Contents

1. Introduction ............................................... 1-1
   HDL Solution ............................................... 1-1
   LeonardoSpectrum Modular Levels .......................... 1-2
      Level 1 ................................................. 1-2
      Level 2 ................................................. 1-3
      Level 3 ................................................. 1-4
   Options ................................................ ' 1-4
      Level 3 - DesktopASIC .................................. 1-5
   All Three Levels ......................................... 1-5
      LeonardoInsight ....................................... 1-5
      HDL Languages ......................................... 1-6
   Standard Features ....................................... 1-6
      Save and Restore Project ............................... 1-6
      P&RIntegrator .......................................... 1-7
      HDL Inventor .......................................... 1-7
      Design Browser ....................................... 1-8
   Still More Features .................................. 1-8
      This LeonardoSpectrum Synthesis and Technology Manual 1-8
      Available Online and Website .......................... 1-8
      Available Online Context-Sensitive Help .............. 1-8
      Available Libraries .................................. 1-9
      Tcl Script Sourcing .................................... 1-9
      XlibCreator for ASIC - Complete Development Kit ... 1-9
      Screen Shots, Reports, Filenames, and Code Examples 1-10
   Three Ways to Synthesis .................................. 1-10
      SynthesisWizard ...................................... 1-10
      Quick Setup ........................................... 1-10
      FlowTabs ............................................... 1-10
   Conventions Used ....................................... 1-11
   PC Hardware and Software Requirements .................. 1-11
   PackagedPower ........................................... 1-12
2. Technology Mapping ............................................... 2-1
   Before Beginning ............................................. 2-2
   Boolean Mapping ............................................. 2-2
   Lookup Table Mapping ....................................... 2-2
   Global Buffers .............................................. 2-6
   I/O Mapping .................................................. 2-6
   Manual I/O Mapping ........................................... 2-8
       Setting Timing Constraints with Constraint Editor ...... 2-8
       PAD and BUFFER_SIG Commands (Attributes) ............ 2-8

3. Timing Analysis .................................................. 3-1
   Before Beginning ............................................. 3-1
   Static Timing Analysis ....................................... 3-1
       Delay Models .............................................. 3-2
       Timing Analysis Before and After Place-and-Route ...... 3-5
   Defining Process, Temperature, and Voltage Values ...... 3-5
   Critical Path Report ......................................... 3-6
   Back-Annotation .............................................. 3-7
       SDF (Standard Delay Format) Writer ....................... 3-7
       Design Flow .............................................. 3-7
       VITAL Libraries .......................................... 3-8
       SDF for Transporting Delays ............................ 3-10
   Post Synthesis RTL (Register Transfer Level) Simulation . 3-10

4. Timing Constraints ............................................... 4-1
   Before Beginning ............................................. 4-1
   Setting Global Constraints .................................. 4-1
   Setting Constraints on Individual Signals .................. 4-3
       Load and Drive Specifications ........................ 4-3
   Timing Specifications ...................................... 4-4
   More Constraint Features ................................... 4-7
       Constraint Driven Timing Optimization ................. 4-7
   Support for Maximum Frequency ............................ 4-8
   Support for Multicycle Path ............................... 4-8
5. Timing Optimization ........................................ 5-1
   Before Beginning .............................................. 5-1
   Optimization Flow .......................................... 5-1
   Delay Optimization ......................................... 5-2
   Setting Timing Constraints ................................. 5-4
      User Control ............................................... 5-5
   Statistics Report ........................................... 5-6
   Automatic Constraining of Designs ....................... 5-15

6. ASIC Synthesis - Level 3 ................................. 6-1
   Before Beginning .............................................. 6-1
   Design Flows ................................................ 6-1
      Hierarchy .................................................. 6-3
      Design Partitioning ...................................... 6-5
   Optimization Features ...................................... 6-5
      Data Path Synthesis ..................................... 6-6
      ASIC Modgen .............................................. 6-6
      Adding an ASIC Library to the GUI ..................... 6-8
      Technology Mapping ...................................... 6-9
      Constraint Driven Timing Optimization ............... 6-10
      Gate Sizing ............................................... 6-10
   Timing Analysis .............................................. 6-11
      Issues in Delay Modeling ................................ 6-11
      Modeling Interconnects .................................. 6-13
      Reporting .................................................. 6-14
   DRC Resolving ................................................. 6-14
      Max Fanout on the GUI .................................... 6-20
      New ASIC Global Variable ................................ 6-21
      FSM Encoding (binary, gray, random, onehot, twohot, auto) 6-21

7. Actel Synthesis .............................................. 7-1
   Before Beginning .............................................. 7-2
   Actel FPGA Architecture .................................... 7-2
   Actel Logic Module ........................................... 7-2
10. Altera MAX Synthesis .......................... 10-1
Before Beginning .................................. 10-1
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using ROMs</td>
<td>11-27</td>
</tr>
<tr>
<td>Mapping to Synchronous Set/Reset Registers</td>
<td>11-28</td>
</tr>
<tr>
<td>Reporting</td>
<td>11-29</td>
</tr>
<tr>
<td>Generation of ORCA Foundry Properties</td>
<td>11-29</td>
</tr>
<tr>
<td>Pin Number Property - LOC</td>
<td>11-29</td>
</tr>
<tr>
<td>Slew Property</td>
<td>11-30</td>
</tr>
<tr>
<td>ORCA DIN/DOUT Attributes for ORCA 2CA/2TA</td>
<td>11-30</td>
</tr>
<tr>
<td>Part Number - PICSPEC</td>
<td>11-32</td>
</tr>
<tr>
<td>Speed Grade - PISCED</td>
<td>11-33</td>
</tr>
<tr>
<td>ORCA Foundry Properties on IOBs</td>
<td>11-33</td>
</tr>
<tr>
<td>NODIN/NODOUT Attributes for ORCA 3C/3T</td>
<td>11-33</td>
</tr>
<tr>
<td>ORCA Preference File Writer</td>
<td>11-34</td>
</tr>
<tr>
<td>ORCA3C and ORCA3T Multicycle Support</td>
<td>11-35</td>
</tr>
<tr>
<td>Additional Example for Logical Preference File</td>
<td>11-36</td>
</tr>
<tr>
<td>ORCA Field-Programmable System Chips (FPSC)</td>
<td>11-41</td>
</tr>
<tr>
<td>Writing Primitives in \texttt{noopt} (Attribute) Instances</td>
<td>11-41</td>
</tr>
<tr>
<td>Propagating \texttt{nopad} (Attribute) to the Top Level</td>
<td>11-41</td>
</tr>
<tr>
<td>Devices Supported for Lucent ORCA FPGA Families</td>
<td>11-43</td>
</tr>
<tr>
<td>Additional Features for ORCA</td>
<td>11-45</td>
</tr>
<tr>
<td>Schematic Symbol Libraries</td>
<td>11-46</td>
</tr>
<tr>
<td>Mapping to ORCALUTS</td>
<td>11-46</td>
</tr>
<tr>
<td>ORCA Wire Load Model</td>
<td>11-46</td>
</tr>
<tr>
<td>Pipelined Multiplier</td>
<td>11-46</td>
</tr>
<tr>
<td>More New Features Supported for ORCA</td>
<td>11-46</td>
</tr>
<tr>
<td>12. QuickLogic Synthesis</td>
<td>12-1</td>
</tr>
<tr>
<td>Before Beginning</td>
<td>12-1</td>
</tr>
<tr>
<td>FPGA Architecture</td>
<td>12-2</td>
</tr>
<tr>
<td>pASIC Logic Cell</td>
<td>12-2</td>
</tr>
<tr>
<td>I/O Buffers</td>
<td>12-2</td>
</tr>
<tr>
<td>Features</td>
<td>12-2</td>
</tr>
<tr>
<td>Design Flow</td>
<td>12-2</td>
</tr>
</tbody>
</table>
Optimization Style Points - Quality ........................................ 12-3
Combinational Logic Loops ................................................. 12-4
Sequential Optimization ..................................................... 12-4
Internal Tristates ............................................................. 12-4
Reporting ........................................................................ 12-4
Process Derating Factors .................................................... 12-5
Design I/O ........................................................................ 12-5
  Manual Assignment of I/O Buffers ....................................... 12-5
  Assigning Pin Locations .................................................. 12-7
  Max Fanout .................................................................... 12-8
Devices Supported ............................................................. 12-9

13. **Xilinx CPLDs Synthesis** ................................................. 13-1
  Before Beginning ............................................................ 13-1
  Xilinx CPLD Architecture .................................................. 13-2
  Design Flow ................................................................... 13-2
  Synthesis Features .......................................................... 13-3
    Device Selection .......................................................... 13-3
    I/O Buffers and Pads ..................................................... 13-4
    Instantiating 0BUFs ..................................................... 13-5
    Pin (LOC) Assignment ................................................... 13-6
  Module Generation ........................................................... 13-6
  Reporting ...................................................................... 13-7
  Xilinx CPLD Family Supported Devices ............................... 13-7

14. **Xilinx LCA Synthesis** ................................................. 14-1
  Before Beginning ............................................................ 14-2
  Xilinx LCA Architecture ................................................... 14-2
     Configurable Logic Block .............................................. 14-2
     Input/Output Block ..................................................... 14-2
     Synthesis Design Flows .............................................. 14-2
     Design Flow Guidelines ............................................. 14-5
  Synthesis and Optimization Features ................................. 14-7
     Fanin Limited Optimization ........................................ 14-7
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup Table (LUT) Mapping</td>
<td>14-10</td>
</tr>
<tr>
<td>Constraint-Driven Timing Optimization</td>
<td>14-12</td>
</tr>
<tr>
<td>CLB Packing for Xilinx XC4000 Technology</td>
<td>14-13</td>
</tr>
<tr>
<td>XC4000 CLB Architecture Constraints</td>
<td>14-13</td>
</tr>
<tr>
<td>Data Path Synthesis and Modgen Implementation</td>
<td>14-18</td>
</tr>
<tr>
<td>Supported Operators</td>
<td>14-18</td>
</tr>
<tr>
<td>User Defined Options and Variables for Modgen</td>
<td>14-19</td>
</tr>
<tr>
<td>GSR Usage and Startup Blocks</td>
<td>14-20</td>
</tr>
<tr>
<td>XNF GSR Specific</td>
<td>14-22</td>
</tr>
<tr>
<td>Using Edge Decoders</td>
<td>14-26</td>
</tr>
<tr>
<td>Using Internal Tristate Buffers</td>
<td>14-28</td>
</tr>
<tr>
<td>Using Clock Enable</td>
<td>14-32</td>
</tr>
<tr>
<td>Using RAMs</td>
<td>14-34</td>
</tr>
<tr>
<td>Using ROMs</td>
<td>14-43</td>
</tr>
<tr>
<td>Using Global Clock Buffers</td>
<td>14-44</td>
</tr>
<tr>
<td>User Options for Global Buffers</td>
<td>14-48</td>
</tr>
<tr>
<td>Design I/O</td>
<td>14-49</td>
</tr>
<tr>
<td>Complex I/O Mapping</td>
<td>14-49</td>
</tr>
<tr>
<td>Pin Location Assignment</td>
<td>14-52</td>
</tr>
<tr>
<td>VHDL Attributes</td>
<td>14-52</td>
</tr>
<tr>
<td>Interactive Command Line Shell</td>
<td>14-54</td>
</tr>
<tr>
<td>Reporting</td>
<td>14-54</td>
</tr>
<tr>
<td>Using Timespecs</td>
<td>14-54</td>
</tr>
<tr>
<td>Specify Timespec Attributes</td>
<td>14-55</td>
</tr>
<tr>
<td>TIMESPECs</td>
<td>14-56</td>
</tr>
<tr>
<td>Using Xilinx Attributes</td>
<td>14-61</td>
</tr>
<tr>
<td>Additional Xilinx Specific Options</td>
<td>14-63</td>
</tr>
<tr>
<td>Using Enabled D Type Flip-Flops</td>
<td>14-63</td>
</tr>
<tr>
<td>Pipelined Multiplier</td>
<td>14-63</td>
</tr>
<tr>
<td>Use of Registered Logic in IOBs</td>
<td>14-63</td>
</tr>
<tr>
<td>Part Number</td>
<td>14-64</td>
</tr>
</tbody>
</table>
Writing XNF and EDIF .................................................. 14-64
Using Xilinx Designs as Input ........................................... 14-67
  Design Restrictions .................................................. 14-67
  XNF File Constructs ............................................... 14-68
Preserving Logic in an XNF Netlist .................................. 14-68
NOOPT ............................................................................... 14-68
I/Os in XNF ........................................................................ 14-71
External Signals .............................................................. 14-75
Interfacing with XBLOX ..................................................... 14-75
Additional Xilinx Architecture Features .............................. 14-76
Xilinx Coregen Support .................................................... 14-76
Modgen Enhancements ....................................................... 14-77
  Non-Square Multipliers .................................................. 14-77
  Improved Operators ....................................................... 14-77
  FSM Encoding (binary, gray, random, onehot, twohot, auto) .... 14-77
Xilinx Libraries ............................................................... 14-78
RAMs .............................................................................. 14-78
Modgen Libraries .............................................................. 14-78
Existing Modgen Library .................................................. 14-78
Components for Modgen Libraries ....................................... 14-78
Fast I/O Buffers .............................................................. 14-78
Global Clock Buffers ........................................................ 14-78
Quality of Area and Delay Results ..................................... 14-79
Fast Xilinx Outputs ........................................................ 14-79
M1 Back Annotation Flow .................................................. 14-80
SDF (Standard Delay Format) Writer .................................. 14-81
M1 Support ........................................................................ 14-81
M2 Support ........................................................................ 14-81
Accurate Delay Estimations .............................................. 14-82
Sophisticated CLB packing - Level 3 ................................. 14-82
15. Xilinx Virtex ................................................. 15-1

Before Beginning ........................................... 15-1
Virtex Features ............................................. 15-1
  Wide Cluster of MUXs mapping ......................... 15-2
  Priority Encoder ........................................ 15-2
  VHDL Example for MUXCY Mapping ..................... 15-2
  Synchronous Set/Reset .................................. 15-3
  Support for Virtex IOB Registers ....................... 15-3
  Tristate Register Replication ......................... 15-4
Support for SRL ............................................ 15-5
  Variables .............................................. 15-5
  VHDL Example SRL Design .............................. 15-5
  Verilog Example SRL .................................. 15-6
Block RAMs for Virtex ..................................... 15-7
  Variables .............................................. 15-7
  Verilog Example: ..................................... 15-7
  VHDL Example of Block RAMs ......................... 15-8
Virtex Single Port RAMs ................................... 15-9
  Verilog Coding Example: .............................. 15-9
  Verilog Coding Example (not used): .................. 15-10
  Variables ............................................. 15-10
GSR Processing ............................................. 15-10
CLKDLL ..................................................... 15-10
  Constraint propagation for Virtex CLKDLL .......... 15-15
Virtex Devices Supported ................................ 15-16
VirtexE Devices Supported ............................................. 15-16
Virtex Spartan2 Devices Supported .................................... 15-17
Additional Information .................................................. 15-17
    FSM Encoding (binary, gray, random, onehot, twohot, auto) ........ 15-17
    Virtex Report ........................................................ 15-18
    Xilinx Coregen Support ............................................ 15-18
Virtex Flow Diagram ................................................... 15-18

16. Additional Supported Technologies. .............................. 16-1
    Atmel .............................................................. 16-1
    Cypress ............................................................ 16-1
    Lattice/Vantis ...................................................... 16-2
        Devices Supported ............................................ 16-2
    Minc ............................................................... 16-3
Welcome to LeonardoSpectrum. LeonardoSpectrum is a suite of high level design tools for a Complex Programmable Logic Device (CPLD), Field Programmable Gate Array (FPGA), or Application Specific Integrated Circuit (ASIC). LeonardoSpectrum offers design capture, VHDL and Verilog entry, register transfer level debugging for logic synthesis, constraint based optimization, timing analysis, encapsulated place-and-route, and schematic viewing. This introduction is divided as follows:

- HDL Solution
- LeonardoSpectrum Modular Levels
- Options
- Standard Features
- Still More Features
- Three Ways to Synthesis
- Conventions Used
- PC Hardware and Software Requirements
- PackagedPower

**HDL Solution**

A complete hardware description language (HDL) solution is here for Windows 95/98/NT; and UNIX HP and Sun. A quality design is completed for you with each of the LeonardoSpectrum tools: Level 1, Level 2, and Level 3. A native Windows graphical user interface (GUI) is common to all three levels and provides these features:
• Same look and feel for all levels
• Windows editing, dragging, and dropping attributes are available
• SynthesisWizard, Quick Setup, and FlowTabs guide you through the design process
• Embedded, interactive, and filtered windows extend task information
• Quick file changes with right mouse button (RMB)
• Popups and pulldowns are prevalent
• Pertinent information is parsed for quick reading
• Clickable buttons assign tasks

LeonardoSpectrum Modular Levels

This section compares and contrasts the three modular tool levels. All three levels are powered by the LeonardoSpectrum core synthesis and optimization engine which yields superior design results with a minimum of tool manipulation; and, at the same time, allows you to control the design domain.

As described in the following paragraphs, the design methodology becomes more detailed with each successive level: Level 1 produces the basic netlist, Level 2 adds more intricate design capabilities; and Level 3 contributes the ultimate in interactive, advanced features.

Level 1

Level 1 is an easy-to-use, single FPGA technology, synthesis tool that uses the LeonardoSpectrum database. A logic designer selects the input design and Xilinx technology (for example) and then clicks the Run button. A high-quality netlist is quickly produced. Level 1 includes the following clearly defined features:
• Windows 95/98/NT node-locked platform
• Single FPGA vendor
• Certified FPGA flows
• Global constraints (frequency, period, input to register, register to register, register to output, input to output)
• Technology-independent specifications
• Technology-specific operator generation
• Architecture-specific optimization
• Place and route sub-invocation (P&RIIntegrator) for Xilinx, Altera, Quartus, and Lattice/Vantis technologies.
• Integrated source code editor (HDLInventor) including template insertion and error/warning cross highlighting
• Single pass area and timing optimization
• Familiar Batch Mode
• TCL Scripting
• Easy upgrade path to Levels 2 and 3

Level 2

Level 2 is an easy-to-use FPGA synthesis and timing analysis device with back-annotation for all FPGA technologies. A logic designer selects the input design and technology and clicks the Run button. A high-quality netlist is quickly produced. In contrast to Level 1, Level 2 is for all FPGA technologies. Level 2 contributes to the extensive features list of Level 1 with the following:

• Platform independent on Windows 95/98/NT or UNIX
• All FPGA technologies
• Certified FPGA flows: Generated netlists and directives successfully pass through the back-end tools; post place-and-route timing information can then be back-annotated for timing analysis, logic verification, and technology retargetting purposes.
• Hierarchy Preservation
• Advanced Constraints
• Advanced Optimization Switches
• Retarget Output Netlist
• Accurate architecture specific timing analysis
• Optimizes designs for area and speed, and accepts designs as either HDL structural netlists or as RTL (register transfer level)
• Vendor specific netlists are produced together with design reports that provide estimates of design performance
• Save and Restore Project
• Back annotated timing analysis from post place-and-route netlist
• Easy upgrade path to Level 3
Level 3

Level 3 is easy-to-use and is a versatile and interactive logic synthesis, optimization, and analysis tool. Level 3 allows the use of technology-independent design methods for FPGA and CPLD devices, and in contrast to Levels 1 and 2, Level 3 optionally supports advanced algorithms to target ASIC technologies. You can perform bottom-up design assembly with technology-mapped netlist. Hierarchy can be preserved, flattened, merged and dissolved. Plus, complex scripts can be written and run through an interactive batch mode operation. The design effort can be accomplished either by an individual engineer or by a team of engineers.

Level 3 utilizes the most powerful state-of-the-art optimization technology to guarantee high-quality results for any FPGA or ASIC technology. Level 3 adds to the long list of Level 2 features with the following:

- Optional ASIC specific module generation, optimization algorithms, design rule resolving, and technology mapping
- Two-way retarget path exists between FPGA synthesis and optional ASIC synthesis
- Mix HDL Design entry - for example, Verilog, VHDL, EDIF
- Interactive Command Line Shell
- Advanced TCL Scripting
- Incremental optimization which allows bottom up, top down, and team design
- RTL and gate-level post-synthesis verification
- Design Browser with commands

You can run Level 3 from the GUI interactive command line shell or on TCL script files. Batch mode is also available for Level 3. LeonardoSpectrum is designed to give you easy access to the Model Technology Model Sim/QuickHDL simulator.

Level 3 provides a top-down verification flow through VHDL or Verilog with an SDF timing file. LeonardoSpectrum is fully integrated with Model Technology, Inc. (MTI) simulation environment.

Options

Options are available as follows:

- Level 3 - DesktopASIC
- All Three Levels - LeonardoInsight and HDL Languages
Level 3 - DesktopASIC

LeonardoSpectrum, Level 3, is available as ASIC only, or is available as an ASIC option to FPGA. The ASIC library tree is added to the technology browser. The ASIC synthesis flow is improved dramatically to handle high-density designs and to improve run time and to reduce memory consumption. New algorithms are present in the ASIC flow for optimization, mapping and design rule checker (DRC). LeonardoSpectrum uses the same methodology for both FPGAs and ASICs. You can use LeonardoSpectrum to prototype an ASIC using FPGAs, or retarget an FPGA to ASIC or ASIC to FPGA for volume production.

All Three Levels

LeonardoSpectrum has two options for all three levels:
- LeonardoInsight
- HDL Languages (VHDL and Verilog)

LeonardoInsight

LeonardoInsight is here to bring the design database into view. LeonardoInsight includes the design browser and schematic viewer. LeonardoInsight allows you to simplify the complexities of synthesis with an advanced debug and analysis environment.

Design Browser

The design browser displays ports, nets, instances, registers, and primitive cells. Interactive and filtered windows of the design browser are available on the GUI after you read in your design. In summary, the design browser is a graphical representation of the design database. Objects selected and highlighted in the design browser may also be highlighted in the schematic viewer. Furthermore, if the selected object initiates cross probing, then that line of code is highlighted in your HDL source code.

Note: The design browser can be used to apply dont_touch, unmap, unfold, group, ungroup, and ungroup_all commands; group and unfold are only available for Level 3.

Note: The design browser is a standard feature for Level 3. In addition, the design browser is available to Levels 1 and 2 from the LeonardoInsight option.
Schematic Viewer

The LeonardoSpectrum schematic viewer - based on the latest in rendering algorithms - produces clear and well-organized schematics. The schematic viewer allows you, for example, to: (1) Cross-probe between HDL source code, RTL schematic, and gate-level schematic. This correlation allows for easy debugging. In addition, you can cross probe a schematic generated in Renoir with a schematic generated in LeonardoSpectrum. (2) You can view the whole critical path in one window, even if the path traverses multiple levels of hierarchy. (3) You can view fanout and fanin cones of logic from a selected net or instance. (4) When the critical path viewer is in query mode, detailed timing popup information is displayed for the objects in the critical path. (5) Query mode provides general popup information for every schematic. (6) The schematic viewer search utility allows you to search for instance, net, and port; and lists these items for you in a window. (7) The schematic viewer can cross probe with Renoir.

HDL Languages

By default you are provided with either the Verilog or VHDL language. You can add either Verilog or VHDL as a second language. Level 3: refer to Special Instructions for Mixing Design Languages in the User’s guide.

Standard Features

The standard features allow you to complete the entire synthesis task within LeonardoSpectrum. These features are:

- Save and Restore Project (Levels 2 and 3)
- P&RIntegrator
- HDLInventor
- Design Browser (Level 3)

Save and Restore Project

Entire design projects can now be restored on the same or a different machine. Before you quit a design, you are prompted to save the entire project. Later you can go back and bring up the discontinued project; the restored project is complete with your specifications and windows environment.
Project Saves and Restores:

- File locations for input files, output files, and current working directory
- Database (RTL, gate level) in XDB format (Level 3)
- Present design information
- Applied constraints, directives, and attributes
- All tab selection information: source technology, designation technology, file type, hierarchy preservation, global constraints, optimization passes, FSM encoding.

Note: Within the v1999.x series your saved project files are forward and backward compatible, plus you can read v1998.x project files in v1999.x. However, if you want to read v1999.x project files in v1998.x then set the following variable in v1999.x: xdb_write_version v1998.x

P&RIntegrator

P&RIntegrator automatically subinvokes vendor backend place and route tools: Xilinx Alliance Series, Altera MAX+PLUS II and Quartus; and Lattice/Vantis from within LeonardoSpectrum. The vendor’s backend tools then create a binary program file which is used to program FPGA devices.

LeonardoSpectrum is the only industry synthesis tool that interfaces directly to selected FPGA and CPLD place and route tools for optimal results. Moreover, LeonardoSpectrum supports back-annotated timing analysis for many vendors. For example, post-routed “simprim” and “neoprime” netlists are generated by the Xilinx Alliance Series environment. Since these libraries are built directly into LeonardoSpectrum, the Xilinx netlist can be read by LeonardoSpectrum. A netlist interface that reads mapped EDIF netlists and SDF back annotation files is also available in LeonardoSpectrum.

HDLInventor

The HDLInventor is an interactive source code editor in LeonardoSpectrum. You can double click on errors, warnings, and information (red, green, and blue dots) in the information window or click on the name of your input file to bring up the HDLInventor. The HDLInventor interactively highlights syntax and synthesis construct errors found during synthesis. You can make your edits in this window and, if required, insert template(s) of HDL code that you frequently use.
Design Browser

The design browser allows you to traverse through the design hierarchy to observe objects like ports, instances, and nets. Refer again to the design browser description in the LeonardoInsight section.

Still More Features

These features are intended to guide you during the synthesis process.

This LeonardoSpectrum Synthesis and Technology Manual

The distinctiveness of the LeonardoSpectrum tools and the steps, descriptions, code examples, tables, and screen shots in this manual allow you to start designing right away.

Note: This manual assumes that the reader is familiar with the LeonardoSpectrum User’s manual and with the Windows environment. LeonardoSpectrum may be referred to as Leonardo in some special cases; for example, in a table or as a program name. Level 3 is highlighted to differentiate between Level 2 and Level 3.

Available Online and Website

This manual is available for viewing online with the Adobe Acrobat Reader after LeonardoSpectrum and the Adobe Acrobat Reader are installed from the CD-ROM. Note: The online, PDF manual may contain the most recent information. In addition, the manual can be viewed and printed with desktop utilities. The LeonardoSpectrum manuals are available for down loading from the Exemplar website: http://www.exemplar.com.

Available Online Context-Sensitive Help

Throughout, LeonardoSpectrum has several avenues of online help: menu bar help, FlowTabs, SynthesisWizard help buttons, and F1 context-sensitive help. While FlowTabs is active, press F1 to open a context-sensitive help or press the help button.

Note: The GUI window must be selected first to be in current focus when using F1.

Note: F1 does not work on UNIX.
Online help is designed around Window’s help properties with the traditional banner, tabs, and buttons. The intent of help is to provide you with synthesis information as quickly as possible. You can continue with your task and get help with context menus at the same time.

**Available Libraries**

The LeonardoSpectrum license automatically enables all synthesis libraries. Check Exemplar’s web site at [http://www.exemplar.com](http://www.exemplar.com) for more the latest information on ASIC or FPGA libraries. Refer to the User’s guide for information on adding ASIC, FPGA and symbol libraries.

**Tcl Script Sourcing**

LeonardoSpectrum provides three ways to source your Tcl script. After you create a Tcl script in a standard text editor, you can source your script from LeonardoSpectrum as follows:

- Interactive Command Line Shell (GUI window) *(Level 3)*
- GUI Menu Bar File -> Run Script
- Command Line with Path to LeonardoSpectrum

**Note:** Refer to Command Reference guide for Tcl script path information.

**XlibCreator for ASIC - Complete Development Kit**

The XlibCreator provides a library development environment for ASIC. The XlibCreator tools and documentation are available at:


Contact your vendor or Exemplar Logic for the Synopsys .lib library file and for a license.

The XlibCreator is a complete library development kit which contains templates, scripts, and “C” programs designed to create Exemplar synthesis libraries from a Synopsys .lib format. The major XlibCreator software tools consist of:

- Syngen converts Synopsys .lib source to an intermediate library format called Lgen.
- Libgen finishes the process by compiling lgen files into binary .lib synthesis files.
Screen Shots, Reports, Filenames, and Code Examples

The screen shots, reports, filenames, and code examples in this manual may differ slightly from the actual or most current screens and examples. Moreover, some screen shots may have options selected and filenames displayed for illustration purposes only.

Three Ways to Synthesis

LeonardoSpectrum provides three ways to synthesize your design:

- SynthesisWizard
- Quick Setup
- FlowTabs

SynthesisWizard

The SynthesisWizard is designed for the first time user. The SynthesisWizard walks you through the synthesis process. Every step, from specifying a technology to input files to design goals, is clearly presented to you in a SynthesisWizard flow.

Quick Setup

Quick Setup is intended for the user who is familiar with LeonardoSpectrum and the synthesis process. Everything that is specified in the SynthesisWizard, can be specified on one condensed tab. Once specified, you can hit the run button to run the entire synthesis flow including synthesis, global constraints, optimizing, and writing netlist. In addition, Quick Setup automatically sets up all options, defaults, and settings in the FlowTabs to assist you when walking through the more advanced tabs.

FlowTabs

The salient FlowTabs are designed for the advanced user who needs access to all the embedded power of LeonardoSpectrum. Nearly every step of the synthesis process can be customized based on FlowTabs. To use the FlowTabs you merely walk through each tab in order while customizing along the way. This is essentially what the SynthesisWizard and Quick Setup accomplish for you with the default settings.
Conventions Used

The reader is alerted to terms, GUI names, and items with the following conventions:

- **Level 3** is shown in bold to emphasize that a particular instruction or GUI item is for **Level 3** only. For example, the main window for **Level 3** differs in appearance from **Level 1** and **Level 2** with the number of FlowTabs, the interactive command line shell, and in the banner area.

- **Level 3**: GUI command line in the Information Window is referred to as the **interactive command line shell**.

- Batch Mode is entered on the **DOS** or **UNIX** command line.

- The **Courier Font** is used for file names, commands and variables.

- **Buttons** and **keys** are typed in bold.

- Arrows indicate a menu or pulldown choice: Click File -> Open.

- Screen Shots: Example defaults, filenames, and field values are for illustration purposes only and may not apply to your particular synthesis task.

- FlowTabs refer to the series of tabs - Quick, Technology,...Output, P&R.

- Tab refers to a single tab - Output, for example.

- Power tab refers to the series of tabs that supports the FlowTabs - EDIF, VHDL, Verilog Options, for example.

- LMB is used for left mouse button.

- RMB is used for right mouse button.

- Choice boxes - ■ selected, □ not selected

- Radio buttons - ● selected, ○ not selected

PC Hardware and Software Requirements

These are the requirements for all levels of LeonardoSpectrum.

**Type of PC**

An IBM compatible PC with a Pentium or Pentium-Pro CPU is recommended. A 486 PC is acceptable, but may run slowly.

**Operating System**

LeonardoSpectrum requires Windows NT/95/98.

**Disk Space**

LeonardoSpectrum requires approximately 70 MBytes of disk space for programs and data files. Plan for an additional 50 MBytes for your files.
System Memory (RAM)

Table 1-1, System Memory, shows the recommended memory for proper operation of Exemplar synthesis tools. The actual requirements may vary; this depends on your design and coding style.

Table 1-1. System Memory Requirements

<table>
<thead>
<tr>
<th>Design Size</th>
<th>Number of Gates</th>
<th>Look Up Tables</th>
<th>Flip-Flops</th>
<th>RAM, MBytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>up to 15,000</td>
<td>up to 1100</td>
<td>500</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>15,000 to 75,000</td>
<td>1100 to 5000</td>
<td>3000</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>75,000 and up</td>
<td>5000 and up</td>
<td>5000</td>
<td>256</td>
</tr>
</tbody>
</table>

Note: A system running with less than the recommended memory may slowdown due to memory swapping.

PackagedPower

PackagedPower integrates Exemplar Logic’s Leonardo Spectrum with Mentor Graphic’s Renoir graphical design and management environment, and with Model Technology’s ModelSim HDL simulator. PackagedPower is intended for engineers who are designing medium to large FPGAs or are in the process of moving across the FPGA/ASIC boundary.

The PackagedPower Environment allows you to launch and run both simulation and synthesis from Renoir and then to analyze results in Leonardo Spectrum for each operation through cross probing, cross highlighting, back annotation, and dynamic design animation.
This chapter describes the mapping techniques used in LeonardoSpectrum to map combinational and sequential logic. Also, I/O pad assignments and setting constraints through the constraint file are described. The mapping algorithm is determined by the targeted technology library. Boolean mapping is used for all Actel FPGA devices, while Lookup table mapping is used for Lookup table based FPGA devices.

When running LeonardoSpectrum in the default mode, all I/O signals are assigned pads. The pads are selected from the target technology library during the technology mapping phase. If more than one size of the same pad is available, LeonardoSpectrum chooses the smallest pad size. If the target library contains complex I/Os - I/Os with registers in the I/O cell - then LeonardoSpectrum maps these complex I/Os as required.

Each architecture has different constraints on the usage of complex I/Os. Currently, LeonardoSpectrum checks for design rule violations during usage of complex I/Os for Actel and Xilinx architectures. For other architectures and for manually assigned I/Os, you are responsible for the validity of the output design. You can override the assignments done by LeonardoSpectrum and assign pads manually. This can be done selectively on each pad. This chapter is divided as follows:

- Before Beginning
- Boolean Mapping
- Look Up Table Mapping
- Global Buffers
- I/O Mapping
- Manual I/O Mapping
Before Beginning

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introduction chapter in this guide.

Boolean Mapping

Boolean mapping is a technology mapping technique that LeonardoSpectrum applies to all Actel FPGAs to utilize the underlying Actel architecture. The technology gates are generated from Actel’s CM8s. This is accomplished by tying CM8s inputs to VCC/GND, and by bridging inputs. This allows the Boolean mapper to use all possible cells that can be derived from the target technology logic cell.

Lookup Table Mapping

LeonardoSpectrum uses lookup table mapping for the following LUT-based FPGAs.

- Xilinx XC3000/A/L, XC3100/A, XC4000/A/E/EX/L/XL/XLA/XV, XC5200, XC7200A/7300, XC9500/9500XL, Spartan/XL, Spartan2, Virtex, VirtexE, and CoolRunner
- Altera FLEX 6K/8K/10K and 10KA/KB/KE and APEX 20K/20KE
- Lucent ORCA 2CA/2TA and 3C/3T

The listed FPGA technologies have logic cells based on LUTs. During the optimization process, combination logic is decomposed to individual logic functions. LUT mapping fits these logic functions into a minimal number of LUTs while meeting timing requirements. For each of the LUT based technologies, a different LUT mapping is performed. LUT mapping finds an optimal coverage that maps these logic functions.

Refer to the following for examples of LUT mapping:

- Table 2-1, LUT Mapping - Altera FLEX
- Table 2-2, LUT Mapping - Xilinx
- Table 2-3, LUT Mapping - ORCA
Table 2-1. LUT Mapping Options for Altera FLEX

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Cascades</td>
<td>on</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set altera_use_cascades false</td>
<td>-nocascades</td>
</tr>
</tbody>
</table>

During LUT mapping, Map Cascades controls mapping logic to cascade gates for Altera FLEX 6K, 8K, 10K.
Table 2-2. LUT Mapping Options and Global Buffers - Xilinx

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write FMAP/HMAP Symbols</td>
<td>on</td>
<td>set use_f5map true</td>
<td>-use_f5map</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>default</td>
<td>default</td>
</tr>
</tbody>
</table>

Controls mapping to F5MAP Symbols for Xilinx XC4000/A/E/L/EX/XL.

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Buffers</td>
<td>on</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set map_global_bufs false</td>
<td>-nomap_global_bufs</td>
</tr>
</tbody>
</table>

Controls the use of global buffers for clocks and other global signals.
Table 2-3. LUT Mapping Options - ORCA

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map 6-input LUTs</td>
<td>on</td>
<td>set use_f6_lut true</td>
<td>-use_f6lut</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>default</td>
<td>default</td>
</tr>
</tbody>
</table>

Controls mapping to 6-Input LUTs for ORCA 2CA/2TA, 3C/3T.
Global Buffers

LeonardoSpectrum assigns clock buffers to I/O signals with high-fanouts. This is useful to speed up clock-to-output timing and input-to-register timing. Also, LeonardoSpectrum checks that assigning a certain clock buffer does not violate any design rule of the target technology. Refer to Tables 2-2 and 2-4.

I/O Mapping

During I/O mapping, LeonardoSpectrum assigns PADs to all I/Os in the top level of a design. LeonardoSpectrum can map input buffers, output buffers, tri-state buffers, bi-directional buffers, and complex I/O cells. LeonardoSpectrum also maps global buffers for clock lines and high fanout input pads. LeonardoSpectrum allows you to specify individual I/Os; LeonardoSpectrum does not map these I/O cells.

Use the NOPAD attribute which is attached to the port to do this. You can apply this attribute with the constraint editor control file, verilog pragma, VHDL attribute, and Tcl command.

There are several technology independent options on the Advanced Settings power tab which controls I/O pad assignments. These options affect the complete design, not just individual I/Os. Refer to Table 2-4, Map Complex I/O Cells and Global Buffers - Actel.
Table 2-4. Map Complex I/O Cells and Global Buffers - Actel

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Complex I/O Cells</td>
<td>on</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set complex_ios false</td>
<td>-no_complex_ios</td>
</tr>
</tbody>
</table>

Controls use of complex I/Os.

<table>
<thead>
<tr>
<th>Global Buffers</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>on</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set_map_global_bufs false</td>
<td>-nomap_global_bufs</td>
</tr>
</tbody>
</table>

Controls use of global buffers for clocks and other global signals.
Manual I/O Mapping

This section describes the methods that allow you to manually assign pads:

- Setting timing constraints with GUI Constraint Editor
- PAD and BUFFER_SIG Commands (Attributes)
- Instantiating Pads in VHDL or Verilog

Setting Timing Constraints with Constraint Editor

Refer to the LeonardoSpectrum User’s guide for more information.

PAD and BUFFER_SIG Commands (Attributes)

There are two commands (attributes) that can be used to manually assign I/O pads: PAD and BUFFER_SIG. These commands can be used as attributes from VHDL or as commands from the interactive command line shell. The PAD and BUFFER_SIG commands can only assign mappable cells. A mappable cell is any cell that LeonardoSpectrum is able to instantiate in the output netlist.

PAD Command (Attribute)

The PAD command (attribute) is the recommended way to assign pads. LeonardoSpectrum is able to map any assigned pad with this attribute. In addition, connections between the pins on the PAD and the design signals do not need to be specified. The PAD attribute only works on inputs and outputs. This is in contrast to the BUFFER_SIG attribute that can be applied to inputs, outputs, and internal signals.

The constraint file syntax is:

```
PAD gate_name signal 1 . . . signal n
```

*Signal 1 . . . signal n* are the names of I/O signals where you want to instantiate the above I/O pad. The I/O pad must be a mappable gate from the target library. The PAD attribute does not work on non-mappable gates. To instantiate the I/O pad,
LeonardoSpectrum makes connections between the pins of the gate and the signals of the design. In the following constraint file syntax example, the Actel Act3 FPGA device family is the target technology.

```
PAD HCLKBUF hclk
```

This command connects the input signal hclk to the input pin of the array registers clock buffer (HCLKBUF), and connects all the elements which were originally driven by hclk to the output pin of hclkbuf. When using this command from VHDL, you set the PAD attribute on the specific I/O signal. The VHDL syntax is:

```
attribute PAD of signal_name: signal is pad_name
```

The signal_name is the I/O on which you want to assign the pad with name pad_name.

This is the interactive command line shell syntax for PAD:

```
set_attribute -net <signal_name> -name pad -value <pad_name>
```

### BUFFER_SIG Command (Attribute)

The BUFFER_SIG attribute can be used to assign buffers only. However, BUFFER_SIG can also be used on internal signals (internal clock buffers). This command only works for I/O buffers (one input, one output).

The interactive command line shell syntax is:

```
set_attribute BUFFER_SIG buffer_name signal_name
```

In the following two constraint file syntax examples, the Actel Act2 FPGA device family is the target technology.

```
BUFFER_SIG clkbuf clk1
```
Connect signal `clk1` to the input pin of the external clock buffer (`clkbuf`), and all the elements which were originally driven by `clk1` will be driven by the output pin of the clock buffer (`clkbuf`).

```
BUFFER_SIG clkint rstn
```

Connect signal `rstn` (reset signal) to the input of the internal clock buffer (`clkint`), and all the elements that were driven by `rstn` are driven by the output pin of the clock buffer (`clkint`). When using this command from VHDL, the **VHDL syntax** is:

```
attribute BUFFER_SIG of signal-name:signal is buffer_name;
```

The `signal-name` is the I/O on which you want to assign buffer with name `buffer_name`.

The **interactive command line shell syntax** is:

```
set_attribute -net <signal_name> -name buffer_sig -value <buffer_name>
```
The following **VHDL example file** is a design using the Actel Act3 architecture. The I/O pads are assigned by LeonardoSpectrum, except for two I/O pads which are assigned manually, using attributes.

```vhdl
-- Example for mapping of orectl in Act3
library ieee ;
use ieee.std_logic_1164.all;
library exemplar ;
use exemplar.exemplar.all;
entity orectl is
  port (a, b, clk, iopcl, e: in std_logic;
        pad : out std_logic);
  -- Instantiate hard wired I/O clock buffer and I/O clear buffer
  attribute pad of iopcl:signal is "IOPCLBUF";
  attribute buffer_sig of clk:signal is "IOCLKBUF";
end orectl;
architecture exemplar of orectl is
begin
  signal sig1, o: std_logic;
  process (clk, sig1, iopcl)
  begin
    if (iopcl = '0') then
      o <= '0';
    elsif (clk'event and clk = '1') then
      o <= sig1;
    end if;
  end process;
  sig1 <= a and b;
  pad <= o when (e = '1') else 'Z';
end exemplar;
```

As shown in the example, the signal pad has been assigned to the ORECTL complex I/O. Since the register is not using the enable signal in the VHDL design, LeonardoSpectrum ties enable to GND on the register. The **PAD** and **BUFFER_SIG**...
attributes are also used to instantiate the IOCLKBUF (for the clock signal) and the IOPCLBUF (for the reset signal). The following VHDL example is for targeting Xilinx XC4000 technology:

```vhdl
architecture exemplar of example is
  component OUTFFT
    port (c, d, t: in std_logic;
          o: out std_logic);
  end component;

b1: OUTFFT port map (c=>clk, d=>intern_out, t=>io_control, o=>inoutp);
end exemplar;
```

The complex I/O OUTFFT registers the intern_out signal and is connected to the inoutp output pad. You must follow these rules:

- In the port section of the component declaration, you should utilize the same formal names which appear in Exemplar’s target technology library.
- Also, you must specify an input technology library. If an input library is not loaded, then LeonardoSpectrum cannot find the instantiated component and treats the component as a black box. This causes LeonardoSpectrum to add additional I/O buffers on the I/O pins of the component.
Timing Analysis

This chapter describes:

- Before Beginning
- Static Timing Analysis
- Defining Process, Temperature, and Voltage Values
- Critical Path Report
- Back Annotation
- Post Synthesis RTL (Register Transfer Level) Simulation

Before Beginning

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.

Static Timing Analysis

Static timing analysis allows for efficient evaluation of timing hot spots in the design. The static timing analyzer enables LeonardoSpectrum to make a decision on area and delay trade-off during synthesis and optimization.

LeonardoSpectrum can also generate a critical path report showing a specific set of critical paths in the design. These critical paths can also be viewed in the schematic viewer. In addition, LeonardoSpectrum provides a path to the HDL simulator for post place-and-route function and timing simulation using Standard Delay Format (SDF).
Timing analysis helps verify the timing performance and correctness of a circuit. Timing analysis traces the clocks to the registers in the circuit, computes the delay along various instances in the circuit, and helps to identify timing critical section of the design. This type of analysis does not require generation of circuit stimuli and requires less time than simulation.

Timing analysis is used in synthesis tools to guide timing optimization and technology mapping. Critical paths in the circuit are reported by checking the slack along the path. Slack is the difference between the required time and the arrival time of a signal. A critical path has a negative slack value. The path with the most negative slack is the most critical path in the circuit. The longest path in the circuit is not necessarily the most critical path, since a long path may have a very late required time.

Arrival times are propagated along the circuit by adding the delay across each gate to the arrival times of its inputs. Delay across a gate not only depends on the delay through the gate (intrinsic delay) but also upon the loading of the gate, the fanout connections, the interconnect load, and the slew of the inputs of the gate. The delay information can be expressed in a variety of delay models.

**Delay Models**

Timing Analysis uses the delay information from a technical library to propagate arrival times (required times) to the end points (start points). The delays in the library can be modeled using a simple linear delay model, a piece-wise linear delay model, or a nonlinear delay model.
The Linear Delay Model

The total delay through a gate is given by:

\[ D_{\text{total}} = D_{\text{intrinsic}} + D_{\text{slope}} + D_{\text{transition}} + D_{\text{connect}} \]

- **D_{\text{intrinsic}}** is the delay through the gate from an input pin to the output pin (or along a path)
- **D_{\text{slope}}** is the additional delay incurred due to the input ramp
  \[ D_{\text{slope}} = D_{\text{transition}} \text{ of prev stage} \times S_s \]
  \[ S_s = \text{slope sensitivity factor} \]
- **D_{\text{transition}}** is the delay due to the loading of the gate
  \[ D_{\text{transition}} = R_{\text{driver}} \times (C_{\text{fanout}} + C_{\text{connect}}) \]
  - The **R_{\text{driver}}** is the drive resistance of the gate
  - **C_{\text{fanout}}** is the load due to the fanouts of the gate
  - **C_{\text{connect}}** is the interconnect load
- **D_{\text{connect}}** = \[ R_{\text{wire}} \times (C_{\text{fanout}} + C_{\text{connect}}) \]
  - The **R_{\text{wire}}** is the resistance of the wire
  - The **C_{\text{fanout}}** and **C_{\text{connect}}** are the loads on the gate due to the capacitance of the driven inputs and the interconnect capacitance

The capacitance and resistance of the interconnect are computed using an interconnect load model. The interconnect load model is an estimation of the load due to the fanout, and the location of the resistive components of the wire. The location of the resistive components effect how much capacitive load the driver sees. Three different models of the capacitance load can be used: best, balanced and worst. Refer to Figure 3-1.

In the best case, the interconnect delay is 0, since the driver does not have to drive the interconnect capacitance through the wire resistance. In the worst case, the driver has to drive all the wire capacitance through the wire resistance, as shown in Figure 3-2. The balanced case divides the capacitance evenly between the driven loads.
Figure 3-1  Modeling Wire Resistance
Timing Analysis Before and After Place-and-Route

Timing analysis can occur before or after place-and-route.

- **Before place-and-route:** only estimated routing delay numbers are available for timing analysis. The accuracy of the estimates varies, depending on the target technology. For many FPGA technologies, the variance between predicted delay and actual delay can be up to 20%.
- **After place-and-route:** actual routing delay numbers can be back annotated into LeonardoSpectrum using an SDF file.

Defining Process, Temperature, and Voltage Values

Derating models the variations in total delay based on your specified process, temperature, and voltage. LeonardoSpectrum derates or scales each deratable parameter to compute the total delay of a path. You can define global parameters in the library to model these effects with respect to the nominal process, temperature, and voltage. Refer to Table 3-1.

LeonardoSpectrum must find the value of the process from the specified process name in technology library. For example, if you selected worst then LeonardoSpectrum looks for a parameter called worst_process in the technical library and begins the derating on process. You may also enter a process name.
Table 3-1. Derating Options

<table>
<thead>
<tr>
<th>ASIC Technology Settings</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>set temp &lt;celsius&gt; -temp=&lt;degrees&gt;</td>
<td></td>
</tr>
<tr>
<td>Timing information is derated during delay computations for this temperature.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voltage</td>
<td>set voltage &lt;volts&gt; -voltage=&lt;volts&gt;</td>
<td></td>
</tr>
<tr>
<td>Timing information is derated during delay computations for this voltage.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td>set process &lt;name&gt; -process=&lt;name&gt;</td>
<td></td>
</tr>
<tr>
<td>Depends on target technology process.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Critical Path Report

A critical path is defined as a path that has negative slack, or slack less than your specified slack threshold. A path, however long, may not be critical if it meets your constraints.

In the LeonardoSpectrum delay report, the header of the path gives the path number, followed by the path slack. All paths are reported from the start point to the end point. If the start point is the output of a flip flop, the rising or falling edge and the arrival time of the clock is reported. The clock at the end point is also reported where appropriate, along with setup information.

The LeonardoSpectrum critical path report is sorted by most critical path first. If there are no critical paths in the design, then the longest path is reported.

The following definitions describe the headings in the critical path report:

NAME The instance name is reported followed by the pin name. By default, only output pins are reported.

ARRIVAL The arrival time at this node is specified. The latest of the rise and fall times is reported followed by up or down indicating rise time or fall time.

LOAD The load being driven by the driver (output pin), is specified.
Nodes which are on combinational loops are identified by the string (loop) next to the arrival time. For example:

```
   i1305/O  AND3  13.0  dn (loop)  0.0
```

When an option is set to report nets, the fanout is reported instead of the load.

**Back-Annotation**

Back-annotation is the process of inserting actual delay numbers into the network after place-and-route. LeonardoSpectrum provides a mechanism for timing back-annotation from the place-and-route tools. For most technologies, a separate Standard Delay Format (SDF) file is written by the place-and-route tool. The SDF format is the industry standard way to represent delay information. For Xilinx, the post place-and-route delay numbers are annotated in the XNF netlist. LeonardoSpectrum XACT flow Time Module can read a timing annotated XNF netlist and write out a Verilog or VHDL structural netlist along with an SDF file with the back-annotated delay information.

**SDF (Standard Delay Format) Writer**

LeonardoSpectrum contains an SDF writer for pre place-and-route delays. The SDF writer calculates delays as used by the timing analyzer and timing optimization routines. The SDF writer writes this information in an SDF format. The SDF format together with the flat VHDL netlist can be read into MTI V-System or any RTL simulator for pre place-and-route timing simulation.

The SDF writer is controlled by the `sdf_write_flat_netlist` Tcl variable. Set this variable to TRUE. An example set of commands may be:

```
set sdf_write_flat_netlist TRUE
ungroup -all /*need to flatten the netlist*/ write design.vhd /*write out flat VHDL or Verilog*/ write design.sdf /*write out SDF*/
```

**Design Flow**

As shown in Figure 3-3, the timing back annotation flow consists of the following steps:
1. LeonardoSpectrum is used to synthesize a behavioral VHDL or Verilog netlist for a particular target technology.

2. The synthesized netlist is input to a place-and-route tool.

3. The place-and-route tool produces a timing annotated netlist (post place-and-route), or an SDF file, along with the post place-and-route netlist.

4. The timing analyzer uses these actual delay numbers to determine post place-and-route critical paths.

5. The schematic viewer is used for graphical analysis of the design. Critical paths can also be highlighted in the schematic.

6. LeonardoSpectrum can read the netlist provided by the place-and-route tool, and can produce a structural VHDL or Verilog netlist along with an SDF file with post place-and-route delays.

7. This structural netlist, along with the SDF file can be input to a VHDL or Verilog simulator.

**VITAL Libraries**

The timing simulation is only as accurate and fast as the libraries that support back-annotation of delay information. VITAL provides for a mechanism for back-annotating delay information to circuit elements through interface constants, called generics, which are declared in the cells in the VITAL library. Generics are declared in the entity section of a cell and are relayed to the port names of the cell and the type of delay information being communicated. Libraries that allow back-annotation through such generics are called VITAL level 0 compliant.

For fast and accurate simulation, functionality and delay propagation need to be expressed efficiently. VITAL specifies a method for developing simulation models and also specifies a set of primitives to efficiently evaluate the functionality. Simulators that take advantage of this can show significant improvement in performance. VITAL libraries that adhere to this specification are considered level 1 compliant.

For delay propagation in VITAL, path delays between an input pin and an output pin \( \text{tpd}_{\text{in}_\text{out}} \) are annotated in the \( \text{tpd}_{\text{in}_\text{out}} \) generic. The interconnect delay leading up to an input pin is annotated in the \( \text{tpd}_{\text{in}} \) generic. The setup and hold timing checks are with respect to a clock edge and are annotated in generics that identify the clock edge.
edge, for instance, tsetup_data_clk_POSEDGE or thold_data_clk_NEGEDGE. A generic, TimingChecksOn, also allows such timing checks to be turned off for increased performance.

Figure 3-3 Design Flow

In the design flow, the values for specific generics are taken from the SDF file which contains post place-and-route delay information. The next section discusses the mapping between specific SDF constructs and the corresponding VITAL generics.
SDF for Transporting Delays

SDF provides a means for transporting delays from one tool to another. SDF allows for specification of computed delays such as pin-to-pin delays, instead of delay parameters such as drive resistance, capacitive load, etc. SDF Delay information is grouped into path specific cell delays and interconnect delays. Path specific delays include:

- The intrinsic delay from the input pin of the gate to the output pin.
- The delay due to the input ramp.
- The delay due to the output loading (including the load due to the interconnect).

The interconnect delay is the delay due to the wire resistance and capacitance between the driver of the net and the input pins it drives.

The path specific delays are modeled as IOPATH constructs in SDF. The interconnect delays are represented by PORT or INTERCONNECT constructs and are lumped at the input pin of the gate. The setup and hold checks are modeled with the SETUP, HOLD or SETUPHOLD constructs. The SDF constructs correspond to generics in the VITAL library into which the delay numbers are to be annotated. Following is an explanation of how various SDF constructs are mapped to corresponding VITAL generics.

- The PORT construct maps to the $\text{tipd}_{\text{ generic}}$. This generic is identified by the port names associated with the SDF construct. For example, (PORT A (:5:) (:6:)) would annotate the $\text{tipd}_A$ generic with rise and fall delays of 5 and 6.
- The IOPATH construct maps to the $\text{tpd}_\text{ generic}$. For example, (IOPATH A Y (:5:) (:6:)) would annotate the $\text{tpd}_A\_Y$ generic with rise and fall delays of 5 and 6.
- The SDF constructs SETUP and HOLD map to timing check generics $\text{tsetup}_{\text{ generic}}$ and $\text{thold}_{\text{ generic}}$ depending if the cell being annotated is a rising edge flip-flop or a falling edge flip-flop.

Post Synthesis RTL (Register Transfer Level) Simulation

LeonardoSpectrum supports writing simulatable models for gate level designs. Technology cells are filled with functional (RTL) information which allows the writing of behavioral descriptions for technology cells.

The $\text{unmap}$ command removes technology cell instances from a design, and replaces the instances with the primitives underneath. This allows VHDL and Verilog writer to write RTL descriptions for the design.
The \texttt{-downto} option on the write command writes descriptions with technology cells \texttt{downto} primitives (RTL).

\textbf{First Application}

LeonardoSpectrum writes RTL VHDL or Verilog for a mapped design. For example, your back-annotated design or a design that already optimized with the \texttt{optimize} command contains technology cells. You can now write the design(s) out in RTL VHDL or Verilog with one of these two methods:

(1) Run write \texttt{-downto} PRIMITIVES.

The design does not change with this method, the technology cells are still in place, but the VHDL or Verilog writer will write the behavioral contents for the used technology cells. Thus, the design is still fully simulatable.

(2) Run command \texttt{unmap} first, then write.

This removes the technology cells from the design. Primitives are put in their place. The design can now be written out in VHDL or Verilog and will be simulatable without a technology library.

\textbf{Note:} \texttt{optimize} maps the design to a technology and \texttt{unmap} unmaps the design back to primitives. After \texttt{unmap}, the structure of the design may not be the same as before \texttt{optimize}. If you run another \texttt{optimize} after \texttt{unmap}, the result may be more unsatisfactory than the result of the first \texttt{optimize} run.

\textbf{Note:} After \texttt{unmap}, the design can contain redundant logic, and may be large. Run \texttt{pre_optimize} \texttt{-common} \texttt{-unused} to reduce the size. This cleans the design, removes constants, shared and unused logic, and does not change the functionality of the design.

\textbf{Second Application}

LeonardoSpectrum writes simulatable models for technology cells. To write one model of \textit{one} library cell, change \texttt{present\_design} to the NETLIST view of the technology cell of choice, and issue a write \texttt{-downto} PRIMITIVES command.

Use the following Tcl script to write \textit{all} library cell models to separate VHDL files:

\begin{verbatim}
# Set library name here:
    set lib lsi300
\end{verbatim}
load_library $lib
foreach i [list_design -short .$lib] {
    present_design .$lib.$i.NETLIST
    write -downto PRIMITIVES $i.vhd
}

If you want all models in a single, large, VHDL file try IO redirection:
load_library $lib
foreach i [list_design -short .$lib] {
    present_design .$lib.$i.NETLIST
    write -downto PRIMITIVES -format VHDL - >> total.vhd
}

Note: write without the downto option still writes the design downto technology cells.
By default unmap always flattens out ALL technology cell instances in a design or
ALL technology cell instances in one level of hierarchy (-single_level).

ungroup does not flatten out technology cell instances. The option -force is added to
the ungroup command, so that specific technology cell instances (by name) can be
flattened out to primitives.

Limitations:

Look Up Table (LUTs) are not unmapped with the unmap command. LUTs (for
example after optimize for xi4) are written out as simulatable RTL equations already in
VHDL or Verilog. If desired, run decompose_luts command then run unmap
command to remove all technology cells, including LUTs.
This chapter describes:

- Before Beginning
- Setting Global Constraints
- Setting Constraints on Individual Signals
- More Constraint Features

**Before Beginning**

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.

**Setting Global Constraints**

Global constraints can be imposed to assist in optimizing the timing of the design. Enter maximum delay and area values, as required, on the Optimize tab, interactive command line shell, or batch mode. LeonardoSpectrum is directed to search for the smallest circuit implementation which meets the specified timing constraint. Refer to Table 4-1.
Table 4-1. Maximum Delay and Area Options

<table>
<thead>
<tr>
<th>Optimize</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Delay</td>
<td>set maxdly &lt;integer&gt;</td>
<td>-maxdly=&lt;integer&gt;</td>
</tr>
<tr>
<td>Specifies maximum delay value acceptable for the optimized circuit.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max Area</td>
<td>set maxarea &lt;integer&gt;</td>
<td>-maxarea=&lt;integer&gt;</td>
</tr>
<tr>
<td>Specifies maximum area value acceptable for the optimized circuit.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Note:** Global constraints are imposed to assist in optimizing the timing of the design. Use the maxdly variable to set global timing constraints. The maxdly variable specifies the maximum delay acceptable for any path in the optimized circuit. LeonardoSpectrum is directed to search for the smallest circuit implementation which meets the specified timing constraint.

**Note:** The maxdly variable is used with -area option for optimize command. If a specific pass does not meet the constraint, then the optimized circuit is remapped in an effort to meet the constraint. If the constraint is not met after remapping, then LeonardoSpectrum performs the next pass and continues the optimization/mapping process.

### Setting Constraints on Individual Signals

You can set constraints on individual signals with the Constraint File Editor as explained in the User’s Guide. Refer also to the Command Reference guide, Attributes chapter for more syntax examples.

### Load and Drive Specifications

Output load and input drive can be specified in the Constraint File Editor. All load numbers are in number of unit loads.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUT_LOAD &lt;value&gt; &lt;output signal 1 ... output signal n&gt;</td>
<td>The OUTPUT_LOAD attribute tells the program the amount of external loading on an output of the design. The value is the number of unit loads driven by the output. This number is used to calculate delays and to make sure that sufficient drive capability is available at the output. This can result in buffering, or in replicating logic to meet the load and drive requirements. If no value is specified for an output, then the default output load for the particular technology is used. This is a constraint file editor syntax example:</td>
</tr>
<tr>
<td>OUTPUT_FANOUT &lt;value&gt; &lt;output signal 1 ... output signal n&gt;</td>
<td>The OUTPUT_FANOUT attribute tells the program the amount of internal loading on an output of the design. The value is the number of units of load from each input to the output. This is used to calculate delays and to make sure that the output can drive the specified number of units of load. If no value is specified for an output, then the default output load for the particular technology is used.</td>
</tr>
</tbody>
</table>
The **OUTPUT_FANOUT** attribute tells the program the amount of external loading on an output of the design. *Value* is the total number of fanout loads driven by the output.

```
INPUT_MAX_LOAD value input signal 1 ... input signal n
```

The **INPUT_MAX_LOAD** attribute defines the maximum load that the synthesized circuit may present at an input to the design. The *value* is the maximum number of unit loads allowed. If the synthesized circuit exceeds this amount of loading, LeonardoSpectrum buffers to reduce the load. In the macro mode, LeonardoSpectrum adds buffers to meet the load constraint unless the input pins have **NOBUFF** attribute.

```
INPUT_MAX_FANOUT value input signal 1 ... input signal n
```

The **INPUT_MAX_FANOUT** attribute defines the maximum fanout load that the synthesized circuit may present at an input to the design. *Value* is the maximum number of total fanout loads allowed. During synthesis, LeonardoSpectrum buffers to reduce the load.

```
INPUT_DRIVE value input signal 1 ... input signal n
```

The **INPUT_DRIVE** attribute specifies the additional delay per unit load for an input port. The *value* is the additional delay in nanoseconds per unit load. This value is used when calculating delays so that the effects of the load the synthesized circuit presents to the gate driving the input can be accurately modeled. Each technology has a default drive defined for inputs, usually the drive of a single inverter gate.

### Timing Specifications

Timing analysis routines that decide where to make an area/delay trade-off in the logic design.

These routines use your specified timing constraints along with delay information for the library elements and do a path analysis of the synthesized circuit. Paths start at primary inputs and at register outputs. Paths end at primary outputs and at register inputs. Paths to the asynchronous set and reset of flip-flops are ignored by default. However, they can be connected and analyzed with the `connect_path` and `disconnect_path` commands.
Table 4-2. Path Analysis

<table>
<thead>
<tr>
<th>Interactive Command Line Shell</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect_path</td>
<td>Marks a timing arc in an instance of a gate as connected or propagating.</td>
</tr>
<tr>
<td>disconnect_path</td>
<td>Detaches a net from a port or port instance.</td>
</tr>
</tbody>
</table>

**Note**: The latest arrival time, and the earliest required time for each signal in the network is determined. The difference between these is the slack at the node. A negative slack at a node indicates the node is on a path which violates some timing constraint.

**Input/Output Timing Constraints**
You may define the required times at output ports and the arrival times at input ports. The **syntax** for these **constraint file** attributes is:

```
REQUIRED_TIME  value  output port 1  ...  output port n
ARRIVAL_TIME   value  input port 1  ...  input port n
```

The required time for an output port defines the time that a signal is required to be available for this port. Arrival times at primary inputs define the maximum delay to that input through logic external to the synthesized design.

**Clock Specification**
Clock timing can be defined for registers (latches and flip-flops). The required and arrival times at the register inputs and outputs are implied through the clock timing definition. The arrival time at a register output is one propagation delay after the leading edge of the clock. The required time at the register input is:

clock_cycle - setup_time

If your specified required times are not assigned to output ports and register inputs, then the required time is set to the value specified with the `maxdly` variable, if any. Otherwise, the required time is set to the latest arrival time in the circuit. The longest path in the circuit then has a zero slack time. The slack times on all of the other nodes indicate how much faster the worst path through that node is when compared with the worst path in the circuit.
You may specify the times from the interactive command line shell, and/or in a VHDL source file, if the input is a VHDL file. The parameters which may be specified are shown in constraint file syntax:

- **CLOCK_OFFSET**
- **CLOCK_CYCLE**
- **PULSE_WIDTH**

These parameters define the behavior of the clock. All clock behaviors assume one single common zero reference. **CLOCK_OFFSET** defines the offset of the leading edge from the common zero. **CLOCK_CYCLE** defines the length of the clock. **PULSE_WIDTH** defines the length of the clock pulse.

For both flip-flops and latches, the leading edge occurs at time **CLOCK_OFFSET**. The arrival time at the register outputs is set to the propagation delay after this time.

```
arrival time at register output = CLOCK_OFFSET + CLK->Q delay
```

The required time at the input to the flip-flops and latches is **CLOCK_OFFSET** + **CLOCK_CYCLE** - setup_time. LeonardoSpectrum does not handle transparent latches.

Clock timing parameters are specified for the actual clock signal (i.e., the signal that connects to the clock pin of the register). This signal may be an input port or the signal may be the output of another register, or the signal may be the output of some combinational logic. **Note:** This is not a recommended method for generating clocks.

Clock timing is not derived automatically for any signals. Timing must be specified explicitly for each clock. For example, a clock which is a divided down version of another clock must have defined timing specifications. The timing is not be determined from the source clock’s timing.

**VHDL Attributes**

VHDL attributes may be used to specify timing parameters in VHDL designs. The names of the attributes are the command names as defined above.
An example of the VHDL syntax for an attribute is:

```
ATTRIBUTE REQUIRED_TIME OF out:SIGNAL IS 10ns;
```

### More Constraint Features

The following features are for all technologies:
- Constraint Driven Timing Optimization
- Support for Maximum Frequency
- Support for Multicycle Path

### Constraint Driven Timing Optimization

Constraint driven timing optimization is available. LeonardoSpectrum optimizes the circuit to meet timing constraints. After optimization and mapping are done, timing violations are determined in the circuit, and an attempt is made to improve timing on critical paths. You can set timing constraints from the constraint file or set a global max frequency constraint that applies to all clocks in the design. If no constraints are specified, LeonardoSpectrum attempts to improve timing on the longest path in the circuit. The following is an example from a LeonardoSpectrum log file that demonstrates the effect of Timing Optimization:

```
Start timing optimization for design .work.ace_bus_ctrl.behavior
Initial Timing Optimization Statistics:
Most Critical Slack    -29.1
Sum of Negative Slacks -1337.7
Longest Path           35.6 ns
Area                   99.0
Final Timing Optimization Statics:
Most Critical Slack    -22.3
Sum of Negative Slacks -1227.4
Longest Path           30.1 ns
Area                   112.0
Total time taken        32 cps secs
```
The constraint driven timing optimization has significant run time improvements (at least 10 times faster), and the quality of results are improved significantly.

You can set global timing constraints by setting clock frequency. You do this by running the clock_frequency <value> constraint, while <value> is in terms of MHz. The clock_frequency constraint applies to all clocks in the design.

**Note:** clock_cycle is set on a particular clock, while clock_frequency is set globally on all clocks.

**Support for Maximum Frequency**

You can set a global timing constraint for maximum clock frequency in a design. LeonardoSpectrum sets this constraint on all global clocks in the design. Constraint driven timing optimization tries to meet the global clock frequency constraint.

You can set the maximum global frequency constraint in LeonardoSpectrum by running `set_attribute clock_frequency <value>` on the LeonardoSpectrum command line. The frequency should be an integer greater than 0 and entered in Mhz.

LeonardoSpectrum reports clock frequency of the clocks after synthesis by running `report_delay -clock` command.

**Support for Multicycle Path**

Use the following **interactive command line shell syntax** to specify multicycle path constraints to LeonardoSpectrum and refer to Table 4-3:

**set a multicycle path:**

`set_multicycle_path -from <> -to <> -value <value> ->`

**delete a multicycle path:**

`set_multicycle_path -from <> -to <> -unset ->`

**options for multicycle path:**

`set_multicycle_path [options] <from> <to> <value>`
Table 4-3. Multicycle Path Syntax

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-setup]</td>
<td>setup time</td>
</tr>
<tr>
<td>[-hold]</td>
<td>set hold time</td>
</tr>
<tr>
<td>[-rise][fall]</td>
<td>set relative to falling edge of clock</td>
</tr>
<tr>
<td>[-src_clk]</td>
<td>name of source clock</td>
</tr>
<tr>
<td>[-dest_clk]</td>
<td>name of destination clock</td>
</tr>
<tr>
<td>[-to &lt;list&gt;]</td>
<td>list of source points</td>
</tr>
<tr>
<td>[-from &lt;list&gt;]</td>
<td>list of end points</td>
</tr>
<tr>
<td>[-value &lt;integer&gt;]</td>
<td>number of cycles</td>
</tr>
<tr>
<td>[-unset]</td>
<td>unset constraint</td>
</tr>
</tbody>
</table>

For Example:

```
set_multicycle_path -setup -rise -from u2.reg_rxdatardy -to u2_regframingerr -value 2
```

This example sets the clock cycle (for setup and clock rising) between `u2.reg_rxdatardy` and `u2.regframingerr` to 2 cycles. Paths that you flag as multicycle are not optimized for timing and are not reported as critical paths.
Timing Optimization

This chapter describes:

- Before Beginning
- Optimization Flow
- Delay Optimization
- Setting Timing Constraints
- Statistics Report
- Automatic Constraining of Designs

Before Beginning

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.

Optimization Flow

This is a brief review of the optimization flow. First, LeonardoSpectrum reads in the design and translates the design into internal data structures. Next, optimization and mapping are run and the design is mapped into technology gates from the target library.

Timing optimization is performed and can be controlled at various levels within LeonardoSpectrum. By choosing the optimization mode to be delay mode you can direct the tool to optimize for delay instead of area. Different optimization algorithms are sensitive to this mode and try to minimize the timing of the circuit. If no timing
constraints are set, optimizing with delay mode causes LeonardoSpectrum to try and minimize the arrival times at all end-points (primary outputs, and data inputs to registers).

In addition, you can set global timing constraints that affect optimization and mapping. By setting timing constraints, you can specify critical paths in the design and enable LeonardoSpectrum to improve the timing on these paths. During technology mapping, the mapper attempts to improve the arrival times on all end-points based on the constraints specified. The timing performance of the design is enhanced with buffering and replication algorithms.

**Delay Optimization**

By default, LeonardoSpectrum optimizes designs to achieve minimum area, or device usage. When running in delay mode, LeonardoSpectrum minimizes the circuit delay. The output netlist includes the fastest circuit implementation found, using the worst case path delay to compare the results from different optimization passes.

By default, LeonardoSpectrum optimizes designs to achieve minimum area, or device usage. You can override this default by running the `optimize` command with the `-delay` option.

```
optimize -delay
```
Table 5-1. Maximum Delay Options

<table>
<thead>
<tr>
<th>Optimize</th>
<th>Interactive Command Line Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Delay</td>
<td>set maxdly &lt;integer&gt; optimize -delay</td>
</tr>
</tbody>
</table>

Specifies maximum delay value acceptable for the optimized circuit.
**Setting Timing Constraints**

You can set timing constraints on primary inputs and primary outputs of each individual block in the hierarchy, as shown in Figure 5-1. You make a block of the present design to set a constraint. For example:

![Diagram of Leonardo Spectrum](LeoRG 01)

*Figure 5-1 Setting Timing Constraints for Leonardo Spectrum*

Use these rules to set constraints on port \( \text{clk} \) of the top level and port \( \text{clk} \) of instance one and of instance two:

- Set constraint on \( \text{clk} \) at top level.
- Change the present design to one by typing `push_design one`.
- Set constraint on \( \text{clk} \).
- Return to top level by typing `pop_design`.
- Change present design to two by typing `push_design two`.
- Set constraint on port `clk`.
- Return to top level by typing `pop_design`.

**Interactive Command Line Shell Syntax:**

```
set_attribute -name REQUIRED_TIME -value n -port port_name
```

**Constraint File Syntax:**

```
REQUIRED_TIME port_name value
```

You can specify the timing constraints in a script for future use. This command sets the required time at the port `port_name`. Similarly, `arrival_time`, `output_load`, and `clock_cycle` can be specified in the constraint file.

**User Control**

Timing optimization works on all levels of hierarchy. It runs on each instance separately and tries to minimize the critical path in each instance. However, `optimize_timing` can be limited to only one level of hierarchy by specifying the `-single_level` option. When limited to one level of hierarchy, the optimize timing runs only on instances in the current level of hierarchy (as pointed by `present_design`).

If the design does not have timing constraints, the `-force` option can be used to force constraints. This option computes the longest path for a given view and uses the longest path as the constraint at all of the end points. This has the effect of optimizing the longest path.

Timing optimization can also be directed to work only on some of the end points by specifying a list of end points with the `-through` option. The `-through` option can also be used to optimize the timing performance at any instance in the design by optimizing the paths through it. Use the following syntax for `optimize_timing`.

```
Interactive Command Line Shell Syntax:

```
optimize_timing -through <node_list>
optimize_timing -force
optimize_timing -single_level
```

**Statistics Report**

Timing optimization reports statistics while evaluating the critical paths. The initial statistics before the view was optimized are reported first.

- **Most critical slack:** This is the biggest negative slack on all critical paths in the optimized view.
- **Sum of negative slacks:** This is the sum of all the negative slacks, at the end points and the start points in the optimized view.
- **Longest path:** This is the delay along the longest path in the view.
- **Area:** Is the area of the view.

As timing optimization evaluates the critical paths, it constantly updates the sum of negative slacks. After completion, it prints the final values of these statistics.
Examples

The following VHDL example illustrates the timing optimization procedure:

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

entity aoi1 is
  generic ( size : integer := 10);
  port ( inp : in std_logic_vector (0 to size);
         outp : out std_logic
      );
end aoi1;

architecture exemplar of aoi1 is
begin
  process (inp)
  variable temp : std_logic;
  begin
    temp := inp(0);
    for i in 1 to size loop
      temp := temp and inp(i);
    end loop;
    outp <= temp;
  end process;
end exemplar;
```
The design before optimization:

Figure 5-2 Design Before Timing Optimization

Targeting Xilinx XC4000

Optimizing this design for XC4000 technology generates the following circuit:

Figure 5-3 Timing Optimization Targeting XC4000
The design has been mapped into three FMAPs and one HMAP. The critical path reporting shows:

**Critical Path Report**

<table>
<thead>
<tr>
<th>NAME</th>
<th>GATE</th>
<th>ARRIVAL</th>
<th>LOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>inp(5)/</td>
<td>0.00</td>
<td>dn</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_INST_13/O</td>
<td>IBUF</td>
<td>3.00</td>
<td>dn</td>
</tr>
<tr>
<td>XMPLR_NET_1/O</td>
<td>F4_LUT</td>
<td>7.50</td>
<td>dn</td>
</tr>
<tr>
<td>XMPLR_NET_3/O</td>
<td>F3_LUT</td>
<td>12.00</td>
<td>dn</td>
</tr>
<tr>
<td>outp_rename/O</td>
<td>H3_LUT</td>
<td>14.50</td>
<td>dn</td>
</tr>
<tr>
<td>XMPLR_INST_7_O1/O</td>
<td>OBUF</td>
<td>21.50</td>
<td>dn</td>
</tr>
<tr>
<td>outp/</td>
<td>21.50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>data arrival time</td>
<td>21.50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data required time not specified

---

Data required time not specified

---

Data arrival time

21.50

---------- unconstrained path

Set the constraints to optimize the design timing in one of two ways:

1. Set the constraints with **constraint file syntax**:

   ARRIVAL_TIME inp(5) 10.0
   REQUIRED_TIME outp 22.0

2. Set the constraints with **interactive command line syntax**:

   set_attribute inp(5) -name ARRIVAL_TIME -value 10.0 -port
   set_attribute outp -name REQUIRED_TIME -value 22.0 -port
Now, because of the constraints, the critical path report shows:

<table>
<thead>
<tr>
<th>NAME</th>
<th>GATE</th>
<th>ARRIVAL</th>
<th>LOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>inp(5)/</td>
<td></td>
<td>10.00</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_INST_13/O</td>
<td>IBUF</td>
<td>13.00</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_NET_1/O</td>
<td>F4_LUT</td>
<td>17.50</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_NET_3/O</td>
<td>F3_LUT</td>
<td>22.00</td>
<td>0.0</td>
</tr>
<tr>
<td>outp_rename/O</td>
<td>H3_LUT</td>
<td>24.50</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_INST_7_O1/O</td>
<td>OBUF</td>
<td>31.50</td>
<td>0.0</td>
</tr>
<tr>
<td>outp/</td>
<td></td>
<td>31.50</td>
<td>0.0</td>
</tr>
<tr>
<td>data arrival time</td>
<td></td>
<td>31.50</td>
<td></td>
</tr>
</tbody>
</table>

Data arrival time 31.50

---

Slack -9.50
Run the timing optimization:

```
LEONARDO: optimize_timing .work.aoi1_10.exemplar
-- Start timing optimization for design .work.aoi1_10.exemplar

Initial Timing Optimization Statistics:
---------------------------------------

Most Critical Slack : -9.5
Sum of Negative Slacks : -19.0
Longest Path : 31.5 ns
Area : 3.0

Final Timing Optimization Statistics:
-------------------------------------

Most Critical Slack : -2.5
Sum of Negative Slacks : -5.0
Longest Path : 24.5 ns
Area : 4.0

Total time taken : 1 cpu secs
```

The final circuit is shown in Figure 5-4.

**Figure 5-4** Design After Running Timing Optimization With Timing Constraints
Figure 5-4 illustrates that the signal inp(5) was promoted and connected to the FMAP that generates the output outp. It goes through only one level of delay.

**Targeting Actel Act3**

Figure 5-5 shows the design after optimization that has been targeting for Actel Act3 technology and before timing optimization has been done.

![Design Before Timing Optimization](Image)

Figure 5-5  Design Before Timing Optimization

Set the constraints in one of the following two ways to optimize design timing:

1. Set constraints with **constraint file syntax**:

   ```plaintext
   ARRIVAL_TIME inp(1) 10.0
   REQUIRED_TIME outp 25.0
   ```
(2) Set constraint with **interactive command line shell syntax**:

```
set_attribute inp(1) -name ARRIVAL_TIME -value 10.0 -port
set_attribute outp -name REQUIRED_TIME -value 25.0 -port
```

The critical path reporting is as follows:

```
Critical Path Report

Critical path #1, (path slack = -10.3):
NAME     GATE     ARRIVAL      LOAD
----------------------------------------------------------------------
inp(1)/   10.00    dn          0.0
XMPLR_INST_17/Y INBUF   14.52    dn          1.1
XMPLR_INST_4/Y AND3    18.92    dn          1.1
XMPLR_INST_5/Y AND4    23.32    dn          1.1
outp_rename/Y AND4    27.72    dn          1.1
XMPLR_INST_7/PAD OUTBUF 35.27    dn          0.0
outp/     35.27    dn          0.0
data arrival time 35.27

data required time 25.00
----------------------------------------------------------------------
```

Timing Optimization 5-13
Run timing optimization:

```
LEONARDO: optimize_timing .work.aoi1_10.exemplar
-- Start timing optimization for design .work.aoi1_10.exemplar

Initial Timing Optimization Statistics:
---------------------------------------
  Most Critical Slack : -10.3
  Sum of Negative Slacks : -21.1
  Longest Path : 35.3 ns
  Area : 15.0

Final Timing Optimization Statistics:
-------------------------------------
  Most Critical Slack : -1.5
  Sum of Negative Slacks : -2.9
  Longest Path : 26.5 ns
  Area : 16.0
```
The design after timing optimization is shown in Figure 5-6 where \text{inp}(1)\text{ is only one level of delay from the output outp.}

\textbf{Figure 5-6  Design After Timing Optimization}

\textbf{Automatic Constraining of Designs}

Set command with \texttt{interactive command line shell syntax}:

\begin{verbatim}
optimize_timing -force
\end{verbatim}

Set variable with \texttt{interactive command line shell syntax}:

\begin{verbatim}
set maxdly <integer>
\end{verbatim}

\textbf{Note:} \texttt{maxdly} is the global maximum delay option on all combinational paths.
Slack is a path that terminates in a register where a clock has been specified. Negative slack indicates that there is a timing violation. However, if the slack is a valid number, then you have specified a required time and/or clock constraint. The constraints are defined as follows:

- **Arrival time only is specified**: The path is unconstrained.
- **Clock only is specified**: The output ports are unconstrained. However, paths ending in register inputs are constrained and `optimize_timing` ignores the `-force` option and attempts to improve the paths with slack.
- **Clock is not specified**: but some is specified on an output port, `optimize_timing` ignores the `-force` option and works only on output ports where the required time is specified. There is negative slack.
- **Clocks and required times on outputs are not specified**: In addition, no maxdly is specified. `optimize_timing -force` then attempts to improve the path that is most critical or the longest path. The command then cleans up the artificial constraint that was applied and returns the circuit without any constraints.

In addition to restructuring paths for all technologies, buffering and replication for Actel and ASICs have been added to the `optimize_timing` command. LeonardoSpectrum attempts to buffer and replicate logic for timing, and then restructures if there are paths that violate timing.

**Note:** Buffering and replication for timing is not available for lookup table (LUT) FPGA technologies: Xilinx, Lucent ORCA, and Altera FLEX.

The `-force` option attempts to improve the circuit if you have not specified any constraints, required time, and/or clock.
LeonardoSpectrum provides a complete synthesis, optimization, and verification environment for Application Specific Integrated Circuit (ASIC) technologies. This chapter is divided as follows:

- Before Beginning
- Design Flows
- Optimization Features
- Timing Analysis
- Design Rule Checking (DRC) Resolving

**Before Beginning**

This chapter assumes that you have read the LeonardoSpectrum User’s guide.

**Design Flows**

Figure 6-1 illustrates the implementation of the top-down design flow by LeonardoSpectrum. You can write a design in VHDL or Verilog according to the functional specification, and run the functional verification through a vendor’s behavioral simulation tool. The RTL design can then be synthesized and optimized to the target ASIC technology. When the run is completed, you can verify design performance by reporting the critical path of the design based on the specified constraints. You can run the timing optimization to ensure that the design meets the timing requirements. An output EDIF netlist can then be generated. Next, you can do scan insertion for testing and layout. The post place and route netlist in an SDF timing file for timing analysis and optimization is feed back into LeonardoSpectrum. You can use a gate level VHDL or Verilog with an SDF timing file to verify timing and functionality.
Figure 6-1  ASIC Top-Down Design Flow
LeonardoSpectrum also allows you to mix HDL descriptions with schematics. You can enter schematics using EDIF netlist and then use VHDL or Verilog. Refer to Special Instructions for Mixing Design Languages in the User’s Guide.

Figure 6-2 shows a synthesis flow. Since an ASIC design is tuned for speed, the emphasis is on setting timing constraints, performing timing analysis, and running timing optimization. In addition, an efficient data path synthesis is important since extensive module optimization and technology specific module generation occur.

**Figure 6-2  Synthesis Design Flow**

**Hierarchy**

You can manage the hierarchy with radio buttons on the Quick Setup GUI for ASIC:

- Hierarchy Auto: Selected by default. Views with 30 or fewer instances are dissolved. Refer also to optimize command, Variables, and Attributes in Command Reference Guide.
- Hierarchy Preserve: If not selected then your design is flattened before optimization; if selected then your design is not changed during optimization.
- Hierarchy Flatten: If selected then your design hierarchy is flattened (dissolved).
The hierarchy manipulation option is available from the GUI, interactive command line shell, and batch mode. Refer also to the Command Reference guide.

If auto (auto_dissolve) is selected, then logic is equated to 2-input NAND gates, and hierarchy is dissolved according to the following rules:

1. FPGA/CPLD auto_dissolve limit is 3000 gates (default).
2. ASIC auto_dissolve limit is 30 gates (default).
3. There is a maximum system limit of gates that can be dissolved in a module. This system limit cannot be modified by a user switch. If this limit is exceeded then auto_dissolve is not completed. The auto_dissolve dissolves instances in a context sensitive manner. If a module is instantiated more than once, then the instance is dissolved only if total number of gates does not exceed the system limit.

**NEW Auto Dissolve Variables and Attribute:**

The auto_dissolve_limit (FPGA/CPLD) and asic_auto_dissolve_limit (ASIC) variables, and the auto_dissolve attribute are available to dissolve blocks using the above three auto_dissolve rules. If the auto_dissolve attribute is set on an instance, then only that instance is dissolved. If the auto_dissolve attribute is set on a view, then all instantiations are dissolved. In addition, auto_dissolve is now the default for the optimization command in both the interactive command line shell and in batch mode.

Hierarchy can be preserved with the -hierarchy_preserve option in batch mode; and with -hierarchy <auto|preserve|flatten> option on the interactive command line shell.

Since the tool is interactive (script driven), different modules in the hierarchy may be optimized with different parameters and constraints. For example, a state machine might be optimized for minimal gate count, while an ALU or data-path function could be optimized for minimal delay.

In some designs, you can merge several instances together to optimize across the boundaries. This is useful when trying to speed up a critical path that goes through several instances.
Design Partitioning

ASIC designs are usually large. The way the design is partitioned has an influence on synthesis results. You can partition the design based on functionality. For example, bus interface unit is one block, while ALU is another block.

Some partitioning can be done for synthesis purposes. The goals of partitioning for synthesis are: produce best synthesis results, speed up optimization time, and simplify the synthesis process. Consider the following guidelines:

- Block size should not exceed 5000 gates. Run-time of synthesis algorithms is nonlinear. Therefore, if the synthesized block is too large synthesis might run for a long time.
- Try to register all inputs and outputs of a block. This is a good practice that simplifies the synthesis process. There is no need to specify required times on outputs since all outputs are registered. Also, all logic is synchronous, which avoids glitches.
- Do not use glue logic between hierarchical blocks. If you preserve hierarchy boundaries, then glue logic is not merged with hierarchical blocks. Glue logic is then optimized separately, which may be detrimental to synthesis results.
- The best results are when the critical paths lies in one hierarchical block as opposed to traversing multiple hierarchical blocks. You can group the logic from several blocks together to ensure that the critical path is in one block.
- Separate sub-designs with different goals.
- Logic with similar characteristics should be grouped together to improve synthesis quality. Timing critical logic in one block with area critical logic in another.

Optimization Features

Specific optimization algorithms are targeted towards ASIC architectures:

- Data path synthesis
- Technology mapping
- Timing optimization
- Gate sizing for timing
- DRC (Design Rule Checking) resolving
Data Path Synthesis

LeonardoSpectrum supports various types of implementations of arithmetic and relational operators used in VHDL or Verilog. Since these implementations are designed specifically for the ASIC technology, the synthesis results are usually smaller and/or faster and take less time to compile. For the following operators, module generators are used to provide a technology-specific implementation:

<table>
<thead>
<tr>
<th>Logical Operators</th>
<th>and</th>
<th>or</th>
<th>nand</th>
<th>nor</th>
<th>xor</th>
<th>xnor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational Operators</td>
<td>=</td>
<td>/=</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>&gt;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>+</td>
<td>-</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Miscellaneous Functions: incrementer, +1; decremented, -1; absolute value, ABS; unary minus, unary -

ASIC Modgen

LeonardoSpectrum supports a variety of new module generator architectures for efficient implementation of data path modules.

The default module generators for adder, subtractor, incrementer, decremented, etc. are enhanced and implemented with two architectures: ripple-carry and carry-look-ahead. If the design is optimized for area, then the ripple architecture is implemented. If the design is optimized for delay, then the carry-look-ahead architecture is implemented. The carry-look-ahead architecture may significantly improve the performance of the design.

Module generators can be supported for technology specific ASIC libraries. Some of the architectures are: ripple-carry/carry-look-ahead adders, subtractors, incrementers, decremented, Baugh-Wooley multiplier with ripple-carry or carry-lookahead final adder stages.

You need a library of modgen primitives for your ASIC technology to be able to use the new modgen architectures. This library of primitives is a VHDL description of technology specific implementations of full-adder cells and other basic cells required for efficient module generation.

This library is included with the ASIC package. The package can be obtained from your ASIC vendor. The Exemplar Logic ftp site also provides many ASIC technology packages.
The ASIC package consists of the following:
- LeonardoSpectrum library (<tech>.syn file).
- LeonardoSpectrum modgen primitives library (<tech>.vhd file)

Presently, there are approximately 60 different ASIC technologies available for LeonardoSpectrum. After you receive the ASIC package, follow the instructions to install the files. Use these steps:

1. Store the <tech>.syn file in the $EXEMPLAR/lib directory
2. Install the <tech>.vhd file in the $EXEMPLAR/data/modgen directory.
3. Next, run LeonardoSpectrum to load both the technology and the modgen architectures:
   - load_library <tech>[.syn] - Load the technology file
   - load_modgen <tech>[.vhd] - Load the modgen architectures
4. You are now ready to load in any design. LeonardoSpectrum uses the new, efficient, technology specific implementations for all data path modules in your design.
 Adding an ASIC Library to the GUI

Follow these steps to add additional libraries for your ASIC technologies.

1. Use a text editor to bring up the device.ini file. This file is located in $EXEMPLAR/lib library. Check for a listing of the ASIC library. If the library is listed continue to step 2. Otherwise, complete the items in Table 6-1 and then continue to step 2.

Table 6-1. Definitions for device.ini file

<table>
<thead>
<tr>
<th>File Line: “Manufacturer”, “Family”, “Library”, “none”, Type”, “I/O”, “”, “”, “”, “” where,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer</td>
</tr>
<tr>
<td>Family</td>
</tr>
<tr>
<td>Library</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>I/O</td>
</tr>
</tbody>
</table>

**Example ASIC:** “sample”, “SAMPLE”, “XCL05U”, “none”, “ASIC”, “BOTH”, “”, “”, “”

Refer to your devices.ini file for additional examples.

2. Copy the <tech>.syn file into the $EXEMPLAR/lib directory. For an ASIC library type:

**UNIX:** `cp sample.syn $EXEMPLAR/lib`

**Windows:** use the Windows explorer to copy the library file to the library subdirectory of LeonardoSpectrum.

**Note:** If the <tech>.vhd file exists in the design kit, then copy <tech>.vhd in the $EXEMPLAR/data/modgen directory.

**Note:** UNIX is case sensitive with respect to the fields of “Library” and “Symbol Library”. For example, if the library is named “Fujitsu50K”, then the library field should be in the same mix of upper and lower case characters. The fields of “Manufacturer” and “Family” are not case sensitive.

The library names in the “Library” and “Symbol Library” fields should be entered without a “.syn” or “.sglib” suffix. The libraries from Exemplar are named according to the convention and do not require a suffix.
**User Switches**

```
set modgen_select auto|smallest|small|fast|fastest
```

This variable switch controls whether modgen implementation is optimized for area or delay. This switch controls modgen resolution for all the operators.

**Technology Mapping**

Pattern matching based technology mapping is used for ASICs technologies. The mapping is done as part of the `optimize` command. You can run the `optimize` command in delay mode or in area mode. Each mode has a different cost function that the mapping is sensitive to. The formula for the cost function is defined in the technology library. For example, in the LSI300 library the cost function is:

```
cost = area_weight * area + delay_weight * delay +
       pin_weight * number of pins + nets_weight * number of nets.
```

When optimizing for area the default values are:

```
area_weight = 1.000; delay_weight = 0.01;
pin_weight = 0.20; net_weight = 0.0;
```

When optimizing for delay the default values are:

```
area_weight = 0.01; delay_weight = 1.000;
pin_weight = 0.20; net_weight = 0.0;
```

You can control the cost function by setting the variables:

```
area_weight and delay_weight
```

before running the `optimize` command. You can assign any value in the range between 0.0 and 1.0.
Constraint Driven Timing Optimization

The `optimize_timing` command can be used after optimization is done to improve the timing quality of the design. The `optimize_timing` command works most effectively when timing constraints are specified. If timing constraints are not specified the `-force` option can be used to force timing constraints, in an attempt to improve the longest path.

You can set timing constraints (attributes) to direct the timing optimization to optimize certain paths. For example: `OUTPUT_LOAD`, `INPUT_DRIVE`, `ARRIVAL_TIME`, and `REQUIRED_TIME`.

The `optimize_timing` command restructures combinational logic to reduce the levels of logic between the critical signal and the output.

The `optimize_timing` may be invoked repeatedly for added improvements if possible. A specific end point or instance to improve can be specified to the `optimize_timing` command with the `-through` option.

Gate Sizing

LeonardoSpectrum performs gate sizing to improve the timing quality of the design. LeonardoSpectrum tries to find the optimal size of a gate so that the cost of the gate is minimal. The cost is controlled by the global cost function that is set from the technology library. Gate sizing is performed as part of the `optimize` and `optimize_timing` commands. You can disable gate sizing by setting the variable `gate_sizing` to `FALSE`. By default this variable is set to `TRUE`. You can experiment with this variable to get optimal results. For example, running the sequence:

```
set gate_sizing TRUE
optimize -ta lsi300 -delay
set gate_sizing FALSE
optimize_timing
```

performs gate sizing during mapping but not during timing optimization.

LeonardoSpectrum does not perform gate sizing for I/O cells and sequential cells.
Timing Analysis

Static timing analysis capability is used internally by optimization algorithms. You can use static timing analysis externally. The technology mapping, timing optimization and gate sizing algorithms use static timing analysis to evaluate the quality of results and guide synthesis. You can run the report_delay command which uses timing analysis to create a critical path report. The nature of the timing analysis depends on the delay model defined in the technology library. LeonardoSpectrum supports three different delay models:

- Linear delay model
- Piece-wise linear delay model
- Nonlinear delay model

Issues in Delay Modeling

Clock Delays

LeonardoSpectrum can model clock delays as an “ideal” delay (clock signal does not have any propagation time), or can model the actual propagation time through the clock signal. You can control that by setting the propagate_clock_delay variable.

Clock Skew

You can specify skew between clocks using the CLOCK_OFFSET command. This command defines the offset of the clock edge from the common zero.

Examples of constraint file syntax:

```
CLOCK_CYCLE 20 clk
CLOCK_CYCLE 20 clk1
CLOCK_OFFSET 5 clk1
```

Defines the clocking schemes as shown in Figure 6-3.
Operating conditions effect the delay estimations done by LeonardoSpectrum. Operating conditions are usually specified in the technology library: operating temperature, voltage, and the process. You can change the operating conditions by setting variables. Refer to Table 6-2.

Table 6-2. Operating Conditions

<table>
<thead>
<tr>
<th>ASIC Technology Settings</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>set temp &lt;celsius&gt;</td>
<td>-temp=&lt;degrees&gt;</td>
</tr>
<tr>
<td>Voltage</td>
<td>set voltage &lt;volts&gt;</td>
<td>-voltage=&lt;volts&gt;</td>
</tr>
<tr>
<td>Process</td>
<td>set process &lt;name&gt;</td>
<td>-process=&lt;name&gt;</td>
</tr>
</tbody>
</table>

Note: If these variables are defined before reading the technology library, LeonardoSpectrum then derates the delay computations. Derating is not currently supported for nonlinear delay computations.
Modeling Interconnects

LeonardoSpectrum supports wire loads to model interconnects of an individual block. By default LeonardoSpectrum will use the first wire load table that is specified in the technology library.

You can define many wire load tables in the technology library and then can select a wire load table by using the wire_table variable. For example, the sequence of commands:

\begin{verbatim}
present_design design1
set wire_table table1
optimize
report_delay
present_design design2
set wire_table table2
optimize
report_delay
\end{verbatim}

Uses wire load table table1 for the sub-design called design1 and wire load table table2 for the sub-design called design2.

You can direct LeonardoSpectrum not to use wire load tables by setting the variable nowire_table to TRUE.

The capacitance and resistance of the interconnect are computed using an interconnect load model. The interconnect load model is an estimation of the load due to the fanout, and the location of the resistive components of the wire. The location of the resistive components effect how much capacitive load the driver see.

The three different models of the capacitance load can be used: best, balanced and worst. In the best case the interconnect delay is 0, since the driver does not have to drive the interconnect capacitance through the wire resistance. In the worst case, the driver has to drive the wire cap through the wire resistance. The balanced case divides the capacitance evenly between the driven loads. This modeling can be controlled by setting the wire_tree variable to either best, balanced, or worst value.

Use the report_wire_table interactive command line shell command to bring up a list of all the wire tables in the library/libraries. Select the appropriate wire table from this list for your design.
**Reporting**

Refer to the LeonardoSpectrum User’s Guide for information on reporting.

**DRC Resolving**

A new DRC resolving algorithm has been implemented in LeonardoSpectrum. After mapping and during the optimize command, DRC resolving occurs automatically. You can switch DRC resolving off in the optimize command with a Tcl variable:

```
set optimize_drc_resolving false
```

DRC resolving can be executed separately with the command `balance_loads`. If your design is bottom up design flow, then run `balance_loads` from the top level since the optimize command resolves DRC in the context of the whole design. This section is divided as follows:

- DRC Checking and Fixing
- Set NOBUFF Attribute
- Set DONT_USE Attribute
- Automatic DRC Resolving
- Descriptions and Example

**Note:** `optimize_drc_resolving`

Enables DRC (design rule checking) resolving during optimization by default. Default value: TRUE.

For example, if you are using script for an ASIC design, then you can

```
set optimize_drc_resolving false
```

**Note:** You must run the `balance_loads` command at the end of your design run to ensure that the final design meets the design rule checking (DRC).

**Note:** DRC resolving may require more runtime. However, DRC resolving can improve the initial timing estimation, and can prevent heavily loaded nets.
DRC Checking and Fixing

DRC resolving checks and fixes design rule violations for every net in the design including:

- max_fanout violations
- max_capacitance violations
- max_transition time violations

The information for fanout, max_fanout, capacitance, max_capacitance, and max_transition time comes from the library cells and user cells connected to each net. DRC resolving accumulates this information and calculates the transition time. The final DRC step checks and fixes any violation.

Set Attributes on the Applicable Ports

You can set the following attributes on the applicable ports of your top level design or blackboxes to control fanout, max_fanout, capacitance, max_capacitance, and max_transition. **Note:** These attributes require a real number value.

- max_fanout
- max_capacitance
- max_transition
- capacitance
- fanout

LeonardoSpectrum assumes a violation if any of the max_fanout, max_capacitance, or max_transition real number values are within 1% of the actual calculated fanout, capacitance, or transition values. For example, if you want to allow a max_fanout of 3 on an input, then set the max_fanout attribute value to at least 3.01.

When LeonardoSpectrum resolves a design rule violation, a safety margin of at least 5% is created. This margin is larger than the DRC check margin. This large margin prevents creation of an infinite loop of fixing design rule violations.

First, LeonardoSpectrum tries to resize a driver to resolve a design rule violation. If this is not possible (i.e. the net is too heavily loaded for even the strongest driver available in the library) then the violation is solved with buffering. A buffer-tree is created.

Second, logic is replicated and the actual load is split between the replicated logic.
Third, LeonardoSpectrum tries to find the lowest cost buffers for each violation. LeonardoSpectrum uses either inverters or buffers for buffering. Many technologies prefer inverters due to lower area cost. LeonardoSpectrum ensures that the logic function remains the same by finding or creating an inverted input to drive the inverter tree.

Finally, if a DRC error cannot be resolved, LeonardoSpectrum provides a warning message. LeonardoSpectrum may insert the largest buffer available. Normally, all design rules can be resolved unless unrealistic values are specified on the ports.

**Note:** For resolving load and fanout violations only, LeonardoSpectrum upsizes the gate on which a load violation has occurred.

**Set nobuff attribute**

You can prevent LeonardoSpectrum from buffering or resolving DRC violations on a net by setting the NOBUFF attribute with interactive command line shell syntax:

```
set_attribute -name NOBUFF -value TRUE -net <net_name>
```

**Set dont_use attribute**

You can also exclude certain gates from the DRC resolving process, by setting the DONT_USE attribute on these gates:

```
set_attribute -name DONT_USE -value TRUE.<techlib>.<cellname>
```

**Note:** This attribute eliminates the specified gate for mapping purposes. The exclude_gates Tcl variable still works and the DONT_USE attribute can be set dynamically between different commands, without reloading the library.

**Automatic DRC Resolving**

DRC resolving is performed automatically during optimize on nets both inside blocks and across hierarchy boundaries. After optimize the design is DRC violation free.

If design rule violations are not found, then the command balance_loads runs fast and without problems in LeonardoSpectrum.

**Note:** Previous LeonardoSpectrum releases required running balance_loads after optimize to fix DRC errors on hierarchy boundaries.
The command `balance_loads` does not run automatically during `optimize_timing` to prevent excessive buffering. You must run `balance_loads` from the interactive command line shell each time you run `optimize_timing`.

**Note:** If you use the ASIC GUI Quick Setup then you must run `balance_loads` in the interactive command line shell to prevent excessive buffering and to ensure against DRC violations.

### DRC Descriptions and Examples

This section is divided as follows:

- Load Violations
- Fanout Violations
- Transition Time Violations
- Additional Fanout Information

#### Load Violations

The load driven by a particular gate is expressed in terms of capacitance, pico Farads (pF). LeonardoSpectrum checks that every output pin in a design has an actual load that does not exceed the allowed (maximum) load. Maximum load for an output pin is set by the `max_cap_load` property in the technology library.

Some libraries may define a default load value to be used for output pins which do not have a `max_cap_load` value set. The default is specified in the library by using the `default_max_cap_load` property.

**Interactive Command Line Shell:** You can overwrite the default value from the interactive command line shell by setting the variable `max_cap_load` before reading the technology library.

**Recommended:** set the load values on all primary outputs of each subdesign using the `output_load` command. If information is available in the technology library, then the actual computed load includes wire load capacitance.

#### Fanout Violations

In the technology library each gate may have a `fanout_load` value for input pins, and a `max_fanout_load` value for output pins. `fanout_load` is expressed in terms of real numbers and represents the number of fanouts that a particular input pin
is loading to the fanout output driver. The `max_fanout_load` specifies how many fanouts a particular output pin can drive. LeonardoSpectrum checks that each output pin in a design does not drive more than the allowed number of fanouts.

A default value can be specified in the library by using the `default_max_fanout_load` property. If specified, the `default_max_fanout_load` assigns `max_fanout_load` values to all unassigned gates.

**Interactive Command Line Shell:** You can overwrite the default value from the interactive command line shell by setting the `max_fanout_load` variable before reading the technology library:

```
set max_fanout_load
```
Example:

The following libgen format describes an adder gate.

```plaintext
GATE AF1H {
    inputs A, B, CI;
    outputs S, CO;

    function {
        CO = A*B*!CI + A!*B*CI + !A*B*CI + A*B*CI;
    }

    area = 12.000000;
    input A {
        fanout_load = 1.000000;
        cap_load = 0.078000;
    };
    input B {
        fanout_load = 1.000000;
        cap_load = 0.071000;
    };
    input CI {
        fanout_load = 1.000000;
        cap_load = 0.079000;
    };
    output S {
        max_cap_load = 1.873000;
        max_fanout_load = 20.0;
    };
    output CO {
        max_cap_load = 1.882000;
        max_fanout_load = 20.0;
    }
};
.....
```

LeonardoSpectrum performs the following checks:

1. Output pin S does not drive actual load that is greater than 1.873 pF.
   Output pin CO does not drive actual load that is greater than 1.882 pF.

2. Output pin S does not drive more than 20 fanouts.
   Output pin CO does not drive more than 20 fanouts.
Transition Time Violations

This violation is applicable for nonlinear timing modeling only. You can specify maximum transition time for input and output pins of gates. LeonardoSpectrum then calculates the actual transition time for input and output pins in the design. If the actual transition time exceeds the maximum transition time, as specified in the library, a violation has occurred.

The transition time of inputs to a particular output pin dictate the transition time of the output from this pin. You can define a default maximum transition time in the technology library by setting the default_max_transition property.

Interactive Command Line Shell: You can overwrite the default transition time from the interactive command line shell by setting the max_transition variable before reading the technology library:

```bash
set max_transition
```

Additional Fanout Information

Use the following if your library does not contain the required max fanout load.

- If the max_fanout_load of the cell is available, then this is used by LeonardoSpectrum.
- If the max_fanout_load specified for the cell is not available, then the library’s default_max_fanout is used.
- DRC compares your set max_fanout_load value with the value of the cell and uses the smaller value.
- If you set both max_fanout_load and force_user_load_values (true) then your max_fanout_load value is used.

Max Fanout on the GUI

Use the Max Fanout field on the Advanced ASIC technology GUI to override the default max fanout load specified in the library. However, a synthesized netlist with high fanout nets may be a problem for the place and route tool. The place and route tool usually splits the net arbitrarily. High fanout nets can cause significant delays on wires and become unroutable. On a critical path, high fanout nets can cause significant delays in a single net segment and cause the timing constraints to fail.
To eliminate the need for splitting of the net by the place and route tool, the synthesis tool must maintain a reasonable number of fanouts for a net. LeonardoSpectrum tries to maintain reasonable fanout limits for each for each target technology. Default fanout limits are derived from the synthesis library.

LeonardoSpectrum maintains reasonable fanouts by replicating the driver which results in net splitting. If replication is not possible, the signal is buffered. The buffering of high fanout primary input signals is an example. Buffering the signal causes the wire to be slower by adding intrinsic delays.

On specific nets, you can set an attribute on the interactive command line shell to control the max_fanout value:

```
set_attribute -net <net_name> -name lut_max_fanout -value <int>
```

**Note:** Setting this attribute takes precedence over any global fanout specifications.

**Note:** Refer to a technology in this guide for setting max fanout commands. For example:

```
set lut_max_fanout <integer>
```

or

```
set max_fanout <integer>
```

**New ASIC Global Variable**

```
set optimize_sequential_cell true
```

You may set this variable false to keep DFF for equivalent checking. This variable improves verifying the state machine.

**FSM Encoding (binary, gray, random, onehot, twohot, auto)**

For auto encoding, in Altera FLEX 6/8/10 technologies, LeonardoSpectrum varies the encoding based on bit width. Moreover, enumerated types with fewer elements than global integer lower_enum_break are encoded as binary; while larger enumerated types are encoded as onehot. Values larger than global integer upper_enum_break are encoded as binary. The auto default allows LeonardoSpectrum to select encoding on a case by case basis. For example, if LeonardoSpectrum selects onehot for your auto encoded design, then “onehot encoding” is printed in the log file of your design.
The encoding variable determines how LeonardoSpectrum encodes enumerated types. The encoding variable determines how LeonardoSpectrum implements a state machine with a state vector of an enumerated type.

**Twohot Encoding**

Twohot encoding is now added to FSM encoding (binary, gray, random, onehot, twohot, auto). Twohot sets two flip flops high for each state. The twohot encoding requires more flip flops than binary and fewer flip flops than onehot. Twohot encoding may be beneficial to large FSMs where onehot uses too many flip flops, and binary requires too much decode logic. Refer also to the HDL Synthesis guide, Chapter 2.
LeonardoSpectrum provides synthesis support for the following Actel architectures:

- Act 1/2/3
- 1200XL/3200DX
- 40MX/42MX/54SX/54SX4/RT54SX
- ProASIC a500K
- A54SX32a/A54SX72a

Designs can be entered using VHDL or Verilog descriptions or EDIF netlists. Gate array and other FPGA designs can be retargeted. This chapter presents:

- Before Beginning
- Actel FPGA Architecture
- Design Flows
- Synthesis and Optimization Features
- Data Path Synthesis and Modgen
- Constraint-Driven Timing Optimization
- Additional Synthesis Features
- User Advanced Options
- Design I/O Mapping
- Complex I/O Design Rule Checker and Modifier
- Reporting
- Process Derating Factors
- Supported Devices
- Actel ProASIC Family
Before Beginning

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.

Actel FPGA Architecture

Actel Logic Module

The basic logic element of the Actel FPGAs is the Logic Module. A logic module is built from three, 2-input multiplexers. The Act1 logic module is an 8-input, 1-output logic circuit, which consists of three 2-to-1 multiplexers, one 2-input OR gate, and one 2-input AND gate. The Act2 architecture creates two different module structures. These structures are the C-Modules and the S-Modules. The C-Modules are capable of implementing only combinational logic, while the S-Modules can implement both combinational and sequential logic. The 1200XL and 3200DX are variations of the Act2 architecture. The 3200DX has on-chip dual port RAMs available.

The Act3 family has basically the same architecture as the Act2 devices, with the exception that the sequential gate in the S-Module does not share the reset logic with the combinational logic. The S-Module has speed improvements and additional clock lines. Since logic modules are created from multiplexers, the multiplexer can be viewed as the most cost effective function, where cost is measured in area and speed.

I/O Buffers

Each I/O pin is configurable as an input, output, tristate, or bidirectional buffer. The Act3 family adds complex I/Os that combine registered logic with the basic I/O buffer. Complex I/Os are absorbed automatically.

Design Flows

A simplified design flow is shown in Figure 7-1 to illustrate how LeonardoSpectrum is used with the Actel tools. Designs are entered using standard methods, and are optimized and mapped to the target Actel technology. The result is an EDIF netlist that is used by Actel place-and-route tools.
Figure 7-1  Simplified Design Flow
Figure 7-2  Simplified Back-Annotation Flow
Synthesis and Optimization Features

- Mux-based optimization that understands and utilizes the multiplexer based structure of the Actel Logic Module. The concept is to restructure combinational logic to be represented in terms of multiplexers, instead of being represented as AND and OR gates.
- Boolean mapping maps directly to the Actel Logic Module.
- LeonardoSpectrum resolves load violations to ensure that the generated designs are valid.
- LeonardoSpectrum assigns clock buffers automatically. You can also assign the clock buffers manually.
- Replication and buffering to resolve timing violations.
- Constraint driven timing optimization can be used to speed up critical paths in the design.
- Additional features are Support for Logic Combinability, Automatic Mapping to Complex I/Os, and Design Rule Checker.
- Boolean mapper is faster. Only Boolean mapper is being used for Actel.
- Mapping to CM7s: Increases the combinability for Act2, 1200xl, 3200dx designs.
- Automatic mapping to available clock buffers.
- All Actel libraries were modified with delay information that is compatible with Designer.
- Act1/2/3/32DX: small/fast non-loadable down counters.
- The Actel technology libraries are smaller.

Actel Technologies

Actel mapping and optimization were enhanced to map to DFM7 and DFM7A gates. This makes area and delay estimates more accurate. When targeting Act2 or 3200DX technologies, the mapper maps to CM7 gates. **Note**: These libraries are available on the Actel Web site. Devices supported for the a54SXa32 technology are: TQ144m PQ208, BO144, BO256, and BO329.

Technology Mapping

LeonardoSpectrum offers a powerful Boolean Mapping algorithm that maps logic directly to Logic Cells. The following libraries are available:
- Act1/2/3
- a12xl/a500K
- a32dx (Act2 modgen plus RAMs)
- Actel 54SX/54SXa: Improved mapping takes advantage of CM8INV.
Data Path Synthesis and Modgen

LeonardoSpectrum supports technology specific implementations of arithmetic and relational operators used in VHDL or Verilog RTL. Since these implementations are designed optimally for the Actel technologies, the synthesis results are in general smaller and/or faster and take less time to compile.

Note: Rows are supported in A3200DX.

Note: All RAMs: support for arbitrary size RAMs

The following operators and miscellaneous functions have technology specific architectures in the modgen libraries:

- Logical Operators: and, or, nand, nor, xor, xnor
  
  reduce_and (Act 1 fast incrementer)
  reduce_nand, reduce_or (all families)

- Relational Operators: = /= < <= > >=

- Shift Operators: sll, srl, sla, sra, rol, ror

- Arithmetic Operators: + - *

Miscellaneous Functions: incrementer/decrementer, absolute value, unary minus, counters, multiplexer

Using RAMs

LeonardoSpectrum supports two types of Inferencing RAMs:

- RAM_DQ: RAM_DQ is a single-port RAM with separate input and output data lines.
- RAM_IO: RAM_IO is a single-port RAM with bidirectional data lines.

Both of these RAM types support synchronous or asynchronous read and write. These RAMs are automatically inferred by LeonardoSpectrum from VHDL or Verilog code. The inferencing process distinguishes between RAMs that perform the read operation with an address clocked or not clocked by the write clock (read address clocked). Both of the following VHDL examples perform synchronous writes (inclock) and synchronous reads (outclock); LeonardoSpectrum recognizes these VHDL processes as RAMs:

- The first entity, entity ram_example1, is when the read operation does not have a clocked address.
The second entity ram_example2, is when the read operation does have a clocked address.

Note: Disable RAM Inferencing Variables

set extract_RAM true (default)
set extract_RAM false to disable.

library ieee, exemplar;
use ieee.std_logic_1164.all;
use exemplar.exemplar_1164.all;
entity ram_example1 is
  port (data: in std_logic_vector(7 downto 0);
        address: in std_logic_vector(5 downto 0);
        we, inclock, outclock: in std_logic;
        q: out std_logic_vector(7 downto 0));
end ram_example1;
architecture ex1 of ram_example1 is
  type mem_type is array (63 downto 0) of std_logic_vector (7 downto 0);
  signal mem: mem_type;
begin
  l0: process (inclock, outclock, we, address) begin
    if (inclock = '1' and inclock'event) then
      if (we = '1') then
        mem(evec2int(address)) <= data;
      end if;
    end if;
    if (outclock = '1' and outclock'event) then
      q <= mem(evec2int(address));
    end if;
  end process;
end ex1;
entity ram_example1, is when the read operation does not have a clocked address.
entity ram_example2 is
  port (data: in std_logic_vector(7 downto 0);
    address: in std_logic_vector(5 downto 0);
    we, inclock, outclock: in std_logic;
    q: out std_logic_vector(7 downto 0));
end ram_example2;
architecture ex2 of ram_example2 is
  type mem_type is array (63 downto 0) of std_logic_vector (7 downto 0);
  signal mem: mem_type;
  signal address_int: std_logic_vector(5 downto 0);
begnin
  10: process (inclock, outclock, we, address) begin
    if (inclock = '1' and inclock'event) then
      address_int <= address;
      if (we = '1') then
        mem(evec2int(address)) <= data;
      end if;
    end if;
    if (outclock = '1' and outclock'event) then
      q <= mem(evec2int(address_int));
    end if;
  end process;
end ex2;

entity ram_example2, is when the read operation does have a clocked address.

Actel A3200DX Modgen Support for RAMs

The Actel A3200DX Modgen Library supports RAMs with synchronous write operation and synchronous or asynchronous read operation. Synchronous reads may or may not have the read address clocked by the write clock.
**Constraint-Driven Timing Optimization**

Constraint driven timing optimization is available. LeonardoSpectrum optimizes the circuit to meet timing constraints. After optimization and mapping are done, timing violations are determined in the circuit, and an attempt is made to improve timing on critical paths. You can set timing constraints from the constraint file or set a global max frequency constraint that will apply to all clocks in the design. If no constraints are specified, LeonardoSpectrum attempts to improve timing on the longest path in the circuit.

The following is an example from a LeonardoSpectrum log file that demonstrates the effect of Timing Optimization:

Start timing optimization for design .work.ace_bus_ctrl.behavior

Initial Timing Optimization Statistics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most Critical Slack</td>
<td>-29.1</td>
</tr>
<tr>
<td>Sum of Negative Slacks</td>
<td>-1337.7</td>
</tr>
<tr>
<td>Longest Path</td>
<td>35.6 ns</td>
</tr>
<tr>
<td>Area</td>
<td>99.0</td>
</tr>
</tbody>
</table>

Final Timing Optimization Statistics:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Most Critical Slack</td>
<td>-22.3</td>
</tr>
<tr>
<td>Sum of Negative Slacks</td>
<td>-1227.4</td>
</tr>
<tr>
<td>Longest Path</td>
<td>30.1 ns</td>
</tr>
<tr>
<td>Area</td>
<td>112.0</td>
</tr>
</tbody>
</table>

Total time taken 32 cps seconds

The `optimize_timing` command can be used after optimization to improve the timing performance of the design. `optimize_timing` command works most effectively when timing constraints are specified. However, the `-force` option can be used to force timing constraints and possibly to improve the longest path.

```bash
optimize_timing -force
```
A specific end point or instance to improve can be specified to the `optimize_timing` command with the `-through` option.

```
optimize_timing -through <node_list>
```

The `optimize_timing` command restructures combinational logic to reduce the levels of logic between the critical signal and the output. This command also maps the restructured portion of the logic effectively to the Actel technologies. `optimize_timing` may be invoked repeatedly for added improvements.

**Interfacing with ACTGen from Actel**

Modgen implementations are fully compatible with ACTGen. This gives you a very high quality of results. You may also interface from LeonardoSpectrum with ACTGen directly. The following steps describe how to interface with ACTGen through VHDL and Verilog.

1. Create your ACTGen modules and generate an EDIF output for each module.
2. Instantiate the ACTGen modules in your design. Be sure to keep the cell and pin names consistent with ACTGen naming convention. Refer to Figure 8-3 and choose the appropriate option.
   - Set the variable `hdl_array_name_style` to the format `%s(%d). This will ensure that LeonardoSpectrum will expand bus names by the format `bus_name bit_number`. For Example if a 32 bit bus is named `A`, it will be expanded as: `A0`, `A1`, `A2`, ..., `A31`. This matches the way ACTGen generates bus names.
   - The option `-simple_port_names` ensures that LeonardoSpectrum expands bus names by the format `%s%d`. For example, if a 32 bit bus is named `A`, the bus name is expanded as: `A0`, `A1`, `A2`, ..., `A31`. This matches the way ACTGen generates bus names.
3. Optimize your design and generate EDIF output. All the ACTGen modules will be treated as black boxes.
4. Merge all EDIF files together using Designer by typing:

```
cae2adl -edn2adl ednin:<file1>+<file2>+<file3>
<design_name>
```
For Example to merge files: *top.edn*, *upcnt.edn* and *dncnt.edn*, type:

```shell
cae2adl -edn2adl ednin:top.edn+upcnt.edn+dncnt.edn top
```

![Diagram](image-url)

*Figure 7-3* Interfacing with ACTgen through VHDL
This example shows a way to instantiate an ACTGen component from VHDL:

```vhdl
library ieee;
use ieee.STD_LOGIC_1164.all;
entity top is
  port(
    st_count, Id, ud: in STD_LOGIC;
    bidien, clk : in STD_LOGIC;
    resetb: in STD_LOGIC;
    data : inout STD_LOGIC_VECTOR(3 downto 0);
    ao,bo,co,do,eo,fo,go : out STD_LOGIC
  );
end top;
architecture ARCH of top is
  signal clear: STD_LOGIC;
  signal v9clr: STD_LOGIC;
  signal in_data, out_data, ddata :STD_LOGIC_VECTOR(3 downto 0);
  --This is an ACTgen sub_block
  component cntfour
    --component and pin names should match ACTgen definition
    port (updown, sload, enable :in STD_LOGIC;
      aclr : in STD_LOGIC;
      clock: in STD_LOGIC;
      data :in STD_LOGIC_VECTOR(3 downto 0);
      q : out STD_LOGIC_VECTOR(3 downto 0);
    end component;
    begin
      actgen:cntfour port map
      (ud, ld, st_count, clear, clk, in_data, out_data);
      data <= "ZZZZ" when bidien = '1' else ddata;
    end ARCH;
```

**Additional Synthesis Features**

**Logic Combinability**

Some combinational cells can be combined in the same logic module with registered logic for the Act2/3, 1200XL, and 3200DX technologies. A significant decrease in the number of logic modules implemented for a design may occur. Figure 7-4 shows logic replication to increase combinability. For example, if an AND gate drives 3 flip-flops in the original design, then LeonardoSpectrum would implement the circuit by replicating the AND gate, and having each AND gate drive a single flip-flop. Without
optimization, this takes 4 logic modules: one each for the AND gate and each flip-flop. With LeonardoSpectrum, the AND gates are combined into the logic modules with the flip-flops, resulting in only 3 logic modules for the design.

![Logic Replication for Combinability](image)

**Figure 7-4  Logic Replication for Combinability**

**Fanout and Load Violations**

A fanout violation occurs when a cell drives a higher number of fanouts than the allowed number of fanouts. The Actel Designer software allows a technology cell to drive up to 24 fanouts. The recommended number of fanouts a cell can drive is less than 10. If a cell drives between 10 and 24 fanouts, Designer issues a warning message. When resolving design rule violations, LeonardoSpectrum considers fanout violations and load violations. The default number for maximum fanout allowed by LeonardoSpectrum is set from the technology library:

- Act1, Act2, 1200XL, 3200DX: 10 fanouts
- Act3: 16 fanouts
Use the Max Fanout field on the GUI to override the default max fanout load specified in the library. However, a synthesized netlist with high fanout nets may be a problem for the place and route tool. The place and route tool usually splits the net arbitrarily. High fanout nets can cause significant delays on wires and become unroutable. On a critical path, high fanout nets can cause significant delays in a single net segment and cause the timing constraints to fail.

To eliminate the need for splitting of the net by the place and route tool, the synthesis tool must maintain a reasonable number of fanouts for a net. LeonardoSpectrum tries to maintain reasonable fanout limits for each for each target technology. Default fanout limits are derived from the synthesis library.

LeonardoSpectrum maintains reasonable fanouts by replicating the driver which results in net splitting. If replication is not possible, the signal is buffered. The buffering of high fanout primary input signals is an example. Buffering the signal causes the wire to be slower by adding intrinsic delays.

**Examples for Fanout and Load Violations**

As shown in Table 7-1, you can override this maximum fanout number in the library by entering an integer value in the Max Fanout window on the GUI, on the interactive command line shell, or in batch mode.

**Pipelined Multiplier**

Refer to the User’s guide, Chapter 10, for more information. The pipelined multiplier is available for Actel a54sx.
Table 7-1. Max Fanout - Actel

<table>
<thead>
<tr>
<th>Technology Settings</th>
<th>Pulldown</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Fanout</td>
<td>Select Value</td>
<td><code>set max_fanout_load &lt;n&gt;</code></td>
<td><code>-max_fanout_load=&lt;n&gt;</code></td>
</tr>
</tbody>
</table>

Max Fanout value overrides library value. `<n>` is an integer.

---

**Logic Replication**

LeonardoSpectrum replicates logic when possible. The logic replication technique is used to avoid fanout violations. Logic replication increases area but maintains delay. Refer to Table 7-2 for logic replication.
Table 7-2. Logic Replication - Actel

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>□ Do Logic Replication</td>
<td>on</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set nologic_rep TRUE</td>
<td>-nologic_rep</td>
</tr>
</tbody>
</table>

Controls use of logic replication to meet fanout limitations.

Note: To resolve fanout violations LeonardoSpectrum replicates logic whenever possible. Otherwise LeonardoSpectrum buffers the relevant signals so that fanout violations are met. Logic replication and buffering resolves timing violations in the optimize_timing command, if constraints are specified.

Using Global Clock Buffers

LeonardoSpectrum supports both automatic mapping of clock signals to clock buffers as well as manual mapping of clocks. Both automatic and manual mapping are discussed.

Note: For all Actel libraries except 40MX and 42MX.

Automatic Mapping of Clocks

LeonardoSpectrum automatically maps primary clocks and primary set/reset signals to clock/global buffers available in the target Actel architecture.
Following architectural constrains are taken into account:

- For Act1 only one CLKBUF is available.
- For Act2/1200XL two CLKBUFs are available.
- For Act3 one HCLKBUF and two CLKBUFs are available.
- For 3200DX two CLKBUFs and four QCLKBUFs are available.
- HCLKBUF (Act3 architecture) for a clock is used only if it drives sequential modules only.
- QCLKBUF (3200DX) for clock/global signal is used only if the logic driven by it fits into a quadrant, and two QCLKBUFs do not lead to any conflicting constraints on placement constraint.

If the number of primary clock/global signals in the design exceeds the number of global buffers available in the target technology, then global buffers will be assigned in decreasing order of the critical clock/global signal. For example:

```vhdl
entity test is
port (clk: bit;
       Din: integer range 0 to 65535;
       Q: buffer integer range 0 to 65535);
end test;

architecture behavior of test is begin
process (clk)
begin
   if (clk='1' and clk'event) then --Clock (edge triggered)
      Q <= Din;
   end if;
end process;
end behavior;
```
**Manual Mapping of Clocks**

Primary as well as internal clocks can be manually mapped to global buffers using `BUFFER_SIG` command. For the VHDL design given above, you can specify with **constraint file syntax**:

```vhdl
BUFFER_SIG HCLKBUF clk
BUFFER_SIG CLKBUF rst
```

For an internal clock, you can specify:

```vhdl
BUFFER_SIG CLKINT int_clk
```

You can also specify a `buffer_sig` attribute in the source VHDL to manually assign global buffers. Refer to the following example with **VHDL syntax**:

```vhdl
entity test is
  port (clk,rst: bit;
       Din: integer range 0 to 65535;
       Q: buffer integer range 0 to 65535);
  attribute BUFFER_SIG of clk: signal is "HCLKBUF";
  attribute BUFFER_SIG of rst: signal is "CLKBUF";
end test;
```

Global buffer mapping does not affect any signal which is already mapped manually.

**FSM Encoding (binary, gray, random, onehot, twohon, auto)**

For auto encoding, LeonardoSpectrum varies the encoding based on bit width. Moreover, enumerated types with fewer elements than global integer `lower_enum_break` are encoded as binary; while larger enumerated types are encoded as onehot. Values larger than global integer `upper_enum_break` are encoded as binary. The auto default allows LeonardoSpectrum to select encoding on a case by case basis.
The encoding variable determines how LeonardoSpectrum encodes enumerated types. The encoding variable determines how LeonardoSpectrum implements a state machine with a state vector of an enumerated type. **Note:** If you select “auto” encoding, then LeonardoSpectrum selects the best encoding for your design. For example, if LeonardoSpectrum selects “onehot” then “onehot” is printed on your log file.

**Twohot Encoding**

Twohot encoding is now added to FSM encoding (binary, gray, random, onehot, twohot, auto). Twohot sets two flip flops high for each state. The twohot encoding requires more flip flops than binary and fewer flip flops than onehot. Twohot encoding may be beneficial to large FSMs where onehot uses too many flip flops, and binary requires too much decode logic. Refer also to the HDL Synthesis guide, Chapter 2.

**User Advanced Options**

On Advanced Settings, the Global Buffers option is selected by default. If this option is not selected then automatic global buffer mapping is disabled. By default LeonardoSpectrum maps to global buffers when possible. By default LeonardoSpectrum does not map to QCLKBUFFs for 3200DX architecture because this might over constrain placement. You can click to select Use Quadrant Clock Buffers option to override the default. Refer to Table 7-3.
Table 7-3. Mapping

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Quadrant Clock Buffers</td>
<td>on</td>
<td>set use_qclk_bufs true</td>
<td>-use_qclk_bufs</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>default</td>
<td>default</td>
</tr>
</tbody>
</table>

Directs LeonardoSpectrum to use quadrant clocks. (Used only in 3200DX family).

Design I/O Mapping

LeonardoSpectrum maps I/O cells and registered I/O cells automatically. If the input description implies a use of a certain I/O cell, then LeonardoSpectrum maps to it. You can assign I/O cells manually by either instantiating them from the register transfer level (RTL) description or using the PAD or BUFFER_SIG constraint file attributes in
the Constraint File Editor. Act3 provides a set of basic complex I/O cells. LeonardoSpectrum enhances this set by adding more variations of these cells. These variant cells are described hierarchically, in terms of Act3 primitives.

**Complex I/O Design Rule Checker and Modifier**

Act3 architecture enables you to use registered I/Os with the following restrictions:

- All complex I/Os should be connected to a specific hardwired clock line. The clocks can be used for synchronizing any number of input and output signals. The clock is connected from an external source to the **IOCLK** pin on the I/O buffer.

- The asynchronous preset/clear must be driven by a dedicated preset/clear network (**IOPCL**). This is a high performance network that can drive any number of I/Os. All complex I/Os must be either preset only or clear only. Mixed combinations are not permitted.

Tying hardwired signals (**IOPCL**) to GND/VCC is not permitted by the Actel software. This forces you to use resetable complex I/Os. You can tie the **IOPCL** pin to VCC/GND at the board level.

After optimization and technology mapping, the complex I/O checker and modifier verifies that a design does not violate any of these restrictions. If the design violates some constraints, it tries to modify the design and to use as many registered I/Os as possible without creating an illegal design. In cases of design rule violations, LeonardoSpectrum issues a warning message such as:

```
Info, internal reset drives registered I/Os. This is illegal in Act3 technology. Disabling complex-io mapping.
```

LeonardoSpectrum uses regular IO pads instead of registered I/Os to ensure a legal output design. The I/O checker ensures that constraints have not been violated. In LeonardoSpectrum the I/O checker is applicable for the Act3 architecture only. Refer to Table 7-4.
Table 7-4. I/O Checker Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Complex I/O Cells</td>
<td>on</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set complex_ios FALSE</td>
<td>-nocomplex_ios</td>
</tr>
</tbody>
</table>

Controls use of complex I/Os.
Example of a VHDL File Containing a Design Rule Violation

```vhdl
library exemplar;
use work.exemplar.all;
entity sample is
  port (
    a, b, d, clr, clk: in bit;
    o, o1 : out bit
  );
end sample;
architecture exemplar of sample is
  signal sig1, sig2, sig3: bit;
beginn
process (clk, clr)
begin
  if (clr = '0') then
    sig2 <= '0';
    sig1 <= '0';
  elsif (clk'event and clk = '1') then
    sig2 <= b;
    sig1 <= sig3;
  end if;
end process;
o <= a and sig2;
o1 <= a and sig1;
sig3 <= a and d;
end exemplar;
```
Figure 7-5  Schematic of the Design Showing the Design Rule Violation
Figure 7-6  Schematic of the Modified Design which Resolves the Design Rule Violation

**Simulation Considerations**

LeonardoSpectrum adds additional ports (IOCLK, IOPCL) to the design to use the complex I/Os properly. You should consider the added ports during simulation.

**Assigning Device Pin Numbers**

LeonardoSpectrum supports assigning device pin numbers to ports. These pin numbers are transferred to the synthesized netlist as an Actel Designer specific attribute (ALSPIN) which is recognized by the place-and-route tool. The pin_numbers can be assigned from:

- LeonardoSpectrum interactive command line shell Syntax
- LeonardoSpectrum Constraint Editor with Constraint File Syntax
- VHDL Syntax
VHDL Example:

```vhdl
library IEEE; use ieee.std_logic_1164.all;
library exemplar; use work.exemplar.all; -- Include the EXEMPLAR pkg

entity EXAMPLE is
    port (  
        CLK: bit;
        DIN: integer range 0 to 4;
        Q: buffer integer range 0 to 4  
    );
end EXAMPLE;

architecture EXEMPLAR of EXAMPLE is
begin
    process (CLK)
    begin
        if (CLK = '1' and CLK'EVENT) then
            Q <= DIN;
        end if;
    end process;
end EXEMPLAR;
```

For the VHDL example, a pin_number attribute can be specified as follows with interactive command line shell syntax:

```
set_attribute -port -name pin_number -value <n> <port>
```

where <n> is 1 and <port> is din(0)
After optimization and technology mapping the resulting netlist will have the ALSPIN attribute which Designer understands as a device pin number.

```
......
(net (rename n11 "din(0)"
(joined
(portRef p8 )
(portRef PAD (instanceRef XMPLR_INST_6 ))
(property ALSPIN (string "1")))
......
```

Attributes can also be specified from VHDL.

```
library IEEE; use ieee.std_logic_1164.all;
library exemplar; use work.exemplar_1164.all; -- Include the exemplar pkg

entity EXAMPLE is
  port (
    CLK: bit;
    DIN: in std_logic_vector (4 downto 0);
    Q: out std_logic_vector (4 downto 0)
  );
  attribute pin_number of clk: signal is "1";
  attribute array_pin_number of din: signal is ("2", "3", "4", "5", "6");
end EXAMPLE;

architecture EXEMPLAR of EXAMPLE is
begin
  process (CLK)
  begin
    if(CLK = '1' and CLK'EVENT) then
      Q <= DIN;
    end if;
  end process;
end EXEMPLAR;
```
The output netlist will have ALSPIN attributes for DIN(0), DIN(1), DIN(2), DIN(3), DIN(4) and CLK.

```plaintext
(net clk
 (joined
   (portRef clk )
   (portRef PAD (instanceRef XMPLR_INST_20 ))
   (property ALSPIN (string "1")))
(net (rename n27 "din(4)"
 (joined
   (portRef p18 )
   (portRef PAD (instanceRef XMPLR_INST_19 ))
   (property ALSPIN (string "2")))
(net (rename n28 "din(3)"
 (joined
   (portRef p19 )
   (portRef PAD (instanceRef XMPLR_INST_18 ))
   (property ALSPIN (string "3")))
(net (rename n29 "din(2)"
 (joined
   (portRef p20 )
   (portRef PAD (instanceRef XMPLR_INST_17 ))
   (property ALSPIN (string "4")))
(net (rename n30 "din(1)"
 (joined
   (portRef p21 )
   (portRef PAD (instanceRef XMPLR_INST_16 ))
   (property ALSPIN (string "5")))
(net (rename n31 "din(0)"
 (joined
   (portRef p22 )
   (portRef PAD (instanceRef XMPLR_INST_14 ))
   (property ALSPIN (string "6")))
```

**Reporting**

Refer to the LeonardoSpectrum User’s guide for reporting information.
**Process Derating Factors**

The following table lists process derating factors for these Actel families:

- A3200DX
- A3265DX
- Act1
- Act2/1200XL
- Act3

**Command Line Definitions:** BC=best case; TC=typical case; WC=worst case; STD=standard, -1, -2, -3, -F=speed grade; V=low voltage; MIL=military; COM=commercial; IND=industrial

**A3265DX Devices** Derating Factors <BC[TC]WC><STD>-1-2-3>_3265

<table>
<thead>
<tr>
<th>Value of Process</th>
<th>Operating Conditions</th>
<th>Speed Grade Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC-F</td>
<td>best case</td>
<td>-F</td>
</tr>
<tr>
<td>TC-F</td>
<td>typical</td>
<td>-F</td>
</tr>
<tr>
<td>WC-F</td>
<td>worst case</td>
<td>-F</td>
</tr>
<tr>
<td>BC-3</td>
<td>best case</td>
<td>-3</td>
</tr>
<tr>
<td>TC-3</td>
<td>typical</td>
<td>-3</td>
</tr>
<tr>
<td>WC-3</td>
<td>worst case</td>
<td>-3</td>
</tr>
<tr>
<td>BC-2</td>
<td>best case</td>
<td>-2</td>
</tr>
<tr>
<td>TC-2</td>
<td>typical</td>
<td>-2</td>
</tr>
<tr>
<td>WC-2</td>
<td>worst case</td>
<td>-2</td>
</tr>
<tr>
<td>BC-1</td>
<td>best case</td>
<td>-1</td>
</tr>
<tr>
<td>TC-1</td>
<td>typical</td>
<td>-1</td>
</tr>
<tr>
<td>WC-1</td>
<td>worst case</td>
<td>-1</td>
</tr>
<tr>
<td>BCSTD</td>
<td>best case</td>
<td>STD</td>
</tr>
<tr>
<td>TCSTD</td>
<td>typical</td>
<td>STD</td>
</tr>
<tr>
<td>WCSTD</td>
<td>worst case</td>
<td>STD</td>
</tr>
</tbody>
</table>
A3265DX Devices (Low Voltage) Derating Factors
<IND|COM><BC|TC|WC><STD|-1|-2|-3>[V]

<table>
<thead>
<tr>
<th>Value of Process</th>
<th>Operating Conditions</th>
<th>Speed Grade</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDBCSTDV</td>
<td>IND best case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>INDTSTDV</td>
<td>IND typical</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>INDWSTDV</td>
<td>IND worst case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>COMBCSTDV</td>
<td>COM best case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>COMTSTDV</td>
<td>COM typical</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>COMWSTDV</td>
<td>COM worst case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
</tbody>
</table>

A3200DX Devices Derating Factors
<BC|TC|WC><STD|-F|-1|-2|-3>[V]

<table>
<thead>
<tr>
<th>Value of Process</th>
<th>Operating Conditions</th>
<th>Speed Grade</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC-F</td>
<td>best case</td>
<td>-F</td>
<td></td>
</tr>
<tr>
<td>TC-F</td>
<td>typical</td>
<td>-F</td>
<td></td>
</tr>
<tr>
<td>WC-F</td>
<td>worst case</td>
<td>-F</td>
<td></td>
</tr>
<tr>
<td>BC-3</td>
<td>best case</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>BC-3</td>
<td>best case</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>WC-3</td>
<td>worst case</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>BC-2</td>
<td>best case</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>BC-2</td>
<td>best case</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>WC-2</td>
<td>worst case</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>BC-1</td>
<td>best case</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>BC-1</td>
<td>best case</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>TC-1</td>
<td>typical</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>TC-1</td>
<td>typical</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>WC-1</td>
<td>worst case</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>BCSTD</td>
<td>best case</td>
<td>STD</td>
<td></td>
</tr>
<tr>
<td>TCSTD</td>
<td>typical</td>
<td>STD</td>
<td></td>
</tr>
<tr>
<td>WCSTD</td>
<td>worst case</td>
<td>STD</td>
<td></td>
</tr>
<tr>
<td>BCSTDV</td>
<td>best case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>TCSTDV</td>
<td>typical</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>WCSTDV</td>
<td>worst case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
</tbody>
</table>
### Actel Devices Derating Factors

<BC|TC|WC><STD|-F|-1|-2|-3>[V|RH0|RH3]

<table>
<thead>
<tr>
<th>Value of Process</th>
<th>Operating Conditions</th>
<th>Speed Grade</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCSTDRH0</td>
<td>best case</td>
<td>RH0</td>
<td></td>
</tr>
<tr>
<td>TCSTDRH0</td>
<td>typical</td>
<td>RH0</td>
<td></td>
</tr>
<tr>
<td>WCSTDRH0</td>
<td>worst case</td>
<td>RH0</td>
<td></td>
</tr>
<tr>
<td>BCSTDRH3</td>
<td>best case</td>
<td>RH3</td>
<td></td>
</tr>
<tr>
<td>TCSTDRH3</td>
<td>typical</td>
<td>RH3</td>
<td></td>
</tr>
<tr>
<td>WCSTDRH3</td>
<td>worst case</td>
<td>RH3</td>
<td></td>
</tr>
<tr>
<td>BCSTDV</td>
<td>best case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>TCSTDV</td>
<td>typical</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>WCSTDV</td>
<td>worst case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>BC-3</td>
<td>best case</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>TC-3</td>
<td>typical</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>WC-3</td>
<td>worst case</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>BC-2</td>
<td>best case</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>TC-2</td>
<td>typical</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>WC-2</td>
<td>worst case</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>BC-1</td>
<td>best case</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>TC-1</td>
<td>typical</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>WC-1</td>
<td>worst case</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>BCSTD</td>
<td>best case</td>
<td>STD</td>
<td></td>
</tr>
<tr>
<td>TCSTD</td>
<td>typical</td>
<td>STD</td>
<td></td>
</tr>
<tr>
<td>WCSTD</td>
<td>worst case</td>
<td>STD</td>
<td></td>
</tr>
</tbody>
</table>
### Act3 Devices Derating Factors

<BC|TC|WC><STD|-1|-2|-3><[V]>

<table>
<thead>
<tr>
<th>Value of Process</th>
<th>Operating Conditions</th>
<th>Speed Grade</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCSTDV</td>
<td>best case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>TCSTDV</td>
<td>typical</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>WCSTDV</td>
<td>worst case</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>BC-3</td>
<td>best case</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>TC-3</td>
<td>typical</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>WC-3</td>
<td>worst case</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>BC-2</td>
<td>best case</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>TC-2</td>
<td>typical</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>WC-2</td>
<td>worst case</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>BC-1</td>
<td>best case</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>TC-1</td>
<td>typical</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>WC-1</td>
<td>worst case</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>BCSTD</td>
<td>best case</td>
<td>STD</td>
<td></td>
</tr>
<tr>
<td>TCSTD</td>
<td>typical</td>
<td>STD</td>
<td></td>
</tr>
<tr>
<td>WCSTD</td>
<td>worst case</td>
<td>STD</td>
<td></td>
</tr>
</tbody>
</table>
**Supported Devices**

The Actel devices supported are: 1200XL, 3200DX, 40MX, 42MX, 54SX, 54SXA, a500k, act1, act2, and act3, RT54SX.

### 1200XL Family

- **Default Speed Grade:** 1
- **Speed Grades supported:** 1, 2, STD, F

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1225XL</td>
</tr>
<tr>
<td>A1240XL</td>
</tr>
<tr>
<td>A1280XL</td>
</tr>
</tbody>
</table>

### 3200DX Family

- **Default Speed Grade:** 1
- **Speed Grades supported:** 1, 2, 3, STD, F

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3265DX</td>
</tr>
<tr>
<td>A32100DX</td>
</tr>
<tr>
<td>A32140DX</td>
</tr>
<tr>
<td>A32200DX</td>
</tr>
<tr>
<td>A32300DX</td>
</tr>
</tbody>
</table>
### 40MX Family

Default Speed Grade: 1  
Speed Grades supported: 3, 2, 1, STD, F

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A40MX02</td>
<td>PL44 PL68 PQ100 VQ80</td>
</tr>
<tr>
<td>A40MX04</td>
<td>PL44 PL68 PL84 PQ100 VQ80</td>
</tr>
</tbody>
</table>

### 42MX Family

Default Speed Grade: 1  
Speed Grades supported: 3, 2, 1, STD, F

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A42MX09</td>
<td>PL84 PQ100 PQ160 TQ176 VQ100</td>
</tr>
<tr>
<td>A42MX16</td>
<td>PL84 PQ100 PQ160 TQ176 VQ100 PQ208</td>
</tr>
<tr>
<td>A42MX24</td>
<td>PL84 PQ160 PQ208 TQ176</td>
</tr>
<tr>
<td>A42MX36</td>
<td>CQ208 PQ208 PQ240 CQ256 BG272</td>
</tr>
</tbody>
</table>

### 54SX Family

Default Speed Grade: 3  
Speed Grades supported: 1, 2, 3, STD

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A54SX08</td>
<td>PL84 VQ100 TQ144 TQ176 PQ208</td>
</tr>
<tr>
<td>A54SX16P</td>
<td>VQ100 TQ176 CQ208 PQ208 PQ240 CQ256 VQ100 TQ144 TQ176 PQ208 PQ240 CQ256</td>
</tr>
<tr>
<td>A54SX32</td>
<td>TQ144 TQ176 CQ208 PQ208 CQ256 BGA313 BGA329 BGA272</td>
</tr>
</tbody>
</table>
### 54SX Family

Default Speed Grade: 3  
Speed Grades supported: 1, 2, 3, STD

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>A54SX32A</td>
</tr>
<tr>
<td>A54SX08A</td>
</tr>
<tr>
<td>A54SX16A</td>
</tr>
<tr>
<td>A54SX72A</td>
</tr>
</tbody>
</table>

### act1 Family

Default Speed Grade: 3  
Speed Grades supported: 1, 2, 3, STD

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3265DX</td>
</tr>
<tr>
<td>A32100DX</td>
</tr>
<tr>
<td>A32140DX</td>
</tr>
<tr>
<td>A32200DX</td>
</tr>
<tr>
<td>A32300DX</td>
</tr>
</tbody>
</table>

### act2 Family

Default Speed Grade: 2  
Speed Grades supported: 1, 2, STD F

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1225</td>
</tr>
<tr>
<td>A1240</td>
</tr>
<tr>
<td>A1280</td>
</tr>
</tbody>
</table>
**act3 Family**

Default Speed Grade: 3

Speed Grades supported: 1, 2, 3, STD

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1415A CP100 PL84 PQ100 VQ100</td>
</tr>
<tr>
<td>A1425A CQ132 CP133 PL84 PQ100 PQ160 VQ100</td>
</tr>
<tr>
<td>A1440A CP175 PL84 PQ160 TQ176 VQ100</td>
</tr>
<tr>
<td>A1460A BG225 CQ196 CP207 PQ160 PQ208 TQ176</td>
</tr>
<tr>
<td>A1460B PBG225 PPQ160 PPQ208 PTQ176</td>
</tr>
<tr>
<td>A14100A BG313 CQ256 CP257</td>
</tr>
<tr>
<td>A14100B PRQ208 PBG313</td>
</tr>
<tr>
<td>A14V15A PL84 VQ100</td>
</tr>
<tr>
<td>A14V25A PL84 PQ160 VQ100</td>
</tr>
<tr>
<td>A14V40A PL84 PQ160 TQ176 VQ100</td>
</tr>
<tr>
<td>A14V60A PQ160 PQ208, TQ176</td>
</tr>
<tr>
<td>A14V100A RQ208 CQ256 BG313</td>
</tr>
<tr>
<td>RP14100A CQ256</td>
</tr>
<tr>
<td>RT1425A CQ132</td>
</tr>
<tr>
<td>RT1460A CQ196</td>
</tr>
<tr>
<td>RT1400A CQ256</td>
</tr>
</tbody>
</table>
Improvements for Actel

CM8INV and New ASIC Flow

Improved mapping to Actel A54SX family. Now the mapper takes advantage of the free CM8INV available in the family. This should improve the Actel A54SX architecture and the ASIC flow.

Actgen Components

Modgen components for Actel A54SX family are added. Data path operators for the A54SX family are enhanced to exploit architectural features to a maximum and to yield improvement in run time and memory usage. Based on the Actgen provided by Actel the following modgen components are added for A54SX:

- ripple adder/subtractor/add/sub
- medium fast adder/subtractor/add/sub
- fast adder/subtractor/add/sub
- incrementer/decrementer
- counter, etc.

New Actel a54sxa Library Components

- Ta54sxa.libgen
- Ta54sxa_def.lst
- Ta54sxa_parts.lst
**Actel ProASIC Technology**

The ProASIC technology combines many of the mask ASICs and reprogrammable logic devices. The key feature is a “flash” memory/switch module.

LeonardoSpectrum supports ProASIC 500K devices. The speed and architecture of the a500k devices is a strong alternative to ASICs. The a500k family of programmable logic is based on .25µ flash process. The ASICmaster P&R tool accepts the EDIF output from LeonardoSpectrum as an input for the P&R function. ASICmaster generates an SDF back annotation file that can be loaded back into LeonardoSpectrum for final timing analysis. A global clock buffer and modified RAM area are added to the ProASIC library.

### Path Constraints for ProASIC

LeonardoSpectrum now writes out SDF path constraints. The SDF file contains path constraints for all the constrained paths in the netlist. The SDF is automatically written out for ProASIC. In the interactive mode, the `auto_write` command automatically writes out the path constraints. For example:

```bash
auto_write uart.edf
    -- write uart.edf
    -- Writing file uart.edf
    -- Writing file uart.sdf
```

If you want to write path constraints separately, use the following command:
write -format sdf_path uart.sdf

Example script

# uart is example design
read uart.vhd

# loading the proasic library
load_library a500k

# setting constraints
set register2register 30
set register2output 30
set input2register 30
set input2output 30

# optimization and mapping
optimize -ta a500k

# writing out netlist and constraint file
auto_write uart.edf

Set Number of SDF Paths

By default only 100 SDF paths are written out. The number of paths written out is controlled by this variable:

sdf_constraints_num_paths <100>

If you require more than 100 SDF paths, then set the variable to that number.

set sdf_constraints_num_paths <integer>
**Turn Off Writing SDF Path Constraints**

You can turn off writing the SDF path constraints in two ways:

- On the interactive command line shell
- On the Output GUI Flow Tab

Refer to Table 7-5 for more explanation.

Table 7-5. Options for SDF Path Constraints

<table>
<thead>
<tr>
<th>Output GUI Flow Tab</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Vendor constraints file (default)</td>
<td>on</td>
<td><code>set novendor_constraint_file false</code></td>
</tr>
<tr>
<td>Write Vendor constraints file Turn off SDF path Constraints.</td>
<td>off</td>
<td><code>set novendor_constraint_file true</code> Turn off SDF path constraints.</td>
</tr>
</tbody>
</table>

Controls writing SDF path constraints.
This chapter presents information specific to the use of Altera APEX as a source or target technology. LeonardoSpectrum supports the APEX 20K/20KE architectures.

This chapter is divided as follows:

- Before Beginning
- APEX 20K/20KE Family
- Quartus Integration
- New Features and Improvements
- APEX Devices Supported
- APEX 20K/20KE Flow Diagram
- User Defined PTERM (Product-Term) Support
- Altera APEX 20K/20KE Compatible Vendor Tools

**Before Beginning**

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.

**APEX 20K/20KE Family**

LeonardoSpectrum supports mapping your design to APEX 20K/20KE. Depending on the options selected, mapping to WYSISYG primitives is either done by LeonardoSpectrum or by Quartus. By default, LeonardoSpectrum does mapping to WYSIWYG primitives. Quartus is the new place and route software from Altera. The Altera APEX technology provides support for WYSIWYG device primitives.
Mapping Options

By default, mapping to WYSIWYG primitives is true and mapping to complex I/Os is false. Currently, these options are not available in the GUI.

The variable `apex_wysiwyg_support` enables mapping to WYSIWYG ATOMs (an ATOM is a WYSIWYG primitive or cell) in APEX 20K/20KE devices. If you need to turn this variable off, you must also set `dont_lock_lcells` to true. For example,

```
set apex_wysiwyg_support false
set dont_lock_lcells true
```

The variable `altera_map_complex_ios` enables mapping to I/O flip flops. This variable is functional if the variable `apex_wysiwyg_support` is also set to true. For example,

```
set altera_map_complex_ios true
set apex_wysiwyg_support true
```

The `altera_map_complex_ios` option is also available on the GUI: Map I/O Registers. Refer to the following screen.

LeonardoSpectrum APEX 20K/20KE Mapping

LeonardoSpectrum support for mapping to Altera APEX 20K/20KE WYSIWYG cells includes:

1. By default, LeonardoSpectrum maps to all modes of Lcells:
   - Counters
   - QFBK_Counters
   - Arithmetic
   - Normal
2. Mapping to I/Os, including various complex I/O configurations is available.

3. RAMs/ROMs are now mapped to LPM_RAMs and LPM_ROMs. In the future, direct mapping to APEX 20K WYSIWYG primitives - RAM slices and Pterms (product terms) - will be provided by LeonardoSpectrum. LPM_RAMs and LPM_ROMs are implemented as RAM slices by Quartus.

4. APEX 20K/20KE libraries are supported.

5. The current P&R GUI for Quartus provides support for the P&R flow using the Quartus NativeLink API features. This allows you to access and modify designs in the Quartus database. Currently, the EDIF format is supported for the output netlist. In contrast to MAX+PLUS II, the choice box selection of "Bring up the Quartus GUI" allows the additional step of setting up a project and completing the design compilation.

6. Wireload model support is functional.

7. When possible, LeonardoSpectrum supports absorption of NOT gate into WYSIWYG primitives.

8. By default, GND/VCC are exported as cells. In Quartus, the preference is to export GND/VCC as undriven nets.
   Set the output->edif out->write power/ground as undriven nets to true in LeonardoSpectrum. For example, script command:
   ```
   set edifout_power_ground_style_is_net TRUE
   ```

APEX Technology Support

The APEX 20K/20KE devices use the best features from the current Altera FLEX 10K and FLEX 6K family architectures as follows:
- The LE (logic element) structure for FLEX 6 remains the same.
- Each LE consists of 4 i/p LUT, D flip flops, carry and cascade chains.
- A LAB (logic array block) consists of 10 LEs.
- A set of 16 LABs + 1 ESB (embedded system block) makes a MegaLAB.

The LAB wide control signals are:
- Synchronous Load - FLEX 10
- Asynchronous Load - FLEX 6
- Synchronous Clear - FLEX 6
- Clock Enable - similar to FLEX 10

Note: References to FLEX 10K and FLEX 6K in this chapter are for comparison purposes only.
Note: There are two new LAB wide clock enable functions. The clock enable in FLEX 10K utilizes one LUT input.

The following are carry chain changes:
  • Counter mode incorporates lab-wide up/down or count enable.
  • Dedicated logic to start counter from LE1.

The following are output configurations. The normal mode LE can implement three distinct outputs:
  • combinational out
  • registered output
  • cascade output

APEX ESB (Embedded System Block)

The ESB can implement RAMs, ROMs, or pterm logic as follows:
  • Single-port RAMs and dual-port RAMs - both synchronous and asynchronous - similar to FLEX 10KE.
  • ROMs - similar to FLEX 10KE.
  • pterm logic: Each ESB contains 32 literals with 16 macrocells in each ESB. Implement the following circuits with pterm: wide multiplexers, state machines, wide input OR gates, and high fan-in sum-of-product equations.

Quartus Integration

Quartus is now encapsulated in the P&R FlowTab of the GUI for LeonardoSpectrum as follows:
  • All constraints are set in LeonardoSpectrum for Quartus.
  • Quartus is executed automatically by P&RIntegrator. The results of cross probing are available in the information window.

Constraint Passing

LeonardoSpectrum supports passing constraints to Quartus using the Quartus NativeLink API features. The following constraints are defined on the GUI or with attributes on the interactive command line shell and passed to Quartus:
  • Global Constraints
  • Clock Frequency
  • Input and Output: Pin Locations
  • Part Numbers and Speed Grades
Constraint Passing Examples

Three project files are generated by LeonardoSpectrum for Quartus.

- traffic.psf (project settings)
- traffic.csf (compiler settings)
- traffic.quartus (project file)

Traffic.psf (project settings)

```plaintext
DEFAULT_DEVICE_OPTIONS
{
  RESERVE_PIN = "AS INPUT TRI-STATED";
  RESERVED_ALL_UNUSED_PINS = "AS OUTPUT DRIVING GROUND";
  HEXOUT_FILE_COUNT_DIRECTION = UP;
  HEXOUT_FILE_START_ADDRESS = 0;
  GENERATE_HEX_FILE = OFF;
  GENERATE_RBF_FILE = OFF;
  GENERATE_TTF_FILE = OFF;
  RESERVE_DATA7_THROUGH_DATA1_AFTER_CONFIGURATION = OFF;
  RESERVE_RDYNBUSY_AFTER_CONFIGURATION = OFF;
  RESERVE_NWS_NRS_NCS_CS_AFTER_CONFIGURATION = OFF;
  DISABLE_CONF_DONE_AND_NSTATUS_PULLUPS_ON_CONFIG_DEVICE = OFF;
  AUTO_INCREMENT_CONFIG_DEVICE_JTAG_USER_CODE = ON;
  CONFIG_DEVICE_JTAG_USER_CODE = FFFFFFFF;
  CONFIGURATION_DEVICE = EPC2LC20;
  USE_CONFIGURATION_DEVICE = ON;
  ENABLE_INIT_DONE_OUTPUT = OFF;
  ENABLE_LOCK_OUTPUT = OFF;
  ENABLE_DEVICE_WIDE_OE = OFF;
  ENABLE_DEVICE_WIDE_RESET = OFF;
  RELEASE_CLEARS_BEFORE_TRI_STATES = OFF;
  AUTO_RESTART_CONFIGURATION = ON;
  USER_START_UP_CLOCK = OFF;
  CONFIGURATION_SCHEME = "PASSIVE SERIAL";
  JTAG_USER_CODE = FFFFFFFF;
}
```

Traffic.csf (compiler settings)

```plaintext
CHIP(traffic)
{
```
AUTO_RESTART_CONFIGURATION = ON;
RELEASE_CLEARS_BEFORE_TRI_STATES = OFF;
USER_START_UP_CLOCK = OFF;
ENABLE_DEVICE_WIDE_RESET = OFF;
ENABLE_DEVICE_WIDE_OE = OFF;
ENABLE_INIT_DONE_OUTPUT = OFF;
ENABLE_LOCK_OUTPUT = OFF;
JTAG_USER_CODE = FFFFFFFF;
CONFIGURATION_SCHEME = "PASSIVE SERIAL";
USE_CONFIGURATION_DEVICE = ON;
CONFIGURATIONDEVICE = EP2LC20;
CONFIGDEVICE_JTAG_USER_CODE = FFFFFFFF;
AUTO_INCREMENTCONFIGDEVICE_JTAG_USER_CODE = ON;
DISABLE_CONF_DONE_AND_NSTATUS_PULLUPS_ONCONFIGDEVICE = OFF;
GENERATE_TTF_FILE = OFF;
GENERATE_RBF_FILE = OFF;
GENERATE_HEX_FILE = OFF;
HEXOUT_FILE_START_ADDRESS = 0;
HEXOUT_FILE_COUNT_DIRECTION = UP;
RESERVED_ALL_UNUSED_PINS = "AS OUTPUT DRIVING GROUND";
RESERVE_NWS_NRS_NCS_CS_AFTER_CONFIGURATION = OFF;
RESERVE_EDYNBUSY_AFTER_CONFIGURATION = OFF;
RESERVE_DATA7_THROUGH_DATA1_AFTER_CONFIGURATION = OFF;
DEVICE = "EP20K200RC208-3";
sensor1 : LOCATION = PIN_3;
red1 : LOCATION = PIN_12;
}

Traffic .quartus (project file)

COMPILER_SETTINGS
{
  FOCUS_ENTITY_NAME = |traffic;
  RUN_TIMING_ANALYSES = ON;
  USE_TIMING_DRIVEN_COMPILATION = ON;
  COMPILEDATION_LEVEL = FULL;
  SAVE_DISK_SPACE = ON;
  SPEED_DISK_USAGE_TRADEOFF = NORMAL;
  FAMILY = APEX20KE;
}
Cross Probing

LeonardoSpectrum provides support for cross probing from within Quartus into the original HDL files.

When targeting APEX 20K/20KE, LeonardoSpectrum generates a cross reference .xrf file together with the EDIF netlist. This allows Quartus users to seamlessly cross probe into the original HDL design files from the floor plan view. Refer to the following example for the traffic.vhd demo.

**Example Demo:** design_name = traffic.vhd

<table>
<thead>
<tr>
<th>instance</th>
<th>comp,</th>
<th>red1_obuf,</th>
<th>red1_obuf,</th>
<th>traffic,</th>
<th>1, 57::57:</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance = comp,</td>
<td>green1_obuf,</td>
<td>green1_obuf,</td>
<td>traffic,</td>
<td>1, 57::57:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>red2_obuf,</td>
<td>red2_obuf,</td>
<td>traffic,</td>
<td>1, 57::57:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>green2_obuf,</td>
<td>green2_obuf,</td>
<td>traffic,</td>
<td>1, 57::57:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>reg_state7,</td>
<td>reg_state7,</td>
<td>traffic,</td>
<td>1, 35::35:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>reg_state6,</td>
<td>reg_state6,</td>
<td>traffic,</td>
<td>1, 57::57:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>reg_state5,</td>
<td>reg_state5,</td>
<td>traffic,</td>
<td>1, 57::57:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>reg_state4,</td>
<td>reg_state4,</td>
<td>traffic,</td>
<td>1, 57::57:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>reg_state3,</td>
<td>reg_state3,</td>
<td>traffic,</td>
<td>1, 35::35:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>reg_state2,</td>
<td>reg_state2,</td>
<td>traffic,</td>
<td>1, 57::57:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>reg_state1,</td>
<td>reg_state1,</td>
<td>traffic,</td>
<td>1, 57::57:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>reg_state0,</td>
<td>reg_state0,</td>
<td>traffic,</td>
<td>1, 35::35:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>clock,</td>
<td>clock,</td>
<td>traffic,</td>
<td>1, 29::29:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>sensor1,</td>
<td>sensor1,</td>
<td>traffic,</td>
<td>1, 29::29:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>sensor2,</td>
<td>sensor2,</td>
<td>traffic,</td>
<td>1, 29::29:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>reset,</td>
<td>reset,</td>
<td>traffic,</td>
<td>1, 29::29:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>red1,</td>
<td>red1</td>
<td>traffic,</td>
<td>1, 30::30:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>yellow1,</td>
<td>yellow1</td>
<td>traffic,</td>
<td>1, 30::30:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>green1,</td>
<td>green1</td>
<td>traffic,</td>
<td>1, 30::30:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>red2,</td>
<td>red2</td>
<td>traffic,</td>
<td>1, 30::30:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>yellow2,</td>
<td>yellow2</td>
<td>traffic,</td>
<td>1, 30::30:</td>
<td></td>
</tr>
<tr>
<td>instance = comp,</td>
<td>green2,</td>
<td>green2</td>
<td>traffic,</td>
<td>1, 30::30:</td>
<td></td>
</tr>
</tbody>
</table>
Simulation with Pre-Layout Verification (Optional)

After optimization is complete, you may write out and compile a VHDL or Verilog netlist, compile the WYSIWYG cell models, and then test and simulate the design.

LPM (Library Parameterized Module) RAMs/ROMs

If you have LPM RAMs and ROMs in a design, then create simulation models with genmem (generate memory). Refer to the Altera FLEX chapter.

Improvements and New Features

FSM Encoding (binary, gray, random, onehot, twohot, auto)

For auto encoding, LeonardoSpectrum varies the encoding based on bit width. Moreover, enumerated types with fewer elements than global integer lower_enum_break are encoded as binary; while larger enumerated types are encoded as onehot. Values larger than global integer upper_enum_break are encoded as binary. The auto default allows LeonardoSpectrum to select encoding on a case-by-case basis. Note: If LeonardoSpectrum selects onehot for your auto encoded design, then “onehot encoding” is printed for the log file of your design.

The encoding variable determines how LeonardoSpectrum encodes enumerated types, and implements a state machine with a state vector of an enumerated type.

Twohot Encoding

Twohot encoding is now added to FSM encoding (binary, gray, random, onehot, twohot, auto). Twohot sets two flip flops high for each state. The twohot encoding requires more flip flops than binary and fewer flip flops than onehot. Twohot encoding may be beneficial to large FSMs where onehot uses too many flip flops, and binary requires too much decode logic. Refer to the HDL Synthesis guide, Chapter 2, for more encoding information.

Note: Encoding is supported for APEX technologies with small enumerated types of up to five elements.
Complex I/O Mapping

The apex_map_complex_ios variable has been renamed to altera_map_complex_ios to allow LeonardoSpectrum to support complex I/O mapping for both Altera APEX and FLEX 10K. Refer also to the Altera FLEX chapter and to the Command Reference guide.

APEX Devices Supported

<table>
<thead>
<tr>
<th>APEX 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 3</td>
</tr>
<tr>
<td>Speed Grades supported: 1, 2, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP20K100</td>
</tr>
<tr>
<td>EP20K200</td>
</tr>
<tr>
<td>EP20K400</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APEX 20KE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 3</td>
</tr>
<tr>
<td>Speed Grades supported: 1, 2, 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP20K100E</td>
</tr>
<tr>
<td>EP20K200E</td>
</tr>
<tr>
<td>EP20K400E</td>
</tr>
<tr>
<td>EP20K600E</td>
</tr>
<tr>
<td>EP20K1000E</td>
</tr>
</tbody>
</table>

APEX 20K/20KE Flow Diagram

Figure 8-1 shows the progress of your design through the synthesis steps to Place and Route.
Figure 8-1. APEX 20K/20KE Flow Diagram.
User Defined PTERM (Product-Term) Support

The "implement in pterm" option in LeonardoSpectrum allows the module for the selected instance to be implemented as sum-of-products. The appropriate project assignments are then passed to the Quartus compiler to map the module into Embedded System Block.

(ESB) PTERM WYSIWYG Elements

The GUI "Implement In PTERM" check box is located on the Constraint flow tab, Module power tab. Refer to the User's guide, Chapters 5 and 10, for more PTERM information.

Altera APEX 20K/20KE: Library Changes:

Cell names for the internal I/O cells used by LeonardoSpectrum for I/O mapping have changed to be consistent with the I/O cell configuration. However, if you have saved an .xdb file for your design after optimization for an APEX 20K/20KE technology, then the .xdb file cannot be read back into LeonardoSpectrum.

Altera APEX 20K/20KE Compatible Vendor Tools

APEX20K Quartus 9910 and Quartus 9906
APEX20KE Quartus 9910
This chapter presents information specific to the use of Altera FLEX as a source or target technology. LeonardoSpectrum supports the FLEX 6K/8K/10K, FLEX 10KE/10KA/10KB architectures. This chapter is divided as follows:

- Before Beginning
- Altera FLEX Architecture
- Design Flow
- Synthesis and Optimization Features
- Data Path Synthesis and Modgen Implementation
- Embedded Array Block (EAB)
- Optimization Style Points
- Reporting
- Design I/Os
- Using FLEX Designs as Input
- How to Get Best Results with MAX+PLUS II
- Features for Altera FLEX
- Altera FLEX Family Supported Devices
- Altera FLEX 10KE/KA/KB devices
- MAX+PLUS II Clique Assignments Information

Before Beginning

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.
Altera FLEX Architecture

Logic Elements (LEs)
LEs are also known as Logic Cells (LCells). The combinational logic part of an LE is a look up table (LUT). LEs are at the core of Altera FLEX architecture and contain both combinational logic and registers. The combinational logic is limited by the number of inputs; functions of up to four inputs can be implemented in one LE. Functions with more inputs must be separated into multiple levels of LEs. Each LE also contains a register with clear and preset inputs. In addition, LEs are grouped in LABs. All LEs in a LAB share control signals and connect to each other through special cascade routing resources.

Input/Output Elements (IOEs)
IOEs in Altera FLEX can be configured as inputs, outputs, tristate outputs, or bi-directional pins, with a register in either input or output path.

Embedded Array Blocks (EABs)
EABs are for FLEX 10K only and implement memory functions. Each EAB provides 2048 bits which can create a RAM, ROM, FIFO, or dual-port RAM.

Design Flow
LeonardoSpectrum accepts hierarchical descriptions in VHDL or Verilog. Input designs can be optimized and mapped to any Altera FLEX architecture. LeonardoSpectrum can also target designs from other FPGA vendors to Altera FLEX architectures. Refer to Figures 9-1 and 9-2. During reading, the input design is translated into intermediate data structures by LeonardoSpectrum. FLEX-specific optimization is then performed. This optimization guarantees that combinational functions do not have more than a limited number of inputs.

This process is called fanin-limited optimization. The next step is to map the design into lookup tables (LUTs) using LUT mapping. Before writing the design, a lut_function equation string is attached to each lookup table to represent the combinational logic function. A hierarchical EDIF netlist in a single file is then written.

After MAX+PLUS II or Quartus is run for place-and-route, LeonardoSpectrum can back-annotate the post place-and-route timing and generate an SDF timing file and a VHDL netlist. LeonardoSpectrum generates the VHDL and SDF files simultaneously for consistency. These files can be used to simulate the design with post place-and-route timing for functional verification.
Figure 9-1  Simplified Design Flow
Figure 9-2  Simplified Back Annotation
Synthesis and Optimization Features

This section is divided as follows:

- Fanin Limited Optimization
- Look Up Table (LUT) Mapping
- Data Path Synthesis
- Library of Parameterized Modules (LPM) Support for FLEX 10K
- Genmen Support
- EAB Support

To take advantage of the Altera FLEX architecture, Altera FLEX-specific optimization algorithms and features are offered by LeonardoSpectrum.

- A fanin limited optimization algorithm is performed. This algorithm utilizes the limited number of inputs of the Altera FLEX logic element.
- A lookup table mapping algorithm (LUT mapping) is then performed. This algorithm tries to find an optimal mapping of combinational logic into lookup tables and cascade gates.
- Data Path Synthesis: RAM components can be instantiated through LPM conventions or inferred and implemented through modgen. Also, counters are automatically inferred and implemented to fit the FLEX architecture. Data path elements are implemented using modgen to take advantage of the following FLEX-specific features: carry chains and cascade chains.
- Clock Enable Detection: Clock Enable logic is automatically detected for registers.
- Constraint driven timing optimization. This algorithm understands the FLEX architecture and speeds up critical paths in the design.

**Fanin Limited Optimization**

The key architectural feature of the FLEX FPGA is that the LE can be any function of four inputs. A 4-input XOR uses the same area and is as fast as a 4-input AND gate. The function shown in Equation E-1 can be solved in two ways. Here are Solutions (1) and (2) for Equation E-1:

\[ X = (A \cdot (B+C)) + (B \cdot D) + (E \cdot F \cdot G \cdot H \cdot I) \]  

(E-1)
Solution (1)

You can decompose a function into a simpler AND/OR equivalent representation, and then split the gates with large fan-in into multiple gates. Represented in AND and OR gates, \( X \) is decomposed as:

\[
X = T1 + T2 + T3 \\
T1 = A \cdot T4 \\
T2 = B \cdot D \\
T3 = E \cdot F \cdot G \cdot H \cdot I \\
T4 = B + C. 
\]

Since \( T3 \) has more than four inputs, further decomposition is required:

\[
T3 = E \cdot F \cdot G \cdot T5 \\
T5 = H \cdot I \]  

(E-3)

After fully decomposing the design, you can use the Altera FLEX MAX+PLUS II place-and-route software to place the design into physical LEs. In this example, \( T3 \) and \( T5 \) cannot be merged because of fan-in limitations. Next, combine \( T1 \) with \( T4 \) and \( X \) with \( T2 \). This gives the following partitioning to four LEs:

\[
\begin{align*}
LE_1: X &= T1 + (B \cdot D) + T3 \\
LE_2: T1 &= A \cdot (B + C) \\
LE_3: T3 &= E \cdot F \cdot G \cdot T5 \\
LE_4: T5 &= H \cdot I \\
\end{align*} \]  

(E-4)

Note – The critical path is \( LE_4 \rightarrow LE_3 \rightarrow LE_1 \), resulting in three levels of LEs for the delay.
Solution (2)

A different decomposition of Equation (E-1) yields partitioning into three LEs:

\[
\begin{align*}
X &= T_1 + (T_2 \cdot E) \\
T_1 &= A \cdot (B + C) + (B \cdot D) \\
T_2 &= F \cdot G \cdot H \cdot I
\end{align*}
\] (E-5)

Since each of the three equations have no more than four inputs, each equation can be placed into a LE. This design, when implemented, has only two LEs in the critical path, resulting in a faster and smaller design.

Lookup Table (LUT) Mapping

Lookup table mapping puts logic into 4-input lookup tables and CASCADE gates with the objective of minimizing the total number of lookup tables or to minimize the delay. In the output EDIF netlist, the lookup table boundaries are marked with LCell buffers. For example, a 4-1 multiplexer description follows:

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

LUT mapping maps this multiplexer to two 4-input LUTs and one cascade gate. The schematic of the mapped circuit is shown in Figure 9-3.

![Figure 9-3 Mapped Circuit Schematic](image.png)
LeonardoSpectrum decomposes the LUTs, Figure 9-3, to AND-OR gates for output to MAX+PLUS II. The output netlist is represented in Figure 9-4.

**Figure 9-4  Output Netlist**

**Max Fanout**

Use the Max Fanout field on the GUI to override the default max fanout load specified in the library. Refer to Table 9-1. However, a synthesized netlist with high fanout nets may be a problem for the place and route tool. The place and route tool usually splits the net arbitrarily. High fanout nets can cause significant delays on wires and become unroutable. On a critical path, high fanout nets can cause significant delays in a single net segment and cause the timing constraints to fail.

To eliminate the need for splitting of the net by the place and route tool, the synthesis tool must maintain a reasonable number of fanouts for a net. LeonardoSpectrum tries to maintain reasonable fanout limits for each for each target technology. Default fanout limits are derived from the synthesis library.

**Note:** The LUT buffering and replication is supported for the Altera FLEX 6/8/10 and 10KA/KE/KB technologies.

LeonardoSpectrum maintains reasonable fanouts by replicating the driver which results in net splitting. If replication is not possible, the signal is buffered. The buffering of high fanout primary input signals is an example. Buffering the signal causes the wire to be slower by adding intrinsic delays.
On specific nets, you can use the interactive command line shell syntax to set an attribute to control the max_fanout value:

```
set_attribute -net <net_name> -name lut_max_fanout -value <int>
```

Note: Setting this attribute takes precedence over any global fanout specifications.

**User Options to Control LUT Mapping**

Refer to Table 9-1 for mapping options.

- Lock LCells - By default this option is selected.

- Lock LCells - If Lock LCells is not selected, then this directs LeonardoSpectrum not to force LCell buffers in the output EDIF netlist. You can then use MAX+PLUS II to map the combinational logic into LCells. FAST is the recommended setting for MAX+PLUS II GLOBAL_PROJECT_SYNTHESIS_STYLE.

- Map Cascades - By default this option is selected, and that directs LeonardoSpectrum to map to cascade gates where applicable.

Note: Before writing an EDIF netlist that can be read by MAX+PLUS II software, LeonardoSpectrum attaches a lut_function property to each lookup table.
Table 9-1. Mapping Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Fanout: window</td>
<td>n/a</td>
<td>set lut_max_fanout &lt;int&gt;</td>
<td>n/a</td>
</tr>
<tr>
<td>Map Cascades</td>
<td>on</td>
<td>default (true)</td>
<td>default (true)</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set altera_use_cascades false -nocascades</td>
<td></td>
</tr>
</tbody>
</table>

Controls mapping logic to cascade gates for Altera FLEX. When false then Lock LCells should be true to allow MAX+PLUS II to map cascades.

| Lock LCells                | on           | default (false)                                            | default (false)     |
|                            | off          | set dont_lock_lcells true -dont_lock_lcells                 |                     |

Controls locking of LCells. When true LCell boundaries are not forced in the netlist, the boundaries are written as soft. MAX+PLUS II can change LCell boundaries.

| Add I/O Pads               | on           | optimize -chip default                                      | -chip default       |
| (see Optimize Flow Tab)    | off          | optimize -macro                                             | -macro              |

In chip mode I/O pads are inserted in design. Refer to the User’s guide for more I/O information.
Data Path Synthesis and Modgen Implementation

LeonardoSpectrum supports various technology-specific implementations of arithmetic and relational operators used in VHDL or Verilog. Since these implementations have been designed optimally for a specific target technology, the synthesis results are usually smaller and/or faster and take less time to compile. LeonardoSpectrum supports module generation for Altera FLEX 6K/8K/10K.

The following operators are supported for Altera FLEX technologies:

- **Relational Operators** = /= < <= > >=
- **Arithmetic Operators** + - *
- **Miscellaneous Functions**: counters (up/down, loadable, etc), RAMs, incrementer/decrementer, absolute value, unary minus

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

Examples:

- Adder in FLEX modgen is implemented using dedicated CARRY chain available in FLEX architectures to implement the adder carry logic. This leads to very fast carry propagation and results in excellent timing performance.
- Counters in FLEX modgen make use of counter modes available in FLEX architectures which results in faster and smaller designs.
- RAMs in FLEX 10K modgen use dedicated RAMs available in FLEX 10K architecture.
User-Defined Switches

As shown in Table 9-2 you can control modgen implementation that is optimized for area or delay with options.

Table 9-2. Optimize Options

<table>
<thead>
<tr>
<th>Optimize Options</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto - Picks smallest if optimization in area mode; and picks fastest if optimization in delay mode.</td>
<td>on (default)</td>
<td>set modgen_select auto</td>
<td>-select_modgen=auto</td>
</tr>
<tr>
<td>Smallest - Picks the best compact implementation available.</td>
<td>off</td>
<td>set modgen_select smallest</td>
<td>-select_modgen=smallest</td>
</tr>
<tr>
<td>Small - Picks a compact implementation.</td>
<td>off</td>
<td>set modgen_select small</td>
<td>-select_modgen=small</td>
</tr>
<tr>
<td>Fast - Picks a fast implementation.</td>
<td>off</td>
<td>set modgen_select fast</td>
<td>-select_modgen=fast</td>
</tr>
<tr>
<td>Fastest - Picks the fastest implementation available.</td>
<td>off</td>
<td>set modgen_select fastest</td>
<td>-select_modgen=fastest</td>
</tr>
</tbody>
</table>

If Use Technology Specific Module Generation Library is selected then a technology-specific modgen library is selected. If not selected, then a default library is used.
LPM (Library of Parameterized Modules) Support for FLEX 10K

This section presents the following:

- ROMs
- RAMs

ROMs

ROM inference support is available.

Description

Currently you can implement ROM behavior in HDL. This is done with case statements or as a table in VHDL or Verilog. The HDL is then converted to a variety of combinational logic in LeonardoSpectrum and mapped to LEs.

The ROM table can be mapped into a combination of decoders, muxes, or constant nets. This depends on the output data bit pattern. Better utilization may be achieved if the ROM tables are mapped to specialized blocks in FPGAs: for example, an EAB in the Altera FLEX10K technology.

LeonardoSpectrum can detect ROMs and can support mapping of ROM tables. Since several types of combinational logic can be mapped to ROMs, a flexibility is needed to determine if a detected ROM should be implemented. Implementation uses default mapping or maps the ROM into EABs.

ROM Detection

By default the minimum size of the ROMs detected is 256. If you want to detect ROMs smaller in size, then set the variable:

```
set mem_minimum_size 64, for example.
```

```
set mem_minimum_size 0, to detect ROMs of all sizes.
```

ROM Data Extraction

After the ROM network is detected, the ROM data for the network must be elaborated. The ROM data is created and converted to the format expected by the target technology. Refer to Verilog or VHDL Design Example with ROM and use the variable.
```verilog
module rom_32x4 (addr, dout);
input [4:0] addr;
output [3:0] dout;
reg [3:0] dout;
always @(addr)
begin
    case (addr)
        0:dout = 4'b1110;
        1:dout = 4'b0100;
        2:dout = 4'b1110;
        3:dout = 4'b1001;
        4:dout = 4'b1111;
        5:dout = 4'b0011;
        6:dout = 4'b1000;
        7:dout = 4'b0001;
        8:dout = 4'b0110;
        9:dout = 4'b0001;
        10:dout = 4'b1100;
        11:dout = 4'b0000;
        12:dout = 4'b0110;
        13:dout = 4'b0000;
        14:dout = 4'b0100;
        15:dout = 4'b0110;
        16:dout = 4'b1110;
        17:dout = 4'b0100;
        18:dout = 4'b1110;
        19:dout = 4'b1001;
        20:dout = 4'b1111;
        21:dout = 4'b0011;
        22:dout = 4'b1000;
        23:dout = 4'b0001;
        24:dout = 4'b0110;
        25:dout = 4'b0001;
        26:dout = 4'b1100;
        27:dout = 4'b0000;
        28:dout = 4'b0110;
        29:dout = 4'b0000;
        30:dout = 4'b0100;
        31:dout = 4'b0110;
    endcase
end
endmodule
```
VHDL Design Example with ROM

Note: set mem_minimum_size 0

Library ieee;
use ieee.std_logic_1164.all;
package Table_rom is
   Type rom_type is array ( 0 to 32 - 1 ) of STD_LOGIC_VECTOR
   ( 8 - 1 downto 0 );
   constant ROM : rom_type := '
end Table_rom;
Model ROM Code

This is an example of VHDL code from model ROM, primitive, block code M01.

```
Library ieee;
use ieee.std_logic_1164.all ;
entity rom_example1 is
generic ( DEPTH : in INTEGER ;
      DATA_WIDTH : in INTEGER ;
      ADDR_WIDTH : in INTEGER );
   port( ADDR : in STD_LOGIC_VECTOR((ADDR_WIDTH-1)downto 0));
   DATAOUT : out STD_LOGIC_VECTOR((DATA_WIDTH-1)downto 0));
end rom_example1 ;
```
Input Data Sets for ROM Inferencing

VHDL example (one of two):

```vhdl
Library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.Table_rom.all;
architecture ex1 OF rom_example1 is
begin
    proc_addr: process (ADDR)
    variable addr_int: integer range 0 to DEPTH-1;
    begin
        addr_int := CONV_INTEGER(UNSIGNED(ADDR));
        DATAOUT <= ROM(addr_int);
    end process;
end ex1;
```
VHDL example (two of two):

```vhdl
Library ieee;
use ieee.std_logic_1164.all;

entity top is
  port ( addr_in : in STD_LOGIC_VECTOR((5-1) downto 0);
         clock : in STD_LOGIC;
         reset : in STD_LOGIC;
         output_enbl : in STD_LOGIC;
         data_out : out STD_LOGIC_VECTOR((8-1) downto 0) );
end top;

architecture top of top is
component rom_example1
  generic ( DEPTH : in INTEGER;
            DATA_WIDTH : in INTEGER;
            ADDR_WIDTH : in INTEGER);
  port (ADDR : in STD_LOGIC_VECTOR((ADDR_WIDTH-1)downto 0);
        DATAOUT :out STD_LOGIC_VECTOR((DATA_WIDTH-1) downto 0));
end component;
begin
  M0 : rom_example1
    generic map (DEPTH => 32 ,
                 DATA_WIDTH => 8 ,
                 ADDR_WIDTH => 5 )
    port map ( ADDR => addr_in,
                  DATAOUT => data_out);
end top;
```
**Mapping**

The detected ROM network is mapped to a LPM_ROM as shown in the table:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LPM_TYPE</td>
<td>element type (set to LPM_ROM)</td>
</tr>
<tr>
<td>LPM_WIDTH</td>
<td>data size</td>
</tr>
<tr>
<td>LPM_WIDTHAD</td>
<td>address size</td>
</tr>
<tr>
<td>LPM_NUMWORDS</td>
<td>memory size</td>
</tr>
<tr>
<td>LPM_ADDRESS_CONTROL</td>
<td>unregistered or registered</td>
</tr>
<tr>
<td>LPM_OUTDATA</td>
<td>unregistered or registered</td>
</tr>
<tr>
<td>LPM_FILE</td>
<td>name of file containing the ROM data</td>
</tr>
</tbody>
</table>

The target place and route tool maps the LPM_ROM to the appropriate logic element in the technology.

**ROM Data File**

LeonardoSpectrum generates a ROM data file that contains the ROM programming data as part of the LPM ROM instantiation. This data is in the Intel Hex Object File format which is supported by Altera tools. The following example is for a 32x5 ROM:

```
:020000040000fa
:080000000030f1f0f030f1f1f68
:08000800071f001f0107011f83
:080010001070107071f0f0f6e
:080018000f7070f1f0f0f1f58
:00000001ff
```

**Type of Inferencing ROMs**

LeonardoSpectrum supports the following types of ROMs:

- Asynchronous ROMs
- Synchronous ROMs

**Altera FLEX 10K Modgen Support for ROMs**

The Altera FLEX 10K Modgen Library supports asynchronous and synchronous ROMs. The \texttt{lpm\_roms} are mapped to Embedded Array Blocks (EABs) in the Altera FLEX 10K technology. This allows for better utilization of the device.
If the selected device contains the number of EABs that are required to implement a ROM of the desired size, then LeonardoSpectrum automatically checks and creates lpm_roms. If the ROM size does not fit the available EABs, then the default implementation is used.

**RAMs**

LeonardoSpectrum supports LPM RAM components for FLEX 10K. There are two levels of support for RAMs in LeonardoSpectrum:

- **RAM instantiation**

  LeonardoSpectrum supports RAM_DQ, RAM_IO through modgen. You can instantiate the RAM_IO, RAM_DQ components from VHDL and Verilog. LeonardoSpectrum then implements them using Altera's LPM components (LPM_RAM_DQ, LPM_RAM_IO) and specifies all the necessary names and properties that are required by MAX+PLUS II. By instantiating the modgen RAM_DQ, RAM_IO, you do not need to know what names and properties are required.

- **Direct LPM components instantiation**

  LPM components can be instantiated directly. You must specify the correct LPM names and properties required by MAX+PLUS II.

**Instantiating RAMs (ram_dq) from VHDL**

Since the description of ram_dq is in file: $EXEMPLAR/data/modgen/flex10.vhd, you should run analyze $EXEMPLAR/data/modgen/flex10.vhd in LeonardoSpectrum, before reading in your design VHDL.
VHDL Example:

Note: before beginning the following VHDL example, read in the lpm_components package from the Altera installation:

read $ALTERA_PATH/vhdl93/lpm/lpm_components.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.LPM_components.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.lpm_ram_dq.all;

entity ram256x6 is
  port (
    RD_ADR : in std_logic_vector (7 downto 9);
    WR_ADR : in std_logic_vector (7 downto 0);
    WR_DATA : in std_logic_vector (5 downto 0);
    WE     : in std_logic;
    WR_ACCESS : in std_logic;
    RAM_DATA : out std_logic_vector (5 downto 0));
end ram256x6;

architecture RAM256x6_A of RAM256x6 is
  signal ADR: std_logic_vector (7 downto 0);
begin
  ADR <= WR_ADR when WR_ACCESS='1' else RD_ADR;
  RAM0:LPM_RAM_DQ
  generic map (LPM_WIDTH=>6;
    LPM_WIDTHAD=>8)
  port map (
    DATA => WR_DATA,
    ADDRESS => ADR,
    WE => WE,
    Q => RAM_DATA);
end RAM256x5_A;
```
**Instantiating LPMs from Verilog**

You must use the `Defparam` construct to specify generic values for the instantiated RAM. For a description of `Defparams` needed for each RAM and their default values, refer to *Library of Parameterized Modules (LPM)* document. `Defparams` values that are not specified use default values.

**Verilog example:**

```verilog
module lpm_ram_dq ( q, data, inclock, outclock, we, address) ;

    parameter lpm_type = "lpm_ram_dq" ;
    parameter lpm_width = 1 ;
    parameter lpm_widthad = 1 ;
    parameter lpm_numwords = 2 ;
    parameter lpm_file = "UNUSED" ;
    parameter lpm_indata = "REGISTERED" ;
    parameter lpm_outdata = "REGISTERED" ;
    parameter lpm_addr_control = "REGISTERED" ;
    parameter polar_data = "NORMAL" ;
    parameter polar_inclock = "NORMAL" ;
    parameter polar_outclock = "NORMAL" ;
    parameter polar_we = "NORMAL" ;
    parameter polar_address = "NORMAL" ;
    parameter polar_q = "NORMAL" ;

    input [lpm_width-1:0] data ;
    input [lpm_widthad-1:0] address ;
    input inclock, outclock, we ;
    output [lpm_width-1:0] q;

endmodule // lpm_ram_dq
```
Verilog Example:

```verilog
module test_lpm_ram (q, data, inclock, outclock, we, address);

  parameter width = 8 ;
  parameter widthad = 2 ;
  parameter numwords = 4 ;

  input [width-1:0] data ;
  input [widthad-1:0] address ;
  input inclock, outclock, we ;
  output [width-1:0] q;

  lpm_ram_dq Instance_r (q, data, we, inclock, outclock, address);
  defparam Instance_r.lpm_width = width;
  defparam Instance_r.lpm_numwords = numwords;
  defparam Instance_r.lpm_widthad = widthad;

endmodule
```

**RAM Inferencing**

LeonardoSpectrum infers RAMs automatically from VHDL and Verilog. If the modgen library is loaded, the library uses the LPM description from modgen to implement the RAMs.

Note: Dual-port RAM implementation for FLEX 10K is improved for LeonardoSpectrum.
For example, consider this Verilog description:

```verilog
module ram (clk, datain, addr, dataout);
input clk;
input [0:3] datain;
input [0:1] addr;
output [0:3] dataout;
reg [0:3] r[0:3];
reg [0:1] addr_out;

assign dataout = r[addr_out];
always @(posedge clk)
begin
    addr_out = addr;
r[addr] = datain;
end
endmodule
```

LeonardoSpectrum infers a LPM `ram_dq` for this design.

The schematic of LeonardoSpectrum output is shown in Figure 9-5:

![Figure 9-5 RAM Inferencing](image_url)
Using RAMs

Types of Inferencing RAMs

LeonardoSpectrum supports two types of RAMs:

- RAM_DQ. RAM_DQ is a single-port RAM with separate input and output data lines.
- RAM_IO. RAM_IO is a single-port RAM with bidirectional data lines.

Both of these RAM types support synchronous or asynchronous read and write. These RAMs are automatically inferred by LeonardoSpectrum from VHDL or Verilog.

The inferencing process distinguishes between RAMs that perform the read operation with an address clocked or not clocked by the write clock (read address clocked). Both of the following VHDL examples perform synchronous writes (inclock) and synchronous reads (outclock); LeonardoSpectrum recognizes these VHDL processes as RAMs:

- The first, entity ram_example1, is when the read operation does not have a clocked address.
- The second, entity ram_example2, is when the read operation does have a clocked address.
The first entity, `ram_example1`, is when the read operation does not have a clocked address.
The second, entity `ram_example2`, is when the read operation has a clocked address.

**Altera FLEX 10K Modgen Support for RAMs**

The Altera FLEX 10K Modgen Library supports asynchronous RAMs and synchronous RAMs that clock the read address with the write clock.
**Genmem Support**

Genmem flow is supported for Altera FLEX devices. Genmem is a utility that generates simulation models for memory. You can generate RAMs, ROMs, FIFOs, and dual port RAMs in different sizes. The memory can be synchronous or asynchronous. Genmem writes the simulation model in VHDL or Verilog.

Use the following flow:

1. Run genmem to generate the desired memory.
   
   For example:

   ```
   genmem asynram 256x15 -verilog
   ```

   This command generates a 256x15 asynchronous RAM model and writes the model in Verilog.

2. Instantiate the RAM module in your Verilog input file.

**LPM Instantiation**

You can instantiate any LPM component from VHDL and Verilog formats.

**Genmem Verilog Design**

Synthesize your Verilog design with LeonardoSpectrum by targeting FLEX 10K.

- The `-simple_port_names` option is supported.
- Generate EDIF netlist as an output. The genmem component is treated as a black-box by LeonardoSpectrum. The EDIF netlist that describes your design and the Verilog file generated by genmem is used as input to MAX+PLUS II. The MAX+PLUS II software merges the genmem description into the top level design and places and routes the design.
- You need to include the module declaration in this file. In the following Verilog genmem example, a module declaration is shown.
- Set the `hdl_array_name_style` variable to `%s%d` to ensure that port names generated by LeonardoSpectrum match port names generated by genmem.
- Generate EDIF netlist as an output. The genmem component is treated as a black-box by LeonardoSpectrum.
The EDIF netlist that describes the design and the Verilog file generated by genmem is used as input to MAX+PLUS II place and route. The MAX+PLUS II software merges the genmem description into the top level design and places and routes it.

**LeonardoSpectrum for Genmem**

The genmem module declaration must be included in this file as shown in the Verilog example:

```verilog
module ram_example ( addr, we, d, o);
  input [0:7] addr;
  input we;
  input [0:14] d;
  output [0:14] o;
  asyn_ram_256x15 i1(.Address(addr), .WE(we), .Q(o), .Data(d));
endmodule

// This module declaration is copied from the genmem file */
module asyn_ram_256x15 (Q, Data, WE, Address);

  parameter LPM_FILE = "UNUSED";
  parameter Width = 15;
  parameter WidthAd = 8;
  parameter NumWords = 256;

  input [WidthAd-1:0] Address;
  input [Width-1:0] Data;
  input WE;
  output [Width-1:0] Q;
endmodule
```

**Embedded Array Block (EAB) Support for FLEX 10K**

LeonardoSpectrum allows you to implement wide functions in the Embedded Array Block available on FLEX 10K devices. Special logic functions like: RAMs, ROMs, FIFOs, multipliers, ALUs are good candidates to be implemented in EABs to achieve a faster implementation.
In Leonardo Spectrum, you can access EABs by setting an attribute on the desired instance. The attribute is `implement_in_eab` and the attribute should be set to `on`. This can be set from either VHDL or Verilog. Following is an example for using EABs from VHDL.

```vhdl
entity mult is
    port (A,B: integer range 0 to 15;
          Q: out integer range 0 to 255);
end mult;

architecture behavior of mult is
begin
    Q <= A*B;
end behavior;

entity eab_test is
    port (CLK,MAC,RST:bit; A,B: integer range 0 to 15;
          Q: buffer integer range 0 to 255);
end eab_test;

architecture behavior of eab_test is
    signal P: integer range 0 to 255;

    component mult;
        port (A,B: in integer range 0 to 15;
              Q: out integer range 0 to 255);
    end component;

EABs example continued...
```
attribute logic_option:string;
attribute noopt:boolean;
attribute logic_option of ul:label is "implement_in_eab=on";
attribute NOOPT of ul:label is true;

begin
    U1:mult port map (A,B,P); -- Product of A and B
    process (RST,CLK)
        begin
            if (RST='1') then --Reset
                Q <= 0;
            else
                if (CLK='1' and CLK'event) then --Clock (edge triggered)
                    if (MAC='1') then
                        Q <= P+Q;
                    else
                        Q <= P;
                    end if;
                end if;
            end if;
        end process;
end behavior;
Optimization Style Points

Quality of Optimization

Certain types of logic are optimized more efficiently than others: arithmetic circuits such as adders and multipliers are not as optimal as a hand design or predefined macro. These constructs are synthesized more efficiently using module generation, when using VHDL or Verilog RTL as the input format. General control logic and state machines are optimized the same as or more efficiently than hand designs.

Sequential Optimization

LeonardoSpectrum eliminates registers and latches if the register or latch is driven by a constant value, and there is no set/reset used on the register. LeonardoSpectrum also eliminates registers or latches that have inputs that are identical to another register or latch; these registers are essentially merged. LeonardoSpectrum does not move registers or change the phase of the signal stored in the registers.

Reporting

Refer to the User’s Guide for information on reports.

Design I/O

Global Signals

Global signals like clocks and output enable for tri-state signals can be specified by using the buffer_sig command from the Constraint File Editor or the LeonardoSpectrum interactive command line shell. The primitive supported by the FLEX architecture is GLOBAL. You must ensure that the design rules for using such a primitive (number of primitives, signals connected to primitives, etc.) are not exceeded. For example with the constraint file syntax:

```text
BUFFER_SIG GLOBAL clock1
BUFFER_SIG GLOBAL oe
```
**Pin Location**

Pins for I/O signals can be defined in Leonardo Spectrum using commands with the **interactive command line shell** syntax:

```
set_attribute -port name -name PIN_NUMBER -val value
```

Pin assignment is accomplished in **VHDL** with attributes. The syntax is as follows:

```
ATTRIBUTE PIN_NUMBER OF signal_name: SIGNAL IS: value
```

For example:

```
attribute pin_number of en: signal is "P14";
```

LeonardoSpectrum translates the PIN_NUMBER property to Altera-specific CHIP_PIN_LC property as shown in the following VHDL example:

```
--library exemplar;
--use exemplar.exemplar_1164.all;
library synth;
use synth.exemplar.all;
entity test is
  port(  
a, b: in bit;
c: out bit  
);
  attribute PIN_NUMBER of a : signal is chip1@4
end test;
architecture exemplar of test is
begin
  process (a, b)
  begin
    c<=(a and (not b)) or ((not a) and b);
  end process;
end exemplar;
```
The following steps are necessary to do pin assignments in MAX+PLUS II:

1. Click on menu Assign->Devices.
2. Click on Rename Chip and rename chip to chip1 in the window that is displayed.
3. Select a device from Devices.
4. Select OK. MAX+PLUS II assigns pin 4 to signal A when compile is done.

Writing EDIF Output

EDIF is the only interface between LeonardoSpectrum and the Altera MAX+PLUS II software. The following are known problems when writing out EDIF for Altera:

- Altera EDIF file name should match the design name. The design name, and the file name are case sensitive. You can change the design name if needed by using the move command in LeonardoSpectrum.
- If the hierarchy is preserved for hierarchical designs, LeonardoSpectrum writes out hierarchical EDIF. If sub-modules have busses on the boundary, usually LeonardoSpectrum renames these busses.

For example:

```
(port (rename p23 "Q(0)") (direction OUTPUT))
(port (rename p22 "Q(1)") (direction OUTPUT))
(port (rename p21 "Q(2)") (direction OUTPUT))
(port (rename p20 "Q(3)") (direction OUTPUT))
(port (rename p19 "Q(4)") (direction OUTPUT))
(port (rename p18 "Q(5)") (direction OUTPUT))
(port (rename p17 "Q(6)") (direction OUTPUT))
(port (rename p16 "Q(7)") (direction OUTPUT))))
```

This causes problems with Altera MAX+PLUS II EDIF reader since the reader does not handle the rename construct properly. You can solve the problem in two ways:

1.

Note: -simple_port_names creates simple name for vector ports. Refer to User’s guide, Batch Mode chapter.
(2)

Set the variable `hdl_array_name_style` to the Altera bus format.

```
set hdl_array_name_style = %s%d
```

LeonardoSpectrum then creates the following EDIF:

```
(port Q0 (direction OUTPUT))
(port Q1 (direction OUTPUT))
(port Q2 (direction OUTPUT))
(port Q3 (direction OUTPUT))
(port Q4 (direction OUTPUT))
(port Q5 (direction OUTPUT))
(port Q6 (direction OUTPUT))
(port Q7 (direction OUTPUT))))))
```

This EDIF is handled correctly by MAX+PLUS II.

**LMF file**: To take the EDIF generated by LeonardoSpectrum into MAX+PLUS II. The `exemplar.lmf` file is required. The `exemplar.lmf` file is installed in the `$EXEMPLAR/data` area. This file maps the cells LeonardoSpectrum writes to MAX+PLUS II primitives and functions. Make sure that your MAX+PLUS II environment is set to pick up this .lmf file:

If using the MAX+PLUS II GUI then go to menu Interfaces-->EDIF Reader Settings and make LMF1 point to `exemplar.lmf` and click on the checkbox for LMF1.

If using the MAX+PLUS II through command line then edit the .acf file. Search for `EDIF_INPUT_LMF` and make `EDIF_INPUT_LMF1 = exemplar.lmf`. Search for `EDIF_INPUT_USE_LMF1` and make `EDIF_INPUT_USE_LMF1 = ON`. 
Using FLEX Designs as Input

**EDIF Input**

To retarget a FLEX design into other technologies or to optimize a FLEX design using LeonardoSpectrum, the design must fit into a single FLEX device. This ensures that the single EDIF netlist generated by MAX+PLUS II captures the whole design. Also, the EDIF writer must be turned on when running the MAX+PLUS II compiler. If the design does not fit into a single device, switch to a bigger device until the design fits. (LeonardoSpectrum does not care what the input device is, only the input device family or technology.) Use the EDIF file generated by MAX+PLUS II as input to LeonardoSpectrum, with the FLEX library as the source technology, and synthesize to the chosen technology. In LeonardoSpectrum, first load the technology library, then read the design:

```
Leonardo>load_lib flex8
--source lib flex8 is loaded
Leonardo>read_altera my_flex_design.edf
Leonardo>load_lib z_tech
Leonardo>optimize -target z_tech
Leonardo>write output_file.edf
--output netlist EDIF file, mapped to target technology z_tech
```

If the design cannot fit into a single device, let MAX+PLUS II partition the design into several devices and generate several EDIF files. Then, manually write an HDL file that connects the devices correctly. The MAX+PLUS II report file (multiple pin connections section) can be helpful in writing this file. Finally, read all the files into LeonardoSpectrum, first the EDIF files, then the top level file.

EDIF files generated by MAX+PLUS II contain, in addition to all functional I/O pins, a VCC pin and a GND pin. These pins are specific to Altera EDIF files. Consequently, LeonardoSpectrum has a special command for reading Altera files: `read_altera`.

**How to Get Best Results with MAX+PLUS II**

LeonardoSpectrum performs architecture specific optimization for Altera FLEX devices. Note: This information is also for Altera MAX. The Altera MAX+PLUS II then places and routes the netlist. This section discusses the settings required to allow MAX+PLUS II and LeonardoSpectrum to produce the best design.
**Altera FLEX Synthesis**

When LeonardoSpectrum is run targeting Altera FLEX, first optimizes the design and then maps the logic into logic cells.

By default, LeonardoSpectrum enforces the mapping by writing LCELLs into the EDIF netlist. LeonardoSpectrum also maps to Altera FLEX primitives, such as carry chains (to accelerate critical paths for data paths) and cascade chains (to implement wide functions).

You may want to rely on MAX+PLUS II to map the logic cells. You can direct LeonardoSpectrum not to enforce the LCELLs in the EDIF output netlist. On Technology Settings select an Altera technology then open Advanced Settings.

The Lock LCells option is selected by default. Click to clear this option. The interactive command line shell variable `set dont_lock_lcells true` can also be used for this option. Refer again to Table 9-1. LeonardoSpectrum produces an EDIF netlist expressed in terms of AND-OR primitives only. Mapping to LCELLs, Carry, and Cascade is done by MAX+PLUS II.

**Note:** LeonardoSpectrum: For FLEX 6K, refer to the Writing EDIF Output section in this chapter. The Library Mapping File (LMF) discussion provides information for using the `data/exemplar.lmf` file with MAX+PLUS II to configure the EDIF reader.

**Place-and-Route using MAX+PLUS II**

You can either use LeonardoSpectrum to map to FLEX primitives or use MAX+PLUS II. The following settings are recommended by Exemplar:

Use MAX+PLUS II to place-and-route the netlist. DO NOT use the default MAX+PLUS II settings for FLEX. With default settings, MAX+PLUS II does not use the carries or cascades that LeonardoSpectrum writes.

**Flow 1: Altera Primitives are Mapped**

Place-and-route is done by MAX+PLUS II:

1. Load MAX+PLUS II with the EDIF netlist from Exemplar. Ensure that any .vhd or .tdf files do not have the same name as the EDIF netlist (e.g. design.edf and design.vhd), otherwise MAX+PLUS II may use one of these files instead of the EDIF netlist.
2. Start the MAX+PLUS II compiler.
   MAX+PLUS II -> Compiler

3. Set the device.
   Assign -> Device

4. Configure the EDIF reader.

**FLEX 8K or FLEX 10K**

5. Launch the EDIF Netlist Reader Settings dialog box.
   Interfaces -> EDIF Netlist Reader Settings...

6. Set Vendor to Exemplar.

**FLEX 6K, FLEX 8K, and FLEX 10K**

7. Make the Global logic settings. Launch the Global Project Logic Synthesis dialog box.
   Assign -> Global Project Logic Synthesis.

8. Set Global Project Synthesis Style to WYSIWYG. The default is NORMAL.
   NORMAL removes cascade and carry chains and generally degrades FLEX results.

**Consider the following options:**

- **Automatic FAST I/O** can be helpful, improving area and clock to out delays. Automatic FAST I/O can also reduce the maximum clock frequency, add input hold time for FLEX 8K and increase input setup time for FLEX 10K.

- **Automatic Register packing** can improve area and delay. If Automatic Register Packing makes the design fail to fit or route, then turn the register off.

- **Automatic implement in EAB.** This option, which only applies to FLEX 10K is often helpful in moving random logic into available EABs, which improves area, and often improves delay. There are some cases where delay may be slower. For example, an 8 input AND gate is faster in LEs than in EABS.

9. Start the compile run.
Flow 2: Altera Primitives are Mapped by MAX+PLUS II

This flow is an alternative to the LeonardoSpectrum mapper. Mapping in MAX+PLUS II may yield better results. Use the following steps to map place-and-route for the netlist with MAX+PLUS II.

1. Load MAX+PLUS II with the EDIF netlist from Exemplar. Ensure that any .vhd or .tdf files do not have the same name as the EDIF netlist (e.g. design.edf and design.vhd), otherwise MAX+PLUS II may use one of these files instead of the EDIF netlist.

2. Start the MAX+PLUS II compiler.

   MAX+PLUS II -> Compiler

3. Set the device.

   Assign -> Device

4. Configure the EDIF reader.

   FLEX 8K or FLEX 10K

5. Launch the EDIF Netlist Reader Settings dialog box.

   Interfaces -> EDIF Netlist Reader Settings...

6. Set Vendor to Exemplar.

   FLEX 6K, FLEX 8K, FLEX 10K

7. Make the Global logic settings. Launch the Global Project Logic Synthesis dialog box.

   Assign -> Global Project Logic Synthesis.

8. Set Global Project Synthesis Style to FAST. The default is NORMAL. NORMAL removes cascade and carry chains, and generally degrades FLEX results.

   Consider the Following Options:
   • Automatic FAST I/O may improve area and clock to out delays. Automatic FAST I/O can reduce the maximum clock frequency, add input hold time for FLEX 8K, and increase input setup time for FLEX 10K. In summary area improves with a delay penalty.
• **Automatic Register Packing** can improve area and delay. Exemplar recommends this option. Turn the register packing off if the design fails to fit or route.

9. Start the compile run.

**Additional Settings to Get Faster Designs**

You can direct MAX+PLUS II to speed up clock frequency, and use a LeonardoSpectrum timing report to find out the estimated clock frequency. The following settings should be made:

1. Launch the Global Project Logic Synthesis dialog box.
   
   Assign -> Global Project Timing Requirements

2. Assign fmax to be 10% to 15% faster than the frequency as estimated by LeonardoSpectrum. This setting improves MAX+PLUS II results by an average of 10%.

**Features for Altera FLEX**

**MAX+PLUS II Integration and Generation of ACF File**

LeonardoSpectrum can automatically setup the ACF (Altera constraint file for MAX+PLUS II and launch MAX+PLUS II from LeonardoSpectrum's GUI. MAX+PLUS II is launched in a batch mode, and the output is directed to LeonardoSpectrum. You can control MAX+PLUS II functionality from the Altera FLEX technology specific dialog box. You have the following options:

- Generate Altera constraint file (*.ACF) file for MAX+PLUS II (default: on)
- Run MAX+PLUS II after optimization step (default: off)
- Run MAX+PLUS II in batch mode or separately (bring up GUI) (default: batch mode)
- Set MAX+PLUS II compiler options -auto_fast_io and -auto_register_packing
- Option to launch MAX+PLUS II Timing Analysis. This allows you to get the Setup/Hold or Register Performance reports.

**Note:** If you are using a PC then specify a MAX+PLUS II location by setting the MAX+PLUS II field in the Preferences menu. The default location is:

C:\maxplus2
**Command Line Switches**

If running LeonardoSpectrum in batch mode, then MAX+PLUS II functionality is available through the following command line switches:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-maxplus2</td>
<td>launch MAX+PLUS II in batch mode</td>
</tr>
<tr>
<td>-no_acf</td>
<td>suppress generation of ACF file</td>
</tr>
<tr>
<td>-max_ta_reg</td>
<td>analyze register performance</td>
</tr>
<tr>
<td>-auto_fast_io</td>
<td>enable auto fast IO</td>
</tr>
<tr>
<td>-auto_register_packing</td>
<td>enable auto register packing</td>
</tr>
<tr>
<td>-max_exe&lt;executable&gt;</td>
<td>provide full path name of MAX+PLUS II executable file if MAX+PLUS II is not in search path.</td>
</tr>
</tbody>
</table>

**ACF File Generation**

LeonardoSpectrum generates Altera constraint file with all necessary settings. **Note:** LeonardoSpectrum cannot process the existing ACF files. These files are REMOVED when generating a new ACF file. If you are experienced with ACF file settings, then you have the option to suppress ACF file generation.

**User Options with MAX+PLUS II**

You have the option of either using the MAX+PLUS II software or of pointing and clicking in LeonardoSpectrum to setup MAX+PLUS II. An ACF file is then created to be read by MAX+PLUS II and applied to your design. These GUI options build your ACF file for MAX+PLUS II.

**MAX+PLUS II Integration**

You can generate simulation files on the P&R FlowTab for MAX+PLUS II by enabling the appropriate VHDL or Verilog writers.
Schematic Symbol Libraries

The following items were added:

- Instance names were added to all symbols.
- Equations were added on LUT symbols.
- New symbols were added.

Altera FLEX 6K

LeonardoSpectrum has support for FLEX 6K technology. LeonardoSpectrum uses a modgen library to support the data path. This includes support for all flavors of counters available in FLEX 6K architecture. You must run MAX+PLUS II version 8.0 or higher for place-and-route. To support FLEX 6K, LeonardoSpectrum uses an Exemplar library mapping file (LMF). Note: You cannot use the Exemplar LMF file that is shipped with MAX+PLUS II; you must use the LMF file shipped by Exemplar under the data directory.

Pipelined Multiplier

Refer to the User’s guide, Chapter 10 for more information.

Dual Port 10K and 10KE RAM Inferencing

Both FLEX 10K/10KE have improved dual port RAM inferencing.

Wire Load Model for 10KE, 10KA, 10KB

A wire load model is added. This model is based on delay calculations that improves LeonardoSpectrum delay estimates. This is based on statistical results.

Complex I/O Mapping

The apex_map_complex_ios variable has been renamed to altera_map_complex_ios to allow LeonardoSpectrum to support complex I/O mapping for both Altera APEX and FLEX 10K. With this option for Altera FLEX10K, LeonardoSpectrum separates boundary registers from internal Lcells in the output netlist to allow I/O register mapping with MAX+PLUSII.
In addition for Altera FLEX10K/KA/KB/KE complex I/O mapping, in the ACF you must check the Auto Fast I/O choice on the P&R tab. This provides directives to MAX+PLUSII to map registers in I/O elements. On the GUI, a choice box for Map IO registers is available for all Altera FPGA.

Refer also to the Altera APEX chapter and to the Command Reference guide.

**Enhancements to Modgen**

**Improved Operators**

- Constant multiplication is improved; constants are swept away.
- Counters are improved; there are many ways to write counters.
- LeonardoSpectrum detects many of the counters.
- Improved >=, etc. for Altera FLEX architecture (FLEX modgen).
- Improved inc/dec for rollover conditions.

**Existing Libraries**

FLEX 10K: adjusted cascade chain lengths, for example, and reduction operators to fit the smaller LAB input counts of some FLEX 10K devices.

All RAMs: Support for arbitrary size RAMs.

**Non-Square Multipliers**

Non-square multipliers were implemented for the following Altera technologies: FLEX 6K/8K/10K

**LPM (Library of Parameterized Modules) Multiplier**

The LPM multiplier provides better results. **Note: Timing and area estimates are not accurate.**

**Inferencing of LPM RAMs and Counters**

LeonardoSpectrum infers RAMs and Counters from the HDL description and uses LPM components to implement them.
**Instantiate LPM components**

You can instantiate Altera LPM simulation models directly in VHDL and Verilog designs. The different parameters of the LPM instances are set to the required value by doing generic or parameter mapping. A special algorithm looks into a generic/parameter 'lpm_type' and moves all the generic/parameter values to attributes, in the final EDIF netlist for Altera.

You can simulate, as well as synthesize the same design without any change - even with instances of technology specific RAMs, counters etc.

**Lookup Table Functionality**

Functionality of lookup table is written in terms of `lut_function`. For example:

```vhdl
(cellref LUT
    (libraryRef FLEX10K)
    (property lut_function
        (string "(IN1 In2)"
    )
)
```

This makes EDIF netlists generated by LeonardoSpectrum much smaller and MAX+PLUSII processing time is faster.

**Quality of Area and Delay Results**

Quality of results were improved by 10%-20% for Altera FLEX area and delay. The most significant improvements are for targeting Altera FLEX technologies. The improvements are due to the following features:

- Clock Enable detection was improved to detect more complex clock enable logic
- Multipliers are implemented more efficiently
- Improvements in LUT based Optimization algorithms
- Improvements in the method LeonardoSpectrum uses to represent and optimize wide functions, like ROMs.

You can expect significant improvements for both area and delay on designs that contain multipliers, clock enable logic, and ROMs.
**FSM Encoding (binary, gray, random, onehot, twohot, auto)**

For auto encoding, in Altera FLEX 6/8/10 technologies, LeonardoSpectrum varies the encoding based on bit width. Moreover, enumerated types with fewer elements than global integer lower_enum_break are encoded as binary; while larger enumerated types are encoded as onehot. Values larger than global integer upper_enum_break are encoded as binary. The auto default allows LeonardoSpectrum to select encoding on a case by case basis. For example, if LeonardoSpectrum selects onehot for your design, then “onehot encoding” is printed in the log file of your design.

The encoding variable determines how LeonardoSpectrum encodes enumerated types. The encoding variable determines how LeonardoSpectrum implements a state machine with a state vector of an enumerated type.

**Twohot Encoding**

Twohot encoding is now added to FSM encoding (binary, gray, random, onehot, twohot, auto). Twohot sets two flip flops high for each state. The twohot encoding requires more flip flops than binary and fewer flip flops than onehot. Twohot encoding may be beneficial to large FSMs where onehot uses too many flip flops, and binary requires too much decode logic. Refer also to the HDL Synthesis guide, Chapter 2.

**Wire Load Model**

LeonardoSpectrum improves delay estimates with the added wire load model based on delay calculations.

**Compatible Altera FLEX Vendor Tools**

FLEX Series MAX+PLUSII 2.9.3 or earlier
## Altera FLEX Family Supported Devices

Devices supported are: APEX 20K/20KE (APEX chapter), Altera FLEX 10K/10KA/10KB/10KE/6K/8K.

### FLEX 10K Family

Default Speed Grade: 3  
Speed Grades supported: 3, 4  

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPF10K10</td>
</tr>
<tr>
<td>EPF10K20</td>
</tr>
<tr>
<td>EPF10K30</td>
</tr>
<tr>
<td>EPF10K40</td>
</tr>
<tr>
<td>EPF10K50</td>
</tr>
<tr>
<td>EPF10K70</td>
</tr>
<tr>
<td>EPF10K100</td>
</tr>
</tbody>
</table>

### FLEX 8K Family

Default Speed Grade: 3  
Speed Grades supported: 4, 3, 2  

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPF8282A</td>
</tr>
<tr>
<td>EPF8452A</td>
</tr>
<tr>
<td>EPF8636A</td>
</tr>
<tr>
<td>EPF8820A</td>
</tr>
<tr>
<td>EPF81188A</td>
</tr>
<tr>
<td>EPF81500A</td>
</tr>
</tbody>
</table>
### FLEX 6K Family

Default Speed Grade: 2  
Speed Grades supported: 2, 3  

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPF6010A</td>
</tr>
<tr>
<td>EPF6016</td>
</tr>
<tr>
<td>EPF6-24A</td>
</tr>
</tbody>
</table>

### FLEX 10KA Family

Default Speed Grade: 1  
Speed Grades supported: 4, 3, 2, 1  

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPF10K10A</td>
</tr>
<tr>
<td>EPF10K30A</td>
</tr>
<tr>
<td>EPF10K50V</td>
</tr>
<tr>
<td>EPF10K100A</td>
</tr>
<tr>
<td>EPF10K130V</td>
</tr>
<tr>
<td>EPF10K250A</td>
</tr>
</tbody>
</table>

### FLEX 10KB Family

Default Speed Grade: 1  
Speed Grades supported: 4, 3, 2, 1  

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPF10K100B</td>
</tr>
</tbody>
</table>
### MAX+PLUS II Clique Assignments (Preliminary) Information

This section contains general information on clique or cluster assignments.

**Note:** This preliminary information was extracted from MAX+PLUS II documentation.

#### Clique Ticket

```
BEGIN

"|ticket0":_ clique;
"|ticket1":_ clique;
"|ticket2":_ clique;
"|ticket3":_ clique;

END;
```

#### Clique Assignments Section

This is the clique assignment and configuration file. The Clique Assignments Section (MAX+PLUS II documentation) specifies which logic functions must remain together, regardless of their exact location. Cliques allow you to partition a project so that only a minimum number of signals travel between LABs, rows, and devices, and to ensure that no unnecessary delays exist on critical timing paths.
You can assign logic functions to different types of cliques to ensure that they are implemented in the same LAB, row, or chip. You can also assign a "best possible" clique that specifies that the clique should be implemented in the smallest possible category of LAB, row, or chip. For example: The Clique Assignments Section stores assignment information entered with the Clique command: Assign menu.

The Clique Assignments Section has the following format:

```
CLIQUE <clique name>
BEGIN
{ <statement> [<source>] ; } 
END ;
```

The `<statements>` in the Clique Assignments Section takes the following BNF format. The letter code corresponding to this format appears as follows:

**Code:** Format: B <name>:<keyword>=<setting>

All clique names must be unique. For example:

**Keyword:** Settings: Usage: Command/Dialog Box/On-Screen
Option:

```
CLIQUE: ON|OFF|NULL
```

**Note:** If you prefer clique or cluster distribution of your HDL coding blocks, instead of random distribution, then refer also to MAX+PLUS II documentation.

**Note:** During timing optimization you can type, for example, clique_size_10 for a clique block.

**Note:** Do the following to create cliques for FLEX technologies:
'set clique_size 10' (any number greater than 0) and then run optimize_timing. You may need to apply -force. If you write out the EDIF, the critical paths are placed together in CLIQUEs. Altera can use these cliques in P&R. In addition, there are clique specific flags: clique_longest_path and clique_in_edif (set to 10 and TRUE, respectively by default).
LeonardoSpectrum provides synthesis for Altera MAX 3000A, 5000; and 7000/A/AE/E/S; and 9000 families of high density PLDs and provide synthesis for these families into any of the other supported technologies. This chapter presents the following:

- Before Beginning
- Altera MAX Architecture
- Design Flow
- Synthesis and Optimization Features
- Optimization Style Points
- Reporting
- Design I/O
- Using MAX Designs as Input
- How to Get Best Results with MAX+PLUS II
- Altera MAX Family Supported Devices

**Before Beginning**

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.
**Altera MAX Architecture**

The basic Altera MAX structures are: macrocells, expanders, and I/O cells.

Macrocells, at the core of the Altera MAX, contain combinational logic and registers. Macrocells can also be referred to as Logic Cells (LCELLs). Combinational logic is limited by the number of product terms. Functions of up to four (MAX5000) or five (MAX7000/9000) product terms can be implemented on one macro cell. Efficient use of the XOR gate resident in the Macrocell can increase the number of product terms.

Functions of more product terms must use Expanders, or be broken into multiple levels of Macrocells. Each Macrocell also contains a register that can be programmed to be a flip-flop or latch with clear and preset inputs. In addition, Macrocells and Expanders are grouped in LABs, which are fanin limited. The Macrocells and Expanders impose a fanin limitation on the product term limitation.

I/O cells in Altera MAX can be configured as inputs, outputs, tri-statable outputs, or bi-directional pins.

**Design Flow**

Figure 10-1 shows a simplified design flow that illustrates the use of LeonardoSpectrum together with the Altera MAX+PLUS II tools to provide a complete design solution. Designs are entered using standard methods and are optimized to the target MAX device. The result is a hierarchical EDIF netlist file which is processed by MAX+PLUS II to produce final fitting.

When targeting the MAX 3000A/5000/7000/9000 devices, optimization in LeonardoSpectrum is used with `-target=max5`, `-target=max7` or `-target=max9`. Output is written using `write -format EDIF outputfile_name` or using a .edf extension for output file name. Output is EDIF file which is for MAX+PLUS II.

**Locking LCells**

By default LeonardoSpectrum locks LCELLS by using LCELL buffers. This usually gives better results after MAX+PLUS II place and route. Refer to Table 10-1.
Table 10-1. Locking LCells

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock LCells</td>
<td>on</td>
<td>default (false)</td>
<td>default (false)</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set dont_lock_lcells true</td>
<td>-dont_lock_lcells</td>
</tr>
</tbody>
</table>

![ALTERA Simulation Software](image_url)
Figure 10-1  Simplified Design Flow
### Library Mapping File (LMF)

The LMF is for taking EDIF into MAX+PLUS II.

| LMF File | An exemplar.lmf file is required to take the EDIF generated by Leonardo into MAX+PLUS II. The exemplar.lmf file is installed in $EXEMPLAR/data area. This file maps the LeonardoSpectrum cells to MAX+PLUS II primitives and functions. Set the MAX+PLUS II environment to select this: .lmf file.

   If you are using the MAX+PLUS II through the command line, edit your .acf file. Search for EDIF_INPUT_LMF1 and set to:
   EDIF_INPUT_LMF1 = exemplar.lmf
   Search for EDIF_INPUT_USE_LMF1 and set to:
   EDIF_INPUT_USE_LMF1 = ON

| Pin Location | Pin locations are specified in the input VHDL file using PIN_NUMBER or ARRAY_PIN_NUMBER attributes. The value of the attribute is chip_name@pin_no. LeonardoSpectrum writes out this on the top net connected to the pin. MAX+PLUS II ignores the pin assignment if the device assignment is AUTO.

   To assign a device through MAX+PLUS II, click on Menu Assign->Device. Use Rename Chip in the same window to rename chip to chip_name. You can specify pin location from the Constraint Editor with:
   PIN_NUMBER,(PIN_NUMBER chip104a;)

**Note:** If using MAX+PLUS II version 8.0 and higher, you can then select the EXEMPLAR entry from the EDIF reader settings pull down menu.
VHDL Examples for pin location assignment:

```
--library exemplar;
--use exemplar.exemplar_1164.all;
library synth;
use synth.exemplar.all;
entity test is
  port( 
    a,b: in bit;
    c: out bit 
  );
  attribute PIN_NUMBER of a : signal is chip1@4 
end test;
architecture exemplar of test is
begin 
  process (a,b)
  begin
    c<=(a and (not b)) or ((not a) and b);
  end process;
end exemplar;
```

```
(edif inp1
  (edifVersion 2 0 0)
  (edifLevel 0)
    (net A
      (joined
        (portRef A)
        (portRef IN (instanceRef II))
        (property CHIP_PIN_LC (string chip1@4)))
    (design inp1
      (cellRef inp1
        (libraryRef inp1))))
```
**Pin Assignments**

The following are the steps necessary to do pin assignments in MAX+PLUS II:

1. Click on menu Assign->Devices.
2. Click on Rename Chip and rename chip to chip1 in the display window.
3. Select a device from Devices.
4. Select OK. MAX+PLUS II assigns pin 4 to signal A when you compile.

**Synthesis and Optimization Features**

Leonardo Spectrum uses dedicated algorithms to optimize logic for the MAX devices given the product term and fanin limitations of the architecture. The optimization takes advantage of all MAX resources such asExpanders, XORs and enabled flip-flops. This enables significant reductions in area and delay for the MAX devices.

Currently, four different, independent optimization passes are used, that may yield different results:

- **Pass 3** – One level implementation. If the results fit, then this is the fastest.
- **Passes 1, 2, and 4** – Multi-level implementations. These are slower but may take less resources than pass 3.

Currently, the number of Macrocells as well as the number of Expanders is reported. The number of Macrocells and Expanders is only an estimate. The exact number can only be determined during fitting, because MAX+PLUS II trades off Macrocells for Expanders, to achieve maximum utilization of resources.

If too many Expanders are used, MAX+PLUS II converts some Expanders into Macrocells, and consequently may use too many Macrocells for the target device. Try to fit the other results as well, even if the first pass seems to be the smallest one.

**Optimization Passes**

If the Extended Optimization Effort is selected on the GUI, then four independent optimization passes are made. Each of Passes 1 to 4 may yield different results.
Optimization Style Points

Quality of Optimization

Certain types of logic are optimized better than others: arithmetic circuits such as adders and multipliers are not as optimal as the best hand design or predefined macro. These constructs are better synthesized using module generation if using VHDL or Verilog as the input format. General control logic and state machines are optimized as well as or better than the best hand designs.

Module generation includes support for MAX7000/9000 architectures. Module generation support for MAX7000 can also be used by designers targeting the MAX5000 architecture, although usually with less optimal results.

Sequential Optimization

LeonardoSpectrum eliminates registers and latches when the register or latch is always latching a constant value, and there is no set or reset used on the register. Leonardo also eliminates registers or latches that are identical in input to another register or latch; these registers are effectively merged together.

LeonardoSpectrum does not “move” registers or change the phase of the signal stored in the registers.

Reporting

Refer to the LeonardoSpectrum User’s Guide for reporting information.

Design I/O

Global Signals

Global signals like clocks and output enables for tri-state signals are specified by using the BUFFER_SIG command. The two global signal primitives that are supported by the MAX architecture are GLOBAL and SCLK.
This is a **constraint file syntax** example:

```
BUFFER_SIG SCLK clock1
BUFFER_SIG GLOBAL oe
```

Design rules for using these primitives (number of primitives, signals connected to primitives) is not violated by LeonardoSpectrum.

This is an **interactive command line shell syntax** example:

```
set_attribute -net <signal_name> -name BUFFER_SIG -value <buffer_name>
```

### Additional Options - Max Fanin and Max Cubes

There are two important options that control the optimization. You can adjust the options and reoptimize if a fit cannot be achieved for any of the passes. Refer to Table 10-2.

**Table 10-2. Max Fanin and Max PT**

<table>
<thead>
<tr>
<th>Technology Settings</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Fanin</td>
<td>set max_fanin &lt;integer&gt;</td>
<td>-max_fanin=&lt;integer&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max PT</td>
<td>set max_pt &lt;integer&gt;</td>
<td>-max_pt=&lt;integer&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This variable controls the maximum fanin into a function. If MAX+PLUS II issues the message:

```
Logic Array Block ... requires too many inputs
```

Use this parameter to reduce fanin. The default values for `max_fanin` are:

- `max5` technology: 80
- `max7` technology: 36
- `max9` technology: 49
This option controls the maximum number of product terms in a function. If MAX+PLUS II issues the message:

```
Logic Array Block ... requires too many expanders or Design
requires too many macrocells/expanders
```

Use this parameter to reduce the number of product terms. This results in more levels of logic. The defaults values for max_pt are:

- max5 technology: 30
- max7 technology: 20
- max9 technology: 20

Appropriate defaults are picked when max_fanin/max_pt is set to 0.

**Using MAX Designs as Input**

To retarget a MAX design into other technologies or to optimize a MAX design, the design must fit into a single MAX device. This ensures that MAX+PLUS II writes a single EDIF netlist that captures the whole design. The EDIF writer option must be enabled when running the MAX+PLUS II compiler. If the design does not fit into a single device, switch to a bigger device until the design fits.

Use the EDIF file generated by MAX+PLUS II as input to LeonardoSpectrum, with the MAX library as the source technology. If the design cannot fit into a single device, let MAX+PLUS II partition the design into several devices and generate several EDIF files. Then, manually write an HDL file that connects the devices correctly. The MAX+PLUS II report file (multiple pin connections section) can be helpful in writing this file.

**How to Get Best Results with MAX+PLUS II**

Refer to the Altera FLEX chapter in this guide.

**LeonardoSpectrum Compatible Altera MAX Vendor Tools**

MAX Series MAX+PLUS II 2.9.3 or higher
**Altera MAX Family Supported Devices**

Devices supported are: MAX 3000A, MAX5000, MAX9000, MAX7000/A/AE/E/S

### MAX 3000A

- Default Speed Grade: 10
- Speed Grades supported: 4, 5, 6, 7, 10

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM3032</td>
</tr>
<tr>
<td>EPM3064</td>
</tr>
<tr>
<td>EPM3128</td>
</tr>
<tr>
<td>EPM3256</td>
</tr>
</tbody>
</table>

### MAX 5000

- Default Speed Grade: 1
- Speed Grades supported: 1, 2, 15, 20, 25

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM5032</td>
</tr>
<tr>
<td>EPM5064</td>
</tr>
<tr>
<td>EPM5128</td>
</tr>
<tr>
<td>EPM5130</td>
</tr>
<tr>
<td>EPM5192</td>
</tr>
</tbody>
</table>
# MAX 9000

Default Speed Grade: 10

Speed Grades supported: 10, 15, 20

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th>EPM9320</th>
<th>LC84 LI84 RC208 RI208 GC280 BC356 ALC84 ALI84 ARC208 ARI208 ABC356</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM9400</td>
<td>LC84</td>
<td>RC208 RC240</td>
</tr>
<tr>
<td>EPM9480</td>
<td>RC208</td>
<td>RC240</td>
</tr>
<tr>
<td>EPM9560</td>
<td>RC208</td>
<td>RI208 RC240 RC304 RI304 GC280 BC356 ARC208 ARI208 ARC240 ARI240 ABC356</td>
</tr>
</tbody>
</table>
### MAX 7000

Default Speed Grade: 10

Speed Grades supported: 10, 4, 5, 6, 7, 10P, 10, 12P, 12, 15, 15T, 20

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EPM7032</strong></td>
</tr>
<tr>
<td><strong>EPM7064</strong></td>
</tr>
<tr>
<td><strong>EPM7096</strong></td>
</tr>
<tr>
<td><strong>EPM7128A</strong></td>
</tr>
<tr>
<td><strong>EPM7256A</strong></td>
</tr>
<tr>
<td><strong>EPM7032AE</strong></td>
</tr>
<tr>
<td><strong>EPM7064AE</strong></td>
</tr>
<tr>
<td><strong>EPM7128AE</strong></td>
</tr>
<tr>
<td><strong>EPM7256AE</strong></td>
</tr>
<tr>
<td><strong>EPM7384AE</strong></td>
</tr>
<tr>
<td><strong>EPM7512AE</strong></td>
</tr>
<tr>
<td><strong>EPM7032S</strong></td>
</tr>
<tr>
<td><strong>EPM7064S</strong></td>
</tr>
<tr>
<td><strong>EPM7128S</strong></td>
</tr>
<tr>
<td><strong>EPM7160S</strong></td>
</tr>
<tr>
<td><strong>EPM7192S</strong></td>
</tr>
<tr>
<td><strong>EPM7256S</strong></td>
</tr>
<tr>
<td><strong>EPM7128E</strong></td>
</tr>
<tr>
<td><strong>EPM7160E</strong></td>
</tr>
<tr>
<td><strong>EPM7192E</strong></td>
</tr>
<tr>
<td><strong>EPM7256E</strong></td>
</tr>
</tbody>
</table>

MAX 7000 continued.....
MAX 7000 continued....

### MAX 7000

- **Default Speed Grade:** 5
- **Speed Grades supported:** 5, 6, 7, 10, 15

#### Devices Supported

<table>
<thead>
<tr>
<th>Model</th>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM7032S</td>
<td>LC44 LI44 TC44 TI44</td>
</tr>
<tr>
<td>EPM7064S</td>
<td>LC44 LI44 TC44 TI44 LC84 LI84 TC100 TI100</td>
</tr>
<tr>
<td>EPM7128S</td>
<td>LC84 LI84 QC100 QI100 TI100 QC160 QI160</td>
</tr>
<tr>
<td>EPM7160S</td>
<td>LC84 LI84 TC100 TI100 QC160 QI160</td>
</tr>
<tr>
<td>EPM7192S</td>
<td>QC160 QI160</td>
</tr>
<tr>
<td>EPM7256S</td>
<td>RC208 RI208 QC208</td>
</tr>
</tbody>
</table>

### MAX 7000A

- **Default Speed Grade:** 6
- **Speed Grades supported:** 6, 7, 10, 12

#### Devices Supported

<table>
<thead>
<tr>
<th>Model</th>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM7128A</td>
<td>LC84 LI84 TC100 TI100 FC100 TC144 FC256</td>
</tr>
<tr>
<td>EPM7256A</td>
<td>TC100 TI100 TC144 TI144 QC208 QI208 FC256 FI256</td>
</tr>
</tbody>
</table>
## MAX 7000AE

Default Speed Grade: 4

Speed Grades supported: 4, 5, 6, 7, 10

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM7032AE</td>
</tr>
<tr>
<td>EPM7064AE</td>
</tr>
<tr>
<td>EPM7128AE</td>
</tr>
<tr>
<td>EPM7384AE</td>
</tr>
<tr>
<td>EPM7512AE</td>
</tr>
</tbody>
</table>

## MAX 7000E

Default Speed Grade: 7

Speed Grades supported: 7, 10, 10P, 12, 12P, 15, 20

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM7128E</td>
</tr>
<tr>
<td>EPM7160E</td>
</tr>
<tr>
<td>EPM7192E</td>
</tr>
<tr>
<td>EPM7256E</td>
</tr>
</tbody>
</table>
### MAX 7000S

Default Speed Grade: 5

Speed Grades supported: 5, 6, 7, 10

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPM7032S</td>
</tr>
<tr>
<td>EPM7064S</td>
</tr>
<tr>
<td>EPM7128S</td>
</tr>
<tr>
<td>EPM7160S</td>
</tr>
<tr>
<td>EPM7192S</td>
</tr>
<tr>
<td>EPM7256S</td>
</tr>
</tbody>
</table>
This chapter provides guidelines for the synthesis of Lucent ORCA field programmable gate arrays (FPGAs). This chapter is divided as follows:

- Before Beginning
- FPGA Architecture
- Synthesis and Optimization Features
- Using ORCA Architectural Features
- Reporting
- Generation of ORCA Foundry Properties
- ORCA DIN/DOUT Attributes
- ORCA Preference File Writer
- Devices Supported for Lucent ORCA FPGA Families
- ORCA Field-Programmable System Chips (FPSC)
- Additional Features for ORCA

Before Beginning

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.
FPGA Architecture

ORCA FPGA consists of the following two basic elements:

- Programmable logic cells (PLCs)
- Programmable input/output cells (PICs)

The array of PLCs is surrounded by PICs. Each PLC contains a programmable function unit (PFU). All combinational logic is performed in Lookup Tables (LUTs) located in the PFU. The PFU can be used in different modes to meet different logic requirements. A PFU can be configured to implement from one to six input combinatorial logic functions:

- Four functions of 4-input variables, some inputs shared (QLUTs)
- Two functions of 5-input variables (HLUTs)
- One function of 6-input variables

The lookup tables in a PFU can also implement nibble-wide ripple functions with high-speed carry logic. Finally, you can also configure the lookup tables as a read/write or read-only memory. The programmable input/output cells (PICs) are located along the perimeter of the device. Each I/O can be configured to be either an input, an output, or a bidirectional I/O. LeonardoSpectrum supports the following families of ORCA devices:

- ORCA-2CA (ORCA-2a)
- ORCA-2TA (ORCA-2a)
- ORCA-3C (ORCA-3c)
- ORCA-3T (ORCA-3t)

When targeting ORCA 2CA/2TA, LeonardoSpectrum selects the technology Lucent ORCA-2A from the Load Library menu. The ORCA3C modgen library is used by ORCA3. ORCA3C modgen library is similar to ORCA2C except for the following differences:

1. Only synchronous RAMs are supported. Asynchronous RAMS are not available in ORCA3C modgen library. Lucent recommends using SLIC (supplemental logic and interconnect cell), to build RAMs greater than 32, instead of using a decoder and tbufs. SCUBA version 9.2a does not support RAM generation for ORCA3C.

2. ORCA3C modgen library does not use PFU gates.

The flow and interface between LeonardoSpectrum and the ORCA Foundry is shown in Figure 11-1. Figure 11-2 is the back annotation flow.
Figure 11-1 Simplified Design Flow
Figure 11-2  Simplified Back Annotation
Synthesis and Optimization Features

Optimization Algorithms

The ORCA-specific synthesis and optimization algorithms were developed to take advantage of the ORCA architecture. A fan-in limited algorithm takes advantage of the lookup table logic of the ORCA FPGAs. In addition, a LUT mapping algorithm maps combinational logic to lookup tables, and a sequential mapping algorithm maps sequential logic to registers in PFUs.

Fanin Limited Optimization

A LUT is any function of 4, 5, and 6 input combinational logic. A 4-input XOR uses the same area and is as fast as a 4-input AND gate. The function in Equation E-1 can be solved in two ways. Here are Solutions (1) and (2) for Equation E-1:

\[
X = (A \cdot (B+C)) + (B \cdot D) + (E \cdot F \cdot G \cdot H \cdot I) \tag{E-1}
\]

Solution (1)
You can decompose a function into its simpler AND/OR equivalent representation, and then split the gates with large fan-in into multiple gates. Represented in AND and OR gates, X is decomposed as:

\[
\begin{align*}
X &= T_1 + T_2 + T_3 \\
T_1 &= A \cdot T_4 \\
T_2 &= B \cdot D \\
T_3 &= E \cdot F \cdot G \cdot H \cdot I \\
T_4 &= B + C
\end{align*} \tag{E-2}
\]

Because \( T_3 \) has more than four inputs, further decomposition is required:

\[
\begin{align*}
T_3 &= E \cdot F \cdot G \cdot T_5 \\
T_5 &= H \cdot I \tag{E-3}
\end{align*}
\]
After fully decomposing the design, you can use the ORCA place-and-route software to place the design into PFUs. In this example, T1, T3, and T5 are packed in the same PFU. T1 and T5 can occupy an HLUT while T3 can occupy the other HLUT. The function X can be placed in another PFU so the design takes two PFUs.

| LE_1: X = T1+(B*D)+T3 |
| LE_2: T1 = A*(B+C)  |
| LE_3: T3 = E*F*G*T5  |
| LE_4: T5 = H*I       |

Note – The critical path is LUT_4 → LUT_3 → LUT_1, resulting in three levels of LUTs for the delay.

**Solution (2)**

Another decomposition of Equation (E-1) yields partitioning into three LUTs:

| X = T1+(T2*E) |
| T1 = A*(B+C)+(B*D) |
| T2 = F*G*H*I     |

Since each of the three equations have no more than four inputs, each equation can be placed into an LUT. When implemented, this design has only two LUTs in the critical path which results in a faster and smaller design.

**Lookup Table (LUT) Mapping**

LUT mapping maps fanin limited logic functions to lookup tables in the ORCA FPGAs. This minimizes the total number of lookup tables when optimizing in area mode and the delay along critical paths when optimizing in delay mode.

LUT mapping maps logic to 4, 5, 6 input lookup tables and various PFU gates (2, 3 inputs PFU-NAND, PFU-XOR, and PFU-MUX). LeonardoSpectrum writes NOMERGE properties in the EDIF output to mark the boundaries of lookup tables.
Consider the following description of a 6 to 1 multiplexer:

\[
\begin{align*}
T_1 &= A \cdot C_2' \cdot C_1' \cdot C_0'; \\
T_2 &= B \cdot C_2' \cdot C_1' \cdot C_0; \\
T_3 &= C \cdot C_2' \cdot C_1 \cdot C_0'; \\
T_4 &= D \cdot C_2 \cdot C_1' \cdot C_0'; \\
T_5 &= E \cdot C_2 \cdot C_1' \cdot C_0; \\
T_6 &= F \cdot C_2 \cdot C_1 \cdot C_0'; \\
\text{OUT} &= T_1 + T_2 + T_3 + T_4 + T_5 + T_6;
\end{align*}
\]

**User Options for Optimization and Mapping**

Use the Max Fanout field on the GUI to override the default max fanout load specified in the library. Refer to Table 11-1. However, a synthesized netlist with high fanout nets may be a problem for the place and route tool. The place and route tool usually splits the net arbitrarily. High fanout nets can cause significant delays on wires and become unroutable. On a critical path, high fanout nets can cause significant delays in a single net segment and cause the timing constraints to fail.
To eliminate the need for splitting of the net by the place and route tool, the synthesis tool must maintain a reasonable number of fanouts for a net. LeonardoSpectrum tries to maintain reasonable fanout limits for each for each target technology. Default fanout limits are derived from the synthesis library.

**Note:** The LUT buffering and replication is supported for the Lucent ORCA 2CA/2TA, ORCA 3C/3T, and 3000 technologies.

LeonardoSpectrum maintains reasonable fanouts by replicating the driver which results in net splitting. If replication is not possible, then the signal is buffered. The buffering of high fanout primary input signals is an example. Buffering the signal causes the wire to be slower by adding intrinsic delays.

**Max Fanout Attribute**

On specific nets, you can set an attribute to control the `max_fanout` value:

```
set_attribute -net <net_name> -name lut_max_fanout -value <int>
```

**Note:** Setting this attribute takes precedence over any global fanout specifications.
## Table 11-1. Mapping and Optimization Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Fanout: window</td>
<td>n/a</td>
<td>set lut_max_fanout &lt;int&gt;</td>
<td>n/a</td>
</tr>
<tr>
<td>Map to 6-input LUTs</td>
<td>on</td>
<td>set use_f6_lut true -use_f6lut</td>
<td></td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>default</td>
<td>default</td>
</tr>
</tbody>
</table>

Controls mapping to 6-Input LUTs. Note: Use the `-k` option to the ORCA Foundry Mapper.

- **Write ORCALUT Symbol**
  - on: default
  - off: set write_lut_binding false -nworite_lut_binding

Controls printing LUT binding.

- **Add I/O Pads**
  - on: optimize -chip (default)
  - off: optimize -macro

Infers I/O pads for chip level designs. Refer to Optimize tab information in the User’s guide.
Constraint-Driven Timing Optimization

LeonardoSpectrum optimizes the circuit to meet timing constraints. After optimization and mapping are done, timing violations are determined in the circuit, and an attempt is made to improve timing on critical paths. You can set timing constraints from the constraint file editor or set a global max frequency constraint that will apply to all clocks in the design. If no constraints are specified, LeonardoSpectrum attempts to improve timing on the longest path in the circuit. The following is an example from a log file that demonstrates the effects of timing optimization:

Start timing optimization for design .work.ace_bus_ctrl.behavior

Initial Timing Optimization Statistics:
Most Critical Slack -29.1
Sum of Negative Slacks -1337.7
Longest Path 35.6 ns
Area 99.0

Final Timing Optimization Statistics:
Most Critical Slack -22.3
Sum of Negative Slacks -1227.4
Longest Path 30.1 ns
Area 112.0
Total time taken 32 cps seconds

The optimize_timing command can be used after optimize to improve the timing performance of the design. optimize_timing command works most effectively when timing constraints are specified. The -force option can be used to force timing constraints, to try to improve the longest path.

optimize_timing -force
The `optimize_timing` command restructures combinational logic to reduce the levels of logic between the critical signal and the output. This command also tries to map the restructured portion of the logic effectively to the ORCA architecture. Using the PFU gate, the `optimize_timing` command tries to map a critical signal directly to the input of the PFU-MUX gate and bypasses the LUTs in the PFU.

Also the specific end point or instance to improve can be specified to the `optimize_timing` command with the `-through` option.

```
optimize_timing -through
```

You can now set global timing constraints by setting clock frequency. You can set the maximum global frequency constraint by using either the constraint editor or by entering the `clock_frequency <value>` on the interactive command line shell. The clock frequency applies to all clocks in the design, while the clock cycle is set on a particular clock. The frequency should be an integer greater than 0 and entered in Mhz.

### Support for Maximum Frequency

You can set a global timing constraint for maximum clock frequency in a design. LeonardoSpectrum will set this constraint on all global clocks in the design. Constraint driven timing optimization will try to meet the global clock frequency constraint. Refer to the Constraints chapter in the LeonardoSpectrum User’s guide.

### Data Path Synthesis and Modgen Implementation

LeonardoSpectrum supports various technology-specific implementations of arithmetic and relational operators used in VHDL or Verilog RTL. Since these implementations have been designed optimally for the ORCA technology, the synthesis results are in general smaller and/or faster and take less time to compile. The following operators are supported for ORCA technology for families 2CA and 2TA:

- **Relational Operators**
  - `=`, `/=`, `<`, `<=`, `>`, `>=`

- **Arithmetic Operators**
  - `+`, `-`, `*`

- **Miscellaneous Functions**: Incrementer, decremter, absolute value, unary minus, counters, RAMS (synchronous/asynchronous for 2A only), multiplexers
Using ORCA Architectural Features

The features are:
- Enabled D Type Flip-Flops
- GSR Usage and Startup Blocks
- Mapping to Synchronous Set/Reset Registers

**Enabled D Type Flip-Flops**

The Extract Clock Enables is selected by default on Optimize Options. Refer to Table 11-2.

Table 11-2. Operator Options

<table>
<thead>
<tr>
<th>Optimize Options</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract Clock Enables</td>
<td>on</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set use_dffenble false</td>
<td>-noenable_dff_map</td>
</tr>
</tbody>
</table>

Controls mapping to enabled D-type flip-flops. Clock enable detection from HDLs occurs by default.

**Notes:** Mapping to enabled D-type flip-flops can be controlled with the `use_dffenable` variable. This variable allows LeonardoSpectrum to infer clock enable from HDL description. The default is TRUE.

When reading HDL designs with `use_dffenable` variable set to TRUE, LeonardoSpectrum infers clock enable logic while parsing HDL code. Detecting the clock enable early in the synthesis flow achieves better results.
**GSR Usage and Startup Blocks**

ORCA flip-flops feature set/reset inputs that can be driven from a dedicated Global Set/Reset (GSR) signal. The GSR is implicit set/reset for all registers, and does not require any routing resources. If the design is suitable for automatic inference of global set/reset, then LeonardoSpectrum would automatically infer a GSR signal and insert a Startup block in the design.

The HDL description should be written with one asynchronous signal that initializes every DFF to the value at POWERUP. DFFs that power up as 0 should use the global signal as an asynchronous reset; DFFs that power up as 1 should use the global signal as an asynchronous set. You can use both set and reset in the same design. Refer to Table 11-3.

**Note:** Setting the `global_sr <my_signal>` Tcl variable allows LeonardoSpectrum to use the specified Active High Signal as Global Set/Reset. The signal is disconnected from the registers Set/Reset pin and connected to a startup block. Specifying the Global Set/Reset on an active low signal will create incorrect logic.

**Note:** LeonardoSpectrum supports GSR hierarchical and flat designs. By default, LeonardoSpectrum infers global set/reset from your design. GSR support is enhanced to handle GSR processing across hierarchy. Auto and manual set/reset are detected on most flat or hierarchical designs.
Table 11-3. Global Set/Reset

<table>
<thead>
<tr>
<th>Technology Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign GSR:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detects the global set/reset signal.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>□ Auto</td>
<td>default true</td>
<td></td>
</tr>
<tr>
<td>□ Manual</td>
<td>set infer_gsr</td>
<td></td>
</tr>
</tbody>
</table>

When □ Auto is selected, □ Manual is not available and the Signal field is grayed out. When □ Manual is selected, you can type in the Signal field.

Assign Global SR: my_reset

Defines an active high signal name as global set/reset.

set global_sr <my_reset>

default not set
In the following VHDL example GSR is used for an active high reset.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity gsr_example is
  port ( reset, clk : in std_logic;
         d_in : in std_logic_vector (7 downto 0);
         q_out : out std_logic_vector (7 downto 0) );
end gsr_example;
architecture active_hi of gsr_example is
begin
  -- Use reset to initialize flip-flops: The q_out byte is initialized
  -- by GSR to be 1’s in the upper nibble, and 0’s in the lower nibble.
  process (clk, reset)
  begin
    if (reset = '1') then
      q_out(7 downto 4) <= "1111"; -- asynchronous set
      q_out(3 downto 0) <= "0000"; -- asynchronous reset
    elsif clk'EVENT and clk='1' then
      q_out <= d_in;
    end if;
  end process p0;
end active_hi;
```
The reset logic for this design can be implemented by using either the `global_sr` or the `infer_gsr` variable. In this VHDL example GSR is used for an active low reset.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity gsr_example is
  port ( reset, clk : in std_logic;
         d_in   : in std_logic_vector (7 downto 0);
         q_out  : out std_logic_vector (7 downto 0) );
end gsr_example;
architecture active_lo of gsr_example is
begin
  -- Use reset to initialize flip-flops
  p0: process (clk, reset)
  begin
    if (reset = '0') then
      q_out(3 downto 0) <= "0000"; -- asynchronous reset
      q_out(7 downto 4) <= "1111"; -- asynchronous set
    elsif clk'EVENT and clk='1' then
      q_out <= d_in;
    end if;
  end process p0;
end active_lo;
```
In this Verilog example GSR is used for an active high reset

```verilog
module gsr_example ( reset, clk, d_in, q_out );
    input reset, clk;
    input [7:0] d_in;
    output [7:0] q_out;
    reg [7:0] q_out;

    // Use reset to initialize flip-flops
    always@ (posedge clk or posedge reset)
        if (reset) begin
            q_out[3:0] = 4’b0000; // asynchronous reset
            q_out[7:4] = 4’b1111; // asynchronous set
        end
    else
        q_out = d_in;
endmodule
```

In this Verilog example GSR is used for an active low reset

```verilog
module gsr_example ( reset, clk, d_in, q_out );
    input reset, clk;
    input [7:0] d_in;
    output [7:0] q_out;
    reg [7:0] q_out;

    // Use reset to initialize flip-flops
    always@ (posedge clk or negedge reset)
        if (!reset) begin
            q_out[3:0] = 4’b0000; // asynchronous reset
            q_out[7:4] = 4’b1111; // asynchronous set
        end
    else
        q_out = d_in;
endmodule
```
Using RAMs

Types of Inferencing RAMs

LeonardoSpectrum supports two types of RAMs:

- **RAM_DQ.** RAM_DQ is a single-port RAM with separate input and output data lines.

- **RAM_IO.** RAM_IO is a single-port RAM with bidirectional data lines.

Both of these RAM types support synchronous or asynchronous read and write. These RAMs are automatically inferred by LeonardoSpectrum from HDL code (VHDL or Verilog). The inferencing process distinguishes between RAMs that perform the read operation with an address clocked or not clocked by the write clock (read address clocked). Both of the following VHDL examples perform synchronous writes (inclock) and synchronous reads (outclock); LeonardoSpectrum recognizes these VHDL processes as RAMs:

- The first, **entity ram_example1**, is when the read operation does not have a clocked address.

- The second **entity ram_example2**, is when the read operation does have a clocked address.
library ieee, exemplar;
use ieee.std_logic_1164.all;
use exemplar.exemplar_1164.all;
entity ram_example1 is
  port (data: in std_logic_vector(7 downto 0);
        address: in std_logic_vector(5 downto 0);
        we, inclock, outclock: in std_logic;
        q: out std_logic_vector(7 downto 0));
end ram_example1;

architecture ex1 of ram_example1 is
  type mem_type is array (63 downto 0) of std_logic_vector (7 downto 0);
  signal mem: mem_type;
begin
  l0: process (inclock, outclock, we, address) begin
    if (inclock = '1' and inclock'event) then
      if (we = '1') then
        mem(evec2int(address)) <= data;
      end if;
    end if;
  end process;
end ex1;

entity ram_example1, is when the read operation does not have a clocked address.
library ieee, exemplar;
use ieee.std_logic_1164.all;
use exemplar.exemplar_1164.all;
entity ram_example2 is
  port (data: in std_logic_vector(7 downto 0);
    address: in std_logic_vector(5 downto 0);
    we, inclock, outclock: in std_logic;
    q: out std_logic_vector(7 downto 0));
end ram_example2;
architecture ex2 of ram_example2 is
  type mem_type is array (63 downto 0) of
    std_logic_vector (7 downto 0);
  signal mem: mem_type;
  signal address_int: std_logic_vector(5 downto 0);
begin
  l0: process (inclock, outclock, we, address)
  begin
    if (inclock = '1' and inclock'event) then
      if (outclock = '1' and outclock'event) then
        mem(evec2int(address)) <= data;
      end if;
    end if;
    if (we = '1') then
      mem(evec2int(address)) <= data;
    end if;
    q <= mem(evec2int(address_int));
  end process;
end ex2;

entity ram_example2, is when the read operation does have a clocked address.

**ORCA 2A Modgen Support for RAMs**

The ORCA2A Modgen Library supports both asynchronous RAMs and synchronous RAMs that do not clock the read address with the write clock.
Examples:

-- Asynchronous Single Port Ram With Bidirectional Data
-- Inference: Leonardo infers RPE16x4 component with tristates

library ieee;
use ieee.std_logic_1164.all;
package mem_pkg is
  type MEM_WORD is array (15 downto 0) of std_logic_vector (3 downto 0);
end mem_pkg;
use work.mem_pkg.all;

library ieee;
use ieee.std_logic_1164.all;
entity mem_tri is
  port (dio : inout std_logic_vector (3 downto 0);
        we : in std_logic;
        addr : integer range (15 downto 0);
end mem_tri;
architecture rtl of mem_tri is
  signal mem : MEM_WORD;
  signal d_int : std_logic_vector (3 downto 0);
begin
  process (we, addr, dio)
  begin
    if (we = '1') then
      mem(addr) <= dio;
    end if;
  end process;
  d_int <= mem (addr);
  dio <= d_int when (we = '0') else "ZZZZ";
end rtl;
-- Asynchronous Single Port Ram
-- Inference: Leonardo infers a RPE16x4 component only

library ieee;
use ieee.std_logic_1164.all;
use work.mem_pkg.all;

entity mem is
  port (din : in std_logic_vector (3 downto 0);
        dout : out std_logic_vector (3 downto 0);
        we : in std_logic;
        addr : integer range (15 downto 0);
"
end entity;

architecture rtl of mem is
begin
  signal mem : MEM_WORD;
  process (we, addr, din)
  begin
    if (we = '1') then
      mem(addr) <= din;
    end if;
  end process;
  dout <= mem(addr);
end rtl;
library ieee;
use ieee.std_logic_1164.all;
package mem_pkg is
  type MEM_WORD is array (15 downto 0) of std_logic_vector (3 downto 0);
end mem_pkg;
use work.mem_pkg.all;

library ieee;
use ieee.std_logic_1164.all;
entity mem_tri is
  port (dio : inout std_logic_vector (3 downto 0);
        we, clk : in std_logic;
        addr : integer range (15 downto 0);
  end mem_tri;
architecture rtl of mem_tri is
  signal mem : MEM_WORD;
  signal d_int : std_logic_vector (3 downto 0);
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        mem(addr) <= dio;
      end if;
      end if;
  end process;
  d_int <= mem(addr);
  dio <= d_int when (we = '0') else "ZZZZ";
end rtl;
When running this example targeting ORCA 2CA or 2TA FPGAs, LeonardoSpectrum will infer a synchronous `ram_io` cell.

### RAM Instantiation

You can use scuba (ORCA Foundry tool) to generate RAMs. This tool can be used in the input HDL design.

When using scuba, you should define the memory in the design as a black box with appropriate ports, and run Leonardo to synthesize the output EDIF netlist. The black box will be instantiated as such in the EDIF netlist.
Then, run scuba to create the EDIF description of the memory. For example, the following memgen invocation creates a 256x6 RAM and writes the RAM to the file myram.edf:

```
scuba -addr_width 8 -data_width 6 -lang vhdl -n myram -type aspram
```

The following two example designs, one in VHDL and one in Verilog, treat the component “myram” as a black box. When these files are run through LeonardoSpectrum, the resulting EDIF file contains a symbol myram.

To set the `initval` attribute on the RAM instance in Verilog, use the `set_attribute` command in LeonardoSpectrum `interactive command line shell`:

```
set_attribute -instance I1 -name initval -type string -value "0x0000"
```
Example 8a. VHDL RAM Usage Example

```vhdl
library exemplar;
use exemplar.exemplar_1164.all;
library ieee;
use ieee.std_logic_1164.all;
entity ram_example is
  port(
    addr : in  bit_vector (7 downto 0);
    din  : in  bit_vector (5 downto 0);
    we   : in  bit;
    o    : out bit_vector (5 downto 0);
  );
end ram_example;

architecture exemplar of ram_example is
  component myram
    port ( a: in bit_vector (7 downto 0);
           d: in bit_vector (5 downto 0);
           we: in bit;
           o: out bit_vector (5 downto 0));
  end component;
begin
  g1: myram port map (a=>addr, we=>we, d=>din, o=>o);
end exemplar;
```
You can also instantiate synchronous single and dual-port RAMs in the input VHDL or Verilog description.

**Using ROMs**

The Lucent ORCA macro library supports ROMs as primitives. Two sizes are supported: RPP16x2 and RPP16x4 as described in the ORCA Foundry Libraries Guide. These can be instantiated for use in a Leonardo design. You can also use scuba (ORCA Foundry tool) to generate ROMs of other sizes and instantiate the ROMs in the input HDL design.

An INITVAL attribute is required for the ROM component to define its memory pattern. The value must be a hexadecimal number of the proper size which depends on the size of the ROM.

An INITVAL attribute is required for the ROM component to define its memory pattern. The value must be a hexadecimal number of the proper size which depends on the size of the ROM.

While using ORCA ROM macro (or any example which directly instantiates a component from the ORCA library), be sure to load the ORCA synthesis library before reading in the RTL design.
Mapping to Synchronous Set/Reset Registers

ORCA FPGAs have a few complex registers that have some combinational logic at the FPGA data inputs. This includes the synchronous set/reset registers and registers with multiplexed data inputs.

By default, LeonardoSpectrum only maps to simple registers and registers with multiplexed data inputs; and does not map to synchronous set/reset registers. Where:

Refer to Table 11-4.

Table 11-4. Synchronous Registers

<table>
<thead>
<tr>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>set include_gates &lt;gate_name&gt;</td>
<td>-include=&lt;gate_name&gt;</td>
</tr>
</tbody>
</table>

By default, does not include gates that are predefined in the target technology. Load the variable before loading the synthesis library.

I/O Mapping

LeonardoSpectrum automatically inserts the I/O buffers in the top-level view and uses the following I/O buffers for ORCA architecture:

- IBM: CMOS input buffer
- OB6: 6ma sink 3ma source output buffer
- OBZ6: 6ma sink 3ma source output buffer with a tristate output
- BMZ6: CMOS input, 6ma/3ma output with a tristate bidirectional buffer

OB6, OBZ6, and BMZ6 can have inverted inputs and are mapped automatically by LeonardoSpectrum where appropriate.

All other I/O buffers, as described in the ORCA Library section of the ORCA Foundry Manual, can be inserted from the interactive command line shell using the PAD command or instantiated by component instantiation in VHDL or Verilog HDL.
Reporting

Refer to the LeonardoSpectrum User’s Guide for examples and information on reporting.

Generation of ORCA Foundry Properties

These properties include:

- Pin Number
- Slew

Pin Number Property - LOC

You can specify IOB locations for external pins of a design. If using VHDL input, you can use pin_number for individual signals and array_pin_number for buses. These attributes are translated into LOC property on the IOB instances in the output EDIF file for further processing by ORCA Foundry place and route tools.

The VHDL attributes pin_number and array_pin_number are declared in the EXEMPLAR and EXEMPLAR_1164 packages. You gain access to these packages by making a package visible with these context clauses before the entity declarations.

```vhdl
library exemplar; use EXEMPLAR.EXEMPLAR_1164.all;
```

If you are not using the EXEMPLAR or EXEMPLAR_1164 package, the attributes pin_number, array_pin_number, and the type string_array must be defined.

For Verilog designs or any other netlist formats, you can use predefined LeonardoSpectrum procedures, or use the set_attribute command to set pin_number constraints on design ports.
Slew Property

Slew property is an ORCA Foundry mapper property that should be placed on IOB instances in the output EDIF. If the slew property is specified on output port, LeonardoSpectrum writes the property on IOB instance. This property specifies the relative speed (slew rate) of the output driver. Value is rated from 1 (slowest speed) to 100 (fastest speed). For example:

```
set_attribute -port a1 -name slew -type string -value "100"
```

ORCA DIN/DOUT Attributes for ORCA 2CA/2TA

The performance of an ORCA design can be improved by passing the Direct In (DIN) and Direct Out (DOUT) attributes to the ORCA Foundry. This is to specify that a register driving or being driven by an I/O be placed next to that I/O during place-and-route.

The DIN and DOUT attributes can be placed on top level ports or on instantiated I/O cells. DIN and DOUT are string attributes that take no arguments.

A LOC property must be attached to the pad for DIN/DOUT to work. If the property is attached to a component that does not feed a register (or is not a register output), an error is generated. An error is also generated if the property is attached to a multi-fanout (fanin) component.
Example:

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

entity demo is port
  in1 : in std_logic;
  clk : in std_logic;
  out1 : inout std_logic);

attribute din : string;
attribute dout: string;
attribute loc : string;

attribute din of in1 : signal is "";
attribute loc of in1 : signal is "19";

attribute dout of out1 : signal is "";
attribute loc of out1 : signal is "20";
end demo;

architecture Exemplar of demo is begin
  p1: process (clk)
    variable int1 :std_logic;
  begin
    if rising_edge(clk) then
      out1 <= int1;
      int1 := in1;
    end if;
  end process p1;
end Exemplar;
```
Part Number - PICSPEC

PICSPEC specifies the ORCA (2CA, 2TA) part number for the device into which the design is programmed. Specify the part number. LeonardoSpectrum writes out the specified Part number as PICSPEC property in output EDIF in the properties section of top-level view. Refer to Table 11-5:

Table 11-5. Part Number

<table>
<thead>
<tr>
<th>Technology Settings</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>This part number is written out as PICSPEC property in the output EDIF.</td>
<td><code>set part &lt;part_number&gt;</code></td>
<td><code>-part=&lt;part_number&gt;</code></td>
</tr>
</tbody>
</table>

Use default part or select part number from pulldown.
Speed Grade - PICSPEED

PICSPEED specifies the speed of the target ORCA device into which the device is programmed. Specify the Speed Grade in LeonardoSpectrum by setting the process variable:

```
set process 2
```

ORCA Foundry Properties on IOBs

In addition to LOC and SLEW, ORCA Foundry supports the following properties on IOB instances:

- DELAYMODE (input buffer, specifies whether the input delay mode is enabled/disabled)
- DRIVE (output buffer, specifies output drive current in milliamps)
- IBUFLEVEL (input buffer, specifies the input voltage level)
- LOAD (output buffer, load values used by timing analysis)
- OBUFLEVEL (output buffer, specifies the output voltage level)

Specify these ORCA Foundry properties on ports of a design specified in VHDL, or from the interactive command line shell by using the `set_attribute` command.

Other Foundry Properties

Other ORCA Foundry properties that can be set on nets or internal logic for the ORCA Foundry tool are written out in the EDIF file if the net or internal logic can be identified clearly. For example, the attributes set on instantiated component are written out in output EDIF.

NODIN/NODOUT Attributes for ORCA 3C/3T

LeonardoSpectrum now maps flips flops to input or output flip flops whenever possible. This helps reduce clock to output time and also frees up the internal resources. LeonardoSpectrum maps to I/O registers if you did not instantiate I/O buffers in your design.

User Options

Use the interactive command line shell command to set the variable:
set complex_ios FALSE

Default is TRUE. This variable is always set to TRUE. If you change the setting to FALSE, then mapping to I/O registers is not done by LeonardoSpectrum.

Use the batch mode:

-nocomplex_ios

Selectively set mapping to input/output I/O registers:

```
set_attribute -port <port_name> -name <nodin|nodout>
```

**Note:** These attributes do not require a value.

To turn off mapping to I/O flip flop, set attribute nodin to the connected input port; and on the output port, set then attribute nodout.

**ORCA Preference File Writer**

ORCA logical preference file writer is available. This feature automatically translates timing constraints into an ORCA logical preference file. You can specify:

- arrival_time
- required_time
- clock_cycle
- clock_offset
- pulse_width
- multicycle_path

These preferences are converted to ORCA logical preferences. These preferences are described in ORCA Foundry 9.2 Properties and Attributes, Lucent Technologies.

**Translation from logical to physical preference file:**

**Note:** You must use Foundry version 9.2 or higher.

- Run LeonardoSpectrum targeting ORCA. LeonardoSpectrum automatically generates an .lprf file (logical preference file).
- Generate the .ngo from EDIF using edif2ngd.
- Build the .ngd file from the .ngo file using ngdbuild.
- Generate .mrp file using MAP.
• Now use lp2prf to translate the logical preference file into the physical preference file.

**Usage:**

```
lp2prf -e <edif_file> -m <mrp_file> -l <logical> -p <physical>
```

**Example Constraint file:**

```
CLOCK_OFFSET 10 clock
ARRIVAL_TIME 5 sensor1
CLOCK_CYCLE 30 clock
PULSE_WIDTH 12 clock
REQUIRED_TIME 22 red1
```

**Resulting logical preference file:**

```
PERIOD PIN ix418/O 30.000000 ns HIGH 12.000000 ns;
INPUT_SETUP sensor1 5.000000 ns CLKNET=clock_int;
CLOCK_TO_OUT red1 12.000000 ns CLKNET=clock_int;
```

**ORCA3C and ORCA3T Multicycle Support**

Multicycle path support is added for ORCA. You can set the Multicycle path in the constraint file and LeonardoSpectrum then passes the path to the ORCA preference file. LeonardoSpectrum timing optimization and delay reporting are sensitive to multicycle paths.

**Usage:**

```
set_multicycle_path -setup -rise -from u2.reg_rxdatardy -to u2.reg_framingerr -value 2
```

```
CLOCK_OFFSET 13 clkx16
CLOCK_CYCLE 30 clkx16
PULSE_WIDTH 23 clkx16
REQUIRED_TIME 31 rxrdy
```

**Logical Preference File**

```
PERIOD PIN ix635/O 30.000000 ns HIGH 23.000000 ns;
CLOCK_TO_OUT rxrdy 18.000000 ns CLKNET=clkx16_int;
MULTICYCLE FROM CELL u2/reg_rxdatardy TO CELL u2/reg_framingerr 2 X;
```
Additional Example for Logical Preference File

This example is for preferences corresponding to timing constraints that are specified in VHDL attributes in demo.vhd. This is in addition to preferences on primary I/Os, and a PERIOD preference on an internal, net.

# File: demo.lprf
PERIOD PIN ix35/O 15.000000 ns HIGH 9.000000 ns;
INPUT_SETUP en 3.000000 ns CLKNET=clk_int;
CLOCK_TO_OUT clk_div_8 16.000000 ns CLKNET=clk_int;
# End file: demo.prf:

# File: demo1.lprf
PERIOD PIN ix35/O 15.000000 ns HIGH 9.000000 ns;
INPUT_SETUP en 3.000000 ns CLKNET=clk_int;
CLOCK_TO_OUT clk_div_8 16.000000 ns CLKNET=clk_int;
PERIOD PIN freq_div_4_inst/clkdiv4 60.000000 ns HIGH 30.000000 ns;
# End file: demo1.prf:
Example 1

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library exemplar;
use exemplar.exemplar_1164.all;
entity freq_div_4 is
  port ( 
    clk : in std_logic;
    rst : in std_logic;
    en : in boolean;
    clkdiv4 : out std_logic
  );
end freq_div_4;
architecture Exemplar of freq_div_4 is begin
  process (clk, rst)
  variable count : std_logic_vector (1 downto 0);
  begin
    if rst = '1' then
      count := (others => '0');
    elsif rising_edge(clk) then
      if en then
        count := count + "1";
      end if;
    end if;
    clkdiv4 <= count(1);
  end Exemplar;
```
Example 2:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library exemplar;
use exemplar.exemplar_1164.all;
entity freq_div_2 is
    port ( 
        clk : in std_logic;
        rst : in std_logic;
        en : in boolean;
        clkdiv2 : out std_logic 
    );
end freq_div_2;
architecture Exemplar of freq_div_2 is begin
    div_by_2: process (clk, rst)
    variable count : std_logic_vector (1 downto 0);
    begin
        if rst = '1' then
            count := (others => '0');
        elsif rising_edge(clk) then
            if en then
                count := count + "1";
            end if;
        end if;
        clkdiv2 <= count(1);
    end process;
end Exemplar;
```
Example 3

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library exemplar;
use exemplar.exemplar_1164.all;
entity top is
  port (clk : in std_logic;
        rst : in std_logic;
        en : in boolean;
        clkdiv8 : out std_logic);
  attribute clock_cycle of clk : signal is 15 ns;
  attribute clock_offset of clk : signal is 8 ns;
  attribute pulse_width of clk : signal is 9 ns;
  attribute required_time of clk_div_8 : signal is 24 ns;
  attribute arrival_time of en : signal is 5 ns;
end top;
architecture Exemplar of top is
  signal clk_div_4 : std_logic;
  attribute clock_cycle of clk_div_4 : signal is 60 ns;
  component freq_div_4
    port (clk : in std_logic;
          rst : in std_logic;
          en : in boolean;
          clkdiv4 : out std_logic);
  end component;
continued...
```
continued...

component freq_div_2
  port (
    clk : in std_logic;
    rst : in std_logic;
    en : in boolean;
    clkdiv2 : out std_logic
  );
end component;
begin
  freq_div_4_inst : freq_div_4 port map(
    clk => clk, en => en, rst => rst, clkdiv4 => clk_div_4);
  freq_div_2_inst : freq_div_2 port map(
    clk => clk_div_4, en => en, rst => rst, 
    clkdiv2 => clk_div_8);
end Exemplar;
ORCA Field-Programmable System Chips (FPSC)

The ORCA FPSC provides you with field-programmable logic and mask-programmable logic on the same device.
- Field-programmable logic is used to develop new parts.
- Mask-programmable logic is used to achieve the highest density, guaranteed performance, and increase in functionality for the tested parts of a design.

This section provides two examples of using LeonardoSpectrum with FPSC, and the resulting solutions or workarounds developed to solve specific problems.
- Writing Primitives in `noopt` (Attribute) Instances
- Propagating `nopad` (Attribute) to Top Level

Writing Primitives in `noopt` (Attribute) Instances

The contents for the `noopt` attribute are written out in the EDIF file. This is controlled by the `edifout_write_noopted_contents` variable. This workaround affects only the EDIF file; the output netlist is not involved.

If you want to prevent LeonardoSpectrum from writing primitives, you must set `edifout_no_prims_in_noopt` to true before you write the EDIF file.

Typical code is:

```
read test.vhd
load_lib orca3c
optimize -ta orca3c
set edifout_no_prims_in_noopt true
auto_write test.edf
```

Note: You must use `auto_write` utility script and not the `write` command.

Propagating `nopad` (Attribute) to the Top Level

The propagating is done with the `prop_nopad.tcl` script.

Note: Open your online browser and download the `prop_nopad.tcl` utility from:

This script traverses to the bottom level of your hierarchy and then bubbles the nopad attribute up to the top level. The prop_nopad command propagates nopad constraints from the blocks on the bottom of the design to the current block.

**Usage:**

prop_nopad [-v]

where: -v verbose flag, dumps messages

**Typical Code is:**

read test.vhd
<set all the constraints>
soure prop_nopad.tcl
prop_nopad
optimize -ta orca3c
set edifout_no_prims_in_noopt true
auto_write test.edf

**Note:** You must use auto_write utility script and not the write command.

**Note:** If you would like prop_nopad.tcl to be automatically sourced when you open LeonardoSpectrum, then add this line to:

$EXEMPLAR/data/exemplar.ini:
source <path>/prop_nopad.tcl
## Devices Supported for Lucent ORCA FPGA Families

The supported devices are: ORCA 2CA/2TA/3C/3T and 3000.

<table>
<thead>
<tr>
<th>Lucent ORCA 2CA family</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 4</td>
<td></td>
</tr>
<tr>
<td>Speed Grades supported: 2, 3, 4, 5</td>
<td></td>
</tr>
<tr>
<td>Devices Supported</td>
<td></td>
</tr>
<tr>
<td>or2c04a</td>
<td>M84 T100 T144 J160 S208</td>
</tr>
<tr>
<td>or2c06a</td>
<td>M84 T100 T144 J160 S208 S240 B256</td>
</tr>
<tr>
<td>or2c08a</td>
<td>M84 J160 S208 S240 B25 M84</td>
</tr>
<tr>
<td>or2c10a</td>
<td>J160 S208 S240 B256 B352</td>
</tr>
<tr>
<td>or2c12a</td>
<td>M84 S208 S240 B256 S304 B352</td>
</tr>
<tr>
<td>or2c15a</td>
<td>M84 S208 S240 B256 S304 B352 SB432</td>
</tr>
<tr>
<td>or2c26a</td>
<td>PS208 PS240 PS304 B352 SB432 SB600</td>
</tr>
<tr>
<td>or2c40a</td>
<td>PS208 PS240 PS304 SB432 SB600</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lucent ORCA 2TA family</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 4</td>
<td></td>
</tr>
<tr>
<td>Speed Grades supported: 2, 3, 4, 5</td>
<td></td>
</tr>
<tr>
<td>Devices Supported</td>
<td></td>
</tr>
<tr>
<td>or2t04a</td>
<td>M84 T100 T144 J160 S208</td>
</tr>
<tr>
<td>or2t06a</td>
<td>M84 T100 T144 J160 S208 S240 B256</td>
</tr>
<tr>
<td>or2t08a</td>
<td>M84 J160 S208 S240 B256</td>
</tr>
<tr>
<td>or2t10a</td>
<td>M84 J160 S208 S240 B256 B352</td>
</tr>
<tr>
<td>or2t12a</td>
<td>M84 S208 S240 B256 B352</td>
</tr>
<tr>
<td>or2t15a</td>
<td>M84 S208 S240 B256 B352 SB432</td>
</tr>
<tr>
<td>or2t26a</td>
<td>PS208 PS240 B352 SB432 SB600</td>
</tr>
<tr>
<td>or2t40a</td>
<td>PS208 PS240 SB432 SB600</td>
</tr>
</tbody>
</table>
### Lucent ORCA ATT3/3000 family

Default Speed Grade: 100

Speed Grades supported: 70, 100, 125, 150, 200, 230

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>3020</td>
</tr>
<tr>
<td>3030</td>
</tr>
<tr>
<td>3042</td>
</tr>
<tr>
<td>3064</td>
</tr>
<tr>
<td>3090</td>
</tr>
</tbody>
</table>

### Lucent ORCA 3C family

Default Speed Grade: 5

Speed Grades supported: 4, 5, 6

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>or3c55PS</td>
</tr>
<tr>
<td>or3c55BA</td>
</tr>
<tr>
<td>or3c80PS</td>
</tr>
<tr>
<td>or3c80SB</td>
</tr>
<tr>
<td>or3c80BA</td>
</tr>
<tr>
<td>or3c80BC</td>
</tr>
</tbody>
</table>
Lucent ORCA 3T family

Default Speed Grade: 5

Speed Grades supported: 4, 5, 6

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>or3t80</td>
</tr>
<tr>
<td>or3t20S</td>
</tr>
<tr>
<td>or3t20B</td>
</tr>
<tr>
<td>or3t30B</td>
</tr>
<tr>
<td>or3t55PS</td>
</tr>
<tr>
<td>or3t55BA</td>
</tr>
<tr>
<td>or3t125S</td>
</tr>
<tr>
<td>or3t125SB</td>
</tr>
<tr>
<td>or3t165S</td>
</tr>
<tr>
<td>or3t165B</td>
</tr>
<tr>
<td>or3t165SB</td>
</tr>
</tbody>
</table>

**Additional Features for ORCA**

- Neoprim Library: The Neoprim Library supports back annotation flow for Lucent ORCA technologies. For accurate timing back annotation use VHDL netlist generated by the following invocation:
  
  `ngd2vhd -n -t <input> <output>`

- RAMs: ORCA 2A library: RAMs with data size mod 4 = 3
- ORCA 3C/3T technologies are supported. Refer to the FPGA Architecture section in this chapter.
Schematic Symbol Libraries

The following items were added:
- Instance names were added to all symbols.
- Equations were added on LUT symbols.
- New symbols were added.

Mapping to ORCALUTs

LeonardoSpectrum maps directly to ORCA Foundry’s ORCALUTs. This improves design results.

ORCA Wire Load Model

The wire load model improves delay estimates for ORCA architectures. The wire load is used by default.

Pipelined Multiplier

Refer to the User’s guide, Chapter 10, for more information.

More New Features Supported for ORCA

LeonardoSpectrum supports the following:
- Improved Modgen to support ORCA counters, RAMs, etc.
- SLIC (supplemental logic and interconnect) modules used to design RAM enable logic.
- Synchronous flip flops are supported.
- PFU wide synchronous mapping is supported.
- FPSC (field programmable system chips) flow is supported. Note: A special Lucent ORCA script can map a design that is in a particular core, like PCI design, to FPGA and appropriate FPSC. Refer to ORCA Field Programmable System Chips section in this chapter.
LeonardoSpectrum provides synthesis support for the QuickLogic pASIC1, pASIC2 and pASIC3 devices and a new modgen library. Designs may be entered using VHDL or Verilog description. In addition, gate array and other FPGA designs may be retargeted to the QuickLogic devices using EDIF netlist. This chapter is divided as follows:

- Before Beginning
- FPGA Architecture
- Features
- Design Flow
- Optimization Style Points
- Reporting
- Process Derating Factors
- Design I/O
- Devices Supported

**Before Beginning**

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.
FPGA Architecture

The FPGA architecture consists of:
- pASIC Logic Cells
- I/O Buffers

pASIC Logic Cell

The basic logic element of the QuickLogic pASIC FPGAs is the Logic Cell. A multiplexer-based structure is the central part of the logic cell. The logic cell consists of two 6-input AND gates, four 2-input AND gates, three 2-to-1 multiplexers and a D flip-flop. In addition to the dedicated flip-flop, logic gates in each cell can be configured to provide two latches. Multiple outputs are also available from the logic cell.

I/O Buffers

In addition to the standard I/O buffers, the pASIC devices have dedicated clock buffers and high drive input buffers.

Features

- Support for pASIC3 is available. These libraries are supported through EDIF output into SPDE back-end tool. You must use SpDE software, version 6.3 or higher.
- Support for process and speed grades is available. The following values are available: WC-2, TYP-2, BC-2, WC-1, TYP-1, BC-1, WC-0, TYP-0, BC-0, WC-X, TYP-X, BC-X

Design Flow

Figure 12-1 shows a simplified design flow that illustrates the use of LeonardoSpectrum with the QuickLogic SpDE tools. Designs can be flat or hierarchical and can be entered using standard methods. The designs are optimized and mapped to the target pASIC technology by LeonardoSpectrum. The result is an EDIF netlist that is used by downstream SpDE tools.
LeonardoSpectrum currently does not accept QDIF files as input. However, pASIC designs in EDIF netlist are accepted. The EDIF netlist is hierarchical if all of the modules are described in a single EDIF file.

**Optimization Style Points - Quality**

Certain types of logic are optimized better than others: arithmetic circuits such as adders and multipliers are not as optimal as the best hand design or predefined macro. These constructs are better synthesized using module generation, if using VHDL or Verilog as the input format.

General control logic and state machines are optimized as well as or better than the best hand designs.
Combinational Logic Loops

By default LeonardoSpectrum attempts to replace combinational loops with latches. Alternatives to consider include removing the loop prior to optimization (but including the loop in the final design), or replacing the loop with a sequential element.

Sequential Optimization

LeonardoSpectrum eliminates registers and latches if the register or latch is latching a constant value and there is no set or reset used on the register. LeonardoSpectrum also eliminates registers or latches that have identical inputs; these registers are effectively merged. LeonardoSpectrum does not move registers or change the phase of the signal stored in the registers.

Internal Tristates

The pASIC architecture has no internal tristate gate. Therefore, all tristate gates must be connected to a pad. If your design has tristates, LeonardoSpectrum issues the message:

Internal tristates not available in this technology.

The internal tristates must be converted to combinational logic. Refer to Table 12-1.

Table 12-1. Tristate Variables

<table>
<thead>
<tr>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>set tristate_map TRUE</td>
<td>-tristate_map</td>
</tr>
<tr>
<td>Converts internal tristates to combinational logic.</td>
<td></td>
</tr>
<tr>
<td>optimize -chip</td>
<td>-chip</td>
</tr>
<tr>
<td>Converts internal tristates to a tristate or bidirectional pad.</td>
<td></td>
</tr>
</tbody>
</table>

Reporting

Refer to the LeonardoSpectrum User’s guide for reporting information.
Process Derating Factors

The following table for pASIC1 and pASIC2, list derating factors for these libraries. These values can be entered on the interactive command line shell or in batch mode.

pASIC1, pASIC2, and pASIC3 Derating Factors

<table>
<thead>
<tr>
<th>Value of Process</th>
<th>Operating Conditions</th>
<th>Speed Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>WC-2</td>
<td>worst case</td>
<td>-2</td>
</tr>
<tr>
<td>TYP-2</td>
<td>typical</td>
<td>-2</td>
</tr>
<tr>
<td>BC-2</td>
<td>best case</td>
<td>-2</td>
</tr>
<tr>
<td>WC-1</td>
<td>worst case</td>
<td>-1</td>
</tr>
<tr>
<td>TYP-1</td>
<td>typical</td>
<td>-1</td>
</tr>
<tr>
<td>BC-1</td>
<td>best case</td>
<td>-1</td>
</tr>
<tr>
<td>WC-0</td>
<td>worst case</td>
<td>0</td>
</tr>
<tr>
<td>TYP-0</td>
<td>typical</td>
<td>0</td>
</tr>
<tr>
<td>BC-0</td>
<td>best case</td>
<td>0</td>
</tr>
<tr>
<td>WC-X</td>
<td>worst case</td>
<td>-X</td>
</tr>
<tr>
<td>TYP-X</td>
<td>typical</td>
<td>-X</td>
</tr>
<tr>
<td>BC-X</td>
<td>best case</td>
<td>-X</td>
</tr>
</tbody>
</table>

Design I/O

LeonardoSpectrum maps I/O cells automatically. Based on the input description, the appropriate I/O cells from the pASIC technology library are used. LeonardoSpectrum does not map automatically to High Drive Input Buffers, Dual Port Input Buffers and Clock Buffers. You can map to these buffers manually using the PAD and BUFFER_SIG command.

Manual Assignment of I/O Buffers

There are two commands that can be used to manually assign I/O pads: PAD and BUFFER_SIG. These commands can be used as attributes from VHDL, or as commands from the interactive command line shell. The PAD command is the recommended way to assign pads. The PAD command can be used to assign any pad that LeonardoSpectrum is able to map. The PAD command is limited, however, and only works on inputs and outputs. The BUFFER_SIG command can be used to assign buffers and also be used on internal signals (internal clock buffers).
**PAD Command**

When using this constraint command the syntax is:

```
PAD IO_gate_name_signal_1 . . . signal_n
```

The following I/O gates can be manually mapped using the PAD command:

<table>
<thead>
<tr>
<th>Input Buffers</th>
<th>INPAD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HDPAD</td>
</tr>
<tr>
<td></td>
<td>HD2PAD</td>
</tr>
<tr>
<td></td>
<td>HD3PAD</td>
</tr>
<tr>
<td></td>
<td>HDIPAD</td>
</tr>
<tr>
<td></td>
<td>HDDPAD</td>
</tr>
<tr>
<td>Output Buffers</td>
<td>OUTPAD</td>
</tr>
<tr>
<td></td>
<td>OUTORPAD</td>
</tr>
<tr>
<td></td>
<td>OUTIPAD</td>
</tr>
<tr>
<td>Tristate Buffers</td>
<td>TRIPAD</td>
</tr>
<tr>
<td></td>
<td>TRIORPAD</td>
</tr>
<tr>
<td></td>
<td>TRIIPAD</td>
</tr>
<tr>
<td>Bidirectional Buffers</td>
<td>BIPAD</td>
</tr>
<tr>
<td></td>
<td>BIORPAD</td>
</tr>
<tr>
<td></td>
<td>BIIPAD</td>
</tr>
</tbody>
</table>

**BUFFER_SIG Command**

You can manually assign clock buffers. For example, use the constraint editor syntax:

```
BUFFER_SIG clk CKPAD
```

The following clock buffers are available: CKPAD, CKDPAD, and CKTPAD.
Assigning Pin Locations

Pin numbers can be assigned to designs targeted to pASIC devices. This can be done from VHDL using PIN_NUMBER and PIN_ARRAY_NUMBER attributes or from the interactive command line shell using the PIN_NUMBER procedure or set attribute commands on I/O ports.

Example for setting pin numbers from VHDL:

```vhdl
library IEEE; use ieee.std_logic_1164.all;
library exemplar; use work.exemplar_1164.all; -- Include the exemplar pkg

entity EXAMPLE is
  port (  
    CLK: bit;
    DIN: in std_logic_vector (4 downto 0);
    Q: out std_logic_vector (4 downto 0)
  );
  attribute pin_number of clk: signal is "1";
  attribute array_pin_number of din: signal is ("2", "3", "4", "5", "6");
end EXAMPLE;

architecture EXEMPLAR of EXAMPLE is
begin
  process (CLK)
  begin
    if (CLK = '1' and CLK'EVENT) then
      Q <= DIN;
    end if;
  end process;
end EXEMPLAR;
```

This is an example of assigning pin numbers from the interactive command line shell:

```
set_attribute -port -name pin_number -value 6 din (0)
```
Max Fanout

Use the Max Fanout field on the GUI to override the default max fanout load specified in the library. Refer to Table 12-2. However, a synthesized netlist with high fanout nets may be a problem for the place and route tool. The place and route tool usually splits the net arbitrarily. High fanout nets can cause significant delays on wires and become unroutable. On a critical path, high fanout nets can cause significant delays in a single net segment and cause the timing constraints to fail. To eliminate the need for splitting of the net by the place and route tool, the synthesis tool must maintain a reasonable number of fanouts for a net. LeonardoSpectrum tries to maintain reasonable fanout limits for each for each target technology. Default fanout limits are derived from the synthesis library. LeonardoSpectrum maintains reasonable fanouts by replicating the driver which results in net splitting. If replication is not possible, then the signal is buffered. The buffering of high fanout primary input signals is an example. Buffering the signal causes the wire to be slower by adding intrinsic delays.

Table 12-2. Mapping and Optimization Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Fanout: window</td>
<td>n/a</td>
<td>set max_fanout &lt;int&gt;</td>
<td>n/a</td>
</tr>
</tbody>
</table>

![Synthesis Switches](image)
Devices Supported

The devices supported are:

- pASIC1
- pASIC2
- pASIC3

<table>
<thead>
<tr>
<th>pASIC1</th>
<th>Default Speed Grade: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed Grades supported: 2, 1, 0, X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>P8x12b</td>
</tr>
<tr>
<td>P12x16b</td>
</tr>
<tr>
<td>P16x24b</td>
</tr>
<tr>
<td>P24x32b</td>
</tr>
</tbody>
</table>

| Packages | PL44 PL68 PL84 CG84 PF100 PV100 CF100 CG144 PF144 CF160 PQ208 CF208 |
| Process | Typical Case (default), Best Case, Worst Case |

<table>
<thead>
<tr>
<th>pASIC2</th>
<th>Default Speed Grade: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed Grades supported: 2, 1, 0, X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>QL2003</td>
</tr>
<tr>
<td>QL2005</td>
</tr>
<tr>
<td>QL2007</td>
</tr>
<tr>
<td>QL2009</td>
</tr>
</tbody>
</table>

| Packages | PF144, PB256, PL84, PQ208 |
| Process | Typical Case (default), Best Case, Worst Case |
### pASIC3

**Default Speed Grade:** 1

**Speed Grades supported:** 2, 1, 0, X

### Devices Supported

<table>
<thead>
<tr>
<th>QL3012</th>
<th>PL84 PF100 PF144</th>
</tr>
</thead>
<tbody>
<tr>
<td>QL3025</td>
<td>PF144 PQ208 PB256</td>
</tr>
<tr>
<td>QL3040</td>
<td>PQ208 PB456</td>
</tr>
<tr>
<td>QL3060</td>
<td>PQ208 PB456</td>
</tr>
</tbody>
</table>

### Packages

- PL84
- PF144
- PQ208
- PB256
- TQ100

### Process

- Typical Case (default), Best Case, Worst Case
The LeonardoSpectrum tools provide synthesis for all families of Xilinx CPLDs, which include XC7200A/7300 and XC9500/9500XL. Designs may be entered using standard VHDL and Verilog descriptions or EDIF netlist. This chapter presents information specific to the use of the Xilinx CPLDs families as a source or target technology.

This chapter is divided as follows:
- Before Beginning
- Xilinx CPLD Architecture
- Design Flow
- Synthesis Features
- Module Generation
- Reporting
- Xilinx CPLD Family Supported Devices

Before Beginning

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.
Xilinx CPLD Architecture

The basic Xilinx CPLD structures consists of:

- Function Blocks (FBs)
- I/O Blocks (IOBs)

Each function block is comprised of a set of macrocells which are capable of implementing a combinational or registered function. In XC9500/9500XL, a function block has a programmable AND-array and product term allocator that feeds the macrocells. Each macrocell can implement a combinational logic function of up to five product terms.

An XOR gate is available in each macrocell which can increase the number of product terms. The product term allocator can reassign other product terms within the FB to increase the logic capacity of a macrocell beyond five direct terms. Each macrocell also contains a register that can be programmed to be a D flip-flop or T flip-flop with clear and preset inputs. The function blocks impose a fanin limitation on top of the product term limitation per macrocell.

The XC7200A/XC7300 family also supports carry lookahead generator that reduces the ripple-carry delay of wide arithmetic functions such as adders, subtractors, and magnitude comparators.

I/O cells in Xilinx CPLDs can be configured as inputs, outputs, tristate outputs or bi-directional pins. LeonardoSpectrum automatically inserts the default I/O buffers in the output netlists.

Design Flow

A simplified design flow is shown in Figure 13-1 to illustrate how LeonardoSpectrum is used with the XACT-CPLD tools to provide a complete design solution. Designs are entered using standard methods and are optimized to the target XC7200A, XC7300 or XC9500 devices. The output design can be written either as an EDIF netlist that is compatible with M1-CPLD software or as an XNF netlist that is compatible with Xilinx XEPLD software.
Synthesis Features

The device selection, I/O buffers and pads, and pin (LOC) assignment features are available for both XC7200A/XC7300 and XC9500 Xilinx CPLD families.

Device Selection

By default, the Xilinx CPLD software automatically selects the appropriate device to fit your design. Optionally, you can specify the Part number and speed grade or process of the target device. Refer to Table 13-1.
Table 13-1. Device Selection

<table>
<thead>
<tr>
<th>Technology Settings</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use default part number or select from pulldown.</td>
<td>set part &lt;part_number&gt; -part=&lt;part_number&gt;</td>
<td>-process=&lt;name&gt;</td>
</tr>
</tbody>
</table>

This is part number that LeonardoSpectrum writes out in the output EDIF.

### I/O Buffers and Pads

LeonardoSpectrum automatically inserts the default I/O buffers in the output netlist. They are IBUF, OBUF, OBUFE (for tristated output ports) and IOBUFE (for bidirectional ports). For Xilinx M1 EDIF, LeonardoSpectrum also inserts I/O PADs in the EDIF netlist. The other I/O Buffers, for example, OBUF_S, IBUF_S, and OBUFE_S are available through component instantiation.
**Instantiating 0BUFs**

A 0BUF has three ports: 1, 0, and GTS. Typically the GTS port is not used. This port must be connected to 'Z' to avoid receiving an error. Refer to the following example if you want to instantiate a Xilinx output buffer without using the global tristate feature. The GTS pin can remain unconnected.

```vhdl
u1 : OBUF_S_6 PORT MAP ( O => O, I => internal, GTS => 'Z' );
```

---

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY slow_slew IS
PORT (
I: IN STD_LOGIC ;
O : OUT STD_LOGIC
);
END slow_slew;
ARCHITECTURE rev1 OF slow_slew IS
signal internal : std_logic;
COMPONENT IBUF
PORT ( I : IN STD_LOGIC ;
O : OUT STD_LOGIC
);
END COMPONENT;
COMPONENT OBUF_S_6
PORT ( I, GTS : IN STD_LOGIC;
O : OUT STD_LOGIC
);
END COMPONENT;
BEGIN
u1 : OBUF_S_6 PORT MAP ( O => O, I => internal, GTS => 'Z' );
u2 : IBUF PORT MAP ( O => internal, I => i );
END rev1;
```
Pin (LOC) Assignment

LeonardoSpectrum supports pin assignment from VHDL, Verilog, or control file. The pin locations are generated in the output netlists as LOC parameters on EXT records in XNF or as EDIF properties on I/O buffer instances in EDIF.

Module Generation

LeonardoSpectrum supports module generation for XC7200A and XC7300 families only.

LeonardoSpectrum supports various technology specific implementations of arithmetic and relational operators used in VHDL or Verilog RTL. Since these implementations have been designed optimally for the specific target technology, the synthesis results are smaller and/or faster and take less time to compile.

The following operators and miscellaneous functions have technology specific architectures in the modgen libraries:

- Relational Operators: = /= < <= > >=
- Arithmetic Operators: + -
- Miscellaneous Functions: incremeneter, decremeneter
### Reporting

Refer to the LeonardoSpectrum User’s Guide for reporting information.

### Xilinx CPLD Family Supported Devices

The devices supported are:
- XC7200A
- XC7300
- XC9500/XL

#### Xilinx CPLD Family - Xilinx XC7200A

<table>
<thead>
<tr>
<th>Default Speed Grade: 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed Grades supported: 16, 20, 25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>7236</td>
</tr>
<tr>
<td>7272</td>
</tr>
</tbody>
</table>

#### Xilinx CPLD Family - Xilinx XC7300

<table>
<thead>
<tr>
<th>Default Speed Grade: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed Grades supported: 5, 7, 10, 12, 15, 20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>7336</td>
</tr>
<tr>
<td>7354</td>
</tr>
<tr>
<td>7372</td>
</tr>
<tr>
<td>73108</td>
</tr>
<tr>
<td>73144</td>
</tr>
</tbody>
</table>
### Xilinx CPLD Family - Xilinx XC9500

**Default Speed Grade:** 7

**Speed Grades supported:** 5, 7, 10, 15, 20

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>9536</td>
</tr>
<tr>
<td>9572</td>
</tr>
<tr>
<td>95108</td>
</tr>
<tr>
<td>95144</td>
</tr>
<tr>
<td>95216</td>
</tr>
<tr>
<td>95288</td>
</tr>
</tbody>
</table>

### Xilinx CPLD Family - Xilinx XC9500XL

**Default Speed Grade:** 7

**Speed Grades supported:** 7, 4, 5, 6, 10, 15, 20

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>9536</td>
</tr>
<tr>
<td>9572</td>
</tr>
<tr>
<td>95144</td>
</tr>
<tr>
<td>95288</td>
</tr>
</tbody>
</table>
### Xilinx CPLD Family - Xilinx 9500XV

- **Default Speed Grade:** 7
- **Speed Grades supported:** 7, 10, 15

#### Devices Supported

<table>
<thead>
<tr>
<th>Device Code</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>9536xv</td>
<td>PC44 CS48 VQ64</td>
</tr>
<tr>
<td>9572xv</td>
<td>PC44 VQ64 TQ100</td>
</tr>
<tr>
<td>95144xv</td>
<td>TQ100 TQ144 CS144</td>
</tr>
<tr>
<td>95288xv</td>
<td>TQ144 PQ208 BG352</td>
</tr>
</tbody>
</table>
LeonardoSpectrum provides full synthesis support for the following Xilinx LCA based architectures:

- XC3000/A/L, XC3100/A
- XC4000/A/E/EX/L/XL/XLA/XV, XC5200
- Spartan/XL, Spartan2 (Virtex)
- CoolRunner

Designs may be entered using VHDL or Verilog descriptions or EDIF and XNF netlists. In addition, gate array and other FPGA designs may be retargeted to the Xilinx devices. This chapter is divided as follows:

- Before Beginning
- Xilinx LCA Architecture
- Synthesis and Optimization Features
- Data Path Synthesis and Modgen Implementation
- GSR Usage and Startup Blocks
- Design I/O
- Reporting
- Using TimeSpecs
- Additional Xilinx Specific Options
- Interfacing with XBLOX
- Additional Xilinx Architecture Features
- Pullups and Pulldowns
- Xilinx FPGA Devices Supported
Before Beginning

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.

Xilinx LCA Architecture

Configurable Logic Block

The Xilinx FPGAs have two basic structures: the configurable logic block (CLB) and the input/output block (IOB). The CLBs are at the core of the Xilinx FPGA architecture. Each CLB contains combinational logic and registers. The combinational logic section of the CLB is limited in the number of inputs.

Input/Output Block

The IOBs in the Xilinx FPGAs can be configured as input, tristate output, or bidirectional pins. Flip-flops and input latches may be used in IOBs. LeonardoSpectrum recognizes these gates when targeting a Xilinx device and utilizes the logic available in the IOBs.

Synthesis Design Flows

A simplified design flow is shown in Figure 14-1. LUT mapping and CLB packing can be enabled/disabled on the GUI. Figure 14-2 shows a simplified back annotation.

**Note:** The output netlist may either be for XACT (XNF) flow or M1 (EDIF) flow. You may also use XNF as an input to M1 software.

**Note:** The XNF flow may not be supported in the future by Xilinx.
Figure 14-1  Simplified Design Flow
Figure 14-2  Simplified Back Annotation
Design Flow Guidelines

During Xilinx synthesis, options are available on Advanced Settings. Refer to Table 14-1.

Max Fanout

Use the Max Fanout field on the GUI to override the default max fanout load specified in the library. However, a synthesized netlist with high fanout nets may be a problem for the place and route tool. The place and route tool usually splits the net arbitrarily. High fanout nets can cause significant delays on wires and become unroutable. On a critical path, high fanout nets can cause significant delays in a single net segment and cause the timing constraints to fail.

To eliminate the need for splitting of the net by the place and route tool, the synthesis tool must maintain a reasonable number of fanouts for a net. LeonardoSpectrum tries to maintain reasonable fanout limits for each for each target technology. Default fanout limits are derived from the synthesis library.

Note: The LUT buffering and replication is supported for the Xilinx XC3000/A/L and XC4000/A/E/EX/L/XL/XLA/XV, XC5200, Spartan/XL, Spartan 2, and Virtex technologies (see Virtex chapter).

LeonardoSpectrum maintains reasonable fanouts by replicating the driver which results in net splitting. If replication is not possible, then the signal is buffered. The buffering of high fanout primary input signals is an example. Buffering the signal causes the wire to be slower by adding intrinsic delays.

Max Fanout Attribute

On specific nets, the user can set an attribute to control the max_fanout value:

```
set_attribute -net <name> - name lut_max_fanout -value <int>
```

Note: Setting this attribute takes precedence over any global fanout specifications.

Note: Max Fanout field is not available for Xilinx 7200a, 7300, or 9500 technologies.
Table 14-1. Output Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Fanout: window</td>
<td>n/a</td>
<td>set lut_max_fanout &lt;integer&gt;</td>
<td>n/a</td>
</tr>
<tr>
<td>Write CLBs</td>
<td>on</td>
<td>set write_clb_packing true</td>
<td>-write_clb_packing</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>default</td>
<td>-nopack_clbs</td>
</tr>
</tbody>
</table>

Controls packing of LUTs into CLBs.

**Note:** If the Write CLBs is not selected then XNF output will not be in terms of MAP (FMAP/HMAP) symbols.

**Note:** CLB packing should be used carefully. It will pack the CLBs tightly and will not consider routing resources, which may cause routing problems in some cases. The other option is to let XACT do the CLB packing. However, in some cases LeonardoSpectrum CLB packing gave superior results over XACT. Also, by packing to CLBs, LeonardoSpectrum can estimate interconnect delays more accurately. Currently, CLB packing is available for the XC4000 architecture only.
**Synthesis and Optimization Features**

This section is divided as follows:
- Fanin Limited Optimization
- Lookup Table (LUT) Mapping
- Constraint-Driven Timing Optimization, **Level 3**
- CLB Packing for Xilinx XC4000 Technology, **Level 3**
- XC4000 CLB Architecture Constraint

LeonardoSpectrum has several optimization features for the Xilinx architectures.
- To take full advantage of the Xilinx architecture, Xilinx-specific optimization techniques have been developed by Exemplar Logic. These include a “fanin limited” optimization algorithm that understands and takes advantage of the input specifications of the Xilinx CLB.
- An important feature is the ability to map logic functions into lookup tables. This mapping is available for XC3000, XC4000/E, and XC5200 technologies.
- You can set timing constraints (`arrival_time` on inputs, `required_time` on outputs and clocking schemes) and apply timing optimization to speed up the design.
- After the logic is mapped to lookup tables, the lookup tables can be packed to CLBs. CLB packing is optional.
- In the output XNF netlist you can specify if LeonardoSpectrum writes out FMaps, HMaps, and CLBs information.
- Module generation is available for implementing data path logic from HDL design source.
- In addition to optimization for the CLB architecture of the Xilinx LCA devices, Leonardo recognizes other Xilinx features such as the global set/reset signal, wide edge decoders, and mapping to complex I/Os.

**Fanin Limited Optimization**

The Xilinx 4000 architecture is used in this example. In this technology any function with up to four inputs can be mapped into a FMAP lookup table. Any function with up to three inputs (with some restrictions) can be mapped into a HMAP. Two FMAPs and one HMAP (with some restrictions) can be packed into one CLB. In summary, a 4-input XOR uses the same area and is as fast as a 4-input AND gate.
One approach to designing in this architecture is to decompose a function into a simpler AND/OR equivalent representation, and then split the gates with large fanin into multiple gates. Consider this example:

\[
X = (A \cdot (B+C)) + (B \cdot D) + (E \cdot F \cdot G \cdot H \cdot I)
\]

Represented in AND and OR gates, X is decomposed as:

\[
\begin{align*}
X &= T1 + T2 + T3 \\
T1 &= A \cdot T4 \\
T2 &= B \cdot D \\
T3 &= E \cdot F \cdot G \cdot H \cdot I \\
T4 &= B + C
\end{align*}
\]

Because T3 has more than four inputs, further decomposition is required:

\[
\begin{align*}
T3 &= E \cdot F \cdot G \cdot T5 \\
T5 &= H \cdot I
\end{align*}
\]

When the design is decomposed, the Xilinx physical place and route software can be used to place the design into physical CLBs. For this design T1, T2, T5, T3 and X will each map into a single lookup table as follows:
T1, T3 and X will be packed into CLB1 while T2 and T5 will be packed into CLB2. The critical path will be two levels of CLBs.

\[ H \rightarrow T5 \rightarrow FMAP \ (T3) \rightarrow HMAP \ (X) \]

When using fanin limited decomposition techniques, the decomposition of the equation yields better results:

\[ X = T1 + (T2 \cdot E) \]
\[ T1 = A \cdot (B + C) + (B \cdot D) \]
\[ T2 = F \cdot G \cdot H \cdot I \]

This fits into two FMAP functions and one HMAP, that can be packed into one CLB. The critical path is reduced to one CLB level.
Look Up Table (LUT) Mapping

LUT mapping maps fanin limited logic functions to Xilinx function generators. This technology is available for XC3000, XC4000/E/EX/XL and XC5200 architectures. When targeting XC4000 LUT mapping maps the logic to F, G and H function generators. Similarly, when targeting XC5200 technology, logic is mapped to F function generators and F5_MUX gates. This minimizes the number of function generators when optimizing for area and the delay when optimizing for delay.

The output is a mapped XNF netlist (i.e., XNF in terms of FMAP and HMAP symbols for the 4000 technology and in terms of FMAP, F5MAP, and F5_MUX symbols for XC5200 technology). Consider the following description of a 4 to 1 multiplexer:

```
T1 = A * C1' * C2';
T2 = B * C1' * C2 ;
T3 = C * C1 * C2';
T4 = D * C1 * C2 ;
OUT = T1 + T2 + T3 + T4;
```
Note – As shown in Figure 14-6, after optimization, decomposition, and mapping the input description requires only one CLB (2 F and 1 H function generator). This input description would require five function generators in the original form.

When targeting XC5200 technology, a F5_MUX gate will be used.
The design is mapped using two F function generators and a F5_MUX gate.

**Constraint-Driven Timing Optimization**

In **Level 3** the `optimize_timing` command can be used after `optimize` to improve the timing quality of the design. The `optimize_timing` command works most effectively when timing constraints are specified. In case timing constraints are not specified, the `-force` option can be used to try to improve the longest path.

```
optimize_timing -force
```

The `optimize_timing` command restructures combinational logic to reduce the levels of logic between the critical signal and the output. It also tries to map the restructured portion of the logic effectively to the particular technology. For the Xilinx 4000 design, it tries to map the critical signal to the direct input of the HMAP in the CLB. The `optimize_timing` may be invoked repeatedly for added improvements.
The specific end point or instance to improve can be specified to the optimize_timing command with the -through option.

```bash
optimize_timing -through
```

**CLB Packing for Xilinx XC4000 Technology**

In **Level 3** the CLB Packing algorithm can be run on a design that is mapped to FMAPs/HMAPs function generators to generate block (CLB) level information. The CLB packing operation has these two attributes:

1. You can get an accurate estimate of the design in terms of occupied CLBs.
2. The delay estimate on a post synthesized design is very accurate, typically within 5 percent of the post place and route design.

CLB packing should be invoked after running the optimize command on a design from the interactive command line shell.

```bash
pack_clbs
```

CLB packing function generates the CLB information on a design using the HBLKNM parameter on FMAPs, HMAPs, and sequential gates.

**XC4000 CLB Architecture Constraints**

Due to architectural constraints of a CLB in XC4000 for LeonardoSpectrum, a CLB can pack up to a maximum of two FMAPs, one HMAP, and two DFFs. But these functional blocks (FMAP/HMAP) are not independent entities because of their connectivity. These blocks have some requirements on the fanouts and number of outputs required outside the CLB, which are handled during the CLB packing operation.
The constraints for LeonardoSpectrum are:

- Only two out of three function generators (two FMAPs, and one HMAP output) can fanout out of a CLB.
- The two DFFs can be used if at least one of the DFFs D input is driven by one function generator (FG) within a CLB.
- At least two out of three inputs of the H FG should be sourced from F or G function generators.
- All the control inputs of two DFFs (clock, clock enable, set/reset) should be identical nets.
An example of CLB packing is shown for the following test case:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library exemplar;
use exemplar.exemplar_1164.all;
entity test is
 port ( 
    CLK : in std_logic;
    RST : in std_logic;
    i1, i2, i3, i4 : in std_logic;
    i5, i6, i7, i8 : in std_logic;
    i9 : in std_logic;
    H : out std_logic;
    qG : inout std_logic;
    Q : inout std_logic_vector(1 downto 0));
end test;

architecture instance of test is
 signal F : std_logic;
 signal G : std_logic;

procedure dffc( signal input : in std_logic;
    signal clr, clk: in std_logic;
    signal output : inout std_logic)
begin
 if (clr='1') then
  output <= '0';
 elsif (clk'event and clk='1') then
  output <= input;
 else
  output <= output;
 end if;
end dffc;
begin
 F <= i1 and i2 and i3 and i4;
 G <= i5 and i6 and i7 and i8;
 H <= F and G and i9;
 dffc (G, RST, CLK, qG);
 dffs: for i in 0 to 1 generate
 dffc (F, RST, CLK, Q(i));
 end generate dffs;
end;
```
Figure 14-8  Logic Packed into CLB
The following is an example XNF netlist:

```
SYM, f, EQN, EQN=((I0*I1*I2*I3)), LIBVER=2.0.0
PIN, I0, I, i1,
PIN, I1, I, i2,
PIN, I2, I, i3,
PIN, I3, I, i4,
PIN, O, O, f,
END

SYM, g, EQN, EQN=((I0*I1*I2*I3)), LIBVER=2.0.0
PIN, I0, I, i5,
PIN, I1, I, i6,
PIN, I2, I, i7,
PIN, I3, I, i8,
PIN, O, O, g,
END

SYM, h_rename, EQN, EQN=((I0*I1*I2)), LIBVER=2.0.0
PIN, I0, I, i9,
PIN, I1, I, f,
PIN, I2, I, g,
PIN, O, O, h,
END

SYM, XMPLR_INST_29_I1_I1, DFF, HBLKNM=CLB0, INIT=R, LIBVER=2.0.0, SCHNM=FDC
PIN, C, I, clk,
PIN, CLR, I, rst,
PIN, Q, O, qg, 3.000000
PIN, D, I, g,
END

SYM, XMPLR_INST_37_I1_I1, DFF, HBLKNM=CLB0, INIT=R, LIBVER=2.0.0, SCHNM=FDC
PIN, C, I, clk,
PIN, CLR, I, rst,
PIN, Q, O, g<1>, 3.000000
PIN, D, I, f,
END

SYM, FMAP_0, FMAP, HBLKNM=CLB0, LIBVER=2.0.0
PIN, II, I, i1
PIN, I2, I, i2
PIN, I3, I, i3
PIN, I4, I, i4
PIN, O, I, f
END

SYM, FMAP_1, FMAP, HBLKNM=CLB0, LIBVER=2.0.0
PIN, II, I, i5
PIN, I2, I, i6
PIN, I3, I, i7
PIN, I4, I, i8
PIN, O, I, g
END

SYM, HMAP_2, HMAP, HBLKNM=CLB0, LIBVER=2.0.0
PIN, I1, I, i9
PIN, I2, I, f
PIN, I3, I, g
PIN, O, I, h
....
```
Data Path Synthesis and Modgen Implementation

LeonardoSpectrum supports technology-specific implementations of arithmetic and relational operators used in VHDL or Verilog RTL. Since these implementations have been designed optimally for the Xilinx target technology, the synthesis results module generation are in general smaller and/or faster and take less time to compile. LeonardoSpectrum supports module generation for the following Xilinx technologies:

<table>
<thead>
<tr>
<th>Technology</th>
<th>Module Generation Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC3000/A/L</td>
<td>xi3 (default), xblox3</td>
</tr>
<tr>
<td>XC4000/A/H</td>
<td>xi4 (default), xblox4</td>
</tr>
<tr>
<td>XC4000E/EX/XL</td>
<td>xi4e</td>
</tr>
<tr>
<td>XC5200</td>
<td>xi5</td>
</tr>
</tbody>
</table>

Supported Operators

The following operators are supported for Xilinx technologies:

Logical Operators: and, or, nand, nor, xor, xnor

Relational Operators: =, /=, <, <=, >, >=

Arithmetic Operators: +, -, *

Miscellaneous Functions: counters (up/down, loadable, etc), RAMs, incrementer/decrementer, absolute value, unary minus, decoders

Module generators use dedicated hardware resources when possible. Consequently, the modgen implementation of operators like addition, subtraction, counters, and relational operators is in general smaller in area and faster in delay. For example:

- Adder in xi4 modgen is implemented using the dedicated carry chain in the XC4000 architecture to implement the adder carry logic. This leads to very fast carry propagation and good timing performance. Modgen implementation of an adder for XC5200 technology uses the CY_MUXs to implement the adder carry logic.
- Relational operators <, <=, >, and >= are implemented using the dedicated carry logic in the XC4000 architecture leading to a small and very fast implementation.
- xblox3 and xblox4 modgen use X-BLOX components to implement data path elements.
### User Defined Options and Variables for Modgen

Refer to Table 14-2 for Optimize Options for modgen.

#### Table 14-2. Optimize Options

<table>
<thead>
<tr>
<th>Optimize Options</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Auto - Picks smallest if optimization in area mode; and picks fastest if optimization in delay mode.</td>
<td>on (default)</td>
<td>set modgen_select auto -select_modgen=auto</td>
<td></td>
</tr>
<tr>
<td>• Smallest - Picks the best compact implementation available.</td>
<td>off</td>
<td>set modgen_select smallest -select_modgen=smallest</td>
<td></td>
</tr>
<tr>
<td>• Small - Picks a compact implementation.</td>
<td>off</td>
<td>set modgen_select small -select_modgen=small</td>
<td></td>
</tr>
<tr>
<td>• Fast - Picks a fast implementation.</td>
<td>off</td>
<td>set modgen_select fast -select_modgen=fast</td>
<td></td>
</tr>
<tr>
<td>• Fastest - Picks the fastest implementation available.</td>
<td>off</td>
<td>set modgen_select fastest -select_modgen=fastest</td>
<td></td>
</tr>
</tbody>
</table>

- Use Technology specific module generation library. If not selected, then default library is used.
**GSR Usage and Startup Blocks**

Xilinx flip-flops feature set/reset inputs that can be driven from a dedicated Global Set/Reset (GSR) signal. The GSR is implicit set/reset for all registers, and does not require any routing resources. This is available in all families except for: XC3000 and XC3100. If the design is suitable for automatic inference of global set/reset, then LeonardoSpectrum would automatically infer a GSR signal and insert a Startup block in the design.

The HDL description should be written with one asynchronous signal that initializes every DFF to the value at POWERUP. DFFs that power up as 0 should use the global signal as an asynchronous reset; DFFs that power up as 1 should use the global signal as an asynchronous set. You can use both set and reset in the same design. Refer to Table 14-3.

**Note: Level 3**: Setting the `global_sr <my_reset>` Tcl variable allows LeonardoSpectrum to use the specified Active High Signal as Global Set/Reset. The signal is disconnected from the registers Set/Reset pin and connected to a startup block.

**Note**: A STARTUP block is instantiated automatically if `global_sr` is set to a signal name (`my_reset`) in the design, or if `infer_gsr` variables are set to TRUE. The purpose of the STARTUP block is to accept the reset signal and to distribute the reset implicitly through the dedicated GSR signal.
Table 14-3. Global Set/Reset

<table>
<thead>
<tr>
<th>Technology Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign GSR:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detects the global set/reset signal.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ Auto</td>
<td>default true</td>
</tr>
<tr>
<td></td>
<td>□ Manual</td>
<td>set infer_gsr</td>
</tr>
</tbody>
</table>

When ■ Auto is selected, □ Manual is not available and the Signal field is grayed out. When ■ Manual is selected, you can type in the Signal field.

Assign Global SR: my_reset
Decorates an active high signal name as global set/reset.

set_global_sr <my_reset>

default not set
XNF GSR Specific

The `xlx_preserve_gsr` and `xlx_preserve_gts` are XNF reader specific only. The purpose of these variables is to preserve the Xilinx chip GSR, GTS behavior on the retargeted chip. The S/R of each register in the output technology is not OR of local and global S/R. It is only Set/Reset with local S/R. Refer to Table 14-4.

Table 14-4. Level 3 Tristate Map

<table>
<thead>
<tr>
<th>Interactive Command Line Shell</th>
<th>Available in Families</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set xlx_preserve_gsr true</code></td>
<td>All</td>
<td>Preserves Xilinx chip GSR.</td>
</tr>
<tr>
<td>default false</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>set xlx_preserve_gts true</code></td>
<td></td>
<td>Preserves Xilinx chip GTS.</td>
</tr>
<tr>
<td>default false</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
library ieee;
use ieee.std_logic_1164.all;
entity gsr_example is
  port ( reset, clk : in std_logic;
         d_in   : in std_logic_vector (7 downto 0);
         q_out  : out std_logic_vector (7 downto 0) );
end gsr_example;
architecture active_hi of gsr_example is
begin
  -- Use reset to initialize flip-flops: The q_out byte is initialized
  -- by GSR to be 1’s in the upper nibble, and 0’s in the lower nibble.
p0: process (clk, reset)
  begin
    if (reset = '1') then
      q_out(7 downto 4) <= "1111"; -- asynchronous set
      q_out(3 downto 0) <= "0000"; -- asynchronous reset
    elsif clk’EVENT and clk='1' then
      q_out <= d_in;
    end if;
  end process p0;
end active_hi;
Example 1b. VHDL Example using GSR for an Active Low Reset

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity gsr_example is
  port ( reset_b, clk : in std_logic;
         d_in    : in std_logic_vector (7 downto 0);
         q_out   : out std_logic_vector (7 downto 0) );
end gsr_example;
architecture active_lo of gsr_example is
begin
  -- Use reset to initialize flip-flops
  p0: process (clk, reset_b)
  begin
    if (reset_b = '0') then
      q_out(3 downto 0) <= "0000";  -- asynchronous reset
      q_out(7 downto 4) <= "1111";  -- asynchronous set
    elsif clk'EVENT and clk='1' then
      q_out <= d_in;
    end if;
  end process p0;
end active_lo;
```
Example 2a. Verilog Example of GSR Usage for an Active-Hi Reset

```verilog
module gsr_example ( reset, clk, d_in, q_out );
    input reset, clk;
    input [7:0] d_in;
    output [7:0] q_out;

    reg [7:0] q_out;
    // Use reset to initialize flip-flops
    always@ (posedge clk or posedge reset)
        if (reset)
            begin
                q_out[3:0] = 4'b0000; // asynchronous reset
                q_out[7:4] = 4'b1111; // asynchronous set
            end
        else
            q_out = d_in;
endmodule
```

Example 2b. Verilog Example of GSR Usage for an Active-Low Reset

```verilog
module gsr_example ( reset_b, clk, d_in, q_out );
    input reset_b, clk;
    input [7:0] d_in;
    output [7:0] q_out;

    reg [7:0] q_out;
    // Invert the input reset_b pin and apply to startup block.
    // Use reset to initialize flip-flops
    always@ (posedge clk or negedge reset_b)
        if (!reset_b)
            begin
                q_out[3:0] = 4'b0000; // asynchronous reset
                q_out[7:4] = 4'b1111; // asynchronous set
            end
        else
            q_out = d_in;
endmodule
```
Using Edge Decoders

The XC4000/E/EX/XL families have dedicated decoder circuitry at each edge of the device. These decoders are useful for wide-input decoding (e.g., address decoders might accept upwards of 16 address bits). These functions are better implemented in dedicated decoders because general-purpose CLBs would require multiple levels of delay.

LeonardoSpectrum allows for utilization of these wide decoders by means of a special Decoder Wired-AND (DWAND) component. You can directly instantiate this component from a source library.

The library must be loaded into LeonardoSpectrum before instantiation, otherwise LeonardoSpectrum will just black box the DWAND component and put an empty SYM record in the output XNF netlist.
**Example 3a. VHDL Example:** Using the GENERATE Statement to Instantiate a DWAND Component

```vhdl
library ieee; use ieee.std_logic_1164.all;
entity decoder_example is
  port ( invect : in std_logic_vector(15 downto 0);
         outport : out std_logic );
end decoder_example;
architecture exemplar of decoder_example is
component dwand
  port ( i : in std_logic; o : out std_logic );
end component;
component pullup
  port ( o : out std_logic );
end component;

attribute noopt: boolean;
attribute noopt of dwand: component is true;
attribute noopt of pullup: component is true;
begin
  -- Generate a DWAND for each bit
  wide_and: for I in invect’low to invect’high generate
    dwandr: dwand port map ( int(i), bingo );
  end generate;
  -- A pullup is required to assert signal when no drivers are active
  pullup_comp: pullup port map ( bingo );
end exemplar;
```
Example 3b. Verilog Example: Using DWAND by Direct Instantiation

```
module decoder (invect, outport);
  input [15:0] invect;
  output outport;

  // Instantiate these DWAND’s individually:
  // Verilog doesn’t have a “generate” equivalent
  dwand dwand0 ( .i(invect [0]), .o(outport) );
  dwand dwand1 ( .i(invect [1]), .o(outport) );
  dwand dwand2 ( .i(invect [2]), .o(outport) );
  ...
  dwand dwand13 ( .i(invect [3]), .o(outport) );
  dwand dwand14 ( .i(invect [4]), .o(outport) );
  dwand dwand15 ( .i(invect [5]), .o(outport) );

  // Escape “pullup” since it is a Verilog primitive name.
  \pullup pullup_comp ( outport );
endmodule

// Define empty modules here.

module dwand (i, o);
  input i;
  output o;
endmodule

module \pullup (o);
  output o;
endmodule
```

Using Internal Tristate Buffers

Xilinx LCA architectures provide internal tristate buffers that may be used as a design alternative to gate implementations. Also, depending on the design, some tristate buffers may be more efficiently implemented with multiplexers.

LeonardoSpectrum allows the designer to experiment with both tristate buffers and multiplexers. The designer can use tristate buffers in the high level language description of the circuit and then instruct LeonardoSpectrum to either keep the internal tristate buffers or convert them to multiplexers.
To keep the tristate buffers, use the default settings. To convert tristate buffers to muxes, set the `tristate_map` variable to true. Refer to Table 14-5.

Table 14-5. Tristate Map

<table>
<thead>
<tr>
<th>Interactive Command Line Shell</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set tristate_map true</code></td>
<td>Converts internal tristates to combinational logic.</td>
</tr>
<tr>
<td>default false</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Potential buss conflict is not checked. Ensure that only one driver is driving the buss at a particular time.
Example 4a. VHDL Example Inferring an Internal Tristate Using Both a Component and Tristate Behavior

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

entity ts_example is
  port ( inbus_a, inbus_b, inbus_c : in std_logic;
        en_a, en_b, en_c : in std_logic;
        common : in std_logic;
        outbus : out std_logic );
end ts_example;

architecture exemplar of ts_example is
  -- Internal tristate buffer.
  component tbuf
    port ( i, t : in std_logic; o : out std_logic );
  end component;

  signal int_bus : std_logic;

begin
  -- RTL description
  int_bus <= inbus_a when en_a = '1' else 'z';
  int_bus <= inbus_b when en_b = '1' else 'z';

  -- Component instantiation
  u0 : tbuf port map ( i => inbus_c, t => en_c, o => int_bus );
  outbus <= common XOR int_bus;
end exemplar;
```
Example 4b. Verilog Example Inferring an Internal Tristate

```verilog
module ts_example ( inbus_a, inbus_b, inbus_c, en_a, en_b, en_c, common, outbus );

input inbus_a, inbus_b, inbus_c;
input en_a, en_b, en_c;
input common;
output outbus;

wire int_bus;

assign int_bus = ( en_a == 1 ) ? inbus_a : l’bz;
assign int_bus = ( en_b == 1 ) ? inbus_b : l’bz;

buft u0 ( .i(inbus_c), .t(en_c), .o(int_bus) );
assign outbus = common ^ int_bus;

endmodule
```
Figure 14-9 and Figure 14-10 are the circuits generated by LeonardoSpectrum.

**Figure 14-9** \texttt{tristate\_map} = FALSE (default) - \textbf{Level 3}

**Figure 14-10** \texttt{tristate\_map} = TRUE - \textbf{Level 3}

**Using Clock Enable**

All DFFs in the Xilinx CLBs have a clock enable input. This feature can be used when using schematic entry. LeonardoSpectrum also allows designers using language-based design methods to use this logic. Since the clock enable is a dedicated input to the Xilinx CLB, the implementation of enabled DFFs can save significant area and setup.
VHDL code must be carefully written in order to achieve enabled DFFs. This is shown in the following code fragment:

```vhdl
if (reset = '1') then
    counter <= B"0000";
else
    if (enable = '1') then
        counter <= counter + B"0001";
    else
        counter <= counter;
    end if;
end if;
```

This does not describe an enable flip-flop because counter is reset regardless of enable. To achieve the enabling function, you should rewrite the code fragment as follows:

```vhdl
if (enable = '1') then
    if (reset = '1') then
        counter <= B"0000";
    else
        counter <= counter + B"0001";
    end if;
else
    counter <= counter;
end if;
```

This yields the expected enabled D flip-flop with synchronous reset.

Clock enable recognition is not limited to behavioral HDL descriptions. LeonardoSpectrum also recognizes clock enable functionality for D-type flip-flops from XNF netlist descriptions. This is important when migrating Xilinx designs to other technologies, including gate arrays. If an enabled DFF has been used in the original design, LeonardoSpectrum recognizes the logic and implements it in the target technology, if possible.
Using RAMs

Types of Inferencing RAMs
LeonardoSpectrum supports two types of RAMs:

- RAM_DQ. RAM_DQ is a single-port RAM with separate input and output data lines.
- RAM_IO. RAM_IO is a single-port RAM with bidirectional data lines.

Both of these RAM types support synchronous or asynchronous read and write. These RAMs are automatically inferred by LeonardoSpectrum from HDL code (VHDL or Verilog). The inferencing process distinguishes between RAMs that perform the read operation with an address clocked or not clocked by the write clock (read address clocked). Both of the following VHDL examples perform synchronous writes (inclock) and synchronous reads (outclock); LeonardoSpectrum recognizes these VHDL processes as RAMs:

- The first entity `ram_example1`, is when the read operation does not have a clocked address.
- The second entity `ram_example2`, is when the read operation does have a clocked address.
The first, entity `ram_example1`, is when the read operation does not have a clocked address.
The second entity ram_example2 is when the read operation does have a clocked address.

**Xilinx 4000/E Modgen Support for RAMs**

The Xilinx 4000 Modgen Library supports asynchronous RAMs and synchronous RAMs that clock the read address with the write clock. The Xilinx 4000E Modgen Library adds support for synchronous RAMs that do not clock the read address with the write clock.
RAM Example:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library exemplar;
use exemplar.exemplar_1164.all;

package my_pkg is
  type MEM_WORD is array (6 downto 0) of std_logic_vector (1 downto 0);
end my_pkg;

library exemplar;
use exemplar.exemplar.all;
use work.my_pkg.all;
entity mem is
  port (dio : inout std_logic_vector (1 downto 0);
        meme, we, inclk, outclk : in bit;
        addr : integer range 6 downto 0;
        ro : out bit);
  attribute clock_node : boolean;
attribute clock_node of inclk : signal is TRUE;
attribute clock_node of outclk : signal is TRUE;
end mem;

architecture junk of mem is
  signal mem : MEM_WORD;
  signal d_int : std_logic_vector (1 downto 0);
beg
process (inclk)
begin
  if (inclk'event and inclk = '1') then
    if (meme = '1' and we = '1') then
      mem(addr) <= dio;
    end if;
  end if;
end process;
process (outclk)
begin
  if (outclk'event and outclk = '1') then
    d_int <= mem (addr);
  end if;
end process;
dio <= d_int when (meme = '1' and we = '0') else "ZZ";
end junk;
```
When running this example targeting XC4000/E LeonardoSpectrum will infer a synchronous *ram_io* cell.

**RAM Instantiation**

You can also use Xilinx memgen utility to generate RAMs and instantiate them in the input HDL design.

To take advantage of memgen, you should define the memory in your design as a black box with appropriate ports, and run Leonardo to synthesize the output XNF netlist. The XNF netlist will include the black box for the RAM.

Next, run memgen to create the XNF description of the memory itself. For example, the following memgen invocation will create a 256x6 RAM and write it to a file *myram.xnf*:

```
% memgen type=ram memory_depth=256 word_width=6 file_name=myram
```

The Xilinx PPR and XNFMERGE tools will look in their default search path (including the current directory) to find the file *myram.xnf* and link it into the original design.

The following two example designs in VHDL and Verilog treat the component *myram* as a black box.
Example 5a. VHDL RAM Usage Example

```vhdl
library exemplar;
use exemplar.exemplar_1164.all;
library ieee;
use ieee.std_logic_1164.all;
entity ram_example is
  port(
    addr : in  bit_vector (7 downto 0);
    din  : in  bit_vector (5 downto 0);
    we   : in  bit;
    o    : out bit_vector (5 downto 0)
  );
end ram_example;

architecture eXemplar of ram_example is
  component myram
    port ( a: in bit_vector (7 downto 0);
           d: in bit_vector (5 downto 0);
           we: in bit;
           o: out bit_vector (5 downto 0))
  end component;
begin
  g1: myram port map (a=>addr, we=>we, d=>din, o=>o);
end eXemplar;
```
Example 5b. Verilog RAM Usage Example

```verilog
module ram_example ( addr, din, we, o);
    input [0:7] addr;
    input [0:5] din;
    input we;
    output [0:5] o;

    myram i1(.a(addr), .d(din), .we(we), .o(o));
endmodule

// Define the interface of the black box myram.
module myram (a, d, we, o);
    input [0:7] a;
    input [0:5] d;
    input we;
    output [0:5] o;
// To generate file myram.xnf run:
// memgen type=ram memory_depth=256 word_width=6 file_name=myram
endmodule
```
You can also instantiate synchronous single and dual-port RAMs in the input VHDL or Verilog description as shown in the examples below.

**Example 6a:**

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY exemplar;
USE exemplar.exemplar_1164.ALL;

ENTITY ramtest IS
  PORT ( a, dpra : IN std_logic_vector (3 downto 0);
         d, we, wclk : IN std_logic;
  END ramtest;

ARCHITECTURE exemplar of ramtest is
  COMPONENT RAM16x1D
    PORT (a3,a2,a1,a0,dpra3,dpra2,dpra1,dpra0,we,wclk,d:IN
          std_logic;
          spo, dpo : OUT std_logic );
  END COMPONENT;

  ATTRIBUTE init: string;
  ATTRIBUTE init OF ram16: LABEL IS "0000";
  BEGIN
    ram16: RAM16x1D PORT MAP (a(3), a(2), a(1), a(0), dpra(3),
                               dpra(2), dpra(1), dpra(0), we, wclk, d, spo, dpo);
END exemplar;
```
The XNF Output for Example 6a.

```
SYM, RAM16, RAMD, SCHNM=RAM16x1D, LIBVER=2.0.0, INIT=0000
PIN, A0, I, A<0> _int, ,
PIN, A1, I, A<1> _int, ,
PIN, A2, I, A<2> _int, ,
PIN, A3, I, A<3> _int, ,
PIN, WE, I, WE_int, ,
PIN, D, I, D_int, ,
PIN, WCLK, I, WCLK_int, ,
PIN, DPRA0, I, DPRA<0> _int, ,
PIN, DPRA1, I, DPRA<1> _int, ,
PIN, DPRA2, I, DPRA<2> _int, ,
PIN, DPRA3, I, DPRA<3> _int, ,
PIN, SPO, O, SPO_int, 7.800000,
PIN, DPO, O, DPO_int, 7.800000,
END
```

Example 6b:

```
module ram_example ( addr, din, we, wclk, o);
    input [0:3] addr;
    input [0:1] din;
    input we, wclk;
    output [0:1] o;

    myrams I1 (.a3 (addr[3]), .a2(addr[2]), .a1 (addr[1]),
    .a0 (addr[0]), .d1 (din[1]), .d0 (din[0]), .we(we),
    .wclk(wclk), .o1 (o[1]), .o0 (o[0]));
endmodule

// Define the interface of the black box myram.
module myrams ( a3, a2, a1, a0, d1, d0, we, wclk, o1, o0);
    input a3, a2, a1, a0;
    input d1, d0;
    input we, wclk;
    output o1, o0;
// To generate file myrams.xnf run:
// memgen type=sync_ram_ memory_depth=16 word_width=2
file_name=myrams
endmodule
```
The XNF Output for Example 6b:

```
USER, AREA, 0
USER, PACKED_ESDTIMAGE, 0
USER, DELAY, 2.500000
USER, LEVELS, 0
EXT, din<1>, I,
EXT, din<0>, I,
EXT, addr<3>, I,
EXT, addr<2>, I,
EXT, addr<1>, I,
EXT, addr<0>, I,
EXT, wclk, I,
EXT, we, I,
EXT, o<1>, O,
EXT, o<0>, O
SYM. I1, myrams
PIN, wclk, I, wclk_int, 
PIN, we, I, we_int, 
PIN, d0, I, din<0>_int, 
PIN, d1, I, din<1>_int, 
PIN, a0, I, addr<0>_int, 
PIN, a1, I, addr<1>_int, 
PIN, a2, I, addr<2>_int, 
PIN, a3, I, addr<3>_int, 
PIN, o0, O, o<0>_int, 
PIN, o1, O, o<1>_int, 
```

Using ROMs

The Xilinx XC4000/E/EX/XL support ROMs as primitives. Two sizes are supported: ROM16X1 and ROM32X1 as described in the XACT Libraries Guide. These can be instantiated for use in a LeonardoSpectrum design.

An INIT attribute is required for the ROM component to define its memory pattern. The value must be a 4-digit or 8-digit hexadecimal number, depending on the size of the ROM.

The following is an example VHDL design using an XC4000 ROM. When synthesizing this example (or any example which directly instantiates a component from the xi4 cell library) be sure to first load the xi4 technology library into LeonardoSpectrum.
Example 7. VHDL Example Using ROMs

Example LIBRARY ieee;
use ieee.std_logic_1164.all;
library exemplar;
use exemplar.exemplar_1164.all;
entity rom_example is
  port ( a1 : in std_logic_vector (4 downto 0);
         x1 : out std_logic;
         a2 : in std_logic_vector (3 downto 0);
         x2 : out std_logic);
end rom_example;
architecture exemplar of rom_example is
  component rom
    port ( a4, a3, a2, a1, a0 : in std_logic;
           o : out std_logic);
  end component;
  attribute init: string;
  -- Initial value for 32 bits = 8 hex digits
  attribute init of rom32: label is "6789ABCD";
  -- Initial value for 16 bits = 4 hex digits
  attribute init of rom16: label is "FEDC";
  constant one : std_logic := '1';
begin
  rom32: rom port map (a1(4), a1(3), a1(2), a1(1), a1(0), x1);
  -- 16x1 ROM is instantiated with high address bit inactive.
  -- Downstream tools will optimize down to 16x1 ROM
  rom16: rom PORT MAP (one, a2(3), a2(2), a2(1), a2(0), x2);
end exemplar;

Using Global Clock Buffers

LeonardoSpectrum supports both automatic mapping of clock signals to clock buffers as well as manual mapping of clocks. Both of these are discussed as follows:

Automatic Mapping of Clocks

LeonardoSpectrum automatically maps primary clocks and other global signals, like set/reset and clock enable, to global buffer BUFG. While doing global buffer mapping, the following architectural constraints are taken into account:

- For XC3000 family only two global buffers (GCLK, ACLK) are available.
- XC4000/E: eight global buffers are available (four BUFGP and four BUFGS). Xilinx recommends using only four of the eight unless you are doing floor planning.
- XC4000EX/XL: eight global buffers are available. LeonardoSpectrum uses BUFG for mapping to global buffers.
- XC5200: four global buffers are available.

LeonardoSpectrum writes all global buffers as BUFG in the XNF output and lets Xilinx place and route tools translate it into appropriate global buffer for the targeted technology. For example:
- ACLK/GCLK for XC3000
- BUFGS/BUFGP for XC4000/E
- BUFGLS/BUFGE/BUFFCLK for XC4000EX/XL

If the number of primary clock/global signals in the design exceeds the number of global buffers available in the target technology, then global buffers will be assigned in decreasing order of criticality of clock/global signal.

Example:

```verbatim
class entity test is
  port (clk: bit;
        Din: integer range 0 to 65535;
        Q: buffer integer range 0 to 65535);
end test;

architecture behavior of test is
begin
  process (clk)
  begin
    if (clk='1' and clk'event) then -- Clock (edge triggered)
      Q <= Din;
    end if;
  end process;
end behavior;
```
The xnf output for this will be:

| SYM, XMPLR_INST_45, BUFG, LIBVER=2.0.0 |
| PIN, I, I, clk, |
| PIN, O, O, clk_int, 6.599999 |
| END |
| SYM, XMPLR_INST_37_01, OUTFF, INIT=R, LIBVER=2.0.0, SCHNM=OFD |
| PIN, D, I, din<0>_int, |
| PIN, C, I, clk_int, |
| PIN, Q, O, q<0>, 7.000000 |
| END |

**Note** – Only primary clock/global signals will be automatically mapped to global buffers. For internal clocks do manual assignment as described in next section.

**Manual Mapping of Clocks**

You can map primary signals as well as internal signals to global clock buffers manually by using the `BUFFER_SIG` command. For example, for the VHDL global buffer design you can specify:

```
BUFFER_SIG BUFGP clk
```

You can also specify a `BUFFER_SIG` attribute in the source VHDL to manually assign global buffers.

```vhdl
entity test is
port(clk: bit);
  Din: integer range 0 to 65535;
  Q: buffer integer range 0 to 65535);
attribute BUFFER_SIG of clk: signal is "BUFGP";
end test;
```
Automatic global buffer mapping will not affect any signal that is already mapped manually. The XNF output is:

```
SYM, XMPLR_INST_45, BUFGP, LIBVER=2.0.0
PIN, I, I, clk,
PIN, O, O, manual_clk, 6.599999
END

SYM, XMPLR_INST_37_01, OUTFF, INIT=R, LIBVER=2.0.0, SCHNM=OFD
PIN, D, I, din<0>_int,
PIN, C, I, manual_clk,
PIN, Q, O, q<0>, 7.000000
END
```

**Internal and Input Buffers**

Xilinx allows BUFG(PS) as internal as well as an input buffer. LeonardoSpectrum allows using these buffers as internal and input buffers on the same signal. For every input buffer like BUFG, there is a corresponding internal buffer called BUFG_INT. The modeling of BUFG(PS) has been changed so you can access them using PAD command. The following is a list of commands that can be used for duality of global buffers:

<table>
<thead>
<tr>
<th>Input Buffer</th>
<th>Internal Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFG</td>
<td>BUFG_INT</td>
</tr>
<tr>
<td>BUFGS</td>
<td>BUFGS_INT</td>
</tr>
<tr>
<td>BUFGP</td>
<td>BUFGP_INT</td>
</tr>
</tbody>
</table>

- Example 1: Use BUFG as an internal buffer driven by an IBUF. Commands used:
  
  BUFFER_SIG BUFGS_INT <signal name>
  PAD IBUF <signal name>

- Example 2: Use BUFGS as an internal buffer driven by an BUFGP. Commands used:
  
  BUFFER_SIG BUFGS_INT <signal name>
  PAD BUFGP <signal name>

- Example 3: Use BUFGS as an input buffer only. Commands used:
  
  PAD BUFGS <signal name>

**Note:** Enter commands as a file in the Constraint Editor. **Level 3** can also enter commands on the interactive command line shell.
User Options for Global Buffers

LeonardoSpectrum tries to map to global buffers whenever possible by default (true). However, you have the option to switch off automatic global buffer mapping on the GUI or with options. LeonardoSpectrum maps to global buffers when possible by default. Refer to Table 14-6.

Table 14-6. Global Buffers Output Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>□ Global Buffers</td>
<td>on</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set insert_global_bufs false</td>
<td>-nomap_global_bufs</td>
</tr>
</tbody>
</table>

If □ Global Buffers is not selected, then the option is disabled.

![Global Buffers GUI Interface](image_url)
Design I/O

LeonardoSpectrum maps design I/Os to I/O buffer cells (IBUF, OBUF, etc.) and registered I/O cells (OUTFF, INFF, etc.) by default. You can also assign I/O cells manually by either instantiating the cells in the input RTL description or by using BUFFER_SIG and PAD commands.

Complex I/O Mapping

Xilinx XC4000 architecture contains flip-flops in I/O blocks. LeonardoSpectrum maps to these registered I/Os when possible. Example VHDL Design:

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

entity outfft is
    port (
        clk, en : in std_logic;
        in1, in2 : in std_logic;
        o : out std_logic
    );
end outfft;

architecture exemplar of outfft is
signal sig1, sig2: std_logic;

begin
    process (clk)
    begin
        if (clk'event and clk = '1') then
            sig1 <= sig2;
        end if;
    end process;

    sig2 <= in1 and in2;
    o <= sig1 when en = '0' else 'Z';

end exemplar;
```
LeonardoSpectrum XNF output netlist for this design is:

```
SYM, i8, OUTFFT,SCHNM=OFDT,LIBVER=2.0.0,INIT=R
PIN, C, I, CLK_int, ,
PIN, D, I, n50, ,
PIN, T, I, EN_int, ,
PIN, O, O, O, 6.500000,
END
  :
  :
SYM, i51, AND,SCHNM=AND2,LIBVER=2.0.0
PIN, I0, I, IN2_int, ,
PIN, I1, I, IN1_int, ,
PIN, O, O, n50, 4.000000,
```

**Note:** EDIF is the preferred Xilinx file.
IOB in Xilinx XC4000E/EX/XL architecture is enhanced and contains flip-flops with clock-enable. For XC4000E architecture, LeonardoSpectrum maps to IOB flip-flops with clock-enable whenever possible.

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

entity outfftx is
   port (clk, en, ce: in std_logic;
in1, in2 : in std_logic;
o : out std_logic);
end outfftx;

architecture exemplar of outfftx is
signal sig1, sig2: std_logic;

begin
   process (clk)
   begin
      if (clk’event and clk = ‘1’) then
         if (ce = ‘1’) then
            sig1 <= sig2;
         end if;
      end if;
   end process;

   sig2 <= in1 and in2;
o <= sig1 when en = ‘0’ else ‘Z’;
end exemplar;
```
LeonardoSpectrum output for this design is:

```
SYM, i8, OUTFFT,SCHNM=OFDTX,LIBVER=2.0.0,INIT=R
PIN, C, I, CLK_int, ,
PIN, D, I, n51, ,
PIN, CE, I, CE_int, ,
PIN, T, I, EN_int, ,
PIN, O, O, O, 6.500000,
END
SYM, i51, AND,SCHNM=AND2,LIBVER=2.0.0
PIN, I0, I, IN2_int, ,
PIN, I1, I, IN1_int, ,
PIN, O, O, n51, 4.000000,
```

*Pin Location Assignment*

You can assign pin locations from:

- VHDL attributes
- **Level 3** Interactive Command Line Shell

You can specify IOB locations for the external pins in the design. When using VHDL, you can use attributes `pin_number` for individual signals and `array_pin_number` for busses. These attributes will be translated into LOC attributes in the output XNF file for further processing by Xilinx place and route software.

*VHDL Attributes*

The VHDL attributes `pin_number` and `array_pin_number` are declared in the EXEMPLAR and EXEMPLAR_1164 packages. You can access these declarations by making one of these packages visible with these context clauses before entity declaration:

```
library exemplar; use EXEMPLAR.EXEMPLAR_1164.all;
```

If the exemplar or exemplar_1164 package is not used, then define the attributes `pin_number` and `array_pin_number` and the type `string_array`, as in the following example.
library ieee;
use ieee.std_logic_1164.all;

entity pin_loc_example is
  port (  
    CLK: bit;
    DIN: in std_logic_vector (4 downto 3);
    Q : out std_logic_vector (4 downto 3)
  );
type string_array is
    array (natural range <>, natural range <>) of character;
  -- these attributes are defined in exemplar and exemplar_1164 packages.
  -- there is no need to define them here if you have a use clause
  -- for the package.
  attribute pin_number : string;
  attribute array_pin_number : string_array;
  attribute pin_number of clk : signal is "P1";
  attribute array_pin_number of din : signal is ("P2", "P3", "P4", "P5", "P6");
end pin_loc_example;

architecture exemplar of pin_loc_example is
begin
  process (CLK)
  begin
    if (CLK = '1' and CLK'EVENT) then
      q <= DIN;
    end if;
  end process;
end EXEMPLAR;

Constraints and attributes can be set with the LeonardoSpectrum Constraint File Editor.

Pin placement can be specified in the constraint file. This allows separating pin information from the VHDL or Verilog description.

Note – From the constraint file there is no equivalent to the array_pin_number attribute. Pin locations for bussed signals need to be expressed bit-by-bit.
Interactive Command Line Shell

Use the interactive command line shell if you wish to keep pin information separate from the VHDL description, or if you are using Verilog.

Note – The interactive command line shell does not currently support busses. This means that there is no equivalent to the `array_pin_number` attribute; pin locations for bussed signals must be expressed bit-by-bit.

Note – Names with parentheses must be enclosed in "" quotes. Parentheses () are special characters in the control file.

Example:

```
pin_number P10 "a(0)"
pin_number P11 "a(1)"
pin_number P12 "a(2)"
pin_number P13 "a(3)"
pin_number P14 "a(4)"
pin_number P15 "a(5)"
pin_number P16 "a(6)"
pin_number P17 "a(7)"
pin_number P18 "a(8)"
pin_number P19 "a(9)"
pin_number P20 "a(10)"
```

Reporting

Refer to the LeonardoSpectrum User's guide for information on reporting.

Using Timespecs

Timespecs are Xilinx-defined primitives that guide Xilinx place and route by specifying timing restrictions on signals in the design. LeonardoSpectrum can write timespec annotated XNF netlists.
Specify Timespec Attributes

You can specify the Timespec design on the Level 3 Interactive command line shell, in VHDL attributes, or in batch mode. Use these rules:

- **Determining when timing specifications should be written.** For PPR, the timing constraints will be written as TIMESPECs in an XNF file.

- **Specification of generic timing constraints.** LeonardoSpectrum allows you to define the behavior of clock signals and the arrival time of primary inputs in either a VHDL source file or as commands from the interactive command line shell. The timing constraints will be inferred from these specifications. Clocks are specified by specifying the clock cycle, pulse width and clock offset. All clock information is relative to a single time reference point. Refer to Table 14-7.

Table 14-7. TimeSpec Output Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Available in Families</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate TimeSpec</td>
<td>on</td>
<td>All</td>
<td>Controls creation of TimeSpec information from your constraints.</td>
</tr>
<tr>
<td></td>
<td>off (default)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 14-11 Clock Timing Constraints

All times are specified in nanoseconds. In VHDL, the specification of a clock and an arrival time would be expressed as follows:

```
ATTRIBUTE CLOCK_CYCLE OF clock:SIGNAL is 20ns;
ATTRIBUTE CLOCK_OFFSET OF clock:SIGNAL is 5ns;
ATTRIBUTE PULSE_WIDTH OF clock:SIGNAL is 10ns;
ATTRIBUTE ARRIVAL_TIME OF inputa:SIGNAL is 3ns;
```

From the interactive command line shell or by using the LeonardoSpectrum constraint editor:

```
CLOCK_CYCLE 20 clock
CLOCK_OFFSET 5 clock
PULSE_WIDTH 10 clock
ARRIVAL_TIME 3 inputa
```

**TIMESPECs**

LeonardoSpectrum writes out all TIMESPECs using End-Point specifications. A TIMEGRP symbol is created for each implied group. Two signals belong to the same TIMEGRP if the following conditions are met:

- (Group 1, a) Both signals are input pins with the same arrival time and are the start points of a set of paths which are identical to the Xilinx predefined group PADS; or
• (Group 2, b) Both signals are D inputs of D flip-flops which share a common CLK and are the endpoints of a set of paths; or

• (Group 2, c) Both signals are output pins and are the endpoints of a set of paths; or

• (Group 1, d) Both signals are Q outputs of D flip-flops with a common CLK and are the start points of a set of paths.

If you specify no arrival times for any inputs in the design, all inputs are assumed to arrive at time 0. For each pair of TIMEGRP symbols for which a connecting path exists, LeonardoSpectrum writes out a TIMESPEC of the form

TS\text{n} = \text{FROM:} group_1 : \text{TO:} group_2 = \text{constraint}

If \text{group}_2 \text{ represents a set of D-inputs clocked by CLK, the value of } \text{constraint} \text{ is determined by the formula:}

\text{constraint} = (CLK.\text{period} + CLK.\text{offset} - \text{group}\_d.\text{setup}) - \text{group}_1.\text{arrival}

For example, read the following Timespec VHDL file into LeonardoSpectrum.

\textbf{Note:} This file together with the constraints, writes out TIMESPEC as shown in the XNF output. The constraint file determines your requirements, which are clock cycle, arrival time on primary inputs and required times on primary outputs.
library exemplar;
use exemplar.exemplar_1164.all;
library ieee;
use ieee.std_logic_1164.all;

-- traffic controller designed using Boolean

ENTITY traffic IS
  PORT (clock, sensor1, sensor2 : IN STD_LOGIC;
        red1, yellow1, green1, red2, yellow2, green2 : OUT STD_LOGIC);
END traffic;

ARCHITECTURE eXemplar OF traffic IS
  SIGNAL state, nxstate : STD_LOGIC_VECTOR (0 TO 2) := O"0";
BEGIN
  dff_v (nxstate, clock, state);
  red1 <= '1' WHEN state = O"4" OR
          state = O"5" OR
          state = O"6" OR
          state = O"7" ELSE '0';
  yellow1 <= '1' WHEN state = O"3" ELSE '0';
  green1 <= '1' WHEN state = O"0" OR
            state = O"1" OR
            state = O"2" ELSE '0';
  red2 <= '1' WHEN state = O"0" OR
         state = O"1" OR
         state = O"2" OR
         state = O"3" ELSE '0';
  yellow2 <= '1' WHEN state = O"7" ELSE '0';
  green2 <= '1' WHEN state = O"4" OR
            state = O"5" OR
            state = O"6" ELSE '0';
  nxstate <= O"1" WHEN state = O"0" AND sensor2 = sensor1 ELSE
               O"2" WHEN state = O"0" AND sensor1 = '0' AND sensor2 = '1' ELSE
               O"0" WHEN state = O"0" AND sensor1 = '1' AND sensor2 = '0' ELSE
               O"2" WHEN state = O"1" ELSE
               O"3" WHEN state = O"2" ELSE
               O"4" WHEN state = O"3" ELSE
               O"5" WHEN state = O"4" AND sensor1 = '0' AND sensor2 = '0' ELSE
               O"6" WHEN state = O"4" AND sensor1 = '1' AND sensor2 = '0' ELSE
               O"5" WHEN state = O"4" ELSE
               O"6" WHEN state = O"5" ELSE
               O"7" WHEN state = O"6" ELSE
               O"0";
END eXemplar;

CLOCK_CYCLE 18 clock
PULSE_WIDTH 9 clock
CLOCK_OFFSET 0 clock
ARRIVAL_TIME 2 SENSOR1 SENSOR2
REQUIRED_TIME 16 red1 yellow1 green1 red2 yellow2 green2
Only the relevant portion of the XNF output with TimeSpec is shown:

```plaintext
LCANET, 5
USER, LIBRARY_NAME, work
USER, CELL_NAME, traffic
USER, VIEW_NAME, exemplar
EXT, clock, I, , TNM=GROUP_1
EXT, sensor1, I, , TNM=GROUP_2
EXT, sensor2, I, , TNM=GROUP_2
EXT, yellow1, O, , FAST, TNM=GROUP_3
EXT, green1, O, , FAST, TNM=GROUP_3
EXT, red2, O, , FAST, TNM=GROUP_3
EXT, yellow2, O, , FAST, TNM=GROUP_3
EXT, green2, O, , FAST, TNM=GROUP_3
...
SYM, state<0>, DFF, TNM=GROUP_0, INIT=R, LIBVER=2.0.0, SCHNM=FDCE
PIN, C, I, clock_int,
PIN, Q, O, state<0>, 1.700000
PIN, D, I, nxstate<0>,
END
SYM, state<1>, DFF, TNM=GROUP_0, INIT=R, LIBVER=2.0.0, SCHNM=FDCE
PIN, C, I, clock_int,
PIN, Q, O, state<1>, 1.700000
PIN, D, I, nxstate<1>,
END
SYM, state<2>, DFF, TNM=GROUP_0, INIT=R, LIBVER=2.0.0, SCHNM=FDCE
PIN, C, I, clock_int,
PIN, Q, O, state<2>, 1.700000
PIN, D, I, nxstate<2>,
END
SYM, ix389, TIMESPEC, TS0=FROM:GROUP_1:TO:GROUP_0=17.70ns, LIBVER=2.0.0
END
SYM, ix390, TIMESPEC, TS1=FROM:GROUP_1:TO:GROUP_3=16.00ns, LIBVER=2.0.0
END
SYM, ix391, TIMESPEC, TS2=FROM:GROUP_2:TO:GROUP_0=15.70ns, LIBVER=2.0.0
END
SYM, ix392, TIMESPEC, TS3=FROM:GROUP_2:TO:GROUP_3=14.00ns, LIBVER=2.0.0
END
SYM, ix393, TIMESPEC, TS4=FROM:GROUP_0:TO:GROUP_0=17.70ns, LIBVER=2.0.0
END
SYM, ix394, TIMESPEC, TS5=FROM:GROUP_0:TO:GROUP_3=16.00ns, LIBVER=2.0.0
END
```

Read the following TimeSpec VHDL example file into LeonardoSpectrum:

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

library exemplar;
use work.exemplar_1164.all;

entity dff_reg is
  port (
    a: in std_logic_vector (1 downto 0);
    clk: in bit;
    o : out std_logic_vector (1 downto 0)
  );
end dff_reg;

architecture exemplar of dff_reg is
begin
  process (clk, a)
  begin
    if (clk'event and clk = '1') then
      o <= a;
    end if;
  end process;
end exemplar;
```

Then, optimize the design and execute the commands:

```vhdl
CLOCK_CYCLE 20 clk
generate_timespec
```
The following XNF Timespec file is generated:

<table>
<thead>
<tr>
<th>LCANET, 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER, LIBRARY_NAME, work</td>
</tr>
<tr>
<td>USER, CELL_NAME, dff_reg</td>
</tr>
<tr>
<td>USER, VIEW_NAME, exemplar</td>
</tr>
<tr>
<td>PROG, Leonardo, V4.0.0b (Beta), &quot;Fri Jul 26 16:06:00 1996&quot;</td>
</tr>
<tr>
<td>EXT, a&lt;1&gt;, I, , TNM=GROUP_1</td>
</tr>
<tr>
<td>EXT, a&lt;0&gt;, I, , TNM=GROUP_1</td>
</tr>
<tr>
<td>EXT, clk, I, , TNM=GROUP_1</td>
</tr>
<tr>
<td>EXT, o&lt;1&gt;, O,</td>
</tr>
<tr>
<td>EXT, o&lt;0&gt;, O,</td>
</tr>
<tr>
<td>SYM, XMPLR_INST_3, IBUF, LIBVER=2.0.0, SCHNM=IBUF</td>
</tr>
<tr>
<td>PIN, I, I, a&lt;0&gt;,</td>
</tr>
<tr>
<td>PIN, O, O, a&lt;0&gt;_int, 3.000000</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>SYM, XMPLR_INST_4, IBUF, LIBVER=2.0.0, SCHNM=IBUF</td>
</tr>
<tr>
<td>PIN, I, I, a&lt;1&gt;,</td>
</tr>
<tr>
<td>PIN, O, O, a&lt;1&gt;_int, 3.000000</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>SYM, XMPLR_INST_9_O1, OUTFF, TNM=GROUP_0, INIT=R, LIBVER=2.0.0, SCHNM=OFD</td>
</tr>
<tr>
<td>PIN, D, I, a&lt;0&gt;_int,</td>
</tr>
<tr>
<td>PIN, C, I, clk_int,</td>
</tr>
<tr>
<td>PIN, Q, O, o&lt;0&gt;, 7.000000</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>SYM, XMPLR_INST_7_O1, OUTFF, TNM=GROUP_0, INIT=R, LIBVER=2.0.0, SCHNM=OFD</td>
</tr>
<tr>
<td>PIN, D, I, a&lt;1&gt;_int,</td>
</tr>
<tr>
<td>PIN, C, I, clk_int,</td>
</tr>
<tr>
<td>PIN, Q, O, o&lt;1&gt;, 7.000000</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>SYM, XMPLR_INST_2, BUFG, LIBVER=2.0.0</td>
</tr>
<tr>
<td>PIN, I, I, clk,</td>
</tr>
<tr>
<td>PIN, O, O, clk_int, 6.599999</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>SYM, XMPLR_INST_0, TIMESPEC, TS0=FROM:GROUP_1:TO:GROUP_0=14.00ns, LIBVER=2.0.0</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>SYM, XMPLR_INST_1, TIMESPEC, TS1=FROM:GROUP_0:TO:GROUP_0=14.00ns, LIBVER=2.0.0</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>EOF</td>
</tr>
</tbody>
</table>

**Using Xilinx Attributes**

Xilinx documentation describes various attributes and constraints which can be attached to signals or instances in an XNF netlist to guide implementation by place and route tools.

These attributes can be attached from VHDL with the use of VHDL attributes or from the interactive command line shell.
An example in VHDL, attaching the FAST attribute to an output net RED1:

```vhdl
attribute fast: boolean;
attribute fast of red1: SIGNAL is true;
```

An example for setting attributes from the interactive command line shell:

```
set_attribute -port red1 -name fast -value TRUE
```

Some examples of attributes of I/O ports at the top level of design:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>MEDFAST</td>
</tr>
<tr>
<td>BLKNM</td>
<td>MEDSLOW</td>
</tr>
<tr>
<td>HBLKNM</td>
<td>NODELAY</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>RES</td>
</tr>
<tr>
<td>FAST</td>
<td>CAP</td>
</tr>
<tr>
<td>SLOW</td>
<td>FREQ</td>
</tr>
<tr>
<td>PERIOD</td>
<td>THI</td>
</tr>
<tr>
<td>TLO</td>
<td></td>
</tr>
</tbody>
</table>

Examples of attributes of instantiated components:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>CAP</td>
</tr>
<tr>
<td>RLOC</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>USE_RLOC</td>
<td>FILE</td>
</tr>
<tr>
<td>U_SET</td>
<td>MAP</td>
</tr>
<tr>
<td>HU_SET</td>
<td>DEF</td>
</tr>
<tr>
<td>RLOC_ORIGIN</td>
<td>DECODE</td>
</tr>
<tr>
<td>RLOC_RANGE</td>
<td>CYMODE</td>
</tr>
<tr>
<td>BLKNM</td>
<td>SCHNM</td>
</tr>
<tr>
<td>HBLKNM</td>
<td>LIBVER</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>SYSTEM</td>
</tr>
<tr>
<td>MEDFAST</td>
<td>CONFIG</td>
</tr>
<tr>
<td>MEDSLOW</td>
<td>BASE</td>
</tr>
<tr>
<td>FAST</td>
<td>ASYNC_VAL</td>
</tr>
<tr>
<td>SLOW</td>
<td>SYNC_VAL</td>
</tr>
<tr>
<td>NODELAY</td>
<td>FLOAT_VAL</td>
</tr>
<tr>
<td>TTL</td>
<td>DUTY_CYCLE</td>
</tr>
<tr>
<td>CMOS</td>
<td>MEMFILE</td>
</tr>
<tr>
<td>RES</td>
<td>DEPTH</td>
</tr>
</tbody>
</table>
Additional Xilinx Specific Options

Using Enabled D Type Flip-Flops

LeonardoSpectrum maps to enabled D-type flip-flops to Xilinx technology.

Pipelined Multiplier

Refer to the User’s guide, Chapter 10.

Use of Registered Logic in IOBs

LeonardoSpectrum then maps I/O to simple input buffers, output buffers, and bidirectional buffers.

Table 14-8. I/O Registers Output Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O Registers</td>
<td>on</td>
<td>set complex_ios true</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>set complex_ios false</td>
<td>-nocomplex_ios</td>
</tr>
</tbody>
</table>

Controls mapping to complex I/O gates. If I/O Registers is not selected, then the use of complex I/O primitives (INLAT, INFF, OUTFF) is inhibited when targeting Xilinx technologies.
Part Number

Table 14-9. Part Number Options

<table>
<thead>
<tr>
<th>Technology Settings</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part Number</td>
<td>set part &lt;part_number&gt;</td>
<td>-part=&lt;part_number&gt;</td>
</tr>
</tbody>
</table>

LeonardoSpectrum writes part number in output netlist. Use default part number on GUI or select from pulldown.

Specifying the part number when targeting a Xilinx FPGA does not affect the output optimization. The target library is the only criteria used by LeonardoSpectrum to determine which algorithms to use. However, the part number must be in the XNF netlist before submission to the Xilinx place and route tools.

The part variable is set by inserting the part number in the top of the output XNF netlist file. If you are unsure of the target part, due to uncertainties in the design (size, or number of I/O pins, for example), run LeonardoSpectrum without specifying the part. When satisfied with the result from LeonardoSpectrum, set part variable and then write XNF.

Writing XNF and EDIF

You can control the flavor or XNF or EDIF outputs with the following options:

- By default the design will be written out in terms of AND/OR gates in output EDIF and EQN symbols in XNFs output. The ![Write EQN Symbols (XNF only)](image) option is selected by default. If this option is selected then LeonardoSpectrum is directed to write EQN symbols when writing XNF.

- By default, LeonardoSpectrum packs function generators, but does not write out the CLB packing information in the output netlist. Refer to Table 14-10.

Notes Level 3: You can control the flavor of the XNF output through the following variables and commands:

- By default LeonardoSpectrum will write hierarchical XNF, i.e., one XNF file for each sub-hierarchy in the design. If you want to write an flat XNF file then flatten the design first (using command ungrou -all -hier) before writing out XNF file. You can also selectively flatten the design for writing.

- By default LeonardoSpectrum does not write out CLB packing information in the output file. To enable writing the CLB, set the LeonardoSpectrum variable set write_clb_packing true.
Table 14-10. Results of Output Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>☐ Write EQN Symbols</td>
<td>off</td>
<td>set write_lut_binding false</td>
<td>default</td>
</tr>
<tr>
<td>(XNF only)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>☑ Write EQN Symbols</td>
<td>on</td>
<td>default</td>
<td>-nowrite_eqn</td>
</tr>
<tr>
<td>Prohibits use of AND/OR when writing XNF only.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram showing synthesis settings](image_url)

---

*Xilinx LCA Synthesis*  
14-65
For example, consider the following input design for Level 3:

```
entity mux2 is
  port (A, B, S :in bit;
        Z: Out bit);
end mux2;

architecture test of mux2 is
begin
  Z <= B when (S = '1') else A;
end test;
```

(1) Default xnf output:

```
SYM, z_rename, EQN, EQN=((I1*I2)+(I0*~I2)), LIBVER=2.0.0
PIN, I0, I, a_int,
PIN, I1, I, b_int,
PIN, I2, I, s_int,
PIN, O, O, z_rename,
END
....
SYM, FMAP_0, FMAP, LIBVER=2.0.0
PIN, I1, I, a_int
PIN, I2, I, b_int
PIN, I3, I, s_int
PIN, O, I, z_rename
END (continued...)
```
Using Xilinx Designs as Input

Use the following guidelines to prepare Xilinx designs for input to LeonardoSpectrum.

Design Restrictions

The input design must not have the following elements:

- BSCAN (IO scan device)
- RDBK, RDCLK

```plaintext
(continued...)

(2) Output with 'set write_lut_binding FALSE'

SYM, z_rename, EQN = ((I1*I2)+(I0*~I2)), LIBVER=2.0.0
PIN, I0, I, a_int,
PIN, I1, I, b_int,
PIN, I2, I, s_int,
PIN, O, O, z_rename,
END

... (no FMAP symbol in xnf output)

(3) Output after 'decompose_luts' command
(set write_lut_binding = TRUE).

SYM, z_rename_duplicate_name_0, OR, LIBVER=2.0.0, SCHNM=OR2
PIN, I1, I, XMPLR_NET_8,
PIN, I0, I, XMPLR_NET_1,
PIN, O, O, z_rename, 4.500000
END

SYM, XMPLR_INST_6, AND, LIBVER=2.0.0, SCHNM=AND2B1
PIN, O, O, XMPLR_NET_8, 0.000000
PIN, I0, I, s_int,, INV
PIN, I1, I, a_int,
END

SYM, XMPLR_INST_7, AND, LIBVER=2.0.0, SCHNM=AND2
PIN, O, O, XMPLR_NET_1, 0.000000
PIN, I0, I, s_int,
PIN, I1, I, b_int,
END

SYM, FMAP_0, FMAP, LIBVER=2.0.0
PIN, I1, I, s_int
PIN, I2, I, a_int
PIN, I3, I, b_int
PIN, O, I, z_rename
END
```

Xilinx LCA Synthesis
• STARTUP (configuration functions)
• Special input pads: TDI, TMS, TCK, MD0, MD2
• Special output pads: TDO, MD1
• Special care needs to be taken if your design has the following elements:
  • An on-chip oscillator macro GXTL GOSC (XC3000). LeonardoSpectrum eliminates the oscillator macro, if any, and connects the oscillator output to an input pad.
  • RAM or ROM (XC4000/E). These are exported to the target technology as a “black box” module.

**XNF File Constructs**

Certain XNF file constructs cannot be read by LeonardoSpectrum. For example, CLB records and designs with hard macros are not allowed.

In general, a design taken through the entire Xilinx process will be readable. This involves running the design through place and route, and obtaining an LCA representation of the design. This can then be processed by LCA2XNF which results in a simulatable XNF file.

**Preserving Logic in an XNF Netlist**

When reading a Xilinx XNF netlist, there are times when you may not want to modify some portion of the logic. For example, you may want to use a pre-optimized macro, a hand-crafted CLB, or CLBs along the critical path in the design. Selectively skipping optimization can also reduce run-time and memory requirements while still optimizing control logic connecting together large datapath elements. This section describes methods to accomplish this.

**NOOPT**

You can add a NOOPT parameter to the SYM definition:

```
SYM, symbol_name, symbol_type, =NOOPT, other_parameters
```

The =NOOPT parameter on a symbol directs LeonardoSpectrum to preserve the symbol; =NOOPT is not removed during network optimization.
As an example, consider the following design consisting of three AND symbols:

```
SYM, and1, AND
  PIN, 1, I, b
  PIN, 2, I, a
  PIN, 0, O, and1
END
SYM, and2, AND
  PIN, 1, I, c
  PIN, 2, I, b
  PIN, 0, O, and2
END
SYM, and3, AND
  PIN, 2, I, and2
  PIN, 1, I, and1
  PIN, 0, O, and3
END
```

This network evaluates to

```
SYM, [46], AND
  PIN, 1, I, c
  PIN, 2, I, b
  PIN, 3, I, a
  PIN, 0, O, and3
END
```

If there is an annotated symbol `and2` with the parameter NOOPT,

```
SYM, and2, AND, =NOOPT
  PIN, 1, I, c
  PIN, 2, I, b
  PIN, 0, O, and2
END
```
then the network evaluates to

```
SYM, [59], AND
    PIN, 1, I, and2
    PIN, 2, I, b
    PIN, 3, I, a
    PIN, 0, O, and3
END
SYM, and2, AND, =NOOPT
    PIN, 1, I, c
    PIN, 2, I, b
    PIN, 0, O, and2
END
```

**Level 3 - Global Set/Reset/Tristate**

XNF netlists, after placement and routing, will have primitives with explicit pins for Global Set/Reset (GSR) and Global Tristate (GTS). The `xlx_preserve_gsr` and `xlx_preserve_gts` variables used to preserve global signals when targeting a non-Xilinx technology. The variables are ignored when targeting a Xilinx technology.

```
set xlx_preserve_gsr (default is FALSE)
set xlx_preserve_gts (default is FALSE)
```

When set to TRUE, the Global Set or Reset (gsr) signal is preserved. For instance, assume the following XNF symbol:

```
SYM, flipflop, DFF
    PIN, D, I, din
    PIN, C, I, clk
    PIN, RD, I, res
    PIN, GR, I, GLOBALRESET-
    PIN, Q, O, qout
END
```
Without the `xlx_preserve_gsr` set, this DFF can only be reset with signal `res`. The GR pin is ignored. When `xlx_preserve_gsr` is set, however, the reset pin of the DFF will be driven by the logical or of `res` and `GLOBALRESET-`.

```
xlx_preserve_gts (default is FALSE)
```

Preserves the global tristate (gts) signal, on XC4000/E designs only. For instance, assume the following XNF symbol:

```
SYM, buffertje, OBUF
PIN, I, I, in
    PIN, O, O, out
    PIN, GTS, I, GT-S
END
```

Without the `xlx_preserve_gts` set, this buffer will translate into a regular output buffer in the target technology. When `xlx_preserve_gts` is set, however, the buffer translates into a tristate output buffer.

The `xlx_preserve_gts` and `xlx_preserve_gsr` variables are ignored when the target technology is a Xilinx architecture, because the global connections are automatically added by the place and route software.

**I/Os in XNF**

Inputs and outputs signals can be explicitly declared in three different ways:

```
EXT, signal_name, direction
```

is used to specify signals that go off chip, and can only connect to I/O symbols.

```
SIG, signal_name, S
```

is used to specify signals that should be treated as externals without going off chip. This is useful for specifying partial designs and macros. Inputs and outputs are implicit if they are connected to an I/O symbol (IBUF, OBUF, INLAT, INREG, INFF, OUTFF, OUTFFT, BDBUF), in which case no EXT record is required.
On the XNF Options, the Preserve Dangling Nets option is not selected by default. This option is used to prevent any loadless net from getting swept away. Any sourceless net in XNF netlist is added as primary input port in the design. Consider an input design in which inputs $i_1$ and $i_2$ are not driving any logic and output signal $or$ is sourceless. Refer to Table 14-11. When Preserve Dangling Nets is selected, LeonardoSpectrum issues the following:

Occurs by Default:
- Warning: No source pin on signal ‘$i_1$’, added as primary input
- Warning: No source pin on signal ‘$i_2$’, added as primary input
- Warning: No load pin on signal ‘$or$’, added as primary output

Table 14-11. Input Options for Preserve Dangling Nets

<table>
<thead>
<tr>
<th>XNF Options</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preserve Dangling Nets</td>
<td>on</td>
<td>set preserve_dangling_net true</td>
<td>-preserve_dangling_net</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>default False</td>
<td>default</td>
</tr>
</tbody>
</table>

Controls preserving of dangling nets. For XNF only

Notes: Any signal that does not have load, and is not declared EXT or SIG or connected to an I/O gate, is considered dangling and is optimized away. This is a recursive process: all logic feeding the dangling net is also swept away.
To prevent automatic sweeping of dangling inputs and outputs, you must select the Preserve Dangling Nets option or set the `preserve_dangling_net` variable to TRUE (the default value is FALSE).

`preserve_dangling_net` is applicable only when the input format is XNF, and by default creates ports to unconnected nets when reading XNF file.

Consider an input design in which inputs ‘i1’ and ‘i2’ are not driving any logic and output signal ‘or’ does not have a source.

With `preserve_dangling_net` set to TRUE, Leonardo issues the following messages:

```
Warning: No load pin on signal ‘or’, added as primary output
Warning: No source pin on signal ‘i1’, added as primary input
Warning: No source pin on signal ‘i2’, added as primary input
```

The resulting netlist is:

```
LCANET, 5
  EXT, i1, I
  SIG, i2, S
  SIG, out, S
  SYM, ibuf, IBUF
  PIN, 1, i1
  PIN, 0, 0, ibuf
  END
  SYM, and, AND
  PIN, 1, i1, ibuf
  PIN, 2, i2
  PIN, 0, 0, and
  END
  SYM, or, OR
  PIN, 1, i1, and
  PIN, 2, i2
  PIN, 0, 0, out
  END
```

This variable should be used with care as it may sometimes result in unexpected behavior. For example, if `preserve_dangling_net` is set to FALSE and no nodes are declared to be primary outputs, then the netlist does not have any output and evaluates to an empty netlist.
If, for example, `preserve_dangling_nets` is set to FALSE (the default):

```
LCANET, 5
  EXT, i1, I
  SIG, out1, S
  SYM, ibuf, IBUF
  PIN, I, I, i1
  PIN, O, O, ibuf
END
SYM, and, AND
  PIN, 1, I, ibuf
  PIN, 2, I, i2
  PIN, O, O, and
END
SYM, or, OR
  PIN, 1, I, and
  PIN, 2, I, i2
  PIN, O, O, out1
END
SYM, inv, INV
  PIN, I, I, and
  PIN, O, O, out2
END
```

LeonardoSpectrum adds `out1` as a primary output and `i1` as a primary input because they are declared as such. Signal `out2` has no fanout and is deleted. In turn, this makes symbol `inv` obsolete. Signal `i2` has no source but cannot be deleted because it fans out into symbol. Therefore, this signal is assumed to be a primary input. Therefore, a warning message is displayed:

```
Warning: No source pin on signal ‘i2’, added as primary input
```
The resulting network is as follows:

```plaintext
LCANET, 5
    EXT, i1, I
SIG, i2, S
SIG, out1, S
SYM, ibuf, IBUF
    PIN, I, I, i1
    PIN, O, O, ibuf
END
SYM, and, AND
    PIN, 1, I, ibuf
    PIN, 2, I, i2
    PIN, O, O, and
END
SYM, or, OR
    PIN, 1, I, and
    PIN, 2, I, i2
    PIN, O, O, out1
END
```

If a circuit is imported from another technology or HDL, then all primary inputs/outputs connected to I/O symbols are declared EXT, and all remaining primary inputs/outputs are declared SIG.

**External Signals**

For example, some XNF files that are produced by ViewLogic’s `wir2xnf`, have all EXT records declared B (bidirectional), even if they are used in a I (input), O (output), or T (tristate) fashion only.

LeonardoSpectrum recognizes the implemented direction of EXT records which overrides the declared direction. A warning message is issued if the implemented direction does not match the declared direction.

**Interfacing with XBLOX**

*Note: XBLOX only exists with XACT software.*
An interface into Xilinx XBLOX is provided through module generation. The package, in $EXEMPLAR/data/modgen/xblox.vhd, maps supported operators into XBLOX primitives.

To use XBLOX primitives for arithmetic and relational operators in a VHDL file, specify load-modgen xblox3 or xblox4 or enable XBLOX as the module generator. Specify the source technology to be xi4hm so LeonardoSpectrum can find and preserve the special macros that are needed for a correct XBLOX netlist.

The output format is XNF. The XNF file is used as input to the XBLOX program, which produces an expanded XNF file (.XG file), and hard macro files (.HM files). (Most XBLOX modules generate only an .XG file. Some implementations may use .HM files.) .XG and .HM files are subsequently used in the PPR program for input to partitioning, placement and routing.

Note: XBLOX3 and XBLOX4: reduction operators now use XBLOX components.

Additional Xilinx Architecture Features

These features include:
- Xilinx Coregen Flow
- Modgen Enhancements
- Fast I/O Buffers
- Quality of Area and Delay Results
- Fast Xilinx Outputs
- M1 Back Annotation Flow
- SDF (Standard Delay Format) Writer
- M1 Support
- M2 Support
- Implementing Boundary Scan for Xilinx 4000 FPGAs

Xilinx Coregen Support

The Exemplar / Xilinx Coregen flow should be as follows:

1. Create the desired core using Coregen GUI with vendor Exemplar.
2. Include the core within the desired design hierarchy. For VHDL create a component declaration and instantiation based on the information contained in the vho file. Do NOT read the *.edn into LeonardoSpectrum. The Core module is treated as a black box. The *.edn file is used by ngdbuild during expansion.

3. Synthesize the top level design, write the output EDIF.

4. Pass the top level EDIF through Xilinx tools. Ngdbuild should pick up the core component.

**Modgen Enhancements**

**Non-Square Multipliers**

Non-square multipliers were implemented for the following technologies:

XC4000/E/L/EX/XL/XV and XC5200

**Improved Operators**

The following operators are improved:
- Constant multiplication is improved; constants are swept away.
- Counters are improved; there are many ways to write counters.
- LeonardoSpectrum detects many of the counters.
- Improved >=, etc.
- Improved incrementer/decrementer for rollover conditions.

**FSM Encoding (binary, gray, random, onehot, twohot, auto)**

For auto encoding, in Xilinx 4000/5000 technologies, LeonardoSpectrum varies the encoding based on bit width. Moreover, enumerated types with fewer elements than global integer lower_enum_break are encoded as binary; while larger enumerated types are encoded as onehot. Values larger than global integer upper_enum_break are encoded as binary. The auto default allows LeonardoSpectrum to select encoding on a case by case basis. If LeonardoSpectrum selects onehot for auto encoding your design, then "onehot encoding" is printed in the log file of the design.

The encoding variable determines how LeonardoSpectrum encodes enumerated types. The encoding variable determines how LeonardoSpectrum implements a state machine with a state vector of an enumerated type.
Twohot Encoding

Twohot encoding is now added to FSM encoding (binary, gray, random, onehot, twohot, auto). Twohot sets two flip flops high for each state. The twohot encoding requires more flip flops than binary and fewer flip flops than onehot. Twohot encoding may be beneficial to large FSMs where onehot uses too many flip flops, and binary requires too much decode logic. Refer also to the HDL Synthesis guide, Chapter 2.

Xilinx Libraries

Support for Xilinx M1 release

RAMs

All RAMs: support for arbitrary size RAMs.

Modgen Libraries

The XI4E: XC4000E/EX modgen library is adding support for synchronous RAMs.

Existing Modgen Library

- XI5: counter area and delay improvements.
- XI3: reductions in area and delay for adders and subtractors.

Components for Modgen Libraries

- XI4/XI4E: pre-setable, non-loadable up counters
- XI4E: RAMs and X15: up/down counters

Fast I/O Buffers

LeonardoSpectrum sets xi4 and xi5 slew rate to Fast for all input and output buffers. This speeds up the design.

Global Clock Buffers

Global clock buffer support - BUFGLS, BUFGE - is added to the pulldown list on the Constraint Input flow tab for the GUI.
Quality of Area and Delay Results

Quality of results were improved by 10%-20% for technology area and delay. The most significant improvements are for targeting Xilinx LUT technologies. The improvements are due to the following features:

- Clock Enable detection was improved to detect more complex clock enable logic
- Multipliers are implemented more efficiently
- Improvements in LUT based Optimization algorithms
- Improvements in the method LeonardoSpectrum uses to represent and optimize wide functions, like ROMs.

You can expect significant improvements for both area and delay on designs that contain multipliers, clock enable logic, and ROMs.

Fast Xilinx Outputs

The following Xilinx Technologies are able to use this feature:

- XC4000/A/E/EX/XL
- XC5200

LeonardoSpectrum can set slew rate on output buffers to FAST. This directs Xilinx software to use output buffers with a fast slew rate and with faster output propagation delay.

Xilinx outputs are set to FAST by default. You can disable this option by clicking to remove selection from FAST Output Buffers, or by specifying `set xlx_fast_slew true` from the Level 3 interactive command line shell. Refer to Table 14-12.

You can also override the default setting. The directive can be an input design HDL (Verilog or VHDL) or a command file in the Constraint File Editor to set slew rate attributes on individual outputs. These attributes can have one of the following values: FAST, MEDFAST, MEDSLOW, SLOW.
Table 14-12. Fast Output Buffers Options

<table>
<thead>
<tr>
<th>Advanced Settings</th>
<th>On/Off</th>
<th>Interactive Command Line Shell</th>
<th>Batch Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST Output Buffers</td>
<td>off</td>
<td>default</td>
<td>default</td>
</tr>
<tr>
<td></td>
<td>on</td>
<td>set xlx_fast_slew true</td>
<td>-noxlx_fast_slew</td>
</tr>
</tbody>
</table>

Controls setting slew rate attributes on individual outputs.

Xilinx outputs are set to FAST by default. You can disable this option by clicking to clear the check box for Use FAST Output Buffers. You can also override the default setting. Your directive can be an input design HDL (Verilog or VHDL) or in the Constraint File Editor to set slew rate attributes on individual outputs. These attributes can have one of the following values: FAST, MEDFAST, MEDSLOW, SLOW.

**M1 Back Annotation Flow**

1. You can generate XNF netlist or EDIF netlist on an output from LeonardoSpectrum and takes it to M1 software.

2. After place and route, M1 software generates VHDL netlist and SDF timing file. The VHDL netlist is simulatable.

3. To read in the VHDL file generated by M1 into Time Module, copy the simprim_VCOMPONENTS.vhd package into your current working directory. The simprim_VCOMPONENTS.vhd package is shipped by Xilinx.

4. Time Module back annotates the timing data from the SDF file generated by M1. You can run timing analysis, review the critical paths report, and look at the critical paths in the schematic viewer.
5. If desired, you can generate a new VHDL netlist and SDF file from Time Module. You must manually add a usage clause for the simprim library (simprim_VCOMPONENTS.vhd) in the VHDL file. The VHDL and SDF files can be read into Model Technology simulator for functional verification.

**SDF (Standard Delay Format) Writer**

An SDF writer for pre place-and-route delays has been added. The SDF writer calculates delays as used by the timing analyzer and timing optimization routines. The SDF writer writes this information in an SDF format. The SDF format together with the flat VHDL netlist can be read into MTI V-System or any RTL simulator for pre place-and-route timing simulation.

The SDF writer is controlled by the `sdf_write_flat_netlist` Tcl variable. Set this variable to TRUE. An example set of Level 3 commands may be:

```
set sdf_write_flat_netlist TRUE
ungroup -all /*need to flatten the netlist*/
write design.vhd /*write out flat VHDL or Verilog*/
write design.sdf /*write out SDF*/
```

**M1 Support**

M1 flow is supported through both EDIF and XNF formats. In EDIF, LeonardoSpectrum writes out unified library cells, and while in XNF, LeonardoSpectrum writes out XNF primitives.

EDIF flow includes:
- writing out FMAPs/HMAPs
- writing CLB information
- writing Timespecs

XNF flow is similar to XACT. In addition, LeonardoSpectrum supports functional verification and timing back-annotation using SimPrims or Neoprims.

**M2 Support**

Xilinx libraries for M2.1 used with build 1999.1d and previous builds: Fixed GND and VCC library cells. If you are using the Xilinx M2.1 library, the new libraries that contain fixes you may need, are located in:

Individual libraries are also in this location if you want to download a specific library.

**Note:** M2.1 may error out if you are using the old LeonardoSpectrum library. The new libraries are available in Release 1e.

**Accurate Delay Estimations**

Wire load models are added for the following Xilinx technologies: XC4000/E/EX/XL and XC5200. These models allow LeonardoSpectrum to accurately predict routing delays. LeonardoSpectrum's estimates are within 5%-10% of (x) delay estimates.

**Sophisticated CLB packing - Level 3**

CLB packing is added for XC4000E, XC4000EX technologies. By default, CLB packing is used for area and delay estimation only. If you want to write CLB information in the output EDIF or XNF netlists, set the variable `write_clb_packing` to TRUE in the interactive command line shell.

```
set write_clb_packing true
```

**Lookup Table**

Functionality of lookup table is written in terms of EQN. For example:

```
(instance ix372 (viewRef NETLIST (cellRef EQN (libraryRef xi4e )))
(property EQN (string "((~I0*~I11)+(~I0*~I2))"))
```

This makes EDIF netlists generated by LeonardoSpectrum much smaller and M1 processing time is faster.

**P&R FlowTab**

The Simulation choice box on the Xilinx P&R FlowTab is for subinvocation. When selected, LeonardoSpectrum instructs the Xilinx Alliance Series to generate VHDL or Verilog simulation netlists targeting the simprim or neoprim cell set and to generate an SDF file for back annotated timing.
Implementing Boundary Scan for Xilinx 4000 FPGAs

The Xilinx 4000 family of FPGAs supports boundary scan through the use of the BSCAN cell. Instantiating this cell implements the boundary scan gate, makes appropriate connections within the FPGA, and implements the TAP controller. Refer to the JTAG 1149.1 standard.

To access the Boundary Scan features of the Xilinx XC4000 devices, you must instantiate the BSCAN Cell and four special components: TDI, TDO, TMS, TCK. The following structure is then internal to the design, and is presented to the Xilinx XACT tools.

Refer to the Libraries Guide in the Programmable Logic Data Book, for more information on the Boundary Scan capabilities of Xilinx.

**VHDL Example:**

ARCHITECTURE structure OF bscan_example IS

-- define boundary scan components --

COMPONENT bscan
PORT (  
  tdi : IN std_logic;
  tms : IN std_logic;
  tck : IN std_logic;
  tdo : OUT std_logic
);
END COMPONENT;

COMPONENT tdo PORT ( o : IN std_logic ); END COMPONENT;

COMPONENT tck PORT ( i : OUT std_logic ); END COMPONENT;

COMPONENT tdi PORT ( i : OUT std_logic ); END COMPONENT;

COMPONENT tms PORT ( i : OUT std_logic ); END COMPONENT;

-- internal signals for TAP controller I/O --

SIGNAL tdo_s : std_logic;
SIGNAL tck_s : std_logic;
SIGNAL tdi_s : std_logic;
SIGNAL tms_s : std_logic;
BEGIN

-- instantiate boundary scan --

u1: TDO PORT MAP ( o => tdo_s );
u2: TCK PORT MAP ( i => tck_s );
u3: TDI PORT MAP ( i => tdi_s );
u4: TMS PORT MAP ( i => tms_s );
u5: bscan PORT MAP ( 
    tdi => tdi_s,
    tms => tms_s,
    tck => tck_s,
    tdo => tdo_s
);

Additional VHDL or Verilog Information

Do not declare TDI, TDO, TMS, TCK as chip-level ports in your VHDL or Verilog description. These ports will be implemented by XACT, even though they are not explicitly declared in the XNF. The appropriate Xilinx library must be designated as the source as well as the target technology:

LeonardoSpectrum treats the BSCAN and T-components as special cases: they are not optimized out of the circuit even though they do not fan into or out of the chip. The BSCAN and T-components remain in the output XNF netlist.

In the design, the TDO, TDI, TCK and TMS pins are still not represented as EXT records (or as primary I/O's) in the XNF. Xilinx XACT tools will consider these as special symbols which will be transformed into pads when the LCA and BIT files are generated.
Example:

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY traffic IS
    PORT (clock, sensor1, sensor2, reset : IN std_logic;
          red1, yellow1, green1, red2, yellow2, green2 : OUT std_logic);
END;
ARCHITECTURE eXemplar OF traffic IS
    TYPE state_t IS (ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7);
    SIGNAL state, nxstate : state_t;
    COMPONENT bscan
        PORT (tdi : IN std_logic;
              tms : IN std_logic;
              tck : IN std_logic;
              tdo : OUT std_logic)
    );
END COMPONENT;
COMPONENT tdo PORT (o : IN std_logic); END COMPONENT;
COMPONENT tck PORT (i : OUT std_logic); END COMPONENT;
COMPONENT tdi PORT (i : OUT std_logic); END COMPONENT;
COMPONENT tms PORT (i : OUT std_logic); END COMPONENT;
BEGIN
    u1: TDO PORT MAP (o => tdo_s);
    u2: TCK PORT MAP (i => tck_s);
    u3: TDI PORT MAP (i => tdi_s);
    u4: TMS PORT MAP (i => tms_s);
    u5: bscan PORT MAP (tdi => tdi_s,
                         tms => tms_s,
                         tck => tck_s,
                         tdo => tdo_s)
END;
update_state : -- Update the state on the clock edge --

    PROCESS (reset, clock)
    BEGIN
        IF (reset='1') THEN
            state <= ST0 ;
        ELSIF clock'event and clock='1' THEN
            state <= nxstate ;
        END IF ;
    END PROCESS;

transitions : -- set the outputs and next state --

    PROCESS (state, sensor1, sensor2)
    BEGIN
        -- Default values for the outputs --
        red1 <= '0'; yellow1 <= '0'; green1 <= '0';
        -- Make sure to always set a value for nxstate, or unwanted latches will occur--
        CASE state IS
            WHEN ST0 =>
                green1 <= '1';
                red2 <= '1';
                IF sensor2 = sensor1 THEN
                    nxstate <= ST1;
                ELSIF (sensor1 = '0' AND sensor2 = '1') THEN
                    nxstate <= ST2;
                ELSE
                    nxstate <= ST0;
                END IF;
            WHEN ST1 =>
                green1 <= '1';
                red2 <= '1';
                nxstate <= ST2;
            WHEN ST2 =>
                green1 <= '1';
                red2 <= '1';
                nxstate <= ST3;
            WHEN ST3 =>
                yellow1 <= '1';
                red2 <= '1';
                nxstate <= ST4;
            WHEN OTHERS =>
                green1 <= '1';
                red2 <= '1';
                nxstate <= ST0;
        END CASE;
    END PROCESS;
WHEN ST4 =>
  red1 <= '1';
  green2 <= '1';
  IF (sensor1 = '0' AND sensor2 = '0') THEN
    nxstate <= ST5;
  ELSIF (sensor1 = '1' AND sensor2 = '0') THEN
    nxstate <= ST6;
  ELSE
    nxstate <= ST4;
  END IF;
WHEN ST5 =>
  red1 <= '1';
  green2 <= '1';
  nxstate <= ST6;
WHEN ST6 =>
  red1 <= '1';
  green2 <= '1';
  nxstate <= ST7;
WHEN ST7 =>
  red1 <= '1';
  yellow2 <= '1';
  nxstate <= ST0;
END CASE;
END PROCESS;
END eXemplar;
Pullups and Pulldowns

The following Verilog, interactive command line shell, and VHDL coding examples illustrate a method for assigning your pullups and pulldowns. **Note for Verilog:**

//exemplar attribute <port_name> pull pullup
//exemplar attribute <port_name> pull pulldn

**Verilog Example**

```verilog
module test (a, oe, o);
    inout [2:0] a ;
    input [3:0] oe ;
    inout o ;
    wire bus;
    assign bus = oe[0] ? a[0] :1 'bz;
    PULLUP i0 (bus);
    PULLUP i1 (a[0]);
endmodule
```

**Interactive Command Line Shell Example**

If you do not want to modify your code, you can assign a pullup or pulldown by setting the following attribute:

```
set_attribute -port <port_name> -name pull -value "pullup"
```
VHDL Example

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY exemplar;
SE exemplar.exemplar_1164.ALL;
ENTITY test2 IS
PORT ( inbus_a, inbus_b, inbus_c : IN std_logic;
        en_a, en_b, en_c : IN std_logic;
        stopit : IN std_logic;
        common : IN std_logic;
        outbus : OUT std_logic );
ATTRIBUTE pull: STRING;
ATTRIBUTE pull OF inbus_a: SIGNAL is "pullup";
ATTRIBUTE pull OF inbus_b: SIGNAL is "pulldn";
END test2;
ARCHITECTURE exemplar OF test2 IS
SIGNAL int_bus : std_logic;
BEGIN
pullup(int_bus);
-- RTL description
int_bus <= inbus_a AND stopit WHEN en_a = '1' ELSE 'Z';
int_bus <= inbus_b AND stopit WHEN en_b = '1' ELSE 'Z';
int_bus <= inbus_c AND stopit WHEN en_c = '1' ELSE 'Z';
bus <= common XOR int_bus;
END exemplar;
Xilinx FPGA Devices Supported

The FPGA devices support are:

- XC3000/A/L, XC3100/A
- XC4000/A/E/EX/L/XL/XLA/XV, XC5200
- Spartan/XL
- Virtex (see Virtex chapter)

<table>
<thead>
<tr>
<th>Xilinx XC3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 100</td>
</tr>
<tr>
<td>Speed Grades supported: 70, 100, 125</td>
</tr>
</tbody>
</table>

Devices Supported

<table>
<thead>
<tr>
<th>3020</th>
<th>PC68 PC84 PG84 PQ100 CB100</th>
</tr>
</thead>
<tbody>
<tr>
<td>3030</td>
<td>PC44 PC68 PC84 PG84 TQ100 PQ100</td>
</tr>
<tr>
<td>3042</td>
<td>PC84 PG84 TQ100 PQ100 CB100 PP132 PG132</td>
</tr>
<tr>
<td>3064</td>
<td>PC84 PG132 PP132 PQ160</td>
</tr>
<tr>
<td>3090</td>
<td>PC84 PQ160 CB164 PG175 PP175 PQ208</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Xilinx XC3000A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 6</td>
</tr>
<tr>
<td>Speed Grades supported: 6, 7</td>
</tr>
</tbody>
</table>

Devices Supported

<table>
<thead>
<tr>
<th>3020A</th>
<th>PC68 PC84 PG84 PQ100 CB100</th>
</tr>
</thead>
<tbody>
<tr>
<td>3030A</td>
<td>PC44 PC68 PC84 PG84 TQ100 PQ100 VQ64 VQ100</td>
</tr>
<tr>
<td>3042A</td>
<td>PC84 PG84 TQ144 PQ100 CB100 PP132 PG132 VQ100</td>
</tr>
<tr>
<td>3064A</td>
<td>PC84 PG132 PP132 PQ160 TQ144</td>
</tr>
<tr>
<td>3090A</td>
<td>PC84 PQ160 CB164 PG175 PP175 PQ208 TQ144 TQ176</td>
</tr>
</tbody>
</table>
### Xilinx XC3000L

<table>
<thead>
<tr>
<th>Default Speed Grade: 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed Grades supported: 8</td>
</tr>
<tr>
<td>Devices Supported</td>
</tr>
<tr>
<td>3020L PC84</td>
</tr>
<tr>
<td>3030L VQ64 PC84 VQ100</td>
</tr>
<tr>
<td>3042L PC84 VQ100 TQ144</td>
</tr>
<tr>
<td>3064L PC84 TQ144</td>
</tr>
<tr>
<td>3090L TQ144 TG176</td>
</tr>
</tbody>
</table>

### Xilinx XC3100

<table>
<thead>
<tr>
<th>Default Speed Grade: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed Grades supported: 2, 3, 4, 5</td>
</tr>
<tr>
<td>Devices Supported</td>
</tr>
<tr>
<td>3120 PC68 PC84 PG84 PQ100</td>
</tr>
<tr>
<td>3130 PC44 PC68 PC84 PG84 PQ100 TQ100</td>
</tr>
<tr>
<td>3142 PC84 PG84 PQ100 TQ100 PP132 PG132 TQ144</td>
</tr>
<tr>
<td>3164 PC84 PP132 PG132 PQ160</td>
</tr>
<tr>
<td>3190 PC84 PP175 PG175 PQ208 PQ160</td>
</tr>
<tr>
<td>3195 PC84 PQ160 PP175 PG175 PQ208 PG223</td>
</tr>
</tbody>
</table>
Xilinx XC3100A
Default Speed Grade: 4
Speed Grades supported: 2, 3, 4, 5

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3120A</td>
<td>PC68 PC84 PG84 PQ100</td>
</tr>
<tr>
<td>3130A</td>
<td>PC44 PC68 PC84 PG84 PQ100 TQ100</td>
</tr>
<tr>
<td>3142A</td>
<td>PC84 PG84 PQ100 TQ100 PP132 PG132 TQ144</td>
</tr>
<tr>
<td>3164A</td>
<td>PC84 PP132 PG132 PQ160</td>
</tr>
<tr>
<td>3190A</td>
<td>PC84 PQ160 PP175 PG175 PQ208</td>
</tr>
<tr>
<td>3195A</td>
<td>PC84 PQ160 3195APP175 PG175 PQ208 PG223</td>
</tr>
</tbody>
</table>

Xilinx XC4000
Default Speed Grade: 5
Speed Grades supported: 4, 5, 6

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4003</td>
<td>PC84 PQ100 PG120</td>
</tr>
<tr>
<td>4005</td>
<td>PC84 PQ100 PG156 PQ160 CB164 PQ208</td>
</tr>
<tr>
<td>4006</td>
<td>PC84 PG156 PQ160 PQ208</td>
</tr>
<tr>
<td>4008</td>
<td>PC84 PQ160 PG191 PQ208 MQ208</td>
</tr>
<tr>
<td>4010</td>
<td>PC84 PQ160 PG191 CB196 PQ208 MQ208 BG225</td>
</tr>
<tr>
<td>4013</td>
<td>PQ160 PQ208 MQ208 PG223 BG225 MQ240 PQ240</td>
</tr>
<tr>
<td>4025</td>
<td>PG223 MQ240 PG299 HQ304</td>
</tr>
</tbody>
</table>
### Xilinx XC4000A

**Default Speed Grade:** 5  
**Speed Grades supported:** 4, 5, 6  
**Devices Supported**

<table>
<thead>
<tr>
<th>Model</th>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>4002A</td>
<td>PC84 PQ100 VQ100 PG120</td>
</tr>
<tr>
<td>4003A</td>
<td>PC84 PQ100 VQ100 CB100 PG120</td>
</tr>
<tr>
<td>4004A</td>
<td>PC84 PG120 TQ144 PQ160</td>
</tr>
<tr>
<td>4005A</td>
<td>PC84 PG120 TQ144 PG156 PQ160 PQ208</td>
</tr>
</tbody>
</table>

### Xilinx XC4000E

**Default Speed Grade:** 3  
**Speed Grades supported:** 1, 2, 3, 4  
**Devices Supported**

<table>
<thead>
<tr>
<th>Model</th>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>4003E</td>
<td>PC84 PQ100 VQ100 PG120</td>
</tr>
<tr>
<td>4005E</td>
<td>PC84 PQ100 TQ144 PG156 PQ160 CB164 PQ208</td>
</tr>
<tr>
<td>4005E</td>
<td>PC84 PQ208</td>
</tr>
<tr>
<td>4006E</td>
<td>PC84 TQ144 PG156 PQ160 PQ208</td>
</tr>
<tr>
<td>4008E</td>
<td>PC84 PQ160 PG191 PQ208</td>
</tr>
<tr>
<td>4010E</td>
<td>PC84 PQ160 PG191 CB196 PQ208 HQ208 BG225</td>
</tr>
<tr>
<td>4013E</td>
<td>PQ160 PQ208 HQ208 PG223 BG225 CB228 HQ240 PQ240</td>
</tr>
<tr>
<td>4013E</td>
<td>BG225</td>
</tr>
<tr>
<td>4020E</td>
<td>HQ208 PG223 HQ240</td>
</tr>
<tr>
<td>4025E</td>
<td>PG223 CB228 HQ240 PG299 HQ304 BG352</td>
</tr>
</tbody>
</table>
### Xilinx XC4000EX

<table>
<thead>
<tr>
<th>Default Speed Grade: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed Grades supported: 2, 3, 4</td>
</tr>
</tbody>
</table>

#### Devices Supported

<table>
<thead>
<tr>
<th>4028EX</th>
<th>HQ208 HQ240 HQ304 PG299 BG352 CB228</th>
</tr>
</thead>
<tbody>
<tr>
<td>4036EX</td>
<td>HQ240 HQ304 BG432 PG411</td>
</tr>
</tbody>
</table>

### Xilinx XC4000L

<table>
<thead>
<tr>
<th>Default Speed Grade: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed Grades supported: 3</td>
</tr>
</tbody>
</table>

#### Devices Supported

<table>
<thead>
<tr>
<th>4005l</th>
<th>PC84 PQ100 VQ100 PC208</th>
</tr>
</thead>
<tbody>
<tr>
<td>4010l</td>
<td>PC84 PQ100 PQ208</td>
</tr>
<tr>
<td>4010l</td>
<td>TQ176</td>
</tr>
<tr>
<td>4013l</td>
<td>BG225 PQ208 PQ240</td>
</tr>
</tbody>
</table>
Xilinx XC4000XL

Default Speed Grade: 3

Speed Grades supported: 3, 09

Devices Supported

<table>
<thead>
<tr>
<th>Device Code</th>
<th>Supported Parts</th>
</tr>
</thead>
<tbody>
<tr>
<td>4005x</td>
<td>1PC84 IPQ100 IVQ100 ITQ144 IPQ160 IPQ208</td>
</tr>
<tr>
<td>4010x</td>
<td>IPQ84 IPQ100 ITQ144 ITQ176 IPQ160 IPQ208 IBG256 ICB228 IPG223</td>
</tr>
<tr>
<td>4013x</td>
<td>IPQ160 IPQ208 IHT144 IHT176 IPQ240 IBG256</td>
</tr>
<tr>
<td>4020x</td>
<td>IHT144 IHT176 IPQ160 IPQ208 IPQ240 IBG256</td>
</tr>
<tr>
<td>4028x</td>
<td>1BG256 IBG352 IHQ208 IHQ240 IGP299 IHQ160</td>
</tr>
<tr>
<td>4036x</td>
<td>1BG352 IBG432 IGP411 IHQ208 IHQ240 1HQ160 ICB228 IHQ304</td>
</tr>
<tr>
<td>4044x</td>
<td>1BG352 IBG432 IHQ240 IGP411 IHQ160 IHQ208 IHQ304</td>
</tr>
<tr>
<td>4052x</td>
<td>1BG432 IBG560 IHQ240 IGP411 1HQ304 ICB228 IHQ304</td>
</tr>
<tr>
<td>4062x</td>
<td>1BG432 IBG560 IHQ240 IGP475 IGP559</td>
</tr>
<tr>
<td>4085x</td>
<td>1PG559 IBG432 IBG560 IGP559</td>
</tr>
<tr>
<td>4002x</td>
<td>1PC84, IPQ100 IVQ100</td>
</tr>
</tbody>
</table>
### Xilinx XC4000XLA

**Default Speed Grade:** 07

**Speed Grades supported:** 07, 08, 09

**Devices Supported**

<table>
<thead>
<tr>
<th>Device Code</th>
<th>Package/Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>4013xla</td>
<td>BG256 PQ160 PQ208 PQ240</td>
</tr>
<tr>
<td>4020xla</td>
<td>BG256 PQ160 PQ208 PQ240</td>
</tr>
<tr>
<td>4028xla</td>
<td>BG256 BG352 HQ160 HQ208 HQ240</td>
</tr>
<tr>
<td>4036xla</td>
<td>BG256 BG352 HQ160 HQ208 HQ240</td>
</tr>
<tr>
<td>4044xla</td>
<td>BG256 BG352 HQ160 HQ208 HQ240 HQ304</td>
</tr>
<tr>
<td>4052xla</td>
<td>BG256 BG352 HQ160 HQ208 HQ240 HQ304 BG560</td>
</tr>
<tr>
<td>4062xla</td>
<td>BG256 BG352 HQ160 HQ208 HQ240 HQ304</td>
</tr>
<tr>
<td>4085xla</td>
<td>BG256 BG352 HQ160 HQ208 HQ240 HQ304 HQ240</td>
</tr>
</tbody>
</table>

### Xilinx XC4000XV

**Default Speed Grade:** 08

**Speed Grades supported:** 08, 09

**Devices Supported**

<table>
<thead>
<tr>
<th>Device Code</th>
<th>Package/Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>40110xv</td>
<td>HQ240 BG352 BG432 BG560</td>
</tr>
<tr>
<td>40150xv</td>
<td>HQ240 BG352 BG432 BG560 PG559</td>
</tr>
<tr>
<td>40200xv</td>
<td>BG560 BG432</td>
</tr>
<tr>
<td>40250xv</td>
<td>BG432 BG560 PG559</td>
</tr>
</tbody>
</table>
### Xilinx XC5200

**Default Speed Grade:** 5

**Speed Grades supported:** 3, 4, 5, 6

**Devices Supported**

<table>
<thead>
<tr>
<th>Speed Grade</th>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>5202</td>
<td>PC84 PQ100 VQ100 5202TQ144 PG156</td>
</tr>
<tr>
<td>5204</td>
<td>PC84 PQ100 VQ100 TQ144 PG156 PQ160</td>
</tr>
<tr>
<td>5206</td>
<td>PC84 PQ100 VQ100 TQ144 PQ160 PQ208 PG191</td>
</tr>
<tr>
<td>5210</td>
<td>PC84 TQ144 PQ160 PQ208 PG223 PQ240 BG225</td>
</tr>
<tr>
<td>5215</td>
<td>PQ208 PQ240 HQ304 PG299</td>
</tr>
</tbody>
</table>

### Spartan

**Default Speed Grade:** 3

**Speed Grades supported:** 3, 4

**Devices Supported**

<table>
<thead>
<tr>
<th>Speed Grade</th>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>S05</td>
<td>PC84 VQ100</td>
</tr>
<tr>
<td>S10</td>
<td>PC84 VQ100 TQ144</td>
</tr>
<tr>
<td>S20</td>
<td>VQ100 TQ144 PQ208</td>
</tr>
<tr>
<td>S30</td>
<td>VQ100 TQ144 PQ208 PQ240 BG256</td>
</tr>
<tr>
<td>S40</td>
<td>PQ208 PQ240 BG256</td>
</tr>
<tr>
<td>Spartan XL</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>Default Speed Grade: 3</td>
<td></td>
</tr>
<tr>
<td>Speed Grades supported: 3, 4</td>
<td></td>
</tr>
</tbody>
</table>

**Devices Supported**

<table>
<thead>
<tr>
<th>Model</th>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>S05xl</td>
<td>PC84, VQ100</td>
</tr>
<tr>
<td>S10xl</td>
<td>PC84 VQ100 TQ144 CS144</td>
</tr>
<tr>
<td>S20xl</td>
<td>VQ100 TQ144 PQ208 CS144</td>
</tr>
<tr>
<td>S30xl</td>
<td>VQ100 TQ144 PQ208 PQ240 BG256</td>
</tr>
<tr>
<td>S40xl</td>
<td>PQ208 PQ240 PQ256</td>
</tr>
</tbody>
</table>
This chapter presents information specific to the use of Xilinx Virtex as a source or target technology. LeonardoSpectrum supports the Xilinx Virtex, VirtexE, and Spartan 2 architectures. This chapter is divided as follows:

• Before Beginning
• Virtex Features
• Xilinx Virtex Family
• Additional Information
• Virtex Flow Diagram

**Before Beginning**

This chapter assumes that you have read the LeonardoSpectrum User’s guide and the introductory chapters in this guide.

**Virtex Features**

The Xilinx Virtex family brings you next generation FPGAs that can break density and performance barriers and also provide unprecedented system level integration. The following quality of results for Virtex have been implemented in LeonardoSpectrum:

• Wide Cluster of MUXs Mapping
• Priority Encoder
• Synchronous Set/Reset
• Support for Virtex IOB Registers
• Tristate Register Replication
• Support for SRL (Shift Register LUTs)
• Block RAMs for Virtex
• Virtex Single Port RAMs
• Support for CLKDLL

Wide Cluster of MUXs mapping

Mapping is improved to map to MUXF5s and MUXF6s efficiently for Virtex. For odd-size muxes (non 2-power input size muxes), more MUXF5s and MUXF6s are utilized.

Priority Encoder

Currently, LeonardoSpectrum maps to Virtex slice MUXCY cell from random logic, to implement priority encoded structures like if-then-else statements in RTL code.

Previously, LeonardoSpectrum used slice MUXCY for mapping only arithmetic logic like adders, counters, multipliers, etc. LeonardoSpectrum now maps to MUXCY by default. The variable that enables this algorithm is map_muxcy. The default of this variable is TRUE.

VHDL Example for MUXCY Mapping

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity priority is
    generic ( size :integer := 8 );
    port(
        clk: in std_logic;
        data: in std_logic_vector(size-1 downto 0);
        cond1: in std_logic_vector(size-1 downto 0);
        cond2: in std_logic_vector(size-1 downto 0);
        q: out std_logic);
    end priority;
architecture syn of priority is
begin
process begin
```
wait until clk = '1';
q <= '0';
for n in data' range loop
    if ((cond1(n) and cond2(n)) = '1') then
        q <= data(n);
    end if;
end loop;
end process;
end syn;

NOTE: If you are using Xilinx M1 software versions prior to M1 2.1i, then set the 
map_muxcy variable to FALSE.

    set map_muxcy false

The earlier versions of Xilinx M1 did not map this primitive very well. This resulted in 
a larger and slower circuit. Refer to Table 15-1 for P&R compatibility.

Table 15-1. Place and Route Compatibility

<table>
<thead>
<tr>
<th>LeonardoSpectrum Version</th>
<th>Xilinx Alliance Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>98.2x</td>
<td>M1 1.5</td>
</tr>
<tr>
<td>99.1a to 99.1f</td>
<td>M1 1.5 or M1 2.1</td>
</tr>
<tr>
<td>99.1g</td>
<td>M1 2.1i (highly recommended)</td>
</tr>
</tbody>
</table>

**Synchronous Set/Reset**

LeonardoSpectrum can now map to most registers with synchronous set/reset that have 
clock enables. There are some boundary cases which are sub-optimal to map to 
synchronous set/reset flip flops. This is now handled properly.

**Support for Virtex IOB Registers**

M1 supports IFDX, 0FDX types of cells for the Xilinx 4K family. In contrast, M1 does 
not support the IFDX, OFDX types of cells as primitives for the Xilinx Virtex family.

LeonardoSpectrum maps IOB registers to internal registers, for example: FD, FDRE, 
FDCE, FDPEs.

If you set the following variable (default false),
set `virtex_map_iob_registers` false

then the processing of IOB registers in LeonardoSpectrum is enabled. If you set this variable to true,

set `virtex_map_iob_registers` true

then LeonardoSpectrum sets an IOB attribute on all internal registers that can be implemented in the IOB register. M1 recognizes the IOB attribute and maps the registers in IOBs.

**Tristate Register Replication**

The `virtex_map_iob_registers` global variable also enables replication of tristate registers driving tristate pins of an external bus. This improves Tck0 time for these tristate registers. Registers driving multiple output ports are also replicated.

You can selectively push some registers and not all registers to IOB by setting the attribute: IOB, on these specific IO ports. The global variable: `virtex_map_iob_registers` is not required. The following example VHDL code shows the selective implementing of specific flip flops in IOB registers:

```vhdl
library ieee;
use ieee.std_LOGIC_1164.ALL;
use ieee.std_LOGIC_arith.ALL;
entity _test IS
  port ( clock, d :IN_ STD_LOGIC;
         q0, q1:OUT STD_LOGIC);
end test;
architecture hds of test is
begin
  ri :process (clock)
  variable x:std_LOGIC;
  begin
    if (clock'event and clock = '1') then
      q0 <= x;
  end process ri;
end hds;
```

**Note:** The IOB attribute enables q0 to be implemented in the Virtex IOB register.

```vhdl
attribute IOB :boolean;
attribute IOB of q0 :signal is true;
end test;
```

LeonardoSpectrum Synthesis and Technology
Support for SRL

LeonardoSpectrum maps to SRLs - an array of flip flops is mapped to SRLs. The SRL mapping provides an improvement in area for designs with similar structures. SRL is used in static addressing mode. This means that the address lines for SRL are tied to a constant. LeonardoSpectrum maps to two flavors of SRL: SRL16 and SRL16E (with clock enable).

Variables

There are two variables:

(1) This variable enables SRL mapping (default true):
set virtex_map_srl true

(2) This variable packs SRL into a single slice (default true).
set virtex_map_srl_pack true

VHDL Example SRL Design

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity pipeline_delay is
   generic (cycle : integer := 16;
            width :integer := 16);
   port (input :in std_logic_vector(width - 1 downto 0);
         clk :in std_logic;
         output :out std_logic_vector(width - 1 downto 0));
   attribute clock_node :boolean;
   attribute clock_node of clk : signal is TRUE;
end pipeline_delay;
architecture behav of pipeline_delay is
x := d;
q1 <= x;
end if;
end process ri;
end hds;
type my_type is array (0 to cycle -1) of std_logic_vector(width -1 downto 0);

signal int_sig :my_type;

begin
main : process (clk)
begin
  if clk'event and clk = '1' then
    int_sig <= input & int_sig(0 to cycle - 2);
  end if;
end process main;

output <= int_sig(cycle -1);
end behav;

Verilog Example SRL

module srle_example (clk, enable, data_in, result);

  parameter cycle=4;
  parameter width = 3;
  input clk, enable;
  input [0:width] data_in;
  output [0:width] result;
  reg [0:width-1] shift [cycle-1:0];
  integer i;

always @(posedge clk)
begin
  if (enable == 1) begin
    for (i = (cycle-1); i > 0; i=i-1) shift[i] = /
      shift[i-1];
    shift[0] = data_in;
  end
end

assign result = shift[cycle-1];
endmodule
**Block RAMs for Virtex**

LeonardoSpectrum can map your memory statements in Verilog or VHDL to the block RAMs on all Virtex devices. The following is a list of the details for block RAMs in LeonardoSpectrum.

- Virtex Block RAMs are completely synchronous - both read and write operations are synchronous.
- LeonardoSpectrum infers single port RAMs - RAMs with both read and write on the same address - and, dual port RAMs - RAMs with separate read and write addresses. Currently, LeonardoSpectrum does not infer dual port RAMs that read both read and write address.
- Virtex Block RAMs support RST (reset) and ENA (enable) pins. Currently, LeonardoSpectrum does not infer RAMs which use the functionality of the RST and ENA pins.

**Variables**

Set the following variable to `false` if you do not want RAM extraction. Default is `true`.

```makefile
set extract_ram false
set extract_ram true (default)
```

By default, RAMs that are mappable to block RAMs, are mapped to block RAMs. You can disable mapping to block RAMs by setting the attribute `block_ram` to `false`.

```makefile
set_attribute -name block_ram -value false
```

In this case, the RAM is implemented using select RAMs if possible.

**Please Note**: The variant of single port RAM that is implemented using block RAMs, cannot be implemented using select RAMs.

**Verilog Example:**

```verilog
module spmem64 (din, wen, wraddr, clk, dout);
input [7:0] din;
input wen, clk;
input [5:0] wraddr;
output [7:0] dout;
integer i;
```
reg [7:0] mem [63:0];
// Exemplar attribute mem block_ram FALSE.
This comment sets the block_ram attribute to FALSE on the signal mem. The block_ram attribute must be set on the memory signal.

reg [5:0] addr_int;
    assign dout = mem[addr_int];
    always @ (posedge clk)
        addr_int <= wraddr;
    always @ (posedge clk)
        if (wen) mem[wraddr] <= din;
endmodule

VHDL Example of Block RAMs

library ieee, exemplar;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ram_example1 is
    port (data: in std_ulogic_vector(7 downto 0);
          address: in unsigned(7 downto 0);
          we, inclock, outclock: in std_ulogic;
          q: out std_ulogic_vector(7 downto 0));
    end ram_example1;
    architecture ex1 of ram_example1 is
   type mem_type is array (255 downto 0) of /
std_ulogic_vector (7 downto 0);
   signal mem: mem_type;
   signal raddress : unsigned(7 downto 0);
   begin
   10: process (inclock, we, address)
   begin
    if (inclock = '1' and inclock'event) then
    raddress <= address;
    if (we = '1') then
    mem(to_integer(raddress)) <= data;
    end if;
   end if;
   end process;
11: process (outclock, address)
   begin
      if (outclock = '1' and outclock'event) then
         q <= mem(to_integer(address));
      end if;
   end process;
end ex1;

Virtex Single Port RAMs

The flavor of single port RAMs that are implemented in Virtex block RAMs is changed as shown in the following Verilog code.

Note: This change is for single port RAMs only; dual port RAMs are unaffected.

LeonardoSpectrum now maps single port RAMs that have synchronous write and read addresses registered in Block RAMs.

Verilog Coding Example:

module new_ram (clk, we, din, addr, dout);
input clk, we;
input [7:0] din;
input [4:0] addr;
output [7:0] dout;
reg [4:0] addri;
reg [7:0] data[0:31];
always @(posedge clk)
begin
   if (we == 1'b1)
      data[addr] = din;
      addri = addr;
end
   assign dout = data[addri];
endmodule

Previously, LeonardoSpectrum implemented single port RAMs that had both synchronous read and write in Block RAMs. This method failed to match the Block RAMs description under certain conditions.
Verilog Coding Example (not used):

```verilog
module old_ram (clk, we, din, addr, dout);

input  clk, we;
input  [7:0] din;
input  [4:0] addr;
output [7:0] dout;
reg    [7:0] dout;
reg    [7:0] data[0:31];
always @(posedge clk)
begin
  if (we == 1'b1)
    data[addr] <= din;
  dout <= data[addr];
end
endmodule
```

Variables

As explained in the Block RAMs for Virtex section, set the following variable to false if you do not want RAM extraction. Default is true.

```
set extract_ram false
```

GSR Processing

GSR processing is not recommended by Xilinx for Virtex. The following variable is false by default since GSR lines can be slower than local asynchronous S/R signals.

```
set virtex_infer_gsr (FALSE)
```

This switch allows LeonardoSpectrum to do "GSR" processing for Virtex. Exemplar recommends using GSR processing sparingly.

CLKDLL

LeonardoSpectrum supports Virtex CLKDLL. CLKDLL may be used in two ways in LeonardoSpectrum.
1. You can instantiate the CLKDLL: CLKDLL input (CLKIN) should always come from IBUFG and the feedback (CLKFB) should be driven by a BUFG. The BUFG input should be driven by same CLKDLL.

**Example VHDL Coding:**

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY traffic IS
  PORT (clock, sensor1, sensor2, reset : IN std_logic; std_logic)
               red1, yellow1, green1, red2, yellow2, green2 : OUT std_logic);
END ;
ARCHITECTURE eXemplar OF traffic IS
  TYPE state_t IS ( ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7 );
  SIGNAL state, nxstate : state_t;
             signal clock_bufg_in : std_logic;
```

Note: The above partial code is from the following example. This partial code describes the function and instantiation of CLKDLL.
signal clock_bufg_out : std_logic;
signal clock_ibufg_out : std_logic;
component CLKDLL
  port (CLKIN : in std_logic;
          CLKFB : in std_logic;
          RST : in std_logic;
          CLK0 : out std_logic;
          CLK90 : out std_logic;
          CLK180 : out std_logic;
          CLK270 : out std_logic;
          CLK2X : out std_logic;
          CLKDV : out std_logic;
          LOCKED : out std_logic);
end component;
component BUFG
  port (I : in std_logic;
        O : out std_logic);
end component;
component IBUFG
  port (I : in std_logic;
        O : out std_logic);
end component;
BEGIN
  u0: IBUFG port map (I => clock, O => clock_ibufg_out);
  u1: CLKDLL port map (CLKIN => clock_ibufg_out, CLKFB => /
                       clock_bufg_out, RST => reset, CLK0 => clock_bufg_in);
  u2: BUFG port map (I => clock_bufg_in, O => clock_bufg_out);
  PROCESS (reset, clock_bufg_out)
  BEGIN
    IF (reset='1') THEN
      state <= ST0;
    ELSIF clock_bufg_out'event and clock_bufg_out='1' THEN
      state <= nxstate;
    END IF;
  END PROCESS;

transitions :--set the outputs and next state

PROCESS (state, sensor1, sensor2)
BEGIN

-- Default values for the outputs
red1 <= '0'; yellow1 <= '0'; green1 <= '0';
red2 <= '0'; yellow2 <= '0'; green2 <= '0';

-- Always set a value for nxstate, or unwanted latches occur
CASE state IS
WHEN ST0 =>
green1 <= '1';
red2 <= '1';
IF sensor2 = sensor1 THEN
  nxstate <= ST1;
ELSIF (sensor1 = '0' AND sensor2 = '1') THEN
  nxstate <= ST2;
ELSE
  nxstate <= ST0;
END IF;
WHEN ST1 =>
green1 <= '1';
red2 <= '1';
  nxstate <= ST2;
WHEN ST2 =>
green1 <= '1';
red2 <= '1';
  nxstate <= ST3;
WHEN ST3 =>
yellow1 <= '1';
red2 <= '1';
  nxstate <= ST4;
WHEN ST4 =>
red1 <= '1';
green2 <= '1';
IF (sensor1 = '0' AND sensor2 = '0') THEN
nxstate <= ST5;
ELSEIF (sensor1 = '1' AND sensor2 = '0') THEN
ELSE
   nxstate <= ST4;
END IF;
WHEN ST5 =>
   red1 <= '1';
   green2 <= '1';
   nxstate <= ST6;
WHEN ST6 =>
   red1 <= '1';
   green2 <= '1';
   nxstate <= ST7;
WHEN ST7 =>
   red1 <= '1';
   yellow2 <= '1';
   nxstate <= ST0;
END CASE;
END PROCESS;
END eXemplar;

2. Automatic insertion of CLKDLL: This supports a limited set of CLKDLL:
   • BUFGDLL : CLKDLL or the low frequency DLL
   • BUFGDLLHF (a macro in LeonardoSpectrum): CLKDLLHF or the high
     frequency DLL.

In both the BUFGDLL and BUFGDLLHF cases for (2.), only the CLK0 output from
the DLL is supported. For other versions, the manual instantiation of DLL is
recommended as shown in (1.).

Procedure
   • load virtex library (load_lib xcv)
   • read the design (read <design>)
   • GUI: in constraint editor: select the clock port and assign a BUFGDLL or
     BUFGDLLHF buffer, based on your clock constraint.
   • Optional Interactive Mode - Use command:
     PAD BUFGDLL|BUFGDLLHF <clock port name>
Example for Interactive Mode
load_lib xcv
read traffic.vhd
PAD BUFGDLL clock
optimize -ta xcv
auto_write traffic.edf

Constraint propagation for Virtex CLKDLL

Xilinx place and route software cannot trace through a CLKDLL, so the clock constraint from the port needs to be propagated to the CLKDLL outputs. LeonardoSpectrum supports automatic propagation of clock constraints to DLL output(s).

For a detailed explanation, refer to Xilinx place and route documentation: “Using Timing Constraints” under “PERIOD Specifications” on CLKDLLs.
Virtex Devices Supported

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 4</td>
<td></td>
</tr>
<tr>
<td>Speed Grades supported: 4, 5, 6</td>
<td></td>
</tr>
<tr>
<td>v50</td>
<td>bg256, pq240, cs144, tq144, fg256</td>
</tr>
<tr>
<td>v100</td>
<td>bg256, cs144, fg256, pq240, tq144</td>
</tr>
<tr>
<td>v150</td>
<td>bg352, fg256, fg456, pq240</td>
</tr>
<tr>
<td>v200</td>
<td>bg352, fg456, pq240</td>
</tr>
<tr>
<td>v300</td>
<td>bg256, bg432, fg256, pq240</td>
</tr>
<tr>
<td>v400</td>
<td>bg432, bg560, fg676, hq240, fg680, bq432, bq560</td>
</tr>
<tr>
<td>v600</td>
<td>bg432, bg560, fg676, fg680, hq240</td>
</tr>
<tr>
<td>v800</td>
<td>bg432, bg560, fg676, fg680, hq240</td>
</tr>
<tr>
<td>v1000</td>
<td>bg560, fg556, fg680</td>
</tr>
</tbody>
</table>

VirtexE Devices Supported

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 6</td>
<td></td>
</tr>
<tr>
<td>Speed Grades supported: 6, 7, 8</td>
<td></td>
</tr>
<tr>
<td>v100e</td>
<td>cs144, pq240, hq240, fg256</td>
</tr>
<tr>
<td>v200e</td>
<td>cs144, pq240, hq240, fg256, fg456</td>
</tr>
<tr>
<td>v300e</td>
<td>pq240, hq240, bg432, fg256, fg456</td>
</tr>
<tr>
<td>v400e</td>
<td>pq240, hq240, bg432, fg676</td>
</tr>
<tr>
<td>v600e</td>
<td>pq240, hq240, bg432, fg676, fg900, fg680</td>
</tr>
<tr>
<td>v1000e</td>
<td>pq240, hq240, bg560, fg900, fg1156, fg680, fg860</td>
</tr>
<tr>
<td>v1600e</td>
<td>bg560, fg900, fg1156, fg680, fg860</td>
</tr>
<tr>
<td>v2000e</td>
<td>bg560, fg1156, fg680, bg860</td>
</tr>
</tbody>
</table>
Virtex Spartan2 Devices Supported

<table>
<thead>
<tr>
<th>Default Speed Grade: 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed Grades supported: 5, 6</td>
</tr>
<tr>
<td>2s15</td>
</tr>
<tr>
<td>2s30</td>
</tr>
<tr>
<td>2s50</td>
</tr>
<tr>
<td>2s100</td>
</tr>
<tr>
<td>2s150</td>
</tr>
</tbody>
</table>

Additional Information

Additional Virtex information includes:
- FSM Encoding
- Virtex Report
- Xilinx Coregen Support

**FSM Encoding (binary, gray, random, onehot, twohot, auto)**

For auto encoding, LeonardoSpectrum varies the encoding based on bit width. Moreover, enumerated types with fewer elements than global integer lower_enum_break are encoded as binary; while larger enumerated types are encoded as onehot. Values larger than global integer upper_enum_break are encoded as binary. The auto default allows LeonardoSpectrum to select encoding on a case-by-case basis. **Note:** If LeonardoSpectrum selects onehot for your auto encoded design, then “onehot encoding” is printed for the log file of your design.

The encoding variable determines how LeonardoSpectrum encodes enumerated types, and implements a state machine with a state vector of an enumerated type.

**Twohot Encoding**

Twohot encoding is now added to FSM encoding (binary, gray, random, onehot, twohot, auto). Twohot sets two flip flops high for each state. The twohot encoding requires more flip flops than binary and fewer flip flops than onehot. Twohot encoding
may be beneficial to large FSMs where onehot uses too many flip flops, and binary requires too much decode logic. Refer to the HDL Synthesis guide, Chapter 2, for more encoding information.

**Virtex Report**

The device utilization report is improved with the number of flip flops, "slices", and function generators. *Note*: The basic building blocks of a CLB are 4 logic cells that are organized in two similar "slices". Two items for the Virtex libraries: (1) the number of function generators equals the number of flip flops; and (2) the number of slices is equal to one half the number of function generators.

**Xilinx Coregen Support**

The Exemplar / Xilinx Coregen flow should be as follows:

1. Create the desired core using Coregen GUI with vendor Exemplar.

2. Include the core within the desired design hierarchy. For VHDL create a component declaration and instantiation based on the information contained in the vho file. Do NOT read the *.edn into LeonardoSpectrum. The Core module is treated as a black box. The *.edn file is used by ngdbuild during expansion.

3. Synthesize the top level design, write the output EDIF.

4. Pass the top level EDIF through Xilinx tools. Ngdbuild should pick up the core component.

**Virtex Flow Diagram**

Figure 15-1 shows the progress of your design through the synthesis steps - Read, Synthesis, Write a Netlist - to Place and Route with Xilinx software.
Figure 15-1. Virtex Flow Diagram.
Additional Supported Technologies

The following FPGA technologies are also supported by LeonardoSpectrum:

- Atmel with modgen library
- Cypress
- Lattice/Vantis
- Minc

**Atmel**

The Atmel modgen library is supported. Devices supported: AT40K, AT6K02, AT6K04

AT40K: Enhanced support includes a new library with timing, LUT mapping, and improved module generation.

Added RAMs: Area and delay modeling for LPMs.

**Cypress**

LeonardoSpectrum supports both Flash370i and Ultra37000 technologies. You generate a VHDL output netlist to be used as input to the Warp place-and-route software. Production Warp software, version 4.3 or higher, MUST be used.
Lattice/Vantis

Lattice/Vantis Devices supported are: pLSI-1000/2000 and isp5000/8000.

LATTICE/VANTIS CPLD devices are supported in LeonardoSpectrum. You select the Vantis library as a target library. LeonardoSpectrum generates an EDIF output netlist based on MINC primitives. The output netlist is used as input to Design Direct place-and-route. Devices supported: Mach1, Mach2, Mach4, Mach 4A, Mach 4 Low Voltage, Mach 5, Mach 5A, Mach5 Low Voltage, and VF1.

**Devices Supported**

The following devices are supported: **NOTE**: Devices for MACH 1 and MACH 2 are currently not listed.

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th>M4: 32/32, 64/32, 96/48, 128N/64, 128/64, 196/96, 256/128</th>
</tr>
</thead>
<tbody>
<tr>
<td>M4LV</td>
<td>M4LV: 32/32, 64/32, 96/48, 128N/64, 128/64, 196/96, 256/128</td>
</tr>
<tr>
<td>M4A</td>
<td>M4A: 32/32, 64/32, 96/48, 128/64, 196/96, 256/128</td>
</tr>
<tr>
<td>Packages</td>
<td>44PLcc 44TQFP 48TQFP 100TQFP 144TQFP 208PQFP</td>
</tr>
<tr>
<td>M5</td>
<td>M5: 384/184, 384/192, 512/120, 512/160, 512/184, 512/192, 512/192</td>
</tr>
<tr>
<td>M5LV</td>
<td>M5LV: 384/184, 384/192, 512/120, 512/160, 512/184, 512/192, 512/256</td>
</tr>
<tr>
<td>M5A</td>
<td>M5A: 384/120, 384/160, 384/192, 512/120, 512/160, 512/184, 512/192, 512/256</td>
</tr>
<tr>
<td>Packages</td>
<td>100PQFP 144PQFP 160PQFP 208PSFP 240PQFP 256BGA 352BGA</td>
</tr>
</tbody>
</table>
Minc

Synario devices.
Index

A
acf file, 10-5
ACTGen, 7-10
adding an ASIC library to the GUI, 6-8
Adobe Acrobat Reader, 1-8
ALSPIN, Actel, 7-25, 7-28
Altera
  APEX 20K/20KE, 8-4
devices supported, 9-46
  Quartus, 8-4
  Quartus P&R Integration, 8-4
Altera MAX
devices supported, 10-11
APEX 20K devices supported, 8-9
APEX 20K/20KE mapping, 8-1
APEX 20KE devices supported, 8-9
APEX technology
  WYSIWYG primitives, 8-1
architecture, Altera MAX, 10-2
arithmetic operator, 7-6, 14-18
array_pin_number, attribute, 10-5, 11-29, 14-52
ARRIVAL_TIME, 4-5, 14-56
ASIC
  Application Specific Integrated Circuit, 1-1
design flow, 6-1
gate sizing, 6-10
XlibCreator, 1-9
ASYNC_VAL, attribute, Xilinx, 14-62
Atmel modgen library, 16-1
attribute, 2-11
  buffer_sig, 2-8, 7-18
dont_use, 6-14, 6-16
  input_drive, 4-4
  input_max_fanout, 4-4
  input_max_load, 4-4
  lut_max_fanout, 6-21, 14-5
  max_capacitance, 6-15
  max_fanout, 6-15, 7-15
  max_fanout_load, 6-20
  max_transition, 6-15
  max_transition_time, 6-15
  nobuff, 4-4, 6-14
  nopad, 2-6
  output_fanout, 4-4
  output_load, 4-3
  pad, 2-9
  required_time, 4-5, 4-6, 5-5
automatic constraining of designs, 5-15

B
back-annotation, 3-7
  Actel, 7-4
  Altera FLEX, 9-2
hold checks, 3-8
setup checks, 3-8
TimingChecksOn, 3-9
Xilinx, 14-2
BASE, attribute, Xilinx, 14-62
batch mode, 1-4
delay and area, 4-2
derating options, 3-6, 6-12
device selection, 13-4
extract clock enables, 11-12
FAST output, 14-80
global buffers, 14-48
I/O registers, 14-63
include gates, 11-28
locking lcells, 10-3
logic replication, 7-16
LUT mapping, 2-3
map complex I/O cells, 2-7, 7-22
mapping, 9-10
mapping and optimization, 11-9, 12-8
max fanin, 10-9
max fanout, 7-15
-nocomplex_ios, 7-22
-nomap_global_bufs, 7-20
optimize options, 14-19
part number, 14-64
preserve dangling nets, 14-72
quadrant clock buffers, 7-20
tristate, 12-4
write CLBs, 14-6
write EQN symbols, 14-65
BDBUF, Xilinx, 14-71
black box, 9-28, 11-24, 11-25, 14-26
BLKNM, attribute, Xilinx, 14-62
Boolean mapping
algorithm, 2-2, 7-5
boundary scan, 14-83
BSCAN, Xilinx, 14-67, 14-83
BUFFER_SIG, 2-8, 2-9, 7-18, 7-20, 9-32, 10-8, 14-46, 14-49
buffering of logic, 4-4, 7-16

C
cae2adl, Actel, 7-10
CAP, attribute, Xilinx, 14-62
capacitance interconnect, 3-3
capacitance load
best, balanced, worst, 6-13
carry
chain, Altera FLEX, 9-5, 9-11
chain, Xilinx, 14-18
lookahead generator, Xilinx CPLD, 13-2
propagation through, 14-18
cascade
chain, Altera FLEX, 9-5
gate, Altera FLEX, 9-7
routing resource, 9-2
CHIP_PIN_LC, Altera FLEX, 9-33
CLB
architecture, 14-2, 14-13
CLB packing, 14-13
packing, 14-6, 14-7, 14-10, 14-13, 14-82
CLKBUF, Actel, 7-17
clock
CLOCK_CYCLE, 4-6
CLOCK_OFFSET, 4-6
common zero, 6-11
cycle, 14-55
delay, 6-11
gate, Altera FLEX, 9-7
enable, 14-14
enable detection, Altera FLEX, 9-5
enable logic, 14-32
enable recognition, 14-33
ideal, 6-11
leading edge, 4-6
manual mapping, Xilinx, 14-46
offset, 14-55
pulse, 4-6
pulse width, 14-55
reference, 4-6
setup time, 4-6, 14-32
clock buffer
Actel, 7-5, 7-16
HCLKBUF, 2-9
QuickLogic, 12-5
clock buffer, Xilinx
ACLK, 14-44
BUFG, 14-44
BUFGP, 14-45
BUFGS, 14-45
GCLK, 14-44
clock skew, 6-11
clock timing constraints, 14-56
CLOCK_CYCLE, 4-6, 14-56
CLOCK_OFFSET, 4-6, 14-56
CMOS, attribute, Xilinx, 14-62
combinational logic loops, 12-4
combinational module, Actel, 7-2
command
   CLOCK_OFFSET, 6-11
command line
   connect_path, 4-4
disconnect_path, 4-4
   optimize_timing, 14-12
   pack_elbs, 14-13
command line options
   -area, 4-3
   -auto, 9-12, 14-19
   -delay, 5-2
   -fast, 9-12, 14-19
   -fastest, 9-12, 14-19
   -force, 5-15, 7-9, 11-10
   -max_fanin <integer>, 10-9
   -max_pt <integer>, 10-9
   -nologic_rep, 7-16
   -nomap_global_bufs, 14-48
   -part <part_number>, 11-32, 13-4
   -simple_port_names, 7-10, 9-28, 9-34
   -small, 9-12, 14-19
   -smallest, 9-12, 14-19
   -through, 11-11, 14-13
   -tristate_map, 14-29
   -use_qclk_bufs, 7-20
complex I/O mapping, 8-9, 9-42
complex I/Os
   Actel, 7-2
   checker, 7-21
   Xilinx, 14-7
CONFIG, attribute, Xilinx, 14-62
configurable logic block, 14-2
constraint
   multicycle path, 4-8
   primary inputs, 4-5
conventions
   manual, 1-11
counter
   Actel, 7-5
   Altera FLEX, 9-11
   ORCA, 11-11
   Xilinx, 14-18
CPLD
   arithmetic operator, 13-6
   complex programmable logic device, 1-1
decrementer, 13-6
I/O buffers and pads, 13-4
incrementer, 13-6
pin location, 13-6
relational operator, 13-6
supported devices, 13-7
XACT-CPLD tools, 13-2
Xilinx architecture, 13-2
critical path
   constraints, 5-10
   minimize, 5-5
   negative slack, 5-6
   specified, 5-2
   unconstrained, 5-9
critical path reporting, 3-1, 3-2, 5-10, 6-11
CYMODE, attribute, Xilinx, 14-62
Cypress
   Flash370i, 16-1
   Ultra37000, 16-1
D
data path synthesis
   Actel, 7-6
   Altera FLEX, 9-11
   Xilinx, 14-18
DECODE, attribute, Xilinx, 14-62
decrementer
   Altera FLEX, 9-11
   ASIC, 6-6
   ORCA, 11-11
   Xilinx, 14-18

Index-3
DEF, attribute, Xilinx, 14-62
default_max_transition, attribute, 6-20
Defparms, Verilog, 9-22
delay model
  connect, 3-3
  input ramp, 3-3
  intrinsic, 3-3
  linear, 3-2, 3-3, 6-11
  nonlinear, 3-2, 6-11
  piece wise linear, 3-2
  transition, 3-3
delay modeling issues
  clock delays, 6-11
  clock skew, 6-11
  derating factors, 6-12
delay models
  linear, 6-11
  piece-wise, 6-11
DEPTH, attribute, Xilinx, 14-62
derating factors, 6-12
  Actel, 7-29
  pASIC1, pASIC2, pASIC3, 12-5
design
  present, 5-4
  rule checker, 7-5
  rule violation, Actel, 7-21
design partition
  block size, 6-5
  critical paths, 6-5
  glue logic, 6-5
  inputs and outputs, 6-5
  logic grouped, 6-5
  sub-designs, 6-5
designer, Actel, 7-13
DesktopASIC, 1-5
devices supported
  Actel, 7-29
  Altera FLEX, 9-46
  Altera FLEX APEX 20K, 9-46
  Altera MAX, 10-11
  Atmel, 16-1
  Cypress, 16-1
  Lattice/Maxplus, 16-2
  Minc Synario, 16-3
ORCA, 11-43
Quick Logic, 12-9
Xilinx, 14-90
Xilinx CoolRunner, 14-1
Xilinx CPLD, 13-7
Xilinx Virtex, 15-16, 15-17
DFF, gate, 14-14
DIN/DOUT Attributes, 11-30
DOUBLE, attribute, Xilinx, 14-62
DRC
  automatic resolving, 6-16
  checking and fixing, 6-14
  design rule checker, 1-5
  NOBUFF attribute, 6-16
  real number value, 6-15
  resolving algorithm, 6-14
DRC attribute
  DONT_USE, 6-16
  nobuff, 6-16
drive
  capability, 4-3
  default, 4-4
  input, 4-4
  requirement, 4-3
  resistance, 3-3
DUTY_CYCLE, attribute, Xilinx, 14-62
DWAND, component, 14-26
E
edf extension, 10-2
edge decoders, Xilinx, 14-26
EDIF
  bus, 9-34
  file, MAX+PLUS II, 10-10
  format, QuickLogic, 12-3
  Input, 9-36
  M1 flow, 14-2
  netlist, 9-2, 9-34, 10-2, 11-24, 14-1
  output, 7-10
EDIF_INPUT_LMF1, Altera MAX, 10-5
EDIF_INPUT_USE_LMF1, Altera MAX, 10-5
embedded array block (EAB), 9-2, 9-29
exemplar.lmf file, 9-35
expander, Altera MAX, 10-7

F
fanin limit, Altera MAX, 10-9
fanin limited optimization, 9-2, 14-7
fanout violations, 6-17, 7-13
fast Xilinx outputs, 14-79
FAST, attribute, Xilinx, 14-62
features
   Altera FLEX, 9-40
   QuickLogic, 12-2
field-programmable system chips
   FPSC, 11-41
   leonardo_scripts, 11-41
   ORCA, 11-41
FIFO, Altera FLEX, 9-2
FILE, attribute, Xilinx, 14-62
FLEX architecture, 9-5
flip-flop
   enable, Altera MAX, 10-7
FLOAT_VAL, attribute, Xilinx, 14-62
FMAP, 5-9, 14-7, 14-14
foundry, ORCA, 11-24, 11-27, 11-29, 11-33
   DELAYMODE, 11-33
   DRIVE, 11-33
   IBUFLEVEL, 11-33
   LOAD, 11-33
   OBUFLEVEL, 11-33
FPGA
   field programmable gate arrays, 1-1
FPSC (Field Programmable System Chips) for
   ORCA, 11-41
FREQ, attribute, Xilinx, 14-62
FSM encoding, 6-21, 9-45
function generator, 14-10, 14-14
functional block, 14-13

G
gate sizing, 6-10
generate_timespec, command, 14-60
genmem

Altera, 9-28
   support, 9-28
global signals, 9-32
GLOBAL, Altera, 9-32, 10-8
GLOBALRESET, Xilinx, 14-70
GND, pin, 9-36
GUI
   graphical user interface, 1-1
GUI options
   add I/O pads, 11-9
   advanced settings, 2-3, 2-4, 2-5, 2-7, 7-16, 7-20, 7-22, 9-10, 10-3, 11-9, 12-8, 14-6, 14-48, 14-55, 14-63, 14-65, 14-80
   ASIC technology settings, 3-6, 6-12
   assign global SR, 11-14, 14-21
   detect the global set/reset signal, 11-14, 14-21
   do logic replication, 7-16
   do not lock LCells, 9-10
   extended optimization effort, 10-7
   extract clock enables, 11-12
   generate TimeSpec, 14-55
   map clock buffers, 2-4, 2-7, 7-19, 7-20, 14-48
   map complex I/O cells, 2-7, 7-22
   map look up tables, 11-9
   map to 6-input LUTs, 2-5, 11-9
   map to CASCADEs during LUT mapping, 2-3, 9-10, 14-63
   max fanin, 10-9
   max fanout, 11-7, 11-9, 12-8, 14-6
   max fanout load, 7-15
   max PT, 10-9
   maximum area, 4-2
   maximum delay, 4-2, 5-3
   optimize, 4-2, 5-3
   optimize options, 9-12, 11-12, 14-19
   part, 14-64
   preserve dangling nets, 14-72
   process, 3-6, 6-12
   technology settings, 7-15, 10-9, 11-14, 11-32, 13-4, 14-21, 14-64
   temperature, 3-6, 6-12
   use FAST output buffers, 14-80
   voltage, 6-12
   write EQN symbols, 14-65
   XNF options, 14-72
H
hard macro, file (.HM), 14-76
HBLKNM, attribute, Xilinx, 14-13, 14-17, 14-62
HCLKBUF, Actel, 7-17
HDL
hardware description language, 1-1
mixing design languages, 1-6
second language, 1-6
hd1_array_name_style, variable, 9-28, 9-35
help
online, 1-9
hierarchy
auto-dissolve, 6-3
flatten, 6-3
preserve, 6-3
HMAP, 5-9, 14-7, 14-12, 14-14
HU_SET, attribute, Xilinx, 14-62
I
I/O
assign manually, 2-1, 2-11, 12-5, 14-49
buffers, Actel, 7-2
buffers, ORCA, 11-28
cells, 7-20
cells, Xilinx, 14-49
complex, 2-1
registered, 7-21, 14-49
set/reset, 7-16
size of, 2-1
I/O mapping, 2-8
Actel, 7-21
ORCA, 11-28
IBUF, Xilinx, 14-71
implementation
multi-level, 10-7
one level, 10-7
incrementer
Altera FLEX, 9-11
ASIC, 6-6
ORCA, 11-11
Xilinx, 14-18
INFF, Xilinx, 14-63, 14-71
INIT, attribute, Xilinx, 14-17, 14-43
INLAT, Xilinx, 14-63, 14-71
input/output block, 14-2
INPUT_DRIVE, 4-4
INPUT_MAX_FANOUT, 4-4
INPUT_MAX_LOAD, 4-4
inputs
primary, 4-4
register, 4-4
INREG, Xilinx, 14-71
interactive command line shell, 2-3, 2-4, 2-5, 2-7, 3-6, 4-
2, 5-3, 6-12, 7-15, 7-16, 7-20, 7-22, 7-40, 9-10,
9-12, 10-3, 10-9, 11-9, 11-12, 11-14, 11-28, 11-
32, 12-4, 12-8, 13-4, 14-6, 14-19, 14-21, 14-22,
14-29, 14-48, 14-63, 14-64, 14-65, 14-72, 14-80
interconnect
load, 3-3
modeling, 6-13
resistance, 3-3, 6-13
INTERNAL, attribute, Xilinx, 14-62
IO pad, 7-21
IOCLK, 7-25
IOPCL, 7-25
L
LABs, Altera, 10-2
latch
identical, 10-8
Lattice/Vantis, 16-2
LATTICE/VANTIS CPLD devices, 16-2
LCA architecture, 14-1, 14-2
LCA2XNF, Xilinx, 14-68
LCELL
primitive, 9-7
LeonardoInsight
design browser, 1-5
schematic viewer, 1-6
LeonardoSpectrum synthesis and technology, 1-8
Level 1 features, 1-2
Level 2 features, 1-3
Level 3
interactive command line shell, 1-11
Level 3 features, 1-4
libraries
  synthesis, 1-9
Library Mapping File (LMF), 10-5
LIBVER, attribute, Xilinx, 14-17, 14-62
LMF file, 10-5
load
  capacitive, 3-3, 6-13
  external, 4-3
  fanout, 4-4
  interconnect, 3-2, 3-3
  maximum, 4-4
  maximum fanout, 4-4
  output, 4-3
  requirements, 4-3
  unit, 4-3
  violation, 6-17, 7-5
LOC, attribute, Xilinx, 14-62
LOC, ORCA, 11-29
Lock LCells
  Altera, 9-9
logic
  array block, Altera, 9-2, 10-9
  buffering, 4-4, 7-16
  combinability, Actel, 7-12
  element, Altera, 9-2
  module, Actel, 7-2
  replication, 4-3
lookup table, 14-82
  Altera FLEX, 9-7
lookup table mapping, 2-2
  Altera, 2-2, 9-2, 9-7
  F5_MUX, gate, 14-11
  ORCA, 2-2
  Xilinx, 2-2, 14-10
LPM
  conventions, 9-5
  instantiation, 9-22
LPM Support for FLEX 10K, 9-13
LPM_RAM_DQ, component, 9-20
LPM_RAM_IO, component, 9-20

M
M1 back annotation flow, 14-80
M1 support, 14-81
macrocell, 10-7
main window
  banner, 1-11
map cascades, Altera, 9-9
MAP, attribute, Xilinx, 14-62
MAX architecture, Altera, 10-8
max cubes, Altera MAX, 10-9
max fanin, Altera MAX, 10-9
MAX+PLUS II, 9-36
MAX+PLUS II, Altera, 10-2, 10-10
MAX+PLUS II, Altera MAX, 10-5
MD0, input port, Xilinx, 14-68
MD1, output port, Xilinx, 14-68
MD2, input port, Xilinx, 14-68
MEDFAST, attribute, Xilinx, 14-62
Medslow, attribute, Xilinx, 14-62
MEMFILE, attribute, Xilinx, 14-62
memgen, Xilinx, 14-38
Minc Synario devices, 16-3
Model Sim/QuickHDL simulator, 1-4
module generator
  Altera MAX, 10-8
  ASIC, 6-6
  library, 14-18
  Xilinx, 13-6
move command, 9-34
multicycle path, 4-8
mux-based, optimization, 7-5

N
Neoprim library, 11-45
NOBUFF, 4-4
NODELAY, attribute, Xilinx, 14-62
NOOPT, Xilinx, 14-68
nowire_table, variable, 6-13
O

OBUF, Xilinx, 14-71
online, 1-8
operating conditions, 6-12
operators
arithmetic, 7-6, 9-11, 11-11, 13-6, 14-18
logical, 7-6, 14-18
relational, 7-6, 9-11, 11-11, 13-6, 14-18
shift, 7-6
optimization
flow, 5-1
passes, Altera MAX, 10-7
sequential, 9-32, 10-8, 12-4
optimize command
-target option, 10-2
optimize_timing command, 5-5, 7-9, 11-10
-force option, 7-9, 14-12
-through option, 5-5
option
HDL languages, 1-6
LeonardoInsight, 1-5
ORCA
architecture, 11-5
back-annotation, 11-2
carry
logic, 11-2
data path synthesis, 11-11
Field-Programmable System Chips (FPSC), 11-41
foundry, 11-2
INITV AL attribute, 11-25, 11-27
lookup table, 11-2
lookup table mapping, 11-6
macro library, 11-27
max fanout attribute, 11-8
NOMERGE, 11-6
PFU-MUX, 11-6
PFU-NAND, 11-6
preference file writer, 11-34
programmable
function unit, 11-2
RAM, 11-18
scuba, 11-24, 11-27
set/reset registers, 11-28
slew property, 11-30
SLIC, 11-2, 11-46
startup blocks, 11-13
timing optimization, 11-10
oscillator, Xilinx, 14-68
OUTFF, Xilinx, 14-63, 14-71
OUTFFT, Xilinx, 14-71
OUTPUT_FANOUT, 4-3
OUTPUT_LOAD, 4-3

P

P&RIntegrator, 1-2, 1-7
pack_clbs, command, 14-15
PackagedPower, 1-12
LeonardoSpectrum, 1-12
ModelSim, 1-12
Renoir, 1-12
PAD, 2-8, 7-20, 14-49
part number, ORCA, 11-32
path analysis, 4-4
PDF manual, 1-8
PERIOD, attribute, Xilinx, 14-62
PFU-XOR, ORCA, 11-6
PICSPEC, ORCA, 11-32
PICSPEED, ORCA, 11-33
pin assignment, Actel, 7-25
pin location
Altera, 9-33, 10-5, 10-6
ORCA, 11-29
Xilinx, 14-52
PIN_NUMBER, 10-5
pin_number, attribute, 7-26, 9-33, 11-29, 14-52
pipelined multiplier, 7-14, 9-42, 11-46, 14-63
place-and-route
Altera, 10-2
PLDs architecture, Altera MAX, 10-1
pop_design, command, 5-4
post synthesis RTL, 3-10
POWERUP, 14-20
PPR, Xilinx, 14-38
preserve_dangling_nets, variable, 14-73, 14-74
process, variable, 11-33
product terms, 10-2, 10-10
project
  save and restore, 1-6
propagation delay, 4-5
PTERM, 8-11
pullups and pulldowns
  verilog example, 14-88
  VHDL Example, 14-89
PULSE_WIDTH, 4-6, 14-56
push_design, command, 5-4

Q
QCLKBUF, Actel, 7-17
Quartus, 9-2
Quick Logic
devices supported, 12-9
QuickLogic
  BIIPAD, 12-6
  BIORPAD, 12-6
  BIPAD, 12-6
  buffer_sig, 12-5
  CKDPAD, 12-6
  CKPAD, 12-6
  CKTPAD, 12-6
clock buffer, 12-5
derating factors, 12-5
format QDIF, 12-3
HD2PAD, 12-6
HD3PAD, 12-6
HDDPAD, 12-6
HDIPAD, 12-6
HDPAD, 12-6
INPAD, 12-6
internal tristate, 12-4
max fanout, 12-8
OUTIPAD, 12-6
OUTORPAD, 12-6
OUTPAD, 12-6
PAD Command, 12-6
pASIC architecture, 12-2
pASIC, logic cell, 12-2
pin location, 12-7
SpDE, 12-2
TRIIPAD, 12-6
TRIORPAD, 12-6
TRIPAD, 12-6
QuickLogic architecture
  tristates, 12-4

R
RAM
  Altera FLEX, 9-2
dual-port, 9-2, 14-41
dual-port, ORCA, 11-27
inferencing, RTL, 9-23
instantiation, Altera FLEX, 9-5, 9-20
instantiation, ORCA, 11-24
instantiation, Xilinx, 14-38
modgen support, 11-20
Xilinx, 14-68
RAM_DQ
  component, 9-20
ram_example1, 7-7, 9-26, 11-19, 14-35
ram_example2, 7-8, 9-27, 11-20, 14-36
RAM_IO
  component, 9-20
RDCLK, Xilinx, 14-67
RDBK, Xilinx, 14-67
registers, 4-5
  identical, 10-8
relational operator, 7-6, 14-18
Renoir graphical design, 1-12
replicate logic, 7-16
report_delay command, 6-11
reports
  Actel, 7-28
  Altera FLEX, 9-32
  Altera MAX, 10-8
  ORCA, 11-29
  QuickLogic, 12-4
  Xilinx, 14-54
  Xilinx CPLD, 13-7
REQUIRED_TIME, 4-5, 4-7
RES, attribute, Xilinx, 14-62
reset
  active high, 11-15, 14-23
active low, 11-16, 14-24
signal, 14-14
right mouse button (RMB), 1-2
RLOC, attribute, in Xilinx, 14-62
RLOC_ORIGIN, attribute, Xilinx, 14-62
RLOC_RANGE, attribute, Xilinx, 14-62
ROM
  Altera FLEX, 9-2
  ORCA, 11-27
  Xilinx, 14-43, 14-68

S
SCHNM, attribute, Xilinx, 14-17, 14-62
SCLK, Altera, 10-8
SDF
  file, 3-7, 3-8
  HOLD, 3-10
  INTERCONNECT delays, 3-10
  IOPATH delays, 3-10
  PORT, 3-10
  SETUP, 3-10
SDF timing file, 6-1
SDF writer, 14-81
sequential
  module, Actel, 7-2, 7-17
set, signal, 14-14
simulation
  Actel, 7-25
  functionality, 3-8
  timing, 3-8
slew input, 3-2
SLOW, attribute, Xilinx, 14-62
speed grade, 11-33
standard features
  design browser, 1-8
  HDLInventor, 1-7
  P&RIntegrator, 1-6
  save and restore project, 1-6
static timing analysis, 3-1
supported devices
  Actel, 7-29
  Altera FLEX, 9-46

ORCA, 11-43
Xilinx, 14-90
SYNC_VAL, attribute, Xilinx, 14-62
syntax
  constraint file, 4-3, 4-5, 4-6
  interactive command line shell, 4-5, 4-8
  VHDL, 4-7
SYSTEM, attribute, Xilinx, 14-62

T
TCK, input port, Xilinx, 14-68
TDI, input port, Xilinx, 14-68
TDO, output port, Xilinx, 14-68
THI, attribute, Xilinx, 14-62
time
  arrival, 3-2
  required, 3-2
TIMEGRP, 14-56, 14-57
TIMESPEC, 14-55, 14-56, 14-57
timing
  required, 4-5
timing analysis, 3-2, 6-11
  annotated XNF netlist, 3-7
  arrival time, 4-5, 5-2
  constraint, 4-5, 5-2
  critical path, 5-2
  end point, 5-2, 5-6
  longest path, 3-2
  negative slack, 3-2
  post place-and-route, 3-5
  reports statistics, 5-6
  routing delay, 3-5
  start point, 4-4
static, 6-11
  worst-case path delay, 5-2
timing constraint
  arrival_time, 6-10
  input_drive, 6-10
  output_load, 6-10
  required_time, 6-10
timing optimization, reporting
  area, 5-6
  longest path, 5-6
most critical slack, 5-6
sum of negative slacks, 5-6
TLO, attribute, Xilinx, 14-62
TMS, input port, Xilinx, 14-68
transition time violations, 6-20
tristatable output, 10-2
tristate, internal
Xilinx, 14-28, 14-71
TTL, attribute, Xilinx, 14-62

U
U_SET, attribute, Xilinx, 14-62
ungroup command
-all option, 14-64
-hier option, 14-64
USE_RLOC, attribute, in Xilinx, 14-62

V
variable
max_fanout_load, 6-17, 6-18
propagate_clock_delay, 6-11
variables
altera_use_cascades, 2-3, 9-10
dont_lock_lcells, 9-10
force_user_load_values, 6-20
hdl_array_name_style, 7-10
infer_gsr, 14-20
max_transition, 6-20
maxdly, 4-3
tristate_map, 14-29
use_f5map, 2-4
use_f6_lut, 2-5, 11-9
xlx_preserve_gsr, 14-22
xlx_preserve_gts, 14-22
VCC, pin, 9-36
vendor tools, 8-11
Verilog, 14-18
Verilog designs, 14-1
VHDL, 14-18
VHDL attributes, 4-6
VHDL designs, 14-1

violation
max_capacitance, 6-15
max_fanout, 6-15
max_transition, 6-15
Virtex
devices, 15-16
devices supported, 15-16
libraries, 15-18
synchronous set/reset, 15-3
wide cluster of MUXs, 15-2
Virtex family for Xilinx, 15-1
VITAL library, 3-8
level 0 compliant, 3-8
level 1 compliant, 3-8

W
web site, 1-9
wir2xnf, 14-75
wire load models, 14-82
wire load tables, 6-13
wire resistance
balanced case, 3-3, 6-13
best case, 3-3, 6-13
worst case, 3-3, 6-13
wire_table, variable, 6-13
wire_tree, variable, 6-13
wireload
table, 6-13
write command, 10-2

X
XACT (XNF) flow, 14-2
XACT, software, 14-6
XBLOX, 14-76
components, 14-18
xblox3 library, 14-76
xblox4 library, 14-76
XBLOX program, 14-76
Xilinx
P&R, 14-82
Xilinx CPLD
devices supported, 13-7
XlibCreator, 1-9
xlx_preserve_gsr, variable, 14-22, 14-70
xlx_preserve_gts, variable, 14-22, 14-70
XNF, 14-52, 14-55
  EXT record, 14-75
  hierarchical, 14-64
  mapped, 14-10
  NOOPT, 14-68
  output netlist, 14-64
  output, EQN symbols, 14-66
  preserving logic, 14-68
  reader, 14-22
  SIG record, 14-75
  simulatable, 14-68
  sourceless signal, 14-72
XNFMERGE, Xilinx, 14-38