode

After running the Performance Analyst, with the target device selected in the Sources window of Project Navigator, double-click Generate Board-level Stamp Model in the Processes window. When the process has completed successfully, the Stamp Model File (`.mod`) and the Stamp Model Data File (`.data`) are generated. You can view the two files in the Report Viewer.

Stand-alone Mode

If you have some timing data other than the current project, you can also invoke the stand-alone Stamp Timing Model Generator using the following command line:

```
stamppar -i design_name [-o new_design_name][-tech tech_file][-vl]
```

The default Stamp files are `<design_name>.mod` and `<design_name>.data`. You can use the `[-o new_design_name]` option to produce new stamp files `<new_design_name>.mod` and `<new_design_name>.data`.

The `[-tech tech_file]` option allows you to specify technology-dependent constraint values for some Stamp timing arcs.

The `[-vl]` option forces the `design_name` and port name in Stamp Model to be upper-case so that they can correspond to those in the board-level netlists created by Viewdraw of Viewlogic.

All timing paths of your design can be produced after running Lattice Stamp Timing Model in its stand-alone mode.

***Note**: In this release of ispLEVER, the stand-alone Stamp Model only supports ispGDX and ispMACH devices.*

Structure of Stamp Model File and Stamp Model Data File

Stamp Model File (`.mod`)	Stamp Model Data File (`.data`)
Header	Header
Port Definitions	Port Data
Arc Definitions	Arc Data

Sample Stamp Model File (.mod)

```
MODEL
MODEL_VERSION "1.0";
DESIGN "and_ff2";
DATE "Fri Dec 01 16:43:56 2000";
VENDOR "Lattice Semiconductor Corporation";
PROGRAM "STAMP Model Generator";

/* port name and type */
INPUT Clk;
INPUT input_1;
INPUT input_2;
OUTPUT output_q;

/* timing arc definitions */
Clk_output_q_delay: DELAY Clk output_q;

/* timing check arc definitions */
input_1_Clk_setup: SETUP(POSEDGE) input_1 Clk;
input_1_Clk_hold: HOLD(POSEDGE) input_1 Clk;
input_2_Clk_setup: SETUP(POSEDGE) input_2 Clk;
input_2_Clk_hold: HOLD(POSEDGE) input_2 Clk;

ENDMODEL
```

Sample Stamp Model Data File (.data)

```
MODELDATA
MODELDATA_VERSION "1.0";
DESIGN "and_ff2";
DATE "Fri Dec 01 16:43:56 2000";
VENDOR "Lattice Semiconductor Corporation";
PROGRAM "STAMP Model Generator";
```

```
/* port drive, max transition and max capacitance */
PORTDATA
Clk: MAXTRANS(0.0);
input_1: MAXTRANS(0.0);
input_2: MAXTRANS(0.0);
output_q: MAXTRANS(0.0);
ENDPORTDATA

/* timing arc data */
TIMINGDATA

ARCDATA
Clk_output_q_delay:
CELL_RISE(scalar) {
VALUES(6.4);
}
CELL_FALL(scalar) {
VALUES(6.4);
}
ENDARCDATA

ARCDATA
input_1_Clk_setup:
RISE_CONSTRAINT(scalar) {
VALUES(0.6);
}
FALL_CONSTRAINT(scalar) {
VALUES(0.6);
}
ENDARCDATA
ENDTIMINGDATA
ENDMODELDATA
```

Device Programming

Programming Devices

Lattice supports device programming for all programmable logic devices with the following tools, which are briefly described below and covered in detail in their respective Help. They are listed in alphabetical order.

ispVM System

The ispVM™ System software (ispVM) supports both serial and concurrent (turbo) programming of all Lattice devices in a PC environment. The ispVM System software is built around a graphical user interface. Device chains can be scanned automatically. Any required JEDEC ISC or Bitstream data files are selected by browsing with a built-in file manager. Non-Lattice devices that are compliant with IEEE 1149.1 can be bypassed once their instruction register length is defined in the chain description. Programmable devices from other vendors can be programmed through the vendor-supplied SVF file. See the ispVM System Help for more information about this tool.

Model 300 Programmer

The ISP Engineering Kit Model 300 programmer is an engineering device programmer that supports prototype development by allowing single-device programming directly from a PC. The Model 300 programmer supports all JTAG devices produced by Lattice, with device Vcc of 1.8, 2.5, 3.3, and 5.0V. See the Model 300 Programmer Help for more information about this tool.

SVF Debugger

The SVF Debugger can be used with ispVM System software to help you debug a Serial Vector Format (SVF) file. The SVF Debugger software allows you to program a device, and then edit, check syntax, debug and trace the process of an SVF file. See the SVF Debugger Help for more information about this tool.

Universal File Writer

The Universal File Writer (UFW) is a separate application that generates bitstream files or an SVF data file for a single device. Using JEDEC or ISC files, the software generates bitstream PCM, Intel Hex and Motorola Hex data files concurrently. It can also generate an SVF file using the parameters you select. You can run the Universal File Writer from the ispVM System toolbar or separately. See the Universal File Writer Help for more information about this tool.

Running ispLEVER from the Command Line

Running from the Command Line

You can run the ispLEVER software from the command line on PC and UNIX using **ispflow** command line software. The ispflow software will attempt to fit the design to the specified part.

***Note**: All examples are shown in PC format. For UNIX, use forward slash instead of back slash.*

Syntax

The format of the command is as follows (on one line):

```
ispflow [-i <design>] [-d <device>] [-imp <yes\no>]
[-sdf <edif\verilog\vhdl\off>]
[-edf <mentor\synopsys\synplicity\viewlogic>]
[-syn <spectrum\synplify>] [-h] [-v]
```

where [ ] denotes optional parameters.

Definitions

-i <design name>	EDIF, ABEL, Verilog, or VHDL design name. The name must include the appropriate file extension in the design name, such as `.edf`, `.abl`, `.v`, or `.vhd`.
-d <device name>	Specifies the device part number that the design will be fitted to. For example, `LFX125B-04F256C`. This option is required if the `<design name>.lci` file does not exist. After running **ispflow**, the specified device will appear in the newly created LCI file.
-imp <yes/no>	Import source constraints. Default is Yes.
-sdf <edif>	Outputs an SDF file in EDIF format.
-sdf <verilog>	Outputs an SDF file in Verilog format.
-sdf <vhdl>	Outputs an SDF file in VHDL format. Default.
-sdf <off>	Switch off SDF output.
-edf <mentor>	Specifies a Mentor Graphics-generated EDIF file. Default.
-edf <synopsys>	Specifies a Synopsys-generated EDIF file.
-edf <synplicity>	Specifies a Synplicity-generated EDIF file.
-edf <viewlogic>	Specifies a Viewlogic-generated EDIF file.
-syn <spectrum>	Specifies a LeonardoSpectrum synthesize file. Default.
-syn <synplify>	Specifies a Synplify synthesize file.
-spd <yes/no/fmax>	Speed: yes (speed), no (area), fmax.
-mpts	Max_PTerm_Split value.
-mptc	Max_PTerm_Collapse value.
-mptl	Max_PTerm_limit value.
-mfan	Max_fanin value.
-msym	Max_symbols value.
-fmll	Fmax_Logic_Level value.
-svf <on/off>	Switch on SVF generation. Default is off.
-r <*.lct/*.lci>	Refit using the constraint file.
-c <yes/no>	Specifies to run vcick (vc checker). Default is no.

-h	Help.
-v	Displays version number

The input source file switch (`-i`) is mandatory. All other switches are not.

If a device name (`-d <device name>`) is specified from the command line, and a `.lci` file exists, the device specified from the command line takes precedence. The **ispflow** software will not run if a device name is not specified from either the command line or in a `.lci` file.

Specifying Options and Pin Assignments

To specify optimization options and pin assignments, you must create or modify a `<design_name>.lci` file. See Constraint Editor Help – "Lattice Constraint File Description" for more information on the LCI file format. If there is no LCI file, the software creates a default file for the specified device. In this case, the `-d` option is required.

***Note 1**: The LCI file is case-sensitive. Once an LCI file is created, the –d option can be omitted from the command line, because the device information will be obtained from the LCI file. The –d option takes precedence over the LCI file. If the –d option is used again with a different device part number, the device information part of the LCI file will be updated to reflect the changes.*

***Note 2**: Pin assignments and certain optimization options in the LCI file could be incorrect if the device is changed on the Command Line.*

The ispflow software runs through the complete flow, including timing analysis.

Input Formats

Acceptable input formats are non-hierarchical EDIF, VHDL, and Verilog HDL source files. The source files must be named with the `.edf`, `.vhd`, or `.v` extensions respectively.

Log Files

A log file of the process will be generated named *`<design_name>`*`.batch.log` file.

Batch Mode Example

The following are examples for fitting a new design, and re-fitting a design after backannotating pin assignments.

***Note**: All examples are shown in PC format. For UNIX, use forward slash instead of back slash.*

To fit a new design:

1. Create a new directory and copy the input files needed.

   ```
   <isptools>\ispcpld\examples\cpld\abel\and_ff2\and_ff2.abl
   <isptools>\ispcpld\examples\cpld\abel\and_ff2\and_ff2.lci
   ```

2. Run `ispflow -i and_ff2.abl`.

   If you have a Lattice Constraint File (*`<design_name>`*`.lci`), the software will get the device from the LCI file and fit the design. The LCI file can be varied to an optimizer setting that you prefer. See Constraint Editor Help – "Lattice Constraint File Description" for additional information.

   or

3. If you do not have a LCI file, run `ispflow -i and_ff2.abl -d M4A3-256/128-55YC`. The software fits the design into the specified CPLD device.

Retaining Pin Assignments Using Batch Mode

You can retain pin assignments by copying LOCATION ASSIGNMENTS from your output Lattice Constraint File (`.lco`) into your input Lattice Constraint File (`.lci`).

To retain pin assignments using batch mode:

1. In the project directory, using a text editor, open the *<design_name>*`.lco` file.
2. In the LCO file, copy the LOCATION ASSIGNMENTS section.
3. In the project directory, using a text editor, open the *<design_name>*`.lci` file.
4. Replace the LOCATION ASSIGNMENTS section in the LCI file with the LOCATION ASSIGNMENTS section you copied from the LCO file.

LCI Files

The Lattice Constraint File (`.lci`) contains the constraints for Part selection as well as the Optimization and Placing and Routing processes.

You do not need to generate a LCI file on the first fit. If you specify a device (`-d`) with ispflow, this will generate a LCI file.

Command Line FAQs

The following are Frequently Asked Questions about processing designs in Command Line Mode using ispflow software.

Q. What report files are available to view after processing a design using Command Line mode?

A. Using a text editor, you can view the Timing Report File (`.trp`) and the Fitting Report File (`.rpt`) in the project directory after command line batch mode processing.

The RPT file displays details about how the current design was fit into the target device. If a fit was not accessible, the RPT file explains the problems preventing a fit. The file is located in your project directory, and is named *<design_name>*`.rpt`

The TRP file contains all information for stamp model generator. The file is located in your project directory, and is named *<design_name>*`.trp`.

Q. Where do I find my programming (JEDEC) file?

A. The programming file can be found in the project directory after command line batch mode processing. The file is located in your project directory, and is named *<design_name>*`.jed`.

Q. Where do I find my back-annotated timing simulation netlist?

A. The back-annotated timing simulation netlist is located in the project directory. The extension of the netlist depends upon the source files used to process your design.

- For VHDL designs, the back-annotated timing simulation netlist is named *<design_name>*`.vho`.
- For Verilog designs, the back-annotated timing simulation netlist is named *<design_name>*`.vo`.

- For EDIF designs, the back-annotated timing simulation netlist is named <*design_name*>.edo.

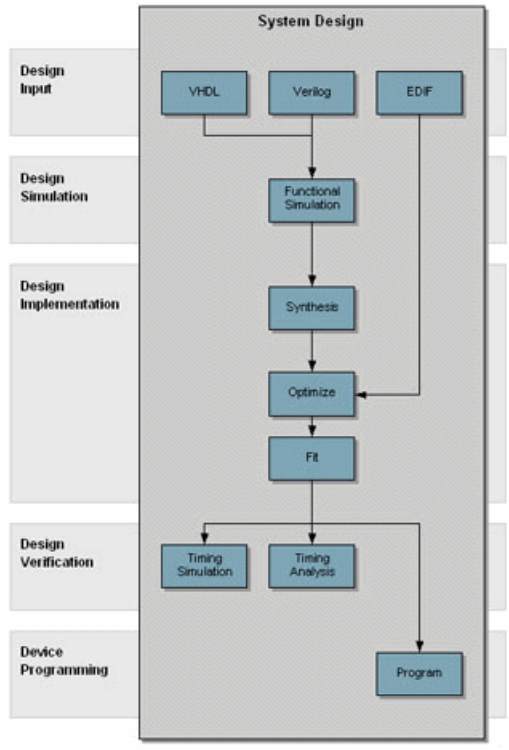
Q. Where do I find my timing delay file?

A. The timing delay file can be found in the project directory after command line batch mode processing. The file is located in your project directory, and is named <*design_name*>.sdf.

CHAPTER 5 *ispGDX Process Flow*

Introduction

ispGDX Design



Lattice's ispLEVER® 4.0 is a new generation of design tool that provides a complete system for ispGDX design including all ispGDX and ispGDX2 devices.

The ispLEVER software includes a fully integrated, push-button design environment and advanced features for schematic and HDL design entry, functional simulation, synthesis, implementation, optimization, and debug. Furthermore, integrated third-party tools let you get an accurate and complete simulation of your design.

Supported Device Families

- ispGDX
- ispGDX2

Overview of ispLEVER for ispGDX

The ispLEVER program offers an integrated environment consisting of several tools necessary to implement Lattice ispGDX devices. These tools are briefly described in the following paragraphs and covered in detail in their respective Help. They are listed in alphabetical order.

Constraint Editor

The Constraint Editor lets you specify pin and node location assignments, group assignments, I/O types settings, power level settings, resource reservations, PLL attributes, as well as output slew rates and JEDEC file options. The Constraint Editor reads the constraint file and displays the constraint settings. Modifications to the constraint file are made via the function dialogs. See the Constraint Editor Help for more information about this tool.

Hierarchy Browser

The Hierarchy Browser allows you to navigate through a design consisting of any combination of schematic and HDL modules. In contrast with the Hierarchy Navigator, the Hierarchy Browser works with designs whose top level is either a schematic or HDL source. Additionally, you can cross probe between design sources and their appropriate tool. See the Hierarchy Navigator/Browser Help for more information about this tool.

ispEXPLORER (ispGDX2 devices only)

The ispEXPLORER lets you run multiple passes of your design using different combinations of Fitter/Optimizer settings and critical timing constraints to achieve the best solution. Results are summarized in a single spreadsheet and detailed reports for each run are accessible. See the ispEXPLORER Help for more information about this tool.

Lattice Logic Simulator

Lattice Logic Simulator performs logic simulation on your design before you implement it into a Lattice device. You can observe not only the gate-level behavior at its inputs and outputs, but also the behavior of internal nodes. See the Lattice Logic Simulator Help for more information about this tool.

LeonardoSpectrum for Lattice

The LeonardoSpectrum™ synthesis environment from Mentor Graphics® lets you can create Lattice ispGDX device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process. LeonardoSpectrum combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features. See the LeonardoSpectrum for Lattice documentation supplied by the manufacturer for more information about this tool.

ModelSim for Lattice

The ispLEVER software supports third-party HDL simulation and verification with ModelSim™ from Mentor Graphics®. Using this integrated package, you can simulate single Verilog or VHDL designs within one environment. See the ModelSim for Lattice documentation supplied by the manufacturer for more information about this tool.

Performance Analyst (ispGDX2 devices only)

The Performance Analyst analyzes the performance of your design after it has been optimized and implemented by the Fitter. See the Performance Analyst Help for more information about this tool.

Project Navigator

The Project Navigator is the primary interface for ispLEVER and provides an integrated environment for managing the project elements and processes, as well as accessing all ispLEVER tools. See the Project Navigator Help for more information about this tool.

Report Viewer

You can use the Report Viewer to view, but not edit, the various report files generated by ispLEVER. See the Report Viewer Help for more information about this tool.

Synplify for Lattice

Synplify® for Lattice is a logic synthesis tool for ispGDX devices, developed by Synplicity®. Synplify starts with high-level designs written in Verilog or VHDL hardware description languages (HDLs). Synplify then converts the HDL into small, high-performance, design netlists that are optimized for Lattice devices. See the Synplify for Lattice documentation supplied by the manufacturer for more information about this tool.

Tcl Editor

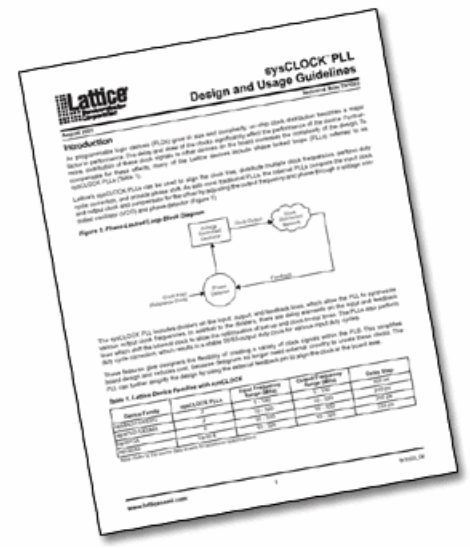
The Tcl Editor is a design tool that allows you to create, edit, and run a series of Tool Control Language (TCL) commands. These commands invoke individual ispLEVER processes for generating a complete programmable logic solution. The default text editor permits you to edit TCL code, and it highlights key TCL language elements in different colors to clarify the nature and use of each language element. You can generate a Tcl script for a project in Project Navigator, open it in the Tcl Editor, edit the script, and run it. The Tcl Editor, together with the robust features of the language, gives you more control over the design environment. See the Tcl Editor Help for more information about this tool.

Text Editor

The Text Editor is the ispLEVER text entry tool. You use this tool to create and edit text-based files, such as ABEL-HDL files, test stimulus files, and project documentation files. See the Text Editor Help for more information about this tool.

ispGDX2 Application Notes

The Lattice Semiconductor web site lists several Application Notes for ispGDX2 devices.


sysCLOCK™ PLL
Design and Usage Guidelines

Design Entry

Verilog HDL Design Entry

The ispLEVER software supports Verilog HDL, a hardware description language used to design and document electronic systems. Verilog HDL allows designers to design at various levels of abstraction.

All Lattice devices support Verilog HDL design.

Adding a Verilog HDL Module to Your Design

To add a Verilog HDL module to a design, you can either import a `.v` file, or create a new Verilog HDL module file with the Text Editor.

Creating a New Verilog HDL Module

You can use the Text Editor to create a new Verilog HDL module.

To create a new Verilog HDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. Select **Verilog Module** and click **OK**. The Text Editor window appears together with the New Verilog Module dialog box.
3. In the dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new Verilog HDL file appears in the Text Editor window.
5. Use the commands on the Edit menu to Cut, Copy, Paste, Find, or Replace text.

Synthesizing Your Verilog HDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor Graphics LeonardoSpectrum. You can synthesize your Verilog HDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

VHDL Design Entry

VHDL is a language for describing the structure and function of integrated circuits. VHDL allows you to:

- Describe the hierarchical structure and interconnect of a design.
- Specify the function of designs using familiar programming language forms.
- Simulate the design before being manufactured, so that design alternatives can be quickly compared and tested.

All Lattice devices support VHDL design.

Adding a VHDL Module to Your Design

To add a VHDL module to a design, you can either import a `.vhd` file, or create a new VHDL module file with the Text Editor.

Creating a New VHDL Module

You can use the Text Editor to create a new VHDL module.

To create a new VHDL module:

1. In the Project Navigator, choose **Source > New** to open the New Source dialog box.
2. In the dialog box, select VHDL Module and click **OK**. The Text Editor window appears together with the New VHDL Source dialog box.
3. In the New VHDL Source dialog box, type the relevant contents into the text fields.
4. Click **OK**. The new VHDL file appears in the Text Editor window.
5. Use the commands in the Edit menu to Cut, Copy, Paste, or Replace text.

Synthesizing Your VHDL Design

The ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity Synplify and Mentor graphics LeonardoSpectrum. You can synthesize your VHDL design as a stand-alone process, or you can synthesize automatically and seamlessly within the Project Navigator.

In Project Navigator, select your synthesis tool in one of two ways:

- Choose **Options > Select RTL Synthesis** and make your selection to run synthesis automatically.
- Choose **Tools** and make your selection to run synthesis stand-alone.

You can also run synthesis stand-alone by choosing the synthesis tool from the Lattice Semiconductor Program folder in your Start menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, you can view ispLEVER Third-Party Manuals.

EDIF Design Entry

The Electronic Design Interchange Format (EDIF) is a format used to exchange design data between different ECAD systems.

The EDIF format is designed to be written and read by computer programs that are constituent parts of EDA systems or tools. Its syntax has been designed for easy machine parsing and is similar to LISP.

The ispLEVER software supports EDIF Version 2 0 0.

All Lattice devices support EDIF design entry.

Importing an EDIF Netlist

You can import a design netlist description into the ispLEVER software from a third-party synthesis or schematic tool if the design file is formatted as EDIF 2 0 0.

***Note**: The project that you are importing the netlist into must be an EDIF project.*

To import an EDIF netlist into your project:

1. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
2. Change Files of type to **EDIF Netlist** (`*.ed*`), and then select the EDIF file that you want to import.
3. Click **Open** to open the Import EDIF dialog box.
4. The default setting for power and ground in the ispLEVER software are the VCC and GND symbols. If you know that the EDIF generated by other tools uses a different convention, you can change it in the window. Select **Custom**. Select either **Symbol** or **Net** representation. Then type the new names for VCC and GND.
5. If you are following the recommendation from Lattice for generating the EDIF file from the supported third party design kit, select **CAE Vendors**. From the list, choose the vendor that generated the EDIF file: Mentor, Synopsys, Synplicity, or Viewlogic.
6. Click **OK**. The software adds the selected EDIF file to the project sources.

Translating EDIF Properties

By default, the ispLEVER software ignores EDIF properties. If you want the ispLEVER software to translate EDIF properties to design constraints for the Fitter, do the following:

1. In Project Navigator, choose **Tools > Import Source Constraint Option** to open the dialog box. The Import Source Constraints Option dialog box lets you import constraints, such as Location (pin/node) Assignments, Group Assignments, and Output Slew Rate, from source files (ABEL, schematic, or EDIF).
2. In the dialog box, select **Auto Import Source Constraints**.
3. Click **OK**.

When you select this option, the ispLEVER software displays a confirmation dialog box prior to implementing the function. This confirmation dialog box appears every time you run the Fit Design process, unless you select the Do Not Import Source Constraints option.

On the warning message dialog box, if you click **Yes**, the constraints from the source files are written into the project constraint file.

***Important**: Constraints from source files and existing constraints in the project constraint file are not merged; existing constraints are overridden by the new constraints.*

Existing constraints (only Location Assignments, Group Assignments, and Output Slew Rate are affected) in the project are cleared, regardless of constraints that might exist in the source file. If there are constraints in the source file, the new constraints are written into the project constraint file. If there are no constraints in the source file, no constraints are written into the file.

EDIF Properties

The ispLEVER software will take design-specific constraints from the properties in the EDIF netlist. The following is the list of properties that the Fitter supports.

PIN LOCATION Property

Name: LOC

Value: {PIN # }

Example: LOC = P20

Scope: IO PORT, net connect to the IO port.

GROUPING Property

Name: GROUPING

Value: GROUP NAME

Example: Use the following command to assign signal locations in your design. In this case, you have a list of internal nodes: a, b, and c, and you want to assign them into a group "mg." The location of this group needs to be Block "A", Segment "2’:

On net a, grouping = mg

On net a, loc = "A, 2"

On net b, grouping = mg

On net b, loc = "A, 2"

On net c, grouping = mg

On net c, loc = "A, 2"

OUTPUT SLEW Property

Name: SLEW

Value: {Fast, Slow}

Example: To set port A to high slew, put the following property on the net or port: SLEW=Fast

Scope: OUTPUT PORT/NET

SIGNAL OPTIMIZATION Property

Name: OPT

Value: {KEEP, COLLAPSE}

Scope: On any net of the design.

OPEN DRAIN Property

Name: OPENDRAIN

Value: {On/Off}

Example: To set port A to an open drain, put the following property on the net or port: OPENDRAIN=on

Scope: OUTPUT PORT/NET

PULL Property

Name: PULL

Value: {On/Off/Hold}

Example: To set port A to pull up, put the following property on the net or port: PULL=on

Scope: OUTPUT PORT/NET.

OUTPUT VOLTAGE Property

Name: VOLTAGE

Value: {VCC/VCCIO}

Example: To set port A to output voltage at VCCIO level, put the following property on the net or port: VOLTAGE=VCCIO

Scope: OUTPUT PORT/NET.

Design Simulation

Running a functional simulation after a design description is complete allows you to verify that the description is functionally correct. Also, by simulating the functionality of your design *before* synthesis, you can find and correct basic design errors sooner. While functional simulation will verify your Boolean equations, it does not indicate timing problems.

The ispLEVER software supports functional simulation for Lattice Semiconductor CPLD and FPGA devices using the Lattice Logic Simulator or *ModelSim* from Mentor Graphics. The RTL design can be simulated for functionality before synthesis using the VHDL or Verilog design description and an input stimulus file.

Simulation Environments

The functional simulators operate in both integrated and stand-alone environments.

Integrated Simulation — To simulate a design inside the current project, the ispLEVER software provides integrated simulation. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the simulation source file, the processes in the tables below are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

For ModelSim, ispLEVER includes scripts that run functional simulation automatically using Lattice-defined settings and preferences. You can customize the run by creating a different script files (DO file), which is a simple script that contain commands that are equivalent to the ModelSim GUI commands. This macro is automatically called when you run ModelSim.

For information about creating your own ModelSim macros, see the *ModelSim User's Manual*, *Chapter 11 Tcl and Macros*, provided with your ispLEVER software (Third-Party Manuals).

CPLD and ispGDX Project Navigator Processes	Simulation Tool Invoked
Functional Simulation	Lattice Logic Simulator
Verilog Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Functional Simulation	ModelSim
Verilog Post-Route Functional Simulation	ModelSim
VHDL Functional Simulation	ModelSim
VHDL Post-Route Functional Simulation	ModelSim

Stand-alone Simulation — The ispLEVER software supports stand-alone functional simulation. This provides an easy entry if you need to simulate a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator **Tools** menu.

Design File Descriptions

Lattice Logic Simulator and ModelSim enable you to simulate the operation of your design in the following design entry formats:

- ABEL-HDL format (*design*.abl) — a hierarchical logic description language that supports a variety of behavioral input forms, including high-level equations, state diagrams, and truth tables. (CPLD designs only)
- Schematic format (*design*.sch) — describes your circuit in terms of the components used and how they connect to each other. (CPLD designs only)
- VHDL format (design.vhd) — Very High Speed IC Hardware Description Language format.
- Verilog HDL format (*design*.v) — an industry-standard hardware description language used to describe the behavior of hardware that can be implemented directly by logic synthesis tools.

The Lattice Logic Simulator also supports mixed design entry as follows:

- Schematic and ABEL-HDL (CPLD designs only)
- Schematic and VHDL
- Schematic and Verilog HDL

ispGDX Test Stimulus Files

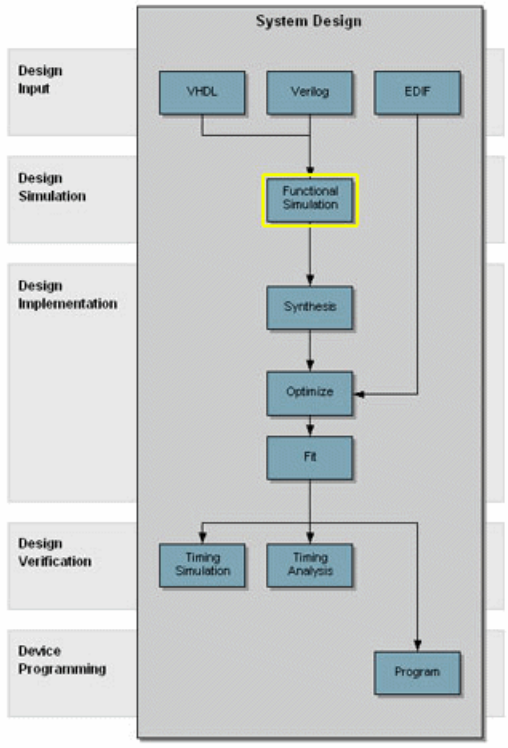
The Lattice Logic Simulator supports ispGDX functional simulation for HDL design entry methods and requires at least one graphic waveform (*.wdl) or test vector (*.abv) stimulus file.

The ModelSim simulator supports functional simulation for HDL design entry methods and requires at least one Verilog test fixture (*.tf) or VHDL test bench (*.vhd) stimulus file.

	Lattice Logic Simulator		ModelSim	
	*.wdl	*.abv	*.tf	*.vhd
Verilog	X	X	X	
VHDL	X	X		X

ispGDX Simulation Process Flow

The figure below shows functional simulation within the ispGDX process flow.



Creating a Waveform Stimulus File using the Waveform Editor

The Waveform Editor lets you graphically create a test stimulus file by clicking and dragging with the mouse. You see exactly what each waveform will look like, as well as its timing relationship to all the other waveforms. The output file is a waveform display file (`.wdl`). This file can be imported into any CPLD project as a waveform stimulus file and used by the Lattice Logic Simulator for simulation.

See the Waveform Editor Help for specific information about creating waveform stimuilus files

Creating ABEL Test Vectors from a Template

***Note**: ispGDX devices do not accept ABEL source files but will accept an ABEL test vector file.*

ABEL test vectors can be specified either in a top-level ABEL-HDL source or in a separate test vector (`.abv`) file. The ABV file is considered a text document and is kept above the device level in the Sources window. Whether the test vectors are part of a top-level ABEL-HDL source (`.abl`) or are in a separate file, they will be compiled and passed to the simulator.

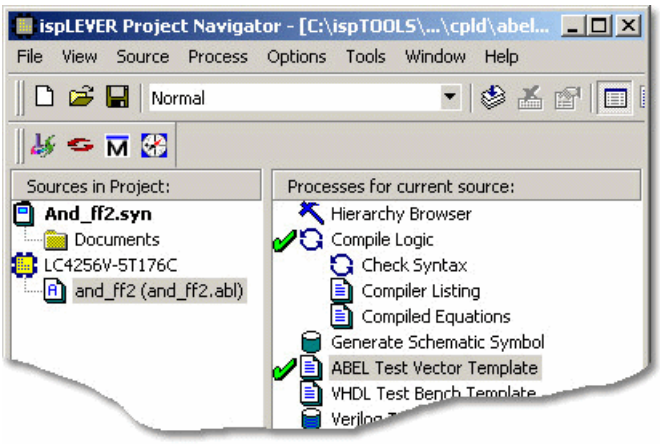
The easiest way to automatically create an ABEL Test Vector template is using the Project Navigator ABEL Test Vector Template process. After the test vector template file (`.abt`) is created, you must add your test vectors and rename it with the extension `.abv` before importing it into your design.

To automatically generate the Verilog test fixture template file and import it into your design:

1. Open your ABEL-HDL design in the Project Navigator.
2. In the Sources window, select the top-level ABEL design source (`*.abl`) file.

   ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

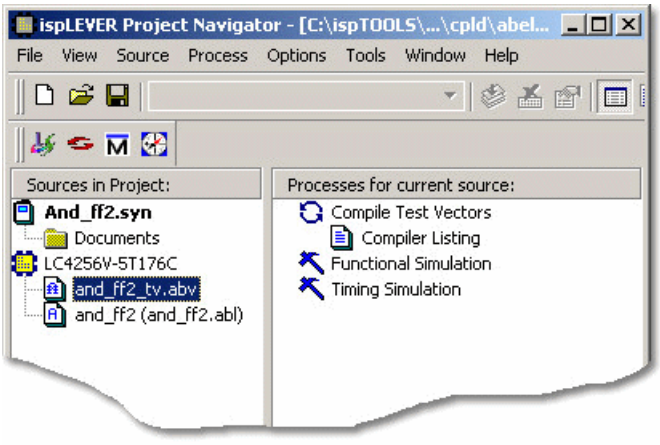
3. In the Processes window, double-click the **Verilog Test Fixture Declarations** process.



   This process creates a template file for an ABEL Test Vector file (`<abel_sourcefile_name>.abt`). However, in order to use this file as a test vector in your design, you must edit it and rename it with the extension `.abv`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `and_ff2_TV.abv`. Add the "`TV`" to the name so that the file will not be overwritten. Change the file extension to "`.abv`" so that it can be imported into the project as a test vector source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test vector file. Click **Open**.

   The file is imported into the project as an ABEL Test Vector. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Creating a Verilog Test Fixture from a Template

Verilog test stimulus can be specified either in the top-level HDL source or in a separate test fixture (`.tf`) file. You can create the test fixture manually using a text editor or use a Verilog Test Fixture template (`.tfi`) file.

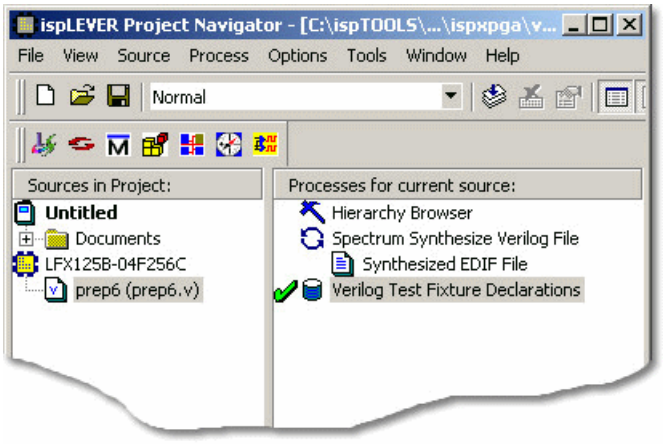
The easiest way to automatically create a Verilog Test Fixture template is using the Project Navigator Verilog Test Fixture Declarations process. After the test fixture template file (`.tfi`) is created, you must add your test vectors and rename it with the extension `.tf` before importing it into your design.

To automatically generate the Verilog test fixture template file and import it into your design:

1. Open your Verilog design in the Project Navigator.
2. In the Sources window, select the top-level Verilog design source (`*.v`) file.

   ***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

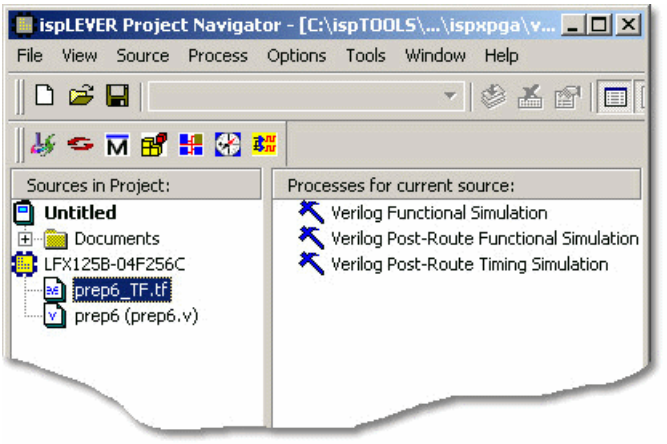
3. In the Processes window, double-click the **Verilog Test Fixture Declarations** process.



   This process creates a template file for a Verilog Test fixture (`<verilog_sourcefile_name>.tfi`). However, in order to use this file as a test fixture in your design, you must edit it and rename it with the extension `.tf`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `prep6_TF.tf`. Add the "`TF`" to the name so that the file will not be overwritten. Change the file extension to "`.tf`" so that it can be imported into the project as a test fixture source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test fixture file. Click **Open**.
7. In the Associate Verilog Test Fixture dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a Verilog Test Fixture. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Creating a VHDL Test Bench from a Template

VHDL test stimulus can be specified either in the top-level HDL source or in a separate test bench (`.vhd`) file. You can create the test bench manually using a text editor or use a VHDL Test Bench template (`.vht`) file.

The easiest way to automatically create a VHDL Test Bench template is using the Project Navigator VHDL Test Bench Template process. After the test bench template file is created, you must add your test stimulus and rename it with the extension `.vhd` before importing it into your design.

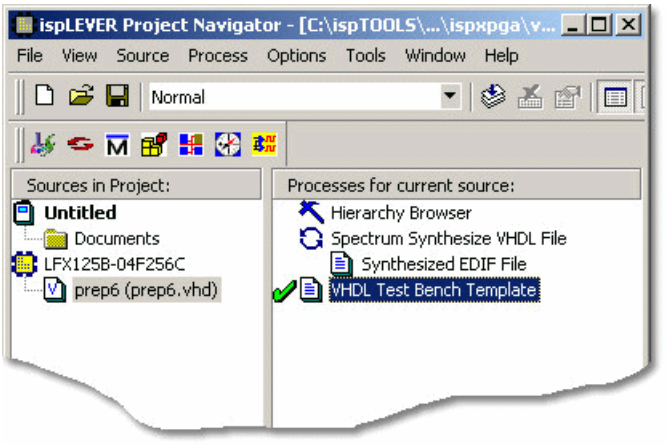
To generate the VHDL test bench template and import it into your design:

1. Open your VHDL design in the Project Navigator.
2. In the Sources window, select the top-level VHDL design source (`*.vhd`) file.

***Note**: You can generate templates for lower-level modules. However, if you use these as test vector templates you can only perform functional simulation and not timing simulation. The top test vector file can perform both simulations.*

---

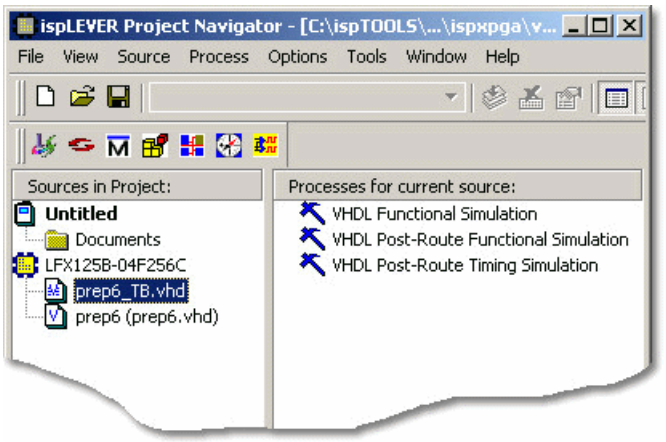
3. In the Processes window, double-click the **VHDL Test Bench Template** process.



This process creates a template file for a VHDL Test Bench (`<vhdl_sourcefile_name>.vht`). However, to use this file as a test bench in your design, you must edit it and rename it with the extension `.vhd`.

4. Using the Windows Explorer, go to the project folder and change the name of the file, for example `prep6_TB.vhd`. Add the "`TB`" to the name so that the file will not be overwritten. Change the file extension to "`.vhd`" so that it can be imported into the project as a test bench source file.
5. In the Project Navigator, choose **Source > Import** to open the Import File dialog box.
6. Select the new test bench file. Click **Open**.
7. In the Import Source Type dialog box, select **VHDL Test Bench** and click **OK**.
8. In the Associate VHDL Test Bench dialog box, associate the file to the target device or a certain module. Click **OK**.

   The file is imported into the project as a VHDL Test Bench. You can open the file from the Project Navigator and edit the user-defined section, adding code to generate the stimulus for your design.



Interfacing with ModelSim

ModelSim functional simulation generates several batch files, such as `.fdo`, `.udo`, and `.tdo`. ModelSim users will frequently take advantage of customizing batch files to control their simulation, for example specifying signals to display, run time, and waveform display options.

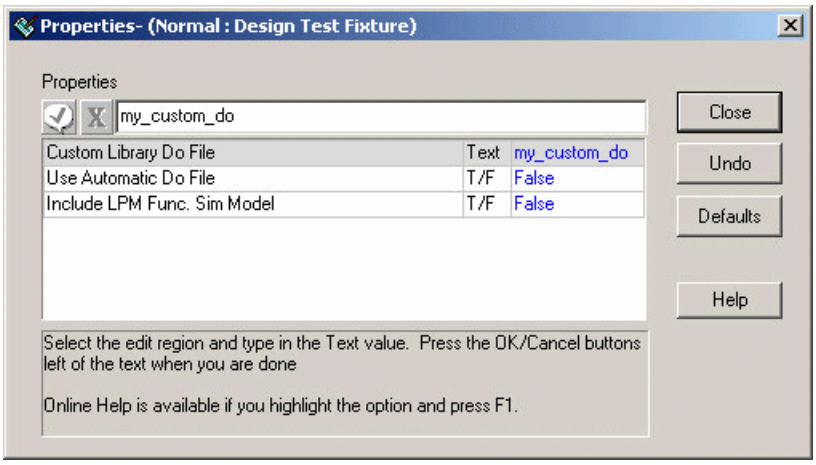
There are two options for creating custom DO files:

Option 1

- In the project folder, edit the `*.udo` file. The Project Navigator will not overwrite this user DO file.

Option 2

1. In the Project Navigator Sources window, select the test bench source.
2. In the Processes window, right-click the functional simulation process to open the Properties dialog box.
3. In the dialog:
   - Type a name for the file in the Custom Library Do File field and click the checkmark icon
   - Set Use Automatic Do File to **False**.
   - Click **Close**.



4. The software will automatically create this file when functional simulation is run. Edit this file as needed.

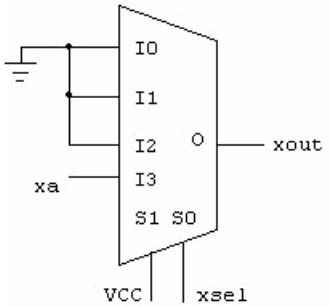
Design Implementation

ispGDX2 Design Guidelines

The ispGDX2 family is Lattice's next generation in-system programmable generic digital cross point switches for high speed bus switching and interface applications. The Lattice ispGDX2 devices combine a flexible switching architecture with advanced sysIO interfaces including 1Gbps sysHSI blocks and sysCLOCK PLLs. It also features in a multiplexer-intensive architecture and on-chip control logic. In order to take full advantage of the device architecture, the following should be considered for ispGDX2 designs.

MUX-based Architecture Ideal for MUX Design

The ispGDX2 family adopts a MUX-based architecture. The logic is implemented through MUX primitive. For example, the implementation of `xout=xa&xsel` is:



Using the MUX-based architecture, the ispGDX2 family is highly efficient for MUX design, such as data switch, data shifter, and some easy logic. However, it is not ideal for complex logic. For example, implementing the equation `xout=xa&xb&xc&xd&xe&xf&xg&xh&xi` may require 4 muxes in an ispGDX2 device, which is not efficient.

Each Four IO Cells Share the Same Select Signal

Based on the ispGDX2 architecture, each 4 adjacent IO cells must be controlled by the same select signal. The following design will fail because the control signals are not from the same source.

```
Input xa, xb;
Input sela, selb;
Output xouta, xoutb;
//mentor attribute xouta loc AB13 //IO0
//mentor attribute xoutb loc AA13 //IO1
xouta <= sela & xa;
xoutb <= selb & xb;
```

LVDS Pin-Locking

The ispGDX2 family supports Low Voltage Differential Signal (LVDS). Each two IOs in an ispGDX2 device form a pair with one X-type IO and one Y-type IO. When specifying pin locking of LVDS signals, you must lock the P-port of LVDS to X-type IO and the software will automatically lock the N-port to the corresponding Y-type IO in the same pair.

Co-Existence of Preset and Reset Signals

The Preset and Reset signals can co-exist in an ispGDX2 device. Each ispGDX2 device has 16 (for ispGDX2-256) or 8 (for ispGDX128) blocks. Although Preset and Reset cannot be applied to the same block, each block can have its own Preset or Reset signals. In the following example, the `xout1` pin is controlled by a Reset signal `rst` while the `xout3` pin is with a Preset signal `pset`.

```
always @(posedge clk or posedge rst)
begin
if (rst)
 begin
  xout <= 'b0; xout1 <= 'b0;
 end
else
 begin
  xout <= xin0 & xin1;
  xout1 <= xin0 & xin2;
 end
end
always @(posedge clk or posedge pset)
begin
if (pset)
 begin
  xout2 <= 'b1; xout3 <= 'b1;
 end
else
 begin
  xout2 <= xin3 & xin4;
  xout3 <= xin3 & xin5;
 end
end
```

Reference Voltage Pin Usage

The ispGDX2 family supports SSTL3_I/II, SSTL2_I/II, and HSTL_I/III attributes. When these attributes are set, the appropriate blocks will need reference voltage to realize the required output voltage, and the Reference Voltage pins (`Vref`) in those blocks can no longer be used as IO pins.

Flexible IO Configuration

The ispGDX2 family is flexible to switching architecture. The IOs can be controlled by various signals including:

- Clock
- Clock Enable
- Out Enable
- Clock Output Enable
- Select signal
- Reset/Preset

As a result, you can configure the IOs of an ispGDX2 device as combine circuit, Latch, Register, or Input Register with clock, clock enable, and/or output enable.

Setting Constraints and Parameters

For an ispGDX2 design, you can set constraints and component parameters:

- In Verilog/VHDL design source file
- Via the Constraint Editor
- Manually in LCT file

Setting Constraints and Parameters in Verilog/VHDL Sources

Constraint information or component parameters can be included in Verilog or VHDL source code. The examples demonstrate how to specify an HSI macro in HDL source. Also shown at the end of this page is a pin-lock example.

Verilog for Simulation

Macro block declaration — The macro declaration should be included in design source or include files.

```
module macro_name (m_port_1, m_port_2, ... m_port_s);

parameter in_freq = "100.0";
parameter div = "1";
parameter mult = "1";
parameter sympat = "11000001011100000101";

input m_port_1;
input m_port_2;
...
output m_port_s;

end module
```

Parameter definition

```
defparam Instance_name.in_freq = "100.0",
Instance_name.mult = "10",
Instance_name.div = "1",
Instance_name.sympat = "11000001011100000101";
```

Macro instantiation

```
macro_name Instance_name
(
. m_port_1 (port_1),
. m_port_2 (port_2),
...
. m_port_s (port_s)
);
```

Verilog for Synthesis

Macro block declaration

```
module macro_name (m_port_1, m_port_2, ... m_port_s);

parameter in_freq = "100.0";
parameter div = "1";
parameter mult = "1";
parameter sympat = "11000001011100000101";

input m_port_1;
input m_port_2;
...
output m_port_s;

endmodule
```

Macro instantiation

```
macro_name Instance_name
(
. m_port_1 (port_1),
. m_port_2 (port_2),
...
. m_port_s (port_s)
);
```

Parameters configuration using Mentor Graphics attribute syntax

```
// mentor attribute Instance_name in_freq 100.0000
// mentor attribute Instance_name mult 10
// mentor attribute Instance_name div 1
// mentor attribute Instance_name sympat 11000001011100000101
```

VHDL for Simulation

Library declaration

```
library gdx2;
use gdx2.components.all;
```

Macro block declaration

```
component Macro_name
generic (
IN_FREQ : string;
MULT : integer;
DIV : integer;
PHASE_ADJ : string
);
port
```

```
(
m_port_1: in std_logic;
m_port_2: in std_logic;
...
m_port_s: out std_logic
);

end component;
```

Macro instantiation and Parameter definition

```
Instance_name: Macro_name
generic map
(
IN_FREQ => "500.0000",
MULT => 2,
DIV => 1,
PHASE_ADJ => "0"
)
port map
(
m_port_1 => port_1,
m_port_2 => port_2,
...
m_port_s => port_s
);
```

VHDL for Synthesis

Macro block declaration

```
component Macro_name
generic (
IN_FREQ : string;
MULT : integer;
DIV : integer;
PHASE_ADJ: string
);
port
(
m_port_1: in std_logic;
m_port_2: in std_logic;
...
m_port_s: out std_logic
);

end component;
```

Macro instantiation

```
Instance_name: Macro_name
port map
(
m_port_1 => port_1,
m_port_2 => port_2,
...
m_port_s => port_s
);
```

Parameters configuration using Mentor Graphics attribute syntax

***Important**: This step is essential for VHDL designs. If you fail to include parameters configuration with the following format, the synthesis tool will not pass these HSI attributes to the Fitter.*

```
attribute in_freq : string;
attribute in_freq of Instance_name : label is "500.0000";

attribute mult : string;
attribute mult of Instance_name : label is "5";

attribute div : string;
attribute div of Instance_name : label is "1";

attribute phase_adj : string;
attribute phase_adj of Instance_name : label is "0";
```

Pin Lock Example

Verilog HDL syntax

```
Output xa, xb;
//mentor attribute xa loc AB13 //IO0
//mentor attribute xb loc AA13 //IO1
```

VHDL syntax

```
entity dataexch is
port (
busAdata : input std_logic_vector (1 downto 0))
;
attribute LOC : string;
attribute LOC of busAdata : signal is "PAA13, PAB13";
end dataexch;
```

***Tip**: prefix 'p' should be added to package pin name.*

Setting Constraints and Parameters with the Constraint Editor

Specifying Bus Information

You can add new busses using the Constraint Editor.

To add a new bus in the Constraint Editor window:

1. In the Constraint Editor, choose **Pin Attribute > Group Assignment** to open the dialog box.
2. In the Group Name box, type a name for the new bus.
3. From the Available Signals list, select the signals you want to be included in the bus. Then, click the **>** button to move them to the Selected Signals list.
4. Click the **Add** button. The new bus is added to the Existing Group Assignment List with all its members displayed in the Signals column.

***Note**: It is not necessary to make selection in the "Assign Group to GLB" field. You just need to accept the default value "Any". If you choose a value other than "Any", the value will be ignored and no error will be reported.*

Locking Pins

You can preassign signals using the Location Assignment dialog box of the Constraint Editor.

To lock pins using the Location Assignment dialog box:

1. In the Constraint Editor, choose **Pin Attribute > Location Assignment** to open the dialog box.
2. In the middle of the dialog box, select **Bus Assignment** to display the Bus Assignment field.

   ***Note**: When **Bus Assignment** is checked, all signals in a bus are considered as a whole and will be assigned together. It is strongly recommended to use Bus Assignment for ispGDX2 designs.*

3. From the Bus List, select the bus for which you want to assign location.
4. From the Pin List, select the desired pin location for the bus.
5. Click **Add**. This new assignment is added to the Existing Location Assignment List.

   ***Note**: For location assignment, whether to specify the GLB location for a signal or not does not effect the fitting result.*

Setting Constraints and Parameters Manually in the LCT File

Specifying Bus Information

Bus information is stored in `[GROUP ASSIGNMENTS]` section. The syntax is:

```
<bus_name> = -, *, $ordered, <signal_name_list>;
<signal_name_list> = <signal_name>[,<signal_name_list>]
```

Locking Pins

Pin lock information is stored in `[LOCATION ASSIGNMENTS]` section. The syntax is:

```
<signal_name> = pin, <pin_name>, -, -, -;
```

To preassign node, you can directly edit the LCT file. The syntax is:

```
<signal_name> = node, <pin_name>, -, -, -;
```

Setting HSI Parameters

HSI parameters are stored in `[HSI Attributes]` section. The syntax is:

```
Instance_name = <macro_name>, <in_freq>, <mult>, <div>, <phase_adj>, <sympat>;
```

***Note**: phase_adj is used for CDRX_SS_4, CDRX_SS_6 and CDRX_SS_8 macros. Sympat is only used for macro CDRX_8B10B.*

Synthesizing

For Verilog and VHDL designs, the ispLEVER software provides two synthesis tools that are integrated into the Project Navigator environment: Synplicity *Synplify* and Mentor Graphics *LeonardoSpectrum*. You can synthesize your Verilog or VHDL design as a stand-alone process by choosing the synthesis tool from the Lattice Semiconductor program group in your Start menu, or you can synthesize automatically and seamlessly within the Project Navigator.

The ispLEVER Tutorials contains synthesis design flow tutorials and is the best place to start if you want to get some hands-on experience. By working through the tutorial lessons, you'll learn how to create sample design projects with some of ispLEVER's most useful and powerful features. You can view the ispLEVER Tutorial manuals by choosing **Help > Tutorials** from the Project Navigator menu.

For additional information about synthesis using LeonardoSpectrum or Synplify, see the ispLEVER Third-Party Manuals.

Synthesis Design Flows

You can run the synthesis tools from within the integrated ispLEVER environment, or as a stand-alone process. Whether using LeonardoSpectrum or Synplify, the high-level design flows are basically the same.

Integrated Flow

This approach lets you create, synthesize, import, and implement a design targeting one of the Lattice devices completely from within the ispLEVER Project Navigator environment.

1. Using the Project Navigator, create a new HDL project.
2. Target a device.
3. Using the Text Editor, create the HDL modules.
4. For mixed-mode designs, use the Schematic Editor to create the schematic files.
5. Using the Project Navigator, import the source files.
6. Select a synthesis tool.
7. Fit (Place and Route) the design.

Stand-alone Flow

The stand-alone approach requires you to create or load a VHDL or Verilog HDL design into the synthesis tool environment. Then you synthesize the design and generate an EDIF netlist that you imported into ispLEVER for implementing into a Lattice device.

1. Using your synthesis tool, create a project.
2. Target a device.
3. Load the source files.
4. Synthesize the design to create an EDIF file.
5. Using the ispLEVER Project Navigator, create an EDIF project.

6. Target a device (same as step 2).
7. Import the EDIF source file.
8. Fit (Place and Route) the design.

Integrated Third-Party Tools

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment.

LeonardoSpectrum

Within the LeonardoSpectrum synthesis environment, you can create Lattice device designs in VHDL or Verilog. The tool is fully integrated in the ispLEVER Project Navigator, or may be run as a stand-alone process.

LeonardoSpectrum from Mentor Graphics combines push-button ease of use with the powerful control and optimization features associated with workstation-based ASIC tools. For challenging designs, you can access advanced synthesis controls within LeonardoSpectrum's exclusive Power Tabs and powerful debugging features.

Synplify

The Synplify solution from Synplicity is a high-performance, sophisticated logic synthesis engine that utilizes proprietary technology to deliver fast, highly efficient FPGA and CPLD designs. Synplify uses Verilog and VHDL Hardware Description Languages as input, and outputs an optimized netlist for the Lattice device.

Selecting the Synthesis Tool

The ispLEVER software supports Verilog HDL and VHDL designs with two synthesis tools that are integrated into the Project Navigator environment. This integrated approach lets you create, synthesize, import, and implement a design targeting a Lattice device completely from within the ispLEVER Project Navigator environment.

To specify the synthesis tool that the ispLEVER software will use:

1. In the Project Navigator, choose **Options > Select RTL Synthesis** to open the dialog box.
2. Select the synthesis tool that you want to use. This tool will be associated with all devices in the current device family. You can also make a synthesis tool the default for all device families.

Setting Constraints

Setting and Editing Constraints

For ispGDX, CPLD, ispXPLD, and ispXPGA devices, the ispLEVER software supports setting and editing of constraints in these ways:

Constraint Editor

Many ispGDX, CPLD, ispXPLD, and ispXPGA constraints can be edited within the Constraint Editor. You can specify pin and node assignments, group assignments, resource reservations, power level settings, output slew-rates, and nodal constraints, as well as PLL and HSI attributes. Modifications to the constraint file are made via the function dialog boxes or directly in the appropriate spreadsheets.

To run the Constraint Editor:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window, double-click the **Constraint Editor** process.

   Any invalid attribute or incorrect assignment is displayed in red in the Constraint Editor constraint sheet as well as in all the dialog boxes of the Constraint Editor. All the default values are displayed in blue. However, you can change the system default colors by choosing View > Set Colors.

Optimization Constraint Editor

For most CPLD and ispXPLD devices, the Optimization Constraint Editor lets you specify the global constraints used in optimization. It reads the constraint file and displays the constraint settings in the Opt Global Constraints sheet. You can directly modify the optimization constraints in the sheet.

To edit global optimization constraints using the Optimization Constraint Editor:

1. In the Project Navigator Sources window, select the target device.
2. In the Processes window, double-click the **Optimization Constraint** process. The software runs the process and opens the Optimization Constraint Editor.
3. In the Opt Global Constraints sheet, double-click a table cell in the **Constraint Value** column.
4. Select a desired option from the list, or directly type your setting in the edit box.
5. Choose **File > Save** to save the edits to the project constraint file (`.lci`).

ispXPGA Floorplanner

For ispXPGA designs using the ispXPGA Floorplanner, you have the option of saving one or a combination of constraint types—PFU, group assignments, pin assignments, region assignments—to the design's constraint file (`.lct`) using teh File > Save Constraints command. The software overwrites the current LCT file with the new constraints and clears the checkmarks from the processes displayed in the Project Navigator Processes window. The software will apply the changes to the physical design file (`.ld2` or `.ld3`) the next time you run Pack & Place or Route.

To save constraints in the ispFloorplanner:

1. Choose **File > Save Constraints**.
2. In the Save Constraints dialog box, select **Region**, **Pin Assignments**, and/or **PFU Packing/Placement**, depending on the types of changes you have made.
3. In the File name box, accept the default name and file type and click **Save**.
4. Click **Yes** to confirm that you want to replace the current LCT file. The Floorplanner saves the changes to the constraint file.

Fitting

Fitting Designs

The ispLEVER software has a single user interface with all options preset to deliver the highest possible push-button performance. At the end of a successful fitter run, the ispLEVER software generates a JEDEC file, as well as a fitter report, so that you can see how the ispLEVER software has routed the design and utilized resources on the part.

Performing Multiple Runs

For CPLD, ispXPLD, and ispXPGA devices, you can use the ispEXPLORER to try many combinations of constraints to achieve a fit. This is especially useful for designs that the software cannot fit because of the constraints. The ispEXPLORER produces a spreadsheet summary of results and settings for each run, making it easy to compare one group of settings with another.

The Fitting Process

After you have entered your design, you are ready to run the Fitter. The fitting process consists of four phases. An understanding of each phase can help you choose the best corrective action if the design does not fit, or your performance criteria are not met.

- Initialization
- Optimization
- Partitioning
- Fitting (Placement and Routing)

Initialization

At the beginning of a new project, the ispLEVER software automatically copies a default constraint file from the ispLEVER directory into your project directory. For first-time users, the default settings allow most designs to achieve a First-Time Fit (FTF). For users requiring more control, the default settings can be easily changed to achieve better fitting density or performance.

You can control the contents of the constraint file using the Global Constraints dialog box and the Location Assignments dialog box.

Using the Global Constraints Dialog Box to Control Optimization

Using the Global Constraints dialog box, you can pack your design, spread your design, or use other advanced options such as specifying device utilization levels.

Using the Location Assignments Dialog Box to Pre-assign Pins and Nodes

The Location Assignments dialog box in the Constraint Editor lets you specify pin and block locations, group signals in specific blocks, or even reserve pins for later use.

Assigning Pin and Node Locations

The ispLEVER software lets you pre-assign pin and node locations. You can use the Location Assignment dialog box in the Constraint Editor to assign input and output pins and buried nodes. The Macrocell, Block, and Segment list boxes are context sensitive to the selected device; only applicable features are available.

You can also use the drag and drop feature in the Package View of the Constraint Editor to assign input, output and bi-directional pins.

Pin and Node Pre-Assignment

Pre-assigning pins lets you lay out your board at the same time as you are doing logic design, thus shortening the design cycle. Pre-assigning nodes is usually not required and is not recommended.

Pin Assignment Guidelines

If you want to pre-place signals (not recommended unless pinout configuration is important), follow these guidelines:

- Do not place large equations to macrocells or pins at the beginning or end of a block.
- Signals that share many common inputs should generally be grouped in the same block (the Partitioner does this automatically). Signals that do not share many common inputs should generally be distributed across several blocks to avoid overburdening the switch matrix for a single block.

Large Functions at the End of a Block

The macrocells at the end of a block have access to fewer product terms than other macrocells.

- Cell number 0, the first cell in all devices, can access the product term clusters from adjacent, higher-numbered cells, but it cannot access any lower-numbered cells (cell 0 being the lowest-numbered cell in the block).
- The last cell in a block can access the product term cluster from the adjacent lower-numbered cell, but it cannot access any higher-numbered cells.

If signals have not been assigned to macrocells, the Fitter will find a macrocell replacement for all the signals that satisfy their product term requirements.

Adjacent Macrocell Use

In MACH devices, adjacent macrocells can share clusters. Therefore, with designs having equations that use a high number of product terms, it is a good idea not to place them in adjacent macrocells.

Modifying Assignments

You can use the Location Assignment dialog box of the Constraint Editor to modify the current location assignment. Modifications can also be made directly in the Pin Attributes sheet of the Constraint Editor.

Deleting Assignments

You can delete project assignments via the Constraint Editor. To do this, select the entire row whose existing assignment(s) you want to delete. From the Edit menu, select **Delete Row(s)**. You may also right-click and select **Delete Row(s)**. In cases where you no longer want any of the current assignments, you can delete all of them at the same time.

Ignoring Assignments

There may be times when you want to ignore, but not delete, the current assignments. For example, after you complete a design, you may want to try fitting it into a different device. In this case, the current pin assignments may not be valid for the new device. The ispLEVER software lets you ignore current constraints for the next Fitter run.

Slew Rate Control

For the majority of Lattice devices, you can set the slew rate to either Slow or Fast. By default, the slew rate is set to Fast. However, changing it to Slow can result in less board noise.

Optimization

If your design failed to fit, or it did not meet your performance criteria, you can apply optimization procedures your design again.

Partitioning

After optimization, the design is partitioned into individual blocks on the specified device. Partitioning is achieved by assigning logic to specific blocks, based on the following considerations:

- Individual signal pre-placements and Grouping assignments
- A block's available internal resources (free macro cells, product terms, clock signals, and so forth)
- The switch-matrix interconnect resources available to the block

The Partitioner considers commonality of signals, macro cell requirements, Set/Reset requirements, product-term requirements, and other factors to determine which partition is most likely to succeed in fitting the design. Only partitions that are likely to succeed (according to the Partitioner's rules) are attempted.

Balanced Partitioning

Controlling how the Partitioner works can be very important. There is one important strategy for partitioning and that is called Balanced Partitioning. By selecting the Balanced Partitioning option in the Global Constraints dialog box, you are telling the Partitioner to spread all of the signals among all the blocks in the device, rather than trying to fill a few blocks to their maximum potential.

There are advantages to either side of the strategy. If you turn balanced partitioning on, you can save room in the device for any future functionality you might want to add to existing logic. However, turning balanced partitioning off lets you "pack" as much logic into the minimum number of blocks in the device as possible, leaving some free blocks for future design enhancements.

Place and Route (Fitting)

Placement is the assignment of physical block resources such as I/O pins, XORs, registers, and product-term clusters to logic equations. *Routing* is the assignment of switch-matrix interconnect resources to logic equations, after the logic equations are placed.

Placement

In the placement phase of the fitting process, individual equations are assigned to physical resources, as follows:

- Logic equations that have been pre-assigned to pins are assigned first.
- Buried logic functions are placed in the remaining unused macrocells.
- Inputs are assigned to any available pin. These pins can be dedicated inputs pins, clock/input pins, or I/O pins that correspond to macrocells that are either unused or used to implement buried logic functions.
- Outputs can be assigned to any unused I/O pin.

Spread Placement

Controlling how the Placer works is also important. When you select the Spread Placement option, you are telling the Placer to spread the signals in the block as far out as possible.

Routing

In the routing phase, the Fitter attempts to route input, output, and feedback signals to and from the physical resources assigned in the placement phase. If the Fitter fails to route all signals, it tries another placement. The Fitter continues trying different placements, and different routing attempts within each placement, until a successful fit is found or the time allotted for fitting is exceeded.

Fitter Options

The Global Constraints dialog box lets you set options for the Fitter. Using the Global Constraints dialog box, you can tell the Fitter to pack as much logic into the device as possible, spread the logic across the entire device, or use other advanced options such as specifying device utilization. The following sections describe these options.

Pack Design

The Pack Design option lets you pack as much logic into the device as possible. This option allows you to achieve the highest possible performance in the smallest possible device, for most designs. Each block may be completely filled, leaving less room for any design changes or logic additions.

Spread Design

The Spread Design option spreads all of the logic across the entire device rather than trying to fill each block to its maximum potential. This option allows you to achieve the highest possible performance, while leaving room for any additional functionality that you may want to add in the future. The fitter leaves room to accommodate design changes to existing logic. Because each block may be incompletely filled, the design may *or may not* require a larger device to achieve a successful fit.

Advanced Options

The advanced options let you individually control the partitioning and placement algorithms.

Balance Partitioning

The Balance Partitioning advanced option partitions the design evenly among all the blocks in the device, so each block should have the same amount of resources used. When this option is cleared, the software partitions the design block-by-block, filling up one block at a time. This means that some blocks may be filled up completely, while others may be unused.

Spread Placement

The Spread Placement advanced option places the signals evenly, or spreads them out, among macrocells in the block. Spreading out the placement lets you make minor changes to the existing output and node signals in the block. When this option is cleared, the software assigns design signals to the first available macrocell, making it easier to add new outputs or nodes to a block.

Fitter Effort

The Fitter Effort option is used to instruct the Fitter how much effort to apply to a fit. The Low option enables a faster fitting process, but will be more likely to result in failures to fit when the utilization gets higher. The High option provides the most exhaustive search of the solution space, but takes more time.

Fitter Report Formats

Two Fitter Report formats are available in the ispLEVER software, text and HTML.

- If you select the Fitter Report process associated with the target device, the Fitter Report is opened in the Output Panel of the Project Navigator or in the Report Viewer.

***Note**: By default, the ispLEVER software opens Fitter Report in the Output Panel of the Project Navigator. If you want it opened in the Report Viewer, select **Using Report Viewer** in the Log tab of the Environment Options dialog box (Project Navigator: Options > Environment).*

- If you select the HTML Fitter Report process associated with the target device, the Fitter Report is opened with your local Internet Browser.

Understanding the Fitter Report

The Fitter Report displays statistics and information on the fitting process of your design, including utilization numbers, pin assignments, etc. The Fitter Report is also written into HTML format to allow user to browse through the report easily.

The Fitter Report is divided into several sections, each briefly described below.

Project Summary

As the name implies, this section summarizes the design. It reports the name and location of the project, and the date it was fitted. This section also reports the targeted device and package, as well as the design source format.

Compilation Times

This section tells you how long it took the Fitter to fit the design in the specified device. The name for each process step is listed, as well as the total elapsed time. Prefit Time consists mainly of run-times of the design compilation and optimization phases. Total Fit Time is the total run-time of the design compilation, optimization, partition, placement and routing phases.

Design Summary

This section reports statistical information about the design, such as the number of Inputs, Outputs, Bidir Signals, Flip-flops, Registered Functions, Product Terms and Reserved Pins. It also points out the number of unique control signals in the design.

Device Resource Summary

This section lists all of the resources available within the device and how much of each resource has been used by the design. It also reports how much of each resource is still available.

GLB Resource Summary

This section lists various GLB (and segment) level resource counts, such as fan-in (or array inputs), I/O pins, input registers, macrocells, logic product terms and product term clusters.

GLB Control Summary

This section lists the totals for all control signals, and how much of each is utilized by individual GLBs.

Optimizer and Fitter Options

This section displays all of the settings that were used to fit and optimize the design. These include things such as Ignoring Constraints to the type of flip-flop synthesis you have chosen. The information in this section is set with the Constraints Options dialog box.

Pinout Listing

This section lists the I/Os and control signals on the device, and how they are assigned.

(Input, Output, Bidir, Buried) Signal List

This section reports information on individual I/Os, such as I/O type, location assignment, the fan-out, and other signal attributes.

Signals Fan-out List

This section lists signal resources and the functions they fan-out to.

GLB (GLB name) Cluster Steering Tables

This section shows information about how functions and inputs are placed in a GLB. It shows how product terms are steered to a macrocell on which a function has been placed. It also contains information about what type of control signal has been used.

GLB (GLB name) Logic Array Fan-in

This section shows how design signals are mapped to individual GLB block inputs.

Product Term Histogram

This section lists and sorts the equations according to the number of product terms they use (in the logic only).

GLB Input Histogram

This section lists and sorts the equations according to their number of inputs, which includes the logic and the ctrl signals, but not the global signals (dedicated routing).

Post-Fit Equations

This section reports the equations in your design, after fitting. It begins with a product term histogram and a GLB input histogram.

Back Annotating Assignments

You can back annotate (write) assignments from the Fitter output to the project constraint file using the Back Annotation tab on the Constraints Options dialog box. This feature lets you retain the assignments made by the Fitter so that they can be used in a future fitting process.

You can back annotate the following location and constraint assignment options:

- Pin Assignments — Only pin assignments are back annotated to the project constraint file. This option lets you retain the fitter pin assignments. All buried nodes assignments are not retained. Existing buried node assignments are removed from the project constraint file.
- Pin and GLB Assignments — Only pin and GLB assignments are back annotated to the project constraint file. The GLB assignments for the back annotated buried nodes are retained, but the associated macrocell assignments are removed. This allows the buried nodes to be placed in the same blocks, but it does not force the Fitter to use the same macrocell assignments.
- Pin, GLB and Macrocell Assignments — The pin, GLB, and macrocell assignments for the design are back annotated. This option retains all the pin and macrocell assignments.
- IO Types — You can also back annotate IO Types constraint settings.

You can only back annotate project assignments after the Fit Design process has been successfully completed. An error message appears if the ispLEVER software detects that this process did not complete successfully.

Design Verification

Verifying Designs

The ispLEVER software supports two types of timing verification: *static timing analysis* and *dynamic timing simulation*. Both of these methods support all Lattice devices.

Static Timing Analysis

Static timing analysis (timing analysis) is the process of verifying 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/hold time requirements, and the maximum frequency. Lattice has two static timing analysis tools, Performance Analyst and TRACE (for FPGAs).

The primary advantage of timing analysis is that it can be run at any time and requires no input test vectors, which can be very time consuming and tedious to create. Another major advantage of static timing analysis is that it exhaustively checks every possible input-to-output path. One shortcoming of all static timing analysis tools is that they detect false paths that will never be exercised during the course of normal operation of a circuit so that you could spend a lot time instructing the analyzer to ignore those paths. In this process, you could accidentally ignore a real issue. Although timing analysis does not give you a complete timing picture, it is an excellent way to quickly verify the speed of critical paths and identify performance bottlenecks.

Dynamic Timing Simulation

This type of analysis is based on an event-driven simulator and requires you to specify a test vector (waveform). Whereas timing analysis returns partial timing information, dynamic timing simulation (timing simulation) will give you detailed information about gate delays and worst-case circuit conditions. Because total delay of a complete circuit will depend on the number of gates the signal sees and on the way the gates have been placed in the device, timing simulation can only be run after the design has been implemented. Timing simulation also requires several input files to run.

There are two basic types of dynamic timing simulation tools, logic simulators (e.g, Lattice's Logic Simulator) and dynamic simulation analyzers (e.g., MTI's ModelSim™). Logic simulators function in a single delay mode, whereas dynamic simulation analyzers will simulate the ambiguity in delay pairs. Dynamic simulation analyzers typically take longer to process simulation results than logic simulators and considerably longer than static timing analysis tools.

Timing Verification Tools

The ispLEVER software offers timing verification with the following tools:

- Performance Analyst — Static timing analysis tool that runs timing analysis (All CPLD, ispXPLD, ispXPGA, and ispGDX2 devices except ispLSI 1K and 2K).
- Lattice Logic Simulator — Logic simulator that runs timing simulation (ispGDX and CPLD devices only).
- ModelSim for Lattice — Dynamic timing analyzer that runs timing simulation (all devices).
- TRACE – The *Timing Reporter and Circuit Evaluator* (TRACE) is an integrated flow tool that provides static timing analysis based on timing preferences (for FPGA devices only).

Additionally, timing simulation for all devices is supported with these tools:

- Text Editor — Used to create test stimulus files

- Waveform Editor — Used to create test stimulus files graphically
- Waveform Viewer — Used to view the results of simulation

Verification Environments

The timing verification tools operate in both integrated and stand-alone environments.

Integrated Timing Analysis

The Performance Analyst is a static timing analysis tool that lets you quickly determine the performance of designs implemented in any Lattice Semiconductor device. To run timing analysis, launch the Performance Analysis from the Project Navigator. The Performance Analyst traces each logical path in the design and calculates the path delays using the device's timing model and worst-case AC specs supplied in the device data sheet.

The timing analysis results are displayed in a graphical spreadsheet with source signals displayed on the vertical axis and destination signals displayed on the horizontal axis. The worst-case delay value is displayed in a spreadsheet cell if there is at least one delay path between the source and destination. To more easily identify performance bottlenecks, you can double-click a cell to view the path delay details.

Integrated Timing Simulation

To verify a design inside the current project, the ispLEVER software provides integrated verification with the Lattice Logic Simulator and ModelSim™ for Lattice from Mentor Graphics®. From the Project Navigator, you can run the appropriate process associated with the design file in the process window. When you select the verification test bench, the following processes are available in the Project Navigator Sources window. Double-click the process to automatically run the corresponding simulator.

CPLD and GDX Project Navigator Process	Simulation Tool Invoked
Timing Simulation	Lattice Logic Simulator
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

FPGA, ispXPGA, and ispXPLD Project Navigator Process	Simulation Tool Invoked
Verilog Timing Simulation	ModelSim
VHDL Timing Simulation	ModelSim

Stand-alone Simulation

The ispLEVER software supports stand-alone timing simulation. This provides an easy entry if you need to verify a design file outside the current project. Also, stand-alone operation lets you choose your own settings and preferences. Even if you have previously opened the Lattice Logic Simulator or ModelSim from a project, you can change to the stand-alone mode flexibly by selecting the appropriate simulator from the Project Navigator Tools menu.

Required Files for Verification

The Lattice Logic Simulator and ModelSim support Lattice ispGDX and CPLD device timing simulation. In addition to your design files, you will need at least one test stimulus file, a netlist file, and a timing delay file.

	Lattice Logic Simulator	ModelSim
Test Stimulus File Formats		
• **.wdl** (Graphic waveform)	X	
• **.abv/.abl** (Test vector)	X	
• **.tf** (Verilog test fixture)		X
• **.vhd** (VHDL test bench)		X
Netlist File Formats		
• **.vo** (Verilog netlist)		X
• **.vho** (VHDL netlist)		X
• **.edo/.sim** (EDIF/simulation)	X	
Delay File Formats		
• **.sdf** (Standard Delay File)	X	X

Verification File Descriptions

Test Stimulus Files

- Graphic Waveforms (`.wdl`) — A file created by the Waveform Editor file that graphically represents a waveform as a sequence of signal states separated by time intervals.
- Test Vectors (`.abv/.abl`) — Test vectors are sets of input stimulus values and corresponding expected outputs that can be used with both functional and timing simulators. Test vectors can be specified either in a top-level ABEL-HDL source or in a separate ABEL-HDL test vector format (`.abv`) file. The ABV file is considered a text document and is kept above the device level in the Sources window. Whether the test vectors are part of a top-level ABEL-HDL source (`.abl`) or are in a separate file, they will be compiled and passed to the simulator.
- Verilog Test Fixtures (`.tf`) — A Verilog test stimulus file that specifies the input waveforms for simulation in ASCII format.
- VHDL Testbench (`.vhd`) — A VHDL test stimulus file that specifies the input waveforms for simulation in ASCII format.

Netlist Files

- Verilog Netlist (`.vo`) — For Verilog designs, the back-annotated timing simulation netlist is named `<design_name>.vo`.
- VHDL Netlist (`.vho`) — For VHDL designs, the back-annotated timing simulation netlist is named `<design_name>.vho`.
- EDIF/Simulation Netlist (`.edo/.sim`) — For EDIF designs, the back-annotated timing simulation netlist is named `<design_name>.edo`.

Timing Delay Files

- Standard Delay Format (`.sdf`) — A file containing delay and timing constraint data for cell instances named `<design_name>.sdf`.

Generating Timing Simulation Files

After you have fit the design, the ispLEVER software lets you export the netlist and delays for timing simulation. For netlist files, ispLEVER supports VHDL, EDIF, and Verilog formats. For timing delay files, ispLEVER supports the standard SDF and Viewlogic DTB timing formats.

To choose a simulation file format:

1. In the Project Navigator, choose **Tools > Generate Timing Simulation Options** to open the dialog box.
2. Select the format options that you want.
   - For netlist formats, you have a choice of Verilog, VHDL, or EDIF (Version 2.0.0). If you choose the EDIF format, you can customize the Power/Ground representation by selecting either Cell or Net.
   - For timing format, SDF (version 2.1) and Viewlogic DTB format are supported.
3. Click **OK** to close the dialog.
4. When you run the **Generate Timing Simulation Files** process, ispLEVER generates the files in the specified formats.

Viewing the Simulation Input Files

You can use the Report Viewer to view simulation input files.

1. In the Project Navigator Sources window, select the target device.
2. In the Project Navigator Processes window, double-click **Report File**. Notice the two output files (Netlist and Delay) listed at the top of the report. The ispLEVER software generates these files and places them in the project directory.
3. To view these files, choose **File** > **View** and select the file that you want to view.

***Note**: You cannot modify files using the Report Viewer. You can only view them.*

ispGDX Verification Summary

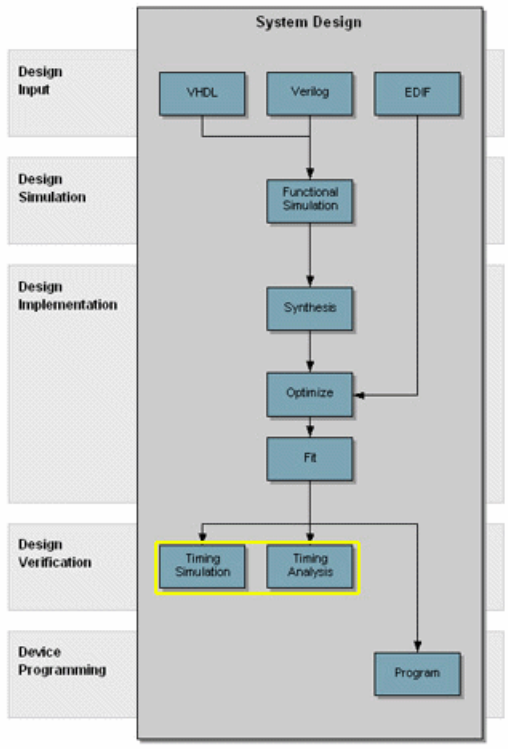
The Lattice Logic Simulator supports ispGDX timing simulation for all design entry methods and requires at least one graphic waveform (`*.wdl`) or test vector (`*.abv`) stimulus file.

The ModelSim simulator supports timing simulation for all design entry methods and requires at least one Verilog test fixture (`*.tf`) or VHDL testbench (`*.vhd`) stimulus file. Additionally, ModelSim requires a netlist file (`*.vo` or `*.vho`) and a timing delay file (`*.sdf`).

	Lattice Logic Simulator		ModelSim	
	.wdl	.abv	.tf	.vhd
			.vo + .sdf	.vho + .sdf
EDIF	X	X	X	X
Verilog	X	X	X	
VHDL	X	X		X

ispGDX Verification Process Flow

The figure below shows timing analysis and simulation within the ispGDX process flow.



Device Programming

Programming Devices

Lattice supports device programming for all programmable logic devices with the following tools, which are briefly described below and covered in detail in their respective Help. They are listed in alphabetical order.

ispVM System

The ispVM™ System software (ispVM) supports both serial and concurrent (turbo) programming of all Lattice devices in a PC environment. The ispVM System software is built around a graphical user interface. Device chains can be scanned automatically. Any required JEDEC ISC or Bitstream data files are selected by browsing with a built-in file manager. Non-Lattice devices that are compliant with IEEE 1149.1 can be bypassed once their instruction register length is defined in the chain description. Programmable devices from other vendors can be programmed through the vendor-supplied SVF file. See the ispVM System Help for more information about this tool.

Model 300 Programmer

The ISP Engineering Kit Model 300 programmer is an engineering device programmer that supports prototype development by allowing single-device programming directly from a PC. The Model 300 programmer supports all JTAG devices produced by Lattice, with device Vcc of 1.8, 2.5, 3.3, and 5.0V. See the Model 300 Programmer Help for more information about this tool.

SVF Debugger

The SVF Debugger can be used with ispVM System software to help you debug a Serial Vector Format (SVF) file. The SVF Debugger software allows you to program a device, and then edit, check syntax, debug and trace the process of an SVF file. See the SVF Debugger Help for more information about this tool.

Universal File Writer

The Universal File Writer (UFW) is a separate application that generates bitstream files or an SVF data file for a single device. Using JEDEC or ISC files, the software generates bitstream PCM, Intel Hex and Motorola Hex data files concurrently. It can also generate an SVF file using the parameters you select. You can run the Universal File Writer from the ispVM System toolbar or separately. See the Universal File Writer Help for more information about this tool.

Running ispLEVER from the Command Line

Running from the Command Line

You can run the ispLEVER software from the command line on PC and UNIX using **ispflow** command line software. The ispflow software will attempt to fit the design to the specified part.

***Note**: All examples are shown in PC format. For UNIX, use forward slash instead of back slash.*

Syntax

The format of the command is as follows (on one line):

```
ispflow [-i <design>] [-d <device>] [-imp <yes\no>]
[-sdf <edif\verilog\vhdl\off>]
[-edf <mentor\synopsys\synplicity\viewlogic>]
[-syn <spectrum\synplify>] [-h] [-v]
```

where [ ] denotes optional parameters.

Definitions

-i <design name>	EDIF, Verilog, or VHDL design name. The name must include the appropriate file extension in the design name, such as `.edf`, `.v`, or `.vhd`.
-d <device name>	Specifies the device part number that the design will be fitted to. For example, `LFX125B-04F256C`. This option is required if the `<design name>.lci` file does not exist. After running **ispflow**, the specified device will appear in the newly created LCI file.
-imp <yes/no>	Import source constraints. Default is Yes.
-sdf <edif>	Outputs an SDF file in EDIF format.
-sdf <verilog>	Outputs an SDF file in Verilog format.
-sdf <vhdl>	Outputs an SDF file in VHDL format. Default.
-sdf <off>	Switch off SDF output.
-edf <mentor>	Specifies a Mentor Graphics-generated EDIF file. Default.
-edf <synopsys>	Specifies a Synopsys-generated EDIF file.
-edf <synplicity>	Specifies a Synplicity-generated EDIF file.
-edf <viewlogic>	Specifies a Viewlogic-generated EDIF file.
-syn <spectrum>	Specifies a LeonardoSpectrum synthesize file. Default.
-syn <synplify>	Specifies a Synplify synthesize file.
-spd <yes/no/fmax>	Speed: yes (speed), no (area), fmax.
-mpts	Max_PTerm_Split value.
-mptc	Max_PTerm_Collapse value.
-mptl	Max_PTerm_limit value.
-mfan	Max_fanin value.
-msym	Max_symbols value.
-fmll	Fmax_Logic_Level value.
-svf <on/off>	Switch on SVF generation. Default is off.
-r <*.lct/*.lci>	Refit using the constraint file.

-c <yes/no>	Specifies to run vcick (vc checker). Default is no.
-h	Help.
-v	Displays version number

The input source file switch (**-i**) is mandatory. All other switches are not.

If a device name (**-d** <device name>) is specified from the command line, and a Lattice Constraint File (.lci) file exists, the device specified from the command line takes precedence. The **ispflow** software will not run if a device name is not specified from either the command line or in a LCI file.

Specifying Options and Pin Assignments

To specify optimization options and pin assignments, you must create or modify a <design_name>.lci file. See Lattice Constraint File Description for more information on the LCI file format. If there is no LCI file, the software creates a default file for the specified device. In this case, the **-d** option is required.

***Note 1**: The LCI file is case-sensitive. Once an LCI file is created, the –d option can be omitted from the command line, because the device information will be obtained from the LCI file. The –d option takes precedence over the LCI file. If the –d option is used again with a different device part number, the device information part of the LCI file will be updated to reflect the changes.*

***Note 2**: Pin assignments and certain optimization options in the LCI file could be incorrect if the device is changed on the Command Line.*

The ispflow software runs through the complete flow, including timing analysis.

Input Formats

Acceptable input formats are non-hierarchical EDIF, VHDL, and Verilog HDL source files. The source files must be named with the .edf, .vhd, or .v extensions respectively.

Log Files

A log file of the process will be generated named <*design_name*>.batch.log file.

Batch Mode Example

The following example is for fitting a new design.

***Note:** The example is shown in PC format. For UNIX, use forward slash instead of back slash.*

To fit a new design:

1. Create a new directory and copy the input files needed.

   <isptools>\ispcpld\examples\ispgdx\ispgdx\verilog\a2bexch\a2bexch.v

   <isptools>\ispcpld\examples\ispgdx\ispgdx\verilog\a2bexch\a2bexch.lci

2. Run ispflow -i a2bexch.v.

   If you have a Lattice Constraint File (<*design_name*>.lci), the software will get the device from the LCI file and fit the design. The LCI file can be varied to an optimizer setting that you prefer. See the Lattice Constraint File Description for additional information.

OR

3. If you do not have a LCI file, run `ispflow -i a2bexch.v -d ispGDX120A-5T176`. The software fits the design into the specified ispGDX device.

Retaining Pin Assignments Using Batch Mode

You can retain pin assignments by copying LOCATION ASSIGNMENTS from your output Lattice Constraint File (`.lco`) into your input Lattice Constraint File (`.lci`).

To retain pin assignments using batch mode:

1. In the project directory, using a text editor, open the *<design_name>*`.lco` file.
2. In the LCO file, copy the LOCATION ASSIGNMENTS section.
3. In the project directory, using a text editor, open the *<design_name>*`.lci` file.
4. Replace the LOCATION ASSIGNMENTS section in the LCI file with the LOCATION ASSIGNMENTS section you copied from the LCO file.

LCI Files

The Lattice Constraint File (`.lci`) contains the constraints for Part selection as well as the Optimization and Placing and Routing processes.

You do not need to generate a LCI file on the first fit. If you specify a device (`-d`) with ispflow, this will generate a LCI file.

Command Line FAQs

The following are Frequently Asked Questions about processing designs in Command Line Mode using ispflow software.

Q. What report files are available to view after processing a design using Command Line mode?

A. Using a text editor, you can view the Timing Report File (`.trp`) and the Fitting Report File (`.rpt`) in the project directory after command line batch mode processing.

The RPT file displays details about how the current design was fit into the target device. If a fit was not accessible, the RPT file explains the problems preventing a fit. The file is located in your project directory, and is named *<design_name>*`.rpt`

The TRP file contains all information for stamp model generator. The file is located in your project directory, and is named *<design_name>*`.trp`.

Q. Where do I find my programming (JEDEC) file?

A. The programming file can be found in the project directory after command line batch mode processing. The file is located in your project directory, and is named *<design_name>*`.jed`.

Q. Where do I find my back-annotated timing simulation netlist?

A. The back-annotated timing simulation netlist is located in the project directory. The extension of the netlist depends upon the source files used to process your design.

- For VHDL designs, the back-annotated timing simulation netlist is named *<design_name>*`.vho`.
- For Verilog designs, the back-annotated timing simulation netlist is named *<design_name>*`.vo`.

- For EDIF designs, the back-annotated timing simulation netlist is named <*design_name*>.edo.

Q. Where do I find my timing delay file?

A. The timing delay file can be found in the project directory after command line batch mode processing. The file is located in your project directory, and is named <*design_name*>.sdf.