Copyright

All Rights Reserved

Trademarks

Exemplar Logic® and its Logo are registered trademarks of Exemplar Logic, Inc.;
Galileo™, Galileo Extreme™, Leonardo™, Galileo FS™ and MODGEN™ are trademarks of
Exemplar Logic, Inc.; Extreme Technology, FAST Synthesis and Synthesizing the next Millennium
are servicemarks of Exemplar Logic, Inc.
V-System/VHDL™ and V-System/Verilog™ are trademarks of Model Technology, Inc.
Verilog® and Verilog-XL® are registered trademarks of Cadence Design Systems, Inc.
All other trademarks remain the property of their respective owners.

Disclaimer

Although Exemplar Logic, Inc. has tested the software and reviewed the documentation, Exemplar
Logic, Inc. makes no warranty or representation, either express or implied, with respect to this soft-
ware and documentation, its quality, performance, merchantability, or fitness for a particular purpose.

Exemplar Logic, Inc.
6503 Dumbarton Circle
Fremont, CA 94555
Telephone: 800-632-3742
email: info@exemplar.com

Part No: EL-60015-R
## Contents

1. **Introduction** ......................................................... 1-1
   - About Leonardo’s Manuals ........................................ 1-2
   - Installation Guide ................................................ 1-2
   - User’s Guide ...................................................... 1-2
   - Synthesis and Technology Guide ................................. 1-3
   - Command Reference Guide ..................................... 1-3
   - HDL Synthesis Guide ............................................. 1-3

   More About Leonardo ............................................. 1-3
   - Available Online ................................................ 1-3
   - Available Help ................................................... 1-3
   - Available Libraries ............................................ 1-3
   - Screen Shots, Reports, and Filenames ....................... 1-4
   - Time Module License ........................................... 1-4

   System Requirements and Runtime ............................... 1-4

2. **Technology Mapping** ............................................... 2-1
   - Before Beginning ................................................ 2-2
   - Boolean Mapping ................................................ 2-2
   - Lookup Table Mapping ......................................... 2-2
   - Command Line and GUI Options ............................... 2-2
   - I/O Mapping ....................................................... 2-7
   - Command Line and GUI Options ............................... 2-7
   - Manual I/O Mapping ............................................. 2-8
   - Setting Timing Constraints with GUI Constraint Editor ... 2-8
   - PAD and BUFFER_SIG Commands .............................. 2-9

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 ........................................... 3-7
4. Timing Constraints ...................................... 4-1
   Before Beginning ........................................ 4-1
   Setting Global Constraints ............................... 4-1
   Setting Constraints on Individual Signals .............. 4-2
       Load and Drive Specifications ....................... 4-2
       Timing Specifications ............................. 4-3
   Usage of Timing Constraints in Leonardo ............... 4-6
   New Constraint Features .............................. 4-8
       Constraint Driven Timing Optimization .......... 4-9
       Support for Maximum Frequency .................. 4-9
       Support for Multicycle Path ....................... 4-9
5. Timing Optimization ................................. 5-1
   Before Beginning ........................................ 5-1
   Optimization Flow ..................................... 5-1
   Delay Optimization .................................... 5-2
       Setting Timing Constraints ......................... 5-3
       User Control ......................................... 5-5
   Statistics Report ...................................... 5-6
   Automatic Constraining of Designs ..................... 5-14
6. Reports ................................................ 6-1
   Xilinx Area/Delay Example Reports .................... 6-1
       Area Reporting ...................................... 6-1
       Delay Estimates ..................................... 6-4
   Altera FLEX Area/Delay Example Reports ............... 6-5

Leonardo Synthesis and Technology Guide
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fanout Violations and Load Violations</td>
<td>8-5</td>
</tr>
<tr>
<td>Reduced Memory Consumption</td>
<td>8-5</td>
</tr>
<tr>
<td>Enhancements to Actel Technologies</td>
<td>8-6</td>
</tr>
<tr>
<td>Modgen Enhancements</td>
<td>8-6</td>
</tr>
<tr>
<td>Synthesis and Optimization Features</td>
<td>8-6</td>
</tr>
<tr>
<td>Technology Mapping</td>
<td>8-7</td>
</tr>
<tr>
<td>Data Path Synthesis</td>
<td>8-7</td>
</tr>
<tr>
<td>Modgen</td>
<td>8-7</td>
</tr>
<tr>
<td>Constraint-Driven Timing Optimization</td>
<td>8-11</td>
</tr>
<tr>
<td>Interfacing with Actel’s ACTGen</td>
<td>8-11</td>
</tr>
<tr>
<td>Additional Synthesis Features</td>
<td>8-14</td>
</tr>
<tr>
<td>Logic Combinability</td>
<td>8-14</td>
</tr>
<tr>
<td>Fanout and Load Violations</td>
<td>8-15</td>
</tr>
<tr>
<td>Using Global Clock Buffers</td>
<td>8-15</td>
</tr>
<tr>
<td>Design I/O</td>
<td>8-18</td>
</tr>
<tr>
<td>I/O Mapping</td>
<td>8-18</td>
</tr>
<tr>
<td>Complex I/O Design Rule Checker and Modifier</td>
<td>8-18</td>
</tr>
<tr>
<td>Assigning Device Pin Numbers</td>
<td>8-21</td>
</tr>
<tr>
<td>Reporting</td>
<td>8-24</td>
</tr>
<tr>
<td>Process Derating Factors</td>
<td>8-25</td>
</tr>
<tr>
<td><strong>9. Altera FLEX Synthesis</strong></td>
<td>9-1</td>
</tr>
<tr>
<td>Before Beginning</td>
<td>9-1</td>
</tr>
<tr>
<td>Altera FLEX Architecture</td>
<td>9-2</td>
</tr>
<tr>
<td>Design Flow</td>
<td>9-2</td>
</tr>
<tr>
<td>Synthesis and Optimization Features</td>
<td>9-5</td>
</tr>
<tr>
<td>Fanin Limited Optimization</td>
<td>9-5</td>
</tr>
<tr>
<td>Lookup Table Mapping</td>
<td>9-7</td>
</tr>
<tr>
<td>Data Path Synthesis and Modgen Implementation</td>
<td>9-9</td>
</tr>
<tr>
<td>LPM Support for FLEX 10K</td>
<td>9-11</td>
</tr>
<tr>
<td>Genmem Support</td>
<td>9-20</td>
</tr>
<tr>
<td>Embedded Array Block (EAB) Support for FLEX 10K</td>
<td>9-22</td>
</tr>
<tr>
<td>Additional Optimization Features</td>
<td>9-24</td>
</tr>
</tbody>
</table>
Sequential Optimization .................................................. 10-8
Reporting ................................................................. 10-8
Design I/O ................................................................. 10-8
Global Signals ........................................................... 10-8
Additional Options ...................................................... 10-9
Max Fanin and Max Cubes ........................................... 10-9
Using MAX Designs as Input to Leonardo ...................... 10-10
EDIF Input ............................................................... 10-10

11. Lucent ORCA Synthesis ............................................. 11-1
   Before Beginning .................................................... 11-1
   FPGA Architecture ................................................ 11-2
   Synthesis and Optimization Features ......................... 11-5
      Optimization Algorithms ..................................... 11-5
      Fanin Limited Optimization ................................ 11-5
      Lookup Table (LUT) Mapping ............................... 11-6
      Leonardo Variables to Control Optimization and Mapping 11-8
      Constraint-Driven Timing Optimization .................... 11-10
      Data Path Synthesis (Modgen) ............................. 11-10
      Using ORCA Architectural Features ....................... 11-11
   ORCA Reporting .................................................... 11-28
   Generation of ORCA Foundry Properties from Leonardo .... 11-28
      Pin Number Property - LOC ................................ 11-28
      Slew Property .................................................... 11-29
   ORCA DIN/DOUT Attributes ..................................... 11-29
      Part Number - PICSPEC ..................................... 11-31
      Speed Grade - PICSPEED .................................... 11-31
      ORCA Foundry Properties on IOBs ......................... 11-31
      Other Foundry Properties .................................. 11-32
   ORCA Preference File Writer ..................................... 11-32
      Preference File ................................................ 11-37
      Additional Writing Preference File Features ............. 11-37
Devices Supported for Lucent ORCA FPGA Families ......................... 11-39
New Features for ORCA .................................................. 11-42
  ORCA Preference File ............................................... 11-42
  ORCA3C and ORCA3T Multicycle Support .......................... 11-43
  Mapping to ORCALUTs ............................................. 11-44

12. QuickLogic Synthesis ............................................... 12-1
  Before Beginning .................................................... 12-1
  FPGA Architecture ................................................ 12-2
    The pASIC Logic Cell .......................................... 12-2
    I/O Buffers ..................................................... 12-2
  New Features for QuickLogic ..................................... 12-2
  Design Flow ....................................................... 12-2
  Optimization Style Points ....................................... 12-3
    Quality of Optimization ...................................... 12-3
    Combinational Logic Loops ................................... 12-4
    Sequential Optimization ...................................... 12-4
    Internal Tristates ............................................. 12-4
  QuickLogic Reports ................................................ 12-4
  Process Derating Factors ........................................ 12-4
  Design I/O ........................................................ 12-5
    Manual Assignment of I/O Buffers ............................. 12-5
    Assigning Pin Locations .................................... 12-7

13. Xilinx CPLDs Synthesis ............................................. 13-1
  Before Beginning .................................................... 13-1
  Xilinx CPLD Architecture .......................................... 13-2
  Design Flow ........................................................ 13-2
  Synthesis and Optimization Features ............................... 13-3
    Special Synthesis Feature .................................... 13-3
  Other Synthesis Features .......................................... 13-4
    Device Selection ................................................ 13-4
    I/O Buffers and Pads .......................................... 13-4
    Pin (LOC) Assignment ........................................... 13-4
## 14. Xilinx LCA Synthesis

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Beginning</td>
<td>14-2</td>
</tr>
<tr>
<td>Xilinx LCA Architecture</td>
<td>14-2</td>
</tr>
<tr>
<td>Configurable Logic Block</td>
<td>14-2</td>
</tr>
<tr>
<td>Input/Output Block</td>
<td>14-2</td>
</tr>
<tr>
<td>Synthesis Design Flows</td>
<td>14-2</td>
</tr>
<tr>
<td>Design Flow Guidelines</td>
<td>14-5</td>
</tr>
<tr>
<td>Synthesis and Optimization Features</td>
<td>14-7</td>
</tr>
<tr>
<td>Fanin Limited Optimization</td>
<td>14-7</td>
</tr>
<tr>
<td>Lookup Table (LUT) Mapping</td>
<td>14-10</td>
</tr>
<tr>
<td>Constraint-Driven Timing Optimization</td>
<td>14-11</td>
</tr>
<tr>
<td>CLB Packing for Xilinx XC4000 Technology</td>
<td>14-12</td>
</tr>
<tr>
<td>Data Path Synthesis (Modgen)</td>
<td>14-17</td>
</tr>
<tr>
<td>Using Xilinx Architectural Features</td>
<td>14-20</td>
</tr>
<tr>
<td>Design I/O</td>
<td>14-50</td>
</tr>
<tr>
<td>Complex I/O Mapping</td>
<td>14-51</td>
</tr>
<tr>
<td>Pin Location Assignment</td>
<td>14-54</td>
</tr>
<tr>
<td>Reporting</td>
<td>14-56</td>
</tr>
<tr>
<td>Using Timespecs</td>
<td>14-56</td>
</tr>
<tr>
<td>User Interaction</td>
<td>14-57</td>
</tr>
<tr>
<td>TIMESPECs</td>
<td>14-58</td>
</tr>
<tr>
<td>Using Xilinx Attributes</td>
<td>14-65</td>
</tr>
<tr>
<td>Additional Xilinx-Specific Options</td>
<td>14-67</td>
</tr>
<tr>
<td>Using Enabled D Type Flip-Flops</td>
<td>14-67</td>
</tr>
<tr>
<td>Use of Registered Logic in IOBs</td>
<td>14-67</td>
</tr>
<tr>
<td>Part Number</td>
<td>14-68</td>
</tr>
<tr>
<td>Writing XNF</td>
<td>14-68</td>
</tr>
<tr>
<td>Using Xilinx Designs as Input to Leonardo</td>
<td>14-71</td>
</tr>
</tbody>
</table>
Welcome to Leonardo with extreme technology. Leonardo is a versatile and interactive logic synthesis, optimization, and analysis tool developed to allow the use of technology-independent design methods for Field Programmable Gate Arrays (FPGAs), Complex Programmable Logic Devices (CPLDs) and ASICs. You can efficiently and economically consolidate multiple designs into one design and preserve and manipulate the design hierarchy. The hardware description languages are used to implement the design.

Leonardo utilizes the most powerful state-of-the-art optimization technology to guarantee high-quality results for any targeted FPGA technology. Leonardo offers the following distinct features:

- Certified FPGA flows: Netlists and directives generated by Leonardo successfully pass through the back-end tools; and post place and route timing information can be read and back-annotated by Leonardo's Time Module for timing and logic verification purposes.
- FPGA architecture specific module generation
- FPGA architecture specific optimization
- Accurate architecture specific timing analysis
- RTL and gate-level post-synthesis verification
- Timing back annotation
- Platform independence
- ASIC specific module generation
Leonardo optimizes your designs for area and speed, and accepts designs in netlist or VHDL and Verilog formats. Leonardo produces vendor specific netlists and design reports which estimate the performance of your design.

You can invoke Leonardo through a graphical user interface (GUI) on all platforms - Unix, Linux, and Windows 95 and NT. You can use Leonardo from the command line and in batch mode and script files. Leonardo is designed to give you easy access to three modules: Leonardo, Time Module, and Model Technology V-System/QuickHDL simulator.

Leonardo provides a top-down verification flow through VHDL or Verilog with an SDF timing file. A VITAL or Verilog simulation library is also provided. Leonardo is fully integrated with Model Technology, Inc. (MTI) simulation environment.

About Leonardo’s Manuals

The following four guides are available in print and online:

- Installation Guide
- User’s Guide
- Synthesis and Technology Guide
- Command Reference Guide
- HDL Synthesis Guide

Installation Guide

The Installation Guide provides you with the steps for multi-platform installation and FLEXlm License Administration.

User’s Guide

The User’s Guide allows you to start designing right away. Clear instructions are provided for the graphical user interface (GUI). In addition, GUI-managed options and switches are listed, and examples of reports and message logs are provided. The constraint file commands are documented.
**Synthesis and Technology Guide**

This Synthesis and Technology Guide provides you with details on synthesis and optimization. This guide explains the various design flows for synthesizing to FPGA technologies.

**Command Reference Guide**

The Command Reference Guide presents a list of commands and a list of variables.

**HDL Synthesis Guide**

The HDL Synthesis Guide presents you with “cookbook” information. This guide provides you with the design guidelines to achieve a circuit implementation that meets the timing and area constraints of a target, while using the high level abstraction of the VHDL or Verilog source code.

**More About Leonardo**

**Available Online**

The four guides are available for viewing online with the Adobe Acrobat Reader after Leonardo and the Adobe Acrobat Reader are installed from the CD-ROM. In addition, the guides can be viewed and printed with desktop utilities.

**Available Help**

When you run Leonardo, balloon help will automatically display when your cursor is positioned on certain fields. Most fields also offer context-sensitive help, which is accessed by tabbing to the field and pressing F1. The online manuals are also accessible from the Help menu after you have installed the Adobe Acrobat Reader.

**Available Libraries**

The Leonardo license automatically enables the synthesis libraries for most FPGA technologies. The VITAL libraries for FPGAs are available. Check Exemplar’s website at [www.exemplar.com](http://www.exemplar.com) for more information.
Screen Shots, Reports, and Filenames

The screen shots, reports, and filename examples in this Synthesis and Technology Guide 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.

Time Module License

You must purchase a Time Module license from Exemplar Logic, Inc., to use back-annotation flows with Leonardo. Back-annotation is supported through VITAL libraries for the following technologies:
- Xilinx XC4000/E/EX/XL/XV (through M1 software only) and XC5200
- Altera FLEX 6K, FLEX 8K, FLEX 10K
- Actel Act1/2/3, 3200DX, 1200XL

System Requirements and Runtime

Table 1-1, System Requirements, shows the recommended system requirements for proper operation of Exemplar synthesis tools. The actual system requirements depend on your design and coding style.

Table 1-1. System Requirements

<table>
<thead>
<tr>
<th>Design Size</th>
<th>RAM, MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Gates</td>
<td>Look Up Tables (LUTs)</td>
</tr>
<tr>
<td>15,000</td>
<td>1,100</td>
</tr>
<tr>
<td>15,000 to 75,000</td>
<td>4,000</td>
</tr>
<tr>
<td>75,000</td>
<td>5,000</td>
</tr>
</tbody>
</table>

Note: A system with less than the recommended requirements may run slower due to memory swapping.

Table 1-2, Runtime, shows the speed of Leonardo for several technologies. Runs were done on a Pentium-Pro 200 Mhz machine with 128 MB RAM.
Table 1-2. Runtime

<table>
<thead>
<tr>
<th>Design Size</th>
<th>Technology</th>
<th>Run Time, cpu seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500 lookup tables</td>
<td>Xilinx 4e</td>
<td>150</td>
</tr>
<tr>
<td>900 lookup tables</td>
<td>Altera FLEX 10</td>
<td>80</td>
</tr>
<tr>
<td>350 lookup tables</td>
<td>Lucent ORCA2c</td>
<td>30</td>
</tr>
<tr>
<td>100 lookup tables</td>
<td>Altera FLEX 10</td>
<td>10</td>
</tr>
<tr>
<td>1300 logic modules</td>
<td>Actel Act3</td>
<td>170</td>
</tr>
</tbody>
</table>
This chapter describes the mapping techniques used in Leonardo 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 Leonardo in the default mode, all I/O signals are assigned pads. The pads are selected from the target technology library during Leonardo’s technology mapping phase. If more than one size of the same pad is available, Leonardo chooses the smallest pad size. If the target library contains complex I/Os - I/Os with registers in the I/O cell - then Leonardo maps these complex I/Os as required.

Each architecture has different constraints on the usage of complex I/Os. Currently, Leonardo checks for design rule violations during usage of complex I/Os for Act3 and Xilinx architectures. For other architectures and for manually assigned I/Os, you are responsible for the validity of the output design. You can also override the automatic assignments done by Leonardo 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
- I/O Mapping
- Manual I/O Mapping
Before Beginning

The synthesis information presented in this chapter introduces you to the technology chapters in this guide. This chapter assumes that you have read the User’s Guide. The HDL Synthesis and Command Reference guides provide additional information.

Boolean Mapping

Boolean mapping is a technology mapping technique that Leonardo applies to all Actel FPGAs to utilize the underlying Actel architecture. The technology gates are generated automatically 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

Leonardo uses lookup table mapping for the following LUT-based FPGAs.
- Xilinx XC3000, XC4000/E/EX/XL/XV and XC5200
- Altera FLEX 6K, FLEX 8K, and FLEX 10K
- Lucent ORCA 1C, 2C, 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.

Command Line and GUI Options

The following choices are available on the GUI to control mapping.
- Map Look Up Tables, Screen 2-1, Table 2-1
- Map Cascades, Screen 2-1, Table 2-1
- Use F5 Map Symbols, Screen 2-2, Table 2-1
- Map 6-Input LUT, Screen 2-3, Table 2-1
<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Option on/off</th>
<th>Command Line**</th>
<th>Available in Technologies</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Look Up Tables</td>
<td>on*</td>
<td>default (true)</td>
<td>All Listed LUT-based FPGAs</td>
<td>Controls LUT mapping.</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>lut_map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map to CASCADEs during LUT</td>
<td>on*</td>
<td>default (true)</td>
<td>Altera FLEX6K</td>
<td>Controls mapping logic to cascade gates for Altera FLEX.</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>flex_use_cascades</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use F5MAP Symbols</td>
<td>on</td>
<td>use_f5map</td>
<td>Xilinx XC5200</td>
<td>Controls mapping to F5MAP Symbols.</td>
</tr>
<tr>
<td></td>
<td>off*</td>
<td>default (false)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map to 6-input LUTs</td>
<td>on</td>
<td>use_f6_lut</td>
<td>ORCA1C</td>
<td>Controls mapping to 6-Input LUTs.</td>
</tr>
<tr>
<td></td>
<td>off*</td>
<td>default (false)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*GUI default options
**variables, options, constraint attributes, commands
Screen 2-1. Example of Altera FLEX 6K Output Options
Screen 2-2. Example of Xilinx 5200 Output Options
Screen 2-3. Example of ORCA 1C Output Options
I/O Mapping

During I/O mapping, Leonardo assigns PADs to all I/Os in the top level of a design. Leonardo can map input buffers, output buffers, tri-state buffers, bi-directional buffers, and complex I/O cells. Leonardo also maps global buffers for clock lines and high fanout input pads.

Command Line and GUI Options

There are several technology independent options in the GUI which control Leonardo’s I/O pad assignments. These GUI options affect the complete design, not just individual I/Os.

Add IO Pads

The ◆ chip and ◆ macro options are available for all technologies. As shown in Table 2-2 and Screen 2-4, the selection of these options enable all I/O Pad assignments in Leonardo.

Table 2-2. Results of Command Line and GUI I/O Mapping Options

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Option on/off</th>
<th>Command Line**</th>
<th>Available in Technologies</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>◆ chip</td>
<td>on*</td>
<td>-chip default (true)</td>
<td>All</td>
<td>Assigns I/O pads around perimeter of the design.</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>◆ macro</td>
<td>on</td>
<td>-macro</td>
<td>Actel, Xilinx, ORCA</td>
<td>Output design is only a part of a design. I/O buffers are not added around the perimeter of design.</td>
</tr>
<tr>
<td></td>
<td>off*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n/a</td>
<td>default (false)</td>
<td>complex_ios</td>
<td></td>
<td>This variable controls use of complex I/Os.</td>
</tr>
</tbody>
</table>

*GUI default options
**variables, options, constraint attributes, commands
Screen 2-4. Example of Altera FLEX 8K Output Options

**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
- Instantiating Pads in VHDL or Verilog

**Setting Timing Constraints with GUI Constraint Editor**

The Constraint Editor is invoked from the Tools menu. Refer to the User’s guide for more information.
**PAD and BUFFER_SIG Commands**

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 Leonardo shell. The **PAD** command is the recommended way to assign pads, since it is the most general and can be used to assign any pad that Leonardo is able to map. In addition, connections between the pins on the **PAD** and the design signals do not need to be specified. The **PAD** command is limited, however, and only works on inputs and outputs.

The **BUFFER_SIG** command can be used to assign buffers only, but it can be used on internal signals (internal clock buffers) as well. The **PAD** and **BUFFER_SIG** commands can only assign mappable cells. A mappable cell is any cell that Leonardo is able to automatically instantiate in the output netlist. For FPGA technologies, all I/O cells are mappable. For ASIC technologies, typically only the simplest I/O cells are mappable. However, other cells can be instantiated through component instantiation, as discussed in the above in section.

**PAD Command**

This is the recommended way to manually assign pads. The command can be used from the Leonardo shell or from VHDL.

When using this command from the Leonardo shell, the syntax is:

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

*Signal 1 . . . signal n* are the names of I/O signals on which you want to instantiate the above I/O pad. The I/O pad must be a mappable gate from the target library. The **PAD** command will not work on non-mappable gates. To instantiate the I/O pad, Leonardo will automatically make connections between the gate’s pins and the design’s signals.

In the example below, the ACTEL Act3 FPGA device family is the target technology.

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

```
attribute PAD of signal-name: signal is pad-name
```

*signal-name* is the I/O on which you want to assign the pad with name *pad-name*.

**BUFFER_SIG Command**

This command will only work for I/O buffers (one input, one output). When using this command from Leonardo's shell, the syntax is:

```
BUFFER_SIG buffer_name signal_name
```

In the examples below, 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 which were originally driven by \( rstn \) will be driven by the output pin of the clock buffer (clkint). When using this command from VHDL, the syntax is:

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

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

---

*Leonardo Synthesis and Technology Guide*
The following example is a design using the Actel Act3 architecture. The example is written in VHDL and the I/O pads are assigned automatically, except for two I/O pads which are assigned manually, using attributes.

The following example is a VHDL design using the Actel Act3 architecture. The I/O pads are assigned automatically, except for two I/O pads which are assigned manually, using attributes.
The VHDL file:

```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
signal sig1, o: std_logic;
begin
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 above, the signal pad has automatically been assigned to the ORECTL complex I/O. Since the register is not using the enable signal in the VHDL design, Leonardo ties the register’s enable to GND. 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 Leonardo cannot find the instantiated component and treats the component as a black box. This causes Leonardo to add additional I/O buffers on the I/O pins of the component.
This chapter describes:

- Before Beginning
- Static Timing Analysis
- Back Annotation
- Post Synthesis RTL (Register Transfer Level) Simulation

**Before Beginning**

The information presented in this chapter introduces you to the technology chapters in this guide. This chapter assumes that you have read the User’s Guide. Refer also to the HDL Synthesis and Command Reference Guides for information.

**Static Timing Analysis**

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

Leonardo can also generate a critical path report showing a specific set of critical paths in the design. These critical paths can also be received in the schematic browser. In addition, Leonardo 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. However, timing analysis does not provide the comprehensive functional and dynamic simulation capabilities of a simulator.

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 (the 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
  
  $D_{\text{slope}} = D_{\text{transition}} \times S_s$
  
  $S_s$ = 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 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 Leonardo 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. Leonardo 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.

Nominal Parameter in Library

Leonardo recognizes the nominal process, temperature, and voltage parameters in a technology library with the following names.

- Process - nominal_process : string
- Temperature - nominal_temp : float
- Voltage - nominal_volt : float
You can define process, temperature, and voltage on Leonardo’s command line as shown in Table 3-1, or on the GUI as shown in Screen 3-1.

Table 3-1. GUI Derating Options

<table>
<thead>
<tr>
<th>GUI Options</th>
<th>Command Line Options</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td><code>set temp &lt;celsius&gt;</code></td>
<td>Timing information is derated during delay computations for this temperature.</td>
</tr>
<tr>
<td>Voltage</td>
<td><code>set voltage &lt;volts&gt;</code></td>
<td>Timing information is derated during delay computations for this voltage.</td>
</tr>
<tr>
<td>Process</td>
<td><code>set process &lt;name&gt;</code></td>
<td>Depends on target technology process. If worst is entered, then Leonardo looks for a parameter called worst_process in technical library.</td>
</tr>
</tbody>
</table>

Note: Leonardo must find the value of the process from the specified process, _process name, in technology library. For example, if you entered `process string as worst` then, Leonardo looks for a parameter called worst_process in the technical library and begins the derating on process.

Screen 3-1. Temperature, Voltage, Process Values
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 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 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. Refer to the Leonardo Command Reference for more details of different options and variables that affect critical path reporting.

Back-Annotation

Back-annotation is the process of inserting actual delay numbers into the network after place-and-route. Leonardo Time Module 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. Leonardo 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**

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 Leonardo commands may be:

```tcl
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. Exemplar’s Leonardo 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. Leonardo’s timing analyzer uses these actual delay numbers to determine post place-and-route critical paths.
5. Leonardo’s schematic viewer is used for graphical analysis of the design. Critical paths can also be highlighted in the schematic.
6. Leonardo 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 $\text{tipd}_{\text{in}}$ generic. The setup and hold timing checks are with respect to a clock edge and are annotated in generics that identify the clock edge, for instance, $\text{tsetup}_{\text{data clk}}_{\text{POSEDGE}}$ or $\text{thold}_{\text{data clk}}_{\text{NEGEDGE}}$. A generic, $\text{TimingChecksOn}$, also allows such timing checks to be turned off for increased performance.
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.
Standard Delay Format (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 tipd_ generic. This generic is identified by the port names associated with the SDF construct. For example, (PORT A (:5:) (:6:)) would annotate the tipd_A generic with rise and fall delays of 5 and 6.
- The IOPATH construct maps to the tpd_ generic. For example, (IOPATH A Y (:5:) (:6:)) would annotate the tpd_A_Y generic with rise and fall delays of 5 and 6.
- The SDF constructs SETUP and HOLD map to timing check generics tsetup_ and thold_ 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

Leonardo now supports writing simulatable models for gate level designs. Previously, if the design contained technology cells, Leonardo could not write the design out in a (simulatable) RTL description (VHDL or Verilog). Technology cells were considered black-boxes in the Leonardo design data base.

Technology cells are now filled with functional (RTL) information. This enables RTL output from technology netlists to write behavior for technology cells.
A new command, `unmap` was added. This 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 command `unmap` is available in the Leonardo Hierarchy menu.

The `-downto` option on the write command is provided to write descriptions with technology cells `downto` primitives (RTL).

**First Application**

Leonardo can now write RTL VHDL or Verilog for a mapped design. For example, your back-annotated design or a design that has already gone through the optimize command contains technology cells. You can now write the design(s) out in RTL VHDL or Verilog with one of the following two methods:

1. **Run command `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.

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

2. **Run write `-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.

   **Note:** `optimize` maps the design to a technology and `unmap` unmaps the design back to primitives. After `unmap`, the structure of the design may not be the same as before `optimize`. If you run another `optimize` after `unmap`, the result may be more undesirable than the result of the first `optimize` run.

**Second Application**

Leonardo can now write simulatable models for technology cells. To write one model of one library cell, change `present_design` to the `NETLIST` view of the technology cell of choice, and issue a write `-downto` PRIMITIVES command.
Use the following Tcl script to write all library cell models to separate VHDL files:

```tcl
# Set library name here:
set lib lsi300
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:

```tcl
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. `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
- Usage of Timing Constraints in Leonardo
- New Constraint Features

**Before Beginning**

The information presented in this chapter is intended to introduce you to the technology chapters in this guide. This chapter assumes that you have read the User’s Guide. Refer also to the HDL Synthesis and Command Reference Guides for information.

**Setting Global Constraints**

Global constraints are imposed to assist in optimizing the timing of the design. Use the `maxdly` variable to set global timing constraints:

```
set maxdly <n>
```
The `maxdly` variable specifies the maximum delay acceptable for any path in the optimized circuit. Leonardo is directed to search for the smallest circuit implementation which meets the specified timing constraint. The `maxdly` variable is used with `-area` option for `optimize` command.

```
optimize -area
```

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 Leonardo 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. Constraints can also be imposed from Leonardo’s shell or as properties in VHDL.

**Load and Drive Specifications**

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

```
OUTPUT_LOAD  value  output signal 1  ...  output signal n
```

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.

```
OUTPUT_FANOUT  value  output signal 1  ...  output signal n
```
The OUTPUT_FANOUT attribute tells the program the amount of external loading on an output of the design. \textit{Value} is the total number of fanout loads driven by the output.

\begin{verbatim}
INPUT_MAX_LOAD value input signal 1 ... input signal n
\end{verbatim}

The \texttt{INPUT_MAX_LOAD} attribute defines the maximum load that the synthesized circuit may present at an input to the design. The \textit{value} is the maximum number of unit loads allowed. If the synthesized circuit exceeds this amount of loading, Leonardo will buffer to reduce the load. In the macro mode, Leonardo will add buffers to meet the load constraint unless the input pins have \texttt{NOBUFF} attribute.

\begin{verbatim}
INPUT_MAX_FANOUT value input signal 1 ... input signal n
\end{verbatim}

The \texttt{INPUT_MAX_FANOUT} attribute defines the maximum fanout load that the synthesized circuit may present at an input to the design. \textit{Value} is the maximum number of total fanout loads allowed. During synthesis, Leonardo will buffer to reduce the load.

\begin{verbatim}
INPUT_DRIVE value input signal 1 ... input signal n
\end{verbatim}

The \texttt{INPUT_DRIVE} attribute specifies the additional delay per unit load for an input port. The \textit{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.

\textit{Timing Specifications}

Leonardo includes timing analysis routines to decide where to make an area/delay tradeoff 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 \texttt{connect_path} and \texttt{disconnect_path} commands.
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 will be 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 Leonardo’s shell, and/or in a VHDL source file, if the input is a VHDL file. The parameters which may be specified are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK_OFFSET</td>
<td>clock_signal_1 ... clock_signal_n</td>
</tr>
<tr>
<td>CLOCK_CYCLE</td>
<td>clock_signal_1 ... clock_signal_n</td>
</tr>
<tr>
<td>PULSE_WIDTH</td>
<td>clock_signal_1 ... clock_signal_n</td>
</tr>
</tbody>
</table>

These parameters define a clock’s behavior. 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.

\[
\text{arrival time at register output} = \text{CLOCK_OFFSET} + \text{CLK->Q delay}
\]

The required time at the input to the flip-flops and latches is \( \text{CLOCK_OFFSET} + \text{CLOCK_CYCLE} - \text{setup_time} \). At this point Leonardo 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 it may be the output of another register, or it may be the output of some combinational logic (although 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 syntax of an attribute is:

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

**Usage of Timing Constraints in Leonardo**

Timing constraints are defined as constraints that affect Leonardo's timing estimates and thus the quality of optimization, mapping and other features.

Table 4-1, Technology Constraints, summarizes what constraints can be used for different technologies. The table also provides a list of applications that are sensitive to the corresponding constraints.
Table 4-1. Technology Constraints

<table>
<thead>
<tr>
<th>Technology</th>
<th>Constraint</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx LUT based</td>
<td>arrival_time</td>
<td>Optimization, Technology Mapping</td>
</tr>
<tr>
<td>XC3000/A/L</td>
<td></td>
<td>TimeSpec Generation</td>
</tr>
<tr>
<td>XC3100/A</td>
<td>arrival_time</td>
<td></td>
</tr>
<tr>
<td>XC4000/A/E/EX/L/XL/XV</td>
<td>required_time</td>
<td></td>
</tr>
<tr>
<td>XC5200</td>
<td>clock_cycle</td>
<td></td>
</tr>
<tr>
<td>Spartan</td>
<td>clock_offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_pulse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>arrival_time</td>
<td>Critical Path Reporting</td>
</tr>
<tr>
<td></td>
<td>required_time</td>
<td>optimize_timing</td>
</tr>
<tr>
<td></td>
<td>clock_cycle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_pulse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>output_load</td>
<td></td>
</tr>
<tr>
<td></td>
<td>input_drive</td>
<td></td>
</tr>
<tr>
<td>Altera</td>
<td>arrival_time</td>
<td>Optimization, Technology Mapping</td>
</tr>
<tr>
<td>FLEX 6</td>
<td></td>
<td>Critical Path Reporting</td>
</tr>
<tr>
<td>FLEX 8</td>
<td>arrival_time</td>
<td>optimize_timing</td>
</tr>
<tr>
<td>FLEX 10</td>
<td>required_time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_cycle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_pulse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>output_load</td>
<td></td>
</tr>
<tr>
<td></td>
<td>input_drive</td>
<td></td>
</tr>
<tr>
<td>Actel</td>
<td>arrival_time</td>
<td>Technology Mapping</td>
</tr>
<tr>
<td>Act1/2/3</td>
<td></td>
<td>Critical Path Reporting</td>
</tr>
<tr>
<td>1200XL, 3200DX</td>
<td>arrival_time</td>
<td>optimize_timing</td>
</tr>
<tr>
<td>40MX, 42MX</td>
<td>required_time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_cycle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clock_pulse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>output_load</td>
<td></td>
</tr>
<tr>
<td></td>
<td>input_drive</td>
<td></td>
</tr>
</tbody>
</table>

Table 4-1 continued...
### New 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**

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>` command, while `<value>` is in terms of MHz. The clock_frequency constraint will apply 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. Leonardo will set this constraint on all global clocks in the design. Constraint driven timing optimization will try to meet the global clock frequency constraint.

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

Leonardo can report clock frequency of the clocks after synthesis by running `report_delay -clock` command.

**Support for Multicycle Path**

Use the following syntax to specify multicycle path constraints to Leonardo:

**set a multicycle path:**

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

**delete a multicycle path:**

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

**options for multicycle path:**

```plaintext
set_multicycle_path [options] <from> <to> <value>
```
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
- Statistics Report
- Automatic Constraining of Designs

Before Beginning

The information presented in this chapter introduces you to the technology chapters in this guide. This chapter assumes that you have read the User’s Guide. Refer also to the HDL Synthesis and Command Reference guides for information.

Optimization Flow

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

Timing optimization is performed and can be controlled at various levels within Leonardo. 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 Leonardo 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 Leonardo 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, Leonardo 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
```

Refer to Screen 5-1 for running optimize in delay mode from the GUI. Click on the Area radio button.
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 the present design to set a constraint. For example:

**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 the present design to set a constraint. For example:
Use these rules to set constraints on port \( clk \) of the top level and port \( clk \) of instance one and of instance two:

- Set constraint on \( clk \) at top level.
- Change the present design to one by typing `push_design one`.
- Set constraint on \( clk \).
- 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`.

Figure 5-1  Setting Timing Constraints
Constraints can be specified for a design by using the constraint editor that is part of the GUI, or by typing one of the following commands:

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

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`, `clock_cycle`, etc. can be specified.

### 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 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`.

```plaintext
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 is 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:

<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>0.00 dn</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_INST_13/O</td>
<td>IBUF</td>
<td>3.00 dn</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_NET_1/O</td>
<td>F4_LUT</td>
<td>7.50 dn</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_NET_3/O</td>
<td>F3_LUT</td>
<td>12.00 dn</td>
<td>0.0</td>
</tr>
<tr>
<td>outp_rename/O</td>
<td>H3_LUT</td>
<td>14.50 dn</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_INST_7_01/O</td>
<td>OBUF</td>
<td>21.50 dn</td>
<td>0.0</td>
</tr>
<tr>
<td>outp/</td>
<td></td>
<td>21.50 dn</td>
<td>0.0</td>
</tr>
<tr>
<td>data arrival time</td>
<td></td>
<td>21.50</td>
<td></td>
</tr>
<tr>
<td>data required time</td>
<td></td>
<td>not specified</td>
<td></td>
</tr>
</tbody>
</table>

Set the following constraints to optimize the timing of the design:

```
 ARRIVAL_TIME inp(5) 10.0
 REQUIRED_TIME outp 22.0
```
Now, because of the constraints, the critical path report shows:

```
Critical Path Report

Critical path #1, (path slack = -9.5):

<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>dn</td>
</tr>
<tr>
<td>XMPLR_INST_13/O</td>
<td>IBUF</td>
<td>13.00</td>
<td>dn</td>
</tr>
<tr>
<td>XMPLR_NET_1/O</td>
<td>F4_LUT</td>
<td>17.50</td>
<td>dn</td>
</tr>
<tr>
<td>XMPLR_NET_3/O</td>
<td>F3_LUT</td>
<td>22.00</td>
<td>dn</td>
</tr>
<tr>
<td>outp_rename/O</td>
<td>H3_LUT</td>
<td>24.50</td>
<td>dn</td>
</tr>
<tr>
<td>XMPLR_INST_7_O1/O</td>
<td>OBUF</td>
<td>31.50</td>
<td>dn</td>
</tr>
<tr>
<td>outp/</td>
<td></td>
<td>31.50</td>
<td>dn</td>
</tr>
</tbody>
</table>

data arrival time 31.50

data required time 22.00

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 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.

After setting the following constraints:

```
ARRIVAL_TIME inp(1) 10.0
REQUIRED_TIME outp 25.0
```
The critical path reporting is as follows:

Critical Path Report

Critical path #1, (path slack = -10.3):

<table>
<thead>
<tr>
<th>NAME</th>
<th>GATE</th>
<th>ARRIVAL</th>
<th>LOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>inp(1)/</td>
<td></td>
<td>10.00</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_INST_17/Y</td>
<td>INBUF</td>
<td>14.52</td>
<td>1.1</td>
</tr>
<tr>
<td>XMPLR_INST_4/Y</td>
<td>AND3</td>
<td>18.92</td>
<td>1.1</td>
</tr>
<tr>
<td>XMPLR_INST_5/Y</td>
<td>AND4</td>
<td>23.32</td>
<td>1.1</td>
</tr>
<tr>
<td>outp_rename/Y</td>
<td>AND4</td>
<td>27.72</td>
<td>1.1</td>
</tr>
<tr>
<td>XMPLR_INST_7/PAD</td>
<td>OUTBUF</td>
<td>35.27</td>
<td>0.0</td>
</tr>
<tr>
<td>outp/</td>
<td></td>
<td>35.27</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Data arrival time: 35.27
Data required time: 25.00

----------

Slack: -10.27

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 inp(1) is only one level of delay from the output outp.

![Design After Timing Optimization](image)

**Figure 5-6 Design After Timing Optimization**

**Automatic Constraining of Designs**

```
optimize_timing -force
```

```
maxdly <number>
```

**Note:** 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. Leonardo 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.
This chapter presents examples of reports for Xilinx and Altera FLEX. Reports are available to all the technologies presented in this guide. This chapter is divided as follows:

- Xilinx Area/Delay Example Reports
- Altera FLEX Area/Delay Example Reports

**Xilinx Area/Delay Example Reports**

Leonardo provides estimates of the area and critical path delay for the synthesized designs. These estimates are provided for comparing different design implementations and for getting a quantitative estimate of design performance.

**Area Reporting**

Area for Xilinx LCA architectures is measured in terms of Function Generators and Configurable Logic Blocks (CLBs). There are two different types of area reports:
Report issued during optimization. Such a report will be issued on each individual block in the hierarchy. Since information about CLBs is not available at this point, only the number of Function Generators is reported. For example the following report:

<table>
<thead>
<tr>
<th>Pass</th>
<th>Area (FGs)</th>
<th>Delay (ns)</th>
<th>DFFs</th>
<th>PIs</th>
<th>POs</th>
<th>--CPU--</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
<td>260</td>
<td>16</td>
<td>7</td>
<td>2</td>
<td>00:15</td>
</tr>
</tbody>
</table>

Resource Use Estimate
Technology: x14
Area: 22 Function Generators
Critical Path: 26 ns
DFFs: 16 (in CLBs or IOBs)
IOFFs: 2 (in IOBs)
HM CLBs: 0
Input Pins: 7
Output Pins: 2

After the whole design has been synthesized a global report can be issued by using the report_area command. For example:

```
LEONARDO: report_area

*******************************************************
Cell: mem  View: example  Library: work
*******************************************************
Number of ports : 10
Number of nets : 74
Number of instances : 73
Number of references to this view : 0
Total accumulated area :
Number of CLB Flip Flops : 14
Number of IOB Output Flip Flops : 2
Number of H Function Generators : 2
Number of FG Function Generators : 22
Number of Packed CLBs : 16
```
The combinational logic in this design occupies 22 FG Function Generators, and 2 H Function Generators. Total of 16 D-flip flops are used, 14 are packed into CLBs and 2 are packed as IOB Output flip flops. The Function Generators and the flip flops were packed (by the pack_clb command) into 16 CLBs.

- When using report_area command with -cell option, a breakdown of the cells that were used in the design will be given. For example:

```plaintext
LEONARDO: report_area -cell
******************************************************
Cell: mem   View: example   Library: work
******************************************************

<table>
<thead>
<tr>
<th>Cell</th>
<th>Library</th>
<th>References</th>
<th>Total Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFG xi4</td>
<td>x14</td>
<td>2 x</td>
<td>14 CLB Flip Flops</td>
</tr>
<tr>
<td>FDE xi4</td>
<td>x14</td>
<td>14 x 1</td>
<td>2 IOB Output Flip Flops</td>
</tr>
<tr>
<td>OFDT xi4</td>
<td>x14</td>
<td>2 x 1</td>
<td>2 IOB Output Flip Flops</td>
</tr>
<tr>
<td>IBUF xi4</td>
<td>x14</td>
<td>7 x</td>
<td></td>
</tr>
<tr>
<td>OR xi4</td>
<td>x14</td>
<td>6 x</td>
<td></td>
</tr>
<tr>
<td>AND xi4</td>
<td>x14</td>
<td>12 x</td>
<td></td>
</tr>
<tr>
<td>INV xi4</td>
<td>x14</td>
<td>6 x</td>
<td></td>
</tr>
<tr>
<td>HMAP_PUC12 xi4</td>
<td>2 x 1</td>
<td>2 H Function Generators</td>
<td></td>
</tr>
<tr>
<td>FMAP_PUC124 xi4</td>
<td>4 x 1</td>
<td>4 FG Function Generators</td>
<td></td>
</tr>
<tr>
<td>F2_LUT xi4</td>
<td>7 x 1</td>
<td>7 FG Function Generators</td>
<td></td>
</tr>
<tr>
<td>F4_LUT xi4</td>
<td>11 x 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Number of ports : 10
Number of nets : 74
Number of instances : 73
Number of references to this view : 0

Total accumulated area :
Number of CLB Flip Flops : 14
Number of IOB Output Flip Flops : 2
Number of H Function Generators : 2
Number of FG Function Generators : 22
Number of Packed CLBs : 16
```
Delay Estimates

Leonardo estimates delays for Xilinx architectures in terms of nanoseconds. These estimates are based on two components:

• Intrinsic delays for individual cells. After the design is mapped to technology cells, Leonardo will estimate the delay through each cell. The timing information for each cell is stored in the technology library.

For example:

The intrinsic delay for a FMAP lookup table targeting XC4000 technology is 4.5 nanoseconds. The intrinsic delay through a FlipFlop (like an FDP gate) is 3.0 nanoseconds.

• Interconnect delays are modeled through a wire load model. The wire load model is a table that associates the number of loads on a certain gate and the interconnect delays. By default, Leonardo will use the wire load information stored in the technology library.

After performing optimization of the design you can receive timing estimates by typing:

```
report_delay
```

This command reports the most critical paths in the design.
For example:

```
For example:

<table>
<thead>
<tr>
<th>NAME</th>
<th>GATE</th>
<th>ARRIVAL</th>
<th>LOAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock information</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>delay thru clock</td>
<td></td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>network</td>
<td></td>
<td>(ideal)</td>
<td></td>
</tr>
<tr>
<td>XMPLR_INST_17_I1/Q</td>
<td>FD</td>
<td>3.00</td>
<td>dn</td>
</tr>
<tr>
<td>XMPLR_INST_287/O</td>
<td>F3_LUT</td>
<td>7.50</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_NET_8/O</td>
<td>F3_LUT</td>
<td>12.00</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_NET_9/O</td>
<td>F4_LUT</td>
<td>16.50</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_NET_155/O</td>
<td>H3_LUT</td>
<td>19.00</td>
<td>0.0</td>
</tr>
<tr>
<td>nxstate(1)/O</td>
<td>F4_LUT</td>
<td>23.50</td>
<td>0.0</td>
</tr>
<tr>
<td>XMPLR_INST_15_I1/D</td>
<td>FD</td>
<td>23.50</td>
<td>0.0</td>
</tr>
<tr>
<td>data arrival time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>data required time</td>
<td></td>
<td>not specified</td>
<td></td>
</tr>
<tr>
<td>data required time</td>
<td></td>
<td>not specified</td>
<td></td>
</tr>
<tr>
<td>data arrival time</td>
<td></td>
<td>23.50</td>
<td></td>
</tr>
</tbody>
</table>
```

Some of the gates that are listed in the above report are internal gates used by Leonardo. The gates F3_LUT, F4_LUT, H3_LUT (represent a 3 input F-function generator, a 4 input F-function generator, and a 3 input H-function generator respectively). The intrinsic delays through these gates are: 4.5 ns, 4.5 ns, and 2.5 ns respectively. The FD gate is a Xilinx unified library gate, the intrinsic delay through this gate is 3.0 ns.

This path is listed as an unconstrained path because no timing constraints are used in the above example.

**Altera FLEX Area/Delay Example Reports**

Leonardo provides estimates of the area and the critical path delay for the synthesized designs.
**Area Reporting**

Area for Altera FLEX architectures is measured in terms of Logic Cells (LCs). The reports are shown in the following examples.

- Report issued during optimization. Such a report will be issued on each individual block in the hierarchy. For example the following report:

```
  Pass  LCs  Delay  DFFs  TRIs  PIs  POs  --CPU--
     1   78    21    62     8     5   14    00:04
```

<table>
<thead>
<tr>
<th></th>
<th>est</th>
<th>est</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>LCs</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>Delay</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>DFFs</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>TRIs</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>PIs</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>POs</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>--CPU--</td>
<td></td>
<td>00:04</td>
</tr>
</tbody>
</table>

Resource Use Estimate

- Technology: flex10
- LCs: 78
- Critical Path: 21 ns
- DFFs: 62
- TRIs: 8
- Input Pins: 5
- Output Pins: 14

Reports 78 Logic Cells for this block, 62 flip-flops and 8 tristate buffers.
• After the whole design has been synthesized a global report can be issued by running the `report_area` command. For example:

```
LEONARDO(4): report_area
******************************************************
Cell: uart   View: exemplar   Library: work
******************************************************
Number of ports : 19
Number of nets : 152
Number of instances : 147
Number of references to this view : 0
Total accumulated area :
Number of TRIs : 8
Number of DFFs : 62
Number of LCs : 78
```

• When running `report_area` with `-cell` option a breakdown of the cells that were used in the design will be given.
For example:

```
LEONARDO: report_area -cell

******************************************************
Cell: uart      View: exemplar    Library: work
******************************************************

<table>
<thead>
<tr>
<th>Cell</th>
<th>Library</th>
<th>References</th>
<th>Total Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRI</td>
<td>flex10</td>
<td>8 x 1</td>
<td>8 TRIs</td>
</tr>
<tr>
<td>OUTBUF</td>
<td>flex10</td>
<td>6 x</td>
<td></td>
</tr>
<tr>
<td>INBUF</td>
<td>flex10</td>
<td>13 x</td>
<td></td>
</tr>
<tr>
<td>LATCH</td>
<td>flex10</td>
<td>8 x 1</td>
<td>8 DFFs</td>
</tr>
<tr>
<td>DFF</td>
<td>flex10</td>
<td>39 x 1</td>
<td>39 DFFs</td>
</tr>
<tr>
<td>DFFE</td>
<td>flex10</td>
<td>15 x 1</td>
<td>15 DFFs</td>
</tr>
<tr>
<td>VCC</td>
<td>flex10</td>
<td>1 x</td>
<td></td>
</tr>
<tr>
<td>F1_LUT</td>
<td>flex10</td>
<td>2 x</td>
<td></td>
</tr>
<tr>
<td>F2_LUT</td>
<td>flex10</td>
<td>19 x</td>
<td></td>
</tr>
<tr>
<td>F3_LUT</td>
<td>flex10</td>
<td>20 x</td>
<td></td>
</tr>
<tr>
<td>F4_LUT</td>
<td>flex10</td>
<td>16 x</td>
<td></td>
</tr>
</tbody>
</table>

Number of ports : 19
Number of nets : 152
Number of instances : 147
Number of references to this view : 0

Total accumulated area :
Number of TRIs : 8
Number of DFFs : 62
Number of LCs : 78
```

Leonardo Synthesis and Technology Guide
The cells F1_LUT, F2_LUT, F3_LUT, F4_LUT are used internally by Leonardo for optimization and mapping. When writing out the output netlist these cells are replaced by Altera primitives (i.e., LCELL).

**Delay Estimates**

```
LEONARDO(6): report_delay -num 1

Critical Path Report

Critical path #1, (unconstrained path)
NAME                GATE     ARRIVAL            LOAD
-----------------------------------------
clock information not specified
delay thru clock network                0.00 (ideal)
xmplr_inst_146/Q  DFF      0.30 dn 0.0
xmplr_net_155/O    DF4_LUT  4.40 dn 0.0
xmplr_net_161/O    DF4_LUT  8.50 dn 0.0
~xmplr_net_185/O   DF4_LUT 12.60 dn 0.0
xmplr_net_95/O     DF2_LUT 16.70 dn 0.0
xmplr_net_107/O    DF2_LUT 20.80 dn 0.0
xmplr_inst_146/D   DDFF    20.80 dn 0.0
data arrival time          20.80

data required time                not specified
-----------------------------------------
data required time                not specified
data arrival time          20.80
                             ------------
                             unconstrained path
```

---

Reports 6-9
Synthesis for ASIC Technologies

Leonardo provides a complete synthesis, optimization, and verification environment for ASIC technologies. You can control and customize the synthesis process for your design. In addition, you can verify timing and functionality after place-and-route is completed. This chapter is divided as follows:

• Before Beginning
• Design Flows
• Optimization Features
• Timing Analysis
• Area and Delay Reporting

Before Beginning

The technology information presented in this chapter assumes that you have read the introductory chapters in this guide.

Design Flows

As shown in Figure 7-1 Leonardo focuses on top-down design flows. You can enter a design in VHDL or Verilog and synthesize it to the target ASIC technology. The behavioral design needs to be simulated first, then optimized and synthesized. An EDIF netlist is generated as an output. In addition, pre-place and pre-route timing analysis can be run and critical path reporting received. Scan insertions for testing and layout are done next. The post-place and post-route netlist can be read to do timing analysis. Gate level simulation is then applied. Leonardo allows you to use gate level VHDL or Verilog with an SDF timing file to verify timing and functionality.
Figure 7-1  ASIC Top-Down Design Flow
Leonardo also allows you to mix HDL descriptions with schematics. You can enter schematics using EDIF netlist and then use VHDL or Verilog.

Figure 7-2 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 is done.

Figure 7-2 Synthesis Design Flow
Hierarchy

You can do the following:

- preserve the hierarchy as it was described in the input design (default behavior)
- flatten certain instances (using the `ungroup` command)
- create new levels of hierarchy (using the `group` command)

Since the tool is interactive (or 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, a bus interface unit is one block, while an 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
• Resolution of design rule constraints

**Data-Path Synthesis**

Leonardo 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>
<tr>
<td>Misc. Functions</td>
<td>incrementer, +1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>decremener, -1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>absolute value, ABS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unary minus, unary -</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**New ASIC Modgen**

Leonardo supports a variety of new module generator architectures for efficient implementation of data path modules. Supported are ripple-carry/carry-save/carry-lookahead adders, subtractors, incrementers, decrementers, Baugh-Wooley multiplier
with ripple-carry or carry-lookahead final adder stage, etc. These module generator architectures give far superior results in terms of area, delay and cpu time when compared to 'default' modgen (without a module generator primitives library) usage in Leonardo, or any previous technology independent module generator technique.

To be able to use the new modgen architectures, you need a library of modgen primitives for the ASIC technology that you are using. This library of primitives is a vhdl description of technology specific implementation of full-adder cells, and other basic cells needed for efficient module generation.

The library is included with the ASIC package. The package can be obtained from your ASIC vendor. The Exemplar ftp site also provides a large number of ASIC packages for various technologies.

If you are currently using an ASIC technology, but don't have the technology specific modgen primitives library, please contact your ASIC vendor, and ask for an update of the ASIC package for the technology you are using.

If you have been using the new ASIC module generators with Leonardo release 4.03, then you will be accustomed to loading the technology independent architecture generator file (asic.vhd) separately. Also, you needed to download a separate asic.vhd architecture generator file from the Exemplar ftp site.

This generator file is not needed in Leonardo 4.1. Now you only need to load the technology specific modgen primitives library file. The architecture generator file is loaded automatically, from the Leonardo 4.1 installation area.

The ASIC package consists of a Leonardo library (<tech>.syn file), a Leonardo modgen primitives library (<tech>.vhd file) plus documentation. Presently, there are approximately 60 different ASIC technologies available for Leonardo.

Once you obtain the ASIC package, follow the instructions that come with the package to install the files: store the <tech>.syn file in the $EXEMPLAR/lib directory, and install the <tech>.vhd file in the $EXEMPLAR/data/modgen directory.

Next, run Leonardo and load both the technology and the modgen architectures

load_library <tech>[.syn]   <-- Load the technology file
load_modgen <tech>[.vhd]    <-- Load the modgen architectures

You are now ready to load in any design. Leonardo will use the new, efficient, technology specific implementations for all data path modules in your design.
User Switches

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

This 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 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**

Leonardo performs gate sizing to improve the timing quality of the design. It will try to find the optimal size of a gate so that the cost of the gate will be 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
```

will perform gate sizing during mapping but not during timing optimization.

Leonardo does not perform gate sizing for I/O cells and sequential cells.
Design Rule Constraints

Leonardo checks for the following design rule constraints load violations, fanout violations, and transition time violations.

Load Violations

The load driven by a particular gate is expressed in terms of capacitance (usually pF). Leonardo checks that every output pin in a given 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. From Leonardo’s shell you can overwrite the default value by setting the variable max_cap_load. This must be done before reading the technology library.

Recommended: set the load values on all primary outputs of each sub-design using the output_load command. The actual computed load includes wire load capacitance, if information is available in the technology library.

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 integer 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. Leonardo checks that each output pin in a design does not drive more than the allowed number of fanouts. 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. You can overwrite the default value from Leonardo’s shell by setting the max_fanout_load variable before reading the technology library.
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;
    }
};
.....
```

Leonardo will perform 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. Leonardo 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.

This computation is more complex than load and fanout violations because the transition time of a particular output pin is dependent on the transition time of its inputs. You can define a default maximum transition time in the technology library by setting the default_max_transition property, and can overwrite the default transition time from Leonardo’s shell by setting the max_transition variable before reading the technology library.

Resolving Load Violations

Leonardo resolves load violations by doing one of the following operations:

- **Gate sizing**
  
  If possible, Leonardo upsizes the gate on which a load violation has occurred. This is applicable for resolving load and fanout violations only.

- **Logic replication**
  
  Logic is replicated and the actual load is split between the replicated logic. This is applicable for resolving load violations only.

- **Buffering**
  
  Building a buffer tree to resolve the violation. This technique is used to resolve transition time violations.
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. Leonardo supports three different delay models:

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

Issues in Delay Modeling

Clock Delays

Leonardo 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.

Example:

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

Defines the clocking schemes as shown in Figure 7-3.
Derating Factors

Operating conditions effect the delay estimations done by Leonardo. Operating conditions are usually specified in the technology library: operating temperature, voltage, and the process. You can change the operating conditions by setting the following variables:

- **process**  Typical, best, or worst process
- **temp** Specifies temperature in Celsius
- **voltage** Specifies voltage in volts

If these variables are defined before reading the technology library Leonardo will derate the delay computations.

Derating is not currently supported for nonlinear delay computations.

Modeling Interconnects

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

Figure 7-3 Wave forms for clk and clk1
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:

```plaintext
present_design design1
set wire_table table1
optimize
report_delay
present_design design2
set wire_table table2
optimize
report_delay
```

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 Leonardo 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.

**Area and Delay Reporting**

Leonardo provides estimates of the area and the critical path delay for the synthesized design.

**Area Report**

Area for ASIC designs is measured in units that are defined in the technology library.
There are two different types of area reports:

1. Report issued during optimization. Such a report will be issued on each individual block in the hierarchy and for every optimization pass.

   For example the following report lists the total accumulated area for this design for every optimization pass and the delay through the longest path. The area and delay of the best pass is also listed.

<table>
<thead>
<tr>
<th>Pass</th>
<th>Area (Gates)</th>
<th>Delay --CPU-- (ns) min:sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>273</td>
<td>3.5 00:11</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
<td>2.0 00:03</td>
</tr>
<tr>
<td>3</td>
<td>234</td>
<td>3.7 00:11</td>
</tr>
<tr>
<td>4</td>
<td>99</td>
<td>2.9 00:04</td>
</tr>
<tr>
<td>5</td>
<td>246</td>
<td>3.4 00:10</td>
</tr>
<tr>
<td>6</td>
<td>102</td>
<td>2.0 00:03</td>
</tr>
<tr>
<td>7</td>
<td>234</td>
<td>3.7 00:10</td>
</tr>
<tr>
<td>8</td>
<td>99</td>
<td>2.9 00:04</td>
</tr>
<tr>
<td>9</td>
<td>266</td>
<td>3.5 00:14</td>
</tr>
<tr>
<td>10</td>
<td>207</td>
<td>3.5 00:10</td>
</tr>
<tr>
<td>11</td>
<td>106</td>
<td>3.2 00:06</td>
</tr>
</tbody>
</table>

Resource Use Estimate

Technology: cg51
Area: 98.6
Critical Path: 2.9 ns
2. After the whole design has been synthesized, a global report can be issued by typing the report_area command. For example:

```
LEONARDO(17): report_area

******************************************************
Cell: traffic    View: exemplar    Library: work
******************************************************
Number of ports :                       9
Number of nets :                       24
Number of instances :                  18
Number of references to this view :     0

Total accumulated area :
Number of gates :                      99
```
This report lists additional information, for example, number of ports, nets, and instances about the synthesized design. To get a breakdown of the technology gates that were used a `report_area -cell` command should be used.

```
LEONARDO(16): report_area -cell

*******************************************************
Cell: traffic    View: exemplar    Library: work
*******************************************************

<table>
<thead>
<tr>
<th>Cell</th>
<th>Library</th>
<th>References</th>
<th>Total Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD1AL</td>
<td>cg51</td>
<td>3 x</td>
<td>7</td>
</tr>
<tr>
<td>RN4L</td>
<td>cg51</td>
<td>1 x</td>
<td>8</td>
</tr>
<tr>
<td>NI3L</td>
<td>cg51</td>
<td>1 x</td>
<td>6</td>
</tr>
<tr>
<td>NI2L</td>
<td>cg51</td>
<td>1 x</td>
<td>4</td>
</tr>
<tr>
<td>AN236L</td>
<td>cg51</td>
<td>1 x</td>
<td>11</td>
</tr>
<tr>
<td>NI1M</td>
<td>cg51</td>
<td>3 x</td>
<td>3</td>
</tr>
<tr>
<td>XI2L</td>
<td>cg51</td>
<td>1 x</td>
<td>5</td>
</tr>
<tr>
<td>AR223L</td>
<td>cg51</td>
<td>1 x</td>
<td>6</td>
</tr>
<tr>
<td>R12L</td>
<td>cg51</td>
<td>2 x</td>
<td>4</td>
</tr>
<tr>
<td>NT2L</td>
<td>cg51</td>
<td>3 x</td>
<td>5</td>
</tr>
<tr>
<td>R12M</td>
<td>cg51</td>
<td>1 x</td>
<td>5</td>
</tr>
</tbody>
</table>

Number of ports : 9
Number of nets : 24
Number of instances : 18
Number of references to this view : 0

Total accumulated area :
Number of gates : 99
```
Delay Report

Leonardo reports delays that are measured in nanoseconds. After each optimization pass on each individual block, Leonardo estimates the longest delay path in this block.

After all blocks have been synthesized you can get a critical path report by typing the `report_delay` command. `report_delay` will report the ‘6’ most critical paths in the design. Leonardo can report critical paths in a design both after synthesis is completed, and after the designs are placed and routed. For example:

```
LENARDO(21): report_delay
Using default wire table: cg51_4000area

Critical Path Report

Critical path #1, (unconstrained path)
NAME GATE ARRIVAL LOAD
---------------------------------------------------------
  clock information not specified
delay thru clock network  0.00 (ideal)
  xmplr_inst_17/XQ FD1AL 0.58 dn 0.27
  xmplr_inst_8/X R12M 1.07 up 0.33
  green2_rename/X R12L 1.50 dn 0.14
  xmplr_inst_12/X AN236L 1.95 up 0.06
  xmplr_inst_14/X NT2L 2.29 up 0.19
  xmplr_inst_16/X N12L 2.54 dn 0.06
  nxstate(2)/X RN4L 2.92 up 0.11
  xmplr_inst_17/D FD1AL 2.92 up 0.00
data arrival time  2.92

data required time not specified
---------------------------------------------------------
data required time not specified
data arrival time  2.92
----------
unconstrained path
```

In this case a wire load table `cg51_4000area` is used for interconnect delays. The clock propagation time is ignored (ideal clock) and no timing constraints are specified so the critical path is an ‘unconstrained path’.
Leonardo provides synthesis support for the following Actel architectures:

- Act1/2/3
- 1200XL
- 3200DX
- 40MX and 42MX

Designs can be entered using VHDL or Verilog HDL descriptions or EDIF netlists. Gate array and other FPGA designs can be retargeted for Actel devices. This chapter presents information on how to use Leonardo for Actel devices.

This chapter is divided as follows:

- Before Beginning
- Actel FPGA Architecture
- Design Flows
- New Features for Actel
- Synthesis and Optimization Features
- Data Path Synthesis
- Additional Synthesis Features
- I/O Mapping
- Complex I/O Design Rule Checker and Modifier
- Actel Reports
- Process Derating Factors
Before Beginning

The technology information presented in this chapter assumes that you have read the introductory chapters in this guide. Refer also to the HDL Synthesis and Command Reference guides for information.

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. It 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 when using Leonardo.

Design Flows

A simplified design flow is shown in Figure 8-1 to illustrate how Leonardo is used with the Actel tools. Designs are entered using standard methods, and are optimized and mapped to the target Actel technology by Leonardo. The result is an EDIF netlist that is used by Actel place-and-route tools.
Figure 8-1  Simplified Design Flow
Figure 8-2  Simplified Back Annotation Flow
New Features for Actel

New Operators

The following operators were added:
- Act 1: fast incrementer, reduce_and
- All families: reduce_nand, reduce_or

Improvements to Mapper

- 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.

Accurate Delay Estimations

All Actel libraries were modified with delay information that is compatible with Designer.

Logic Replication and Buffering

Logic replication and buffering resolves timing violations.

Fanout Violations and Load Violations

When resolving design rule violations, Leonardo considers fanout violations and also load violations. You can disable either fanout or load violations from the GUI.

Reduced Memory Consumption

The Actel technology libraries are smaller.
**Enhancements to Actel Technologies**

Actel mapping and optimization were enhanced to map to DFM7 and DFM7A gates. This will make area and delay estimates more accurate. When targeting Act2 or 3200DX technologies, the mapper will map to CM7 gates. It is a known problem that the CM7 gate was not included in Designer's simulation libraries. This problem has been fixed and you can get the new simulation libraries from Actel.

**Note:** These libraries are available on Actel's Web site. Beginning with the R1/98 release, CM7 simulation models will be part of Actel's release.

**Modgen Enhancements**

**New Modgen Libraries**

A32DX: Actel 3200DX Modgen Library (Act2 modgen plus RAMs)

**Existing Library Improvement**

All RAMs: support for arbitrary size RAMs

**New Components**

Act1/2/3/32DX: small/fast non-loadable down counters.

**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 AND and OR gates).
- Boolean mapping maps directly to the Actel Logic Module.
- Leonardo resolves load violations to ensure that the generated designs are valid.
- Leonardo 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.

Technology Mapping

Leonardo offers a powerful Boolean Mapping algorithm that maps logic directly to Logic Cells. The following libraries are available:

- **Act1** For the Act1 technology
- **Act2** For the Act2 technology
- **Act3** For the Act3 technology
- **a12x1** For the 1200XL technology
- **a32dx** For the 3200DX technology

Data Path Synthesis

Modgen

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

• Logical Operators: `and` `or` `nand` `nor` `xor` `xnor`
• Relational Operators: `=` `=/=` `<` `<=` `>` `>=`
• Shift Operators: `sll` `srl` `sla` `sra` `rol` `ror`
• Arithmetic Operators: `+` `-` `*`
• Misc. Functions: `incrementer` `decrementer` `absolute value` `unary minus` `counters` `multiplexer`
Using RAMs

Types of Inferencing RAMs

Leonardo 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 Leonardo 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); Leonardo 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.

Disable RAM Inferencing

`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);
begin
  address_int <= address;
  if (we = '1') then
    mem(evec2int(address)) <= data;
  end if;
  if (outclock = '1' and outclock'event) then
    q <= mem(evec2int(address_int));
  end if;
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

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, in which case it will try to improve the longest path.

```
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 Actel’s ACTGen

Modgen implementations are fully compatible with Actel’s ACTGen. This gives you a very high quality of results. You may also interface from Leonardo 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.
3. Set the Leonardo variable `hdl_array_name_style` to the format `%s(%d)`. This will ensure that Leonardo 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.
4. Optimize your design and generate EDIF output. All the ACTGen modules will be treated as black boxes.
5. Merge all EDIF files together using Designer by typing:

```typescript
cae2adl -edn2adl ednin:<file1>+<file2>+<file3> <design_name>
```

For Example to merge files: top.edn, upcnt.edn and dncnt.edn, type:

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

---

**Figure 8-3** Interfacing with ACTgen Through VHDL
The following example shows how 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

In the Act2, Act3 1200XL and 3200DX technologies, some combinational cells can be combined in the same logic module with registered logic. This can result in a significant decrease in the number of logic modules implemented for a design. Figure 8-4 shows logic replication to increase combinability. For example, if an AND gate drives 3 flip-flops in the original design, Leonardo would implement the circuit by replicating the AND gate, and having each AND gate drive a single flip-flop. Without optimization by Leonardo, this takes 4 logic modules: one each for the AND gate and each flip-flop. With Leonardo, the AND gates are combined into the logic modules with the flip-flops, resulting in only 3 logic modules required for the design.

Figure 8-4  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. Actel’s 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 will issue a warning message.

The default number for maximum fanout allowed by Leonardo is set from the technology library and is as follows:

- Act1, Act2, 1200XL, 3200DX: 10 fanouts
- Act 3: 16 fanouts

You can overwrite this number by setting the variable from Leonardo’s shell.

```
set max_fanout_load <n>
```

To resolve fanout violations Leonardo will replicate logic whenever possible. Otherwise Leonardo will buffer the relevant signals so that fanout violations are met. You can disable logic replication by setting the variable `nologic_rep` to `TRUE`.

```
set nologic_rep TRUE
```

**Using Global Clock Buffers**

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

**Automatic Mapping of Clocks**

Leonardo 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:

```vhdl
BUFFER_SIG HCLKBUF clk
BUFFER_SIG CLKBUF rst
```
For an internal clock, you can specify:

```
BUFFER_SIG CLKINT int_clk
```

You can also specify a `buffer_sig` attribute in the source VHDL to manually assign global buffers. For example:

```
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;
```

Automatic global buffer mapping will not affect any signal which is already mapped manually.

**User Options**

You have the option to switch off automatic global buffer mapping by specifying the following command:

```
set insert_global_bufs FALSE
```

By default Leonardo maps to global buffers.

By default Leonardo will not map to QCLKBUFs for 3200DX architecture, since this may over constrain placement. You can switch on the mapping to QCLKBUFs by specifying:

```
set use_qclk_bufs TRUE
```
Design I/O

I/O Mapping

Leonardo maps I/O cells and registered I/O cells automatically. If the input description implies a use of a certain I/O cell, then Leonardo maps to it. You can assign I/O cells manually by either instantiating them from the (RTL description) VHDL or Verilog, or using the PAD or BUFFER_SIG commands from Leonardo’s shell.

Complex I/O Design Rule Checker and Modifier

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

- All registered I/Os are controlled by one hardwired clock line: IOCLK.
- All registered I/Os should use either clear or preset. There is only one hardwired clear/preset line to be used (IOPCL).

Leonardo verifies that a design (after optimization and technology mapping) does not violate any of these restrictions. If the design violates some constraints, Leonardo tries to modify the design and to use as many registered I/Os as possible without creating an illegal design.
library exemplar;
use work.exemplar.all;
entity sample is
  port (a, b, d, clr, clk: in bit;
        o, o1: out bit
    );
end sample;
arithmetic exemplar of sample is
  signal sig1, sig2, sig3: bit;
begin
  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;

Example of a VHDL File Containing a Design Rule Violation
Figure 8-5  Schematic of the Design Showing the Design Rule Violation
Figure 8-6  Schematic of the Modified Design which Resolves the Design Rule Violation

**Simulation Considerations**

Leonardo 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**

Leonardo 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:

- Leonardo shell
- Leonardo constraint editor
- 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:

where <n> is 1 and <port> is din(0)

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

```
.......  
|net clk 
|joined 
| (portRef clk )  
| (portRef PAD (instanceRef XMPLR_INST_20 )))  
| (property ALSPIN (string "1")))  
|net (rename n37 "din(4)*)  
|joined 
| (portRef p18 )  
| (portRef PAD (instanceRef XMPLR_INST_19 )))  
| (property ALSPIN (string "4")))  
|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 "2")))  
|net (rename n30 "din(1)*)  
|joined 
| (portRef p21 )  
| (portRef PAD (instanceRef XMPLR_INST_16 )))  
| (property ALSPIN (string "1")))  
|net (rename n31 "din(0)*)  
|joined 
| (portRef p22 )  
| (portRef PAD (instanceRef XMPLR_INST_14 )))  
| (property ALSPIN (string "0")))  
.......  
```

**Reporting**

Leonardo reports on estimates of area of the optimized design as follows:

- Reports on the number of logic modules and IO modules can be issued during each optimization pass.
- Detailed reports on all cells used in the design can be issued after optimization.
  
  **Type:** report_area -cell

Refer to Chapter 6, Reports, for more information and examples.
Process Derating Factors

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

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

As shown in Chapter 3, Timing Analysis, process values are entered on the Leonardo GUI or on the command line.

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

<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>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>
### 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>INDTCSTDV</td>
<td>IND typical</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>INDWCSTDV</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>COMTCSTDV</td>
<td>COM typical</td>
<td>STD</td>
<td>low voltage</td>
</tr>
<tr>
<td>COMWCSTDV</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>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>BCASTD</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>BCASTDV</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>
### Act1 Devices Derating Factors

<BC|TC|WC><STD|-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>
This chapter presents information specific to the use of Altera FLEX as a source or target technology. Leonardo supports the following architectures:

- FLEX 6K
- FLEX 8K
- FLEX 10K

This chapter is divided as follows:

- Before Beginning
- Altera FLEX Architecture
- Design Flow
- Synthesis and Optimization Features
- Reporting
- Design I/Os
- Writing EDIF Output
- Using EDIF from MAX+PLUS II
- How to Get Best Results with MAX+PLUS II
- New Features for Altera FLEX

Before Beginning

The technology information presented in this chapter assumes that you have read the introductory chapters in this guide. Refer also to the HDL Synthesis and Command Reference guides for information.
Altera FLEX Architecture

Logic Elements (LEs)
LEs are also known as Logic Cells (LCells). The combinational logic part of an LE is often called 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 an 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
Leonardo accepts hierarchical HDL descriptions in VHDL or Verilog. Input designs can be optimized and mapped to any Altera FLEX architecture. Leonardo can also target designs from other FPGA vendors to Altera architectures. Refer to Figures 9-1 and 9-2.

During reading, the input design is translated into Leonardo's intermediate data structures. 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, Leonardo attaches an equation string, lut_function, 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 is run for place-and-route, Leonardo can back-annotate the post place-and-route timing and generate an SDF timing file and a VHDL netlist. Leonardo 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
• 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 Leonardo.

• 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 inferenced 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):
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. For example:

\[
X = (A \cdot (B+C)) + (B \cdot D) + (E \cdot F \cdot G \cdot H \cdot I) \tag{E-1}
\]

Represented in AND and OR gates, \(X\) is decomposed as:

\[
\begin{align*}
X &= T1 + T2 + T3 \\
T1 &= A \cdot T4 \\
T2 &= B \cdot D \\
T3 &= E \cdot F \cdot G \cdot H \cdot I \\
T4 &= B + C.
\end{align*}
\tag{E-2}
\]

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

\[
\begin{align*}
T3 &= E \cdot F \cdot G \cdot T5 \\
T5 &= H \cdot I \tag{E-3}
\end{align*}
\]

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:

\[
\begin{align*}
\text{LE}_1: X &= T1 + (B \cdot D) + T3 \\
\text{LE}_2: T1 &= A \cdot (B+C) \\
\text{LE}_3: T3 &= E \cdot F \cdot G \cdot T5 \\
\text{LE}_4: T5 &= H \cdot I \tag{E-4}
\end{align*}
\]

Note – The critical path is \(\text{LE}_4 \rightarrow \text{LE}_3 \rightarrow \text{LE}_1\), resulting in three levels of LEs for the delay.
Solution (2)

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

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

```verilog
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.
Before writing an EDIF netlist that can be read by MAX+PLUS II software, Leonardo attaches a lut_function property to each lookup table. This step is done automatically when using the auto_write command. This is necessary to describe the LUTs functionality.

**User Options to Control LUT Mapping**

Use these rules and refer to Table 9-1 and Screen 9-1.

- **dont_lock_lcells** - Default value is set to FALSE. If this variable is set to TRUE, then Leonardo does not force LCell boundaries in the output EDIF netlist. Instead, it writes them as SOFT. In this case, MAX+PLUS II can change the LCell boundaries.
- **flex_use_cascades** - If set **flex_use_cascades** is set to FALSE, **dont_lock_lcells** should also be set to TRUE. This way MAX+PLUS II can still map to cascades wherever appropriate.
- **lut_map** - If this variable is set to FALSE, LUT mapping is off.

### Table 9-1. Mapping Options

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Option on/off</th>
<th>Command Line**</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>■ Map Look Up Tables</td>
<td>on* default (true)</td>
<td></td>
<td>Controls LUT mapping. If lut_map is false, set dont_lock_lcells to true.</td>
</tr>
<tr>
<td></td>
<td>off lut_map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>■ Map to CASCADEs during LUT mapping</td>
<td>on* default (true)</td>
<td></td>
<td>Controls mapping logic to cascade gates for Altera FLEX.</td>
</tr>
<tr>
<td></td>
<td>off flex_use_cascades</td>
<td></td>
<td></td>
</tr>
<tr>
<td>□ Do not lock LCells</td>
<td>on dont_lock_lcells</td>
<td></td>
<td>Controls locking of LCells.</td>
</tr>
<tr>
<td></td>
<td>off* default (false)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*GUI default options

**variables, options, constraint attributes, commands
Screen 9-1. Output Options for Altera FLEX 6K

*Data Path Synthesis and Modgen Implementation*

Leonardo 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. Leonardo supports module generation for Altera FLEX 6K, 8K, and 10K.
The following operators are supported for Altera FLEX technologies:

- **Relational Operators**: = /= < <= > >=
- **Arithmetic Operators**: + - *
- **Misc. Functions**: counters (up/down, loadable etc.)
  - RAMs
  - incrementer
  - decremener
  - 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**

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

You can control modgen implementation that is optimized for area or for delay by using this switch. This switch controls modgen resolution for all operators. You can also use the GUI as explained in Table 9-2 and shown in Screen 9-2 to control modgen implementation.
Table 9-2. GUI Modgen Select and Results

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>GUI Radio Button Options on/off</th>
<th>Command Line**</th>
<th>Available in Families</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modgen Select:</td>
<td>Auto on*</td>
<td>Auto (default)</td>
<td>All</td>
<td>Picks smallest if optimization in area mode; and picks fastest if optimization in delay mode.</td>
</tr>
<tr>
<td></td>
<td>Smallest off</td>
<td>Smallest</td>
<td></td>
<td>Picks the most compact implementation available.</td>
</tr>
<tr>
<td></td>
<td>Small off</td>
<td>Small</td>
<td></td>
<td>Picks a compact implementation.</td>
</tr>
<tr>
<td></td>
<td>Fast off</td>
<td>Fast</td>
<td></td>
<td>Picks a fast implementation.</td>
</tr>
<tr>
<td></td>
<td>Fastest off</td>
<td>Fastest</td>
<td></td>
<td>Picks the fastest implementation available.</td>
</tr>
</tbody>
</table>

* GUI default option  
** variables, options, constrained attributes, commands

Screen 9-2. Modgen Implementation

**LPM Support for FLEX 10K**

**RAMs**

Leonardo supports LPM RAM components for FLEX 10K. The component specifications are derived from the EDIF 2.0 LPM Specification. Refer to this document for more detailed information regarding ports. There are two levels of support for RAMs in Leonardo:
• RAM instantiation

Leonardo supports RAM_DQ, RAM_IO through modgen. You can instantiate the RAM_IO, RAM_DQ components from VHDL and Verilog. Leonardo 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

Instead of going through modgen, LPM components can be instantiated directly. In this case, you must specify the correct LPM names and properties required by MAX+PLUS II.

Instantiating RAMs from VHDL

Below is an example of ram_dq instantiation from VHDL. Because the description of ram_dq is in file $EXEMPLAR/data/modgen/flex10.vhd, you should run analyze $EXEMPLAR/data/modgen/flex10.vhd in Leonardo first before reading in your design VHDL.
For example:

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

entity ramload is
  port (clock: in std_logic;
         load, en: in std_logic;
         data, address : in std_logic_vector (3 downto 0);
         dout: out std_logic_vector (3 downto 0));
end ramload;

architecture exemplar of ramload is
  signal mout: std_logic_vector (3 downto 0);
begin
  -- the component declaration of ram_dq is in file
  -- $EXEMPLAR/data/modgen/modgen.vhd.
  i0: ram_dq generic map (size => 4, sizead => 4, read_addr_clked => true)
     port map (data, address, en, clock, load, mout);
  dout <= mout;
end exemplar;
```

**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 Version 2.0.1. Defparms values that are not specified use default values.
For 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
```
Leonardo infers RAMs automatically from VHDL and Verilog. If the modgen library is loaded, it uses the LPM description from modgen to implement the RAMs.
For example, consider the Verilog description shown below:

```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
```

Leonardo will infer a LPM `ram_dq` for this design.

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

![Figure 9-4 RAM Inferencing](image_url)

Figure 9-4  RAM Inferencing
Using RAMs

Types of Inferencing RAMs

Leonardo 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 Leonardo 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); Leonardo 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.
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
        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;
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.
The module declaration must be included in this file as shown in the 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
```
3. Synthesize the Verilog design with Leonardo by targeting FLEX 10K.
   a. Set the `hdl_array_name_style` variable to `%s%d` to make sure that port
      names generated by Leonardo match port names generated by genmem.
   b. Generate EDIF netlist as an output.
      The genmem component is treated as a black-box by Leonardo.
      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.

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

Leonardo 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. Usually it will
achieve a faster implementation.

In Leonardo, you can access EABs by setting an attribute on the desired instance. The
attribute is `implement_in_eab` and it should be set to on. This can be set from
either VHDL or Verilog. Following is an example for using EABs from 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...
...continued EABs example.

```
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
  Ui: 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;
```

**Additional Optimization Features**

**Sequential Optimization**

Leonardo 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. Leonardo also eliminates registers or latches that have inputs that are identical to another register or latch; these registers will effectively be merged. Leonardo does not “move” registers or change the phase of the signal stored in the registers.

**Altera FLEX Reports**

Refer to Chapter 6, Reports, in this guide for examples and more information. Note: Chapter 6 only contains report examples for two technologies.
Design I/O

Global Signals

Global signal like clocks and output enable for tri-state signals can be specified by using the buffer_sig command from Leonardo’s shell. The primitive supported by the FLEX architecture is GLOBAL. You are responsible to ensure that the design rules for using such a primitive (number of primitives, signals connected to primitives, etc.) are not exceeded. For example:

```plaintext
BUFFER_SIG GLOBAL clock1
BUFFER_SIG GLOBAL oe
```

Pin Location

Pins for I/O signals can be defined in Leonardo using commands from Leonardo’s shell or from the VHDL design source.

Use the following syntax from Leonardo’s shell:

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

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

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

For example:

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

Leonardo translates the PIN_NUMBER property to Altera-specific CHIP_PIN_LC property as shown in the following example:
Examples for pin location assignment:

```vhdl
--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;
```

```edf
---------- EDIF File inp.edf----------
(edif inpl
  (edifVersion 2 0 0)
  (edifLevel 0)

  (net A
   (joined
    (portRef A)
    (portRef IN (instanceRef II)))
   (property CHIP_PIN_LC (string chip1@4)))

  (design inpl
   (cellRef iinpl
    (libraryRef inpl))))
```

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 Leonardo and Altera's 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 Leonardo.

- If the hierarchy is preserved for hierarchical designs, Leonardo will write out hierarchical EDIF. If sub-modules have busses on the boundary, usually Leonardo will rename 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 will cause problems with Altera MAX+PLUS II EDIF reader since it does not handle the rename construct properly.

You can solve the problem by setting the Leonardo variable hdl_array_name_style to the Altera bus format.

```
set hdl_array_name_style = %s%d
```
In such a case Leonardo will create the following EDIF:

\[
\text{(port Q0 (direction OUTPUT))} \\
\text{(port Q1 (direction OUTPUT))} \\
\text{(port Q2 (direction OUTPUT))} \\
\text{(port Q3 (direction OUTPUT))} \\
\text{(port Q4 (direction OUTPUT))} \\
\text{(port Q5 (direction OUTPUT))} \\
\text{(port Q6 (direction OUTPUT))} \\
\text{(port Q7 (direction OUTPUT))})})
\]

Such EDIF will be handled correctly by MAX+PLUS II.

A Leonardo script called `write_altera` will automate the above operations. The `write_altera` script writes your design out as an EDIF file that is acceptable by MAX+PLUS II and should be used in place of the Leonardo `write` command.

**LMF file:** To take the EDIF generated by Leonardo 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 Leonardo 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 turn 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 it `EDIF_INPUT_LMF1 = exemplar.lmf`. Search for `EDIF_INPUT_USE_LMF1` and make it `EDIF_INPUT_USE_LMF1 = ON`.

**Using FLEX Designs as Input to Leonardo**

**EDIF Input**

To retarget a FLEX design into other technologies or to optimize a FLEX design using Leonardo, 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. (Leonardo
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 Leonardo, with the FLEX library as the source technology, and synthesize to the chosen technology. In Leonardo, first load the technology library, then read the design:

Leonardo>load_lib flex 8
--source lib flex 8 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 Leonardo, 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, Leonardo has a special command for reading Altera files: read_altera.

How to Get Best Results with MAX+PLUS II

Leonardo performs architecture specific optimization for Altera FLEX devices. Altera’s MAX+PLUS II then places and routes the netlist. This section discusses the settings required to allow MAX+PLUS II and Leonardo to produce the best design.

Altera Synthesis with Leonardo

When Leonardo is run targeting Altera FLEX, it will first optimize the design and then map the logic into logic cells.

By default, Leonardo enforces the mapping by writing LCELLs into the EDIF netlist. Leonardo 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 direct Leonardo not to enforce the LCELLs in the EDIF output netlist. In the Output Design Technology field, select one of the Altera FLEX technologies, then click on the Output Options button to bring up the Altera FLEX Output Options dialog.
The ▶ Lock LCells option is selected by default. Click to clear this option. The command line option `-dont_lock_lcells` can also be used for this option. Refer again to Table 9-1 and Screen 9-1. Leonardo will produce an EDIF netlist expressed in terms of AND-OR primitives only. Mapping to LCELLs, Carry, and Cascade will be done by MAX+PLUS II.

**Note:** 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 Leonardo to map to FLEX primitives or 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 will not use the carries or cascades that Leonardo writes.

#### Flow 1: Altera Primitives are Mapped by Leonardo

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

1. Load MAX+PLUS II with the EDIF netlist from Exemplar. Ensure that any .xnf, .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 will remove cascade and carry chains and will generally degrade FLEX results.

**Consider the following options:**

- **Automatic FAST I/O** can be helpful, improving area and clock to out delays. But it can 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. However, if it makes the design fail to fit or route, then turn it off.

- **Automatic implement in EAB**. This option, which only applies to FLEX 6K and FLEX 10K, is often helpful in moving random logic into available EABs, which improves area, and often improves delay. There are some cases where it can hurt delay. 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**

If you are not satisfied with the results from Leonardo's mapper, then this is an alternative flow. In some cases, mapping in MAX+PLUS II will 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 .xnf, .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 will remove 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. But it can reduce the maximum clock frequency, add input hold time for FLEX 8K, and increase input setup time for FLEX 10K. So it improves area, but can impose a delay penalty.
   - **Automatic Register Packing** can improve area and delay. Exemplar recommends this option. If it makes the design fail to fit or route, then turn it off.

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 Leonardo 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 $f_{max}$ to be 10% to 15% faster than the frequency as estimated by Leonardo. From the experiments completed at Exemplar, this setting improves MAX+PLUS II results by an average of 10%.

**New Features for Altera FLEX**

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

Leonardo can automatically setup the ACF (Altera constraint file for MAX+PLUS II and launch MAX+PLUS II from Leonardo's GUI. MAX+PLUS II will be launched in a batch mode, and its output will be directed to a Leonardo screen. You can control MAX+PLUS II functionality from Altera's 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

**New Command Line Switches**

If running Leonardo in batch mode MAX+PLUS II functionality will be available through the following new command line switches:

- `-maxplus2` launch MAX+PLUS II in batch mode
- `-no_acf` suppress generation of ACF file
- `-max_ta_reg` analyze register performance
- `-auto_fast_io` enable auto fast IO
- `-auto_register_packing` enable auto register packing
- `-max_exe<executable>` provide full path name of MAX+PLUS II executable file if MAX+PLUS II is not in search path.
**ACF File Generation**

Leonardo generates Altera constraint file with all necessary and recommended settings. **Note:** Leonardo cannot process the existing ACF files. These files will be 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

You have the option of using the MAX+PLUS II software or of pointing and clicking in Leonardo to setup MAX+PLUS II to create an ACF file to be read by MAX+PLUS II and applied to your design. Refer to Table 9-3 and Screen 9-3. These GUI options build your ACF file for MAX+PLUS II.
Table 9-3. GUI Options and Results

<table>
<thead>
<tr>
<th>GUI Options</th>
<th>Option on/off</th>
<th>Comments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock LCells</td>
<td>on*</td>
<td>click to clear default</td>
<td>Allows MAX+PLUS II to map logic cells</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>command line option <code>-dont_lock_lcells</code></td>
<td></td>
</tr>
<tr>
<td>Input EDIF File</td>
<td></td>
<td>EDIF file that is output from optimization.</td>
<td>Part of the ACF file.</td>
</tr>
<tr>
<td>Input Target</td>
<td></td>
<td>Target technology</td>
<td>Part of the ACF file</td>
</tr>
<tr>
<td>Input Part</td>
<td></td>
<td>Part number for target technology</td>
<td>Part of the ACF file</td>
</tr>
<tr>
<td>Input Speed Grade</td>
<td></td>
<td>Speed grade for target technology</td>
<td>Part of the ACF file</td>
</tr>
</tbody>
</table>

**RUN (Select ☐ radio button)**

- Batch Mode Use Batch Mode to setup ACF file. Runs MAX+PLUS II in batch mode.
- GUI Use GUI. Runs MAX+PLUS II GUI.
- Auto Fast IO off* option disabled Compiles MAX+PLUS II on enable auto fast io
- Auto Register Packing off* option disabled Compiles MAX+PLUS II on enable auto register packing

**OPTIMIZE (Select ☐ radio button)**

- Area Minimize design area in MAX+PLUS II Run Area again to improve results
- Delay Minimize design delay in MAX+PLUS II Run Delay again to improve results

**TIMING ANALYSIS (Select ☐ radio button)**

- None Select None if timing analysis is not needed. Timing analysis is disabled.
- Setup/Hold Report Select a report if desired. Enables timing analysis reports

*GUI Default Options
Screen 9-3. MAX+PLUS II Input Options for ACF File
New Features for Altera FLEX

Altera FLEX 6K

Leonardo has support for the new FLEX 6K technology. It 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, Leonardo 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.

Enhancements to Modgen

Existing Libraries

FLEX 10K: adjusted cascade chain lengths for eq 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, FLEX 8K, FLEX 10K

LPM (Library of Parameterized Modules) Multiplier

The LPM multiplier provides better results. **Note:** Timing and area estimates will not be accurate.

Inferencing of LPM RAMs and Counters

Leonardo 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:

```vhd
(cellref LUT
 (libraryRef FLEX10K)
 )

(property lut_function
 (string "(IN1 In2)"
 )
```

This makes EDIF netlists generated by Leonardo 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 Leonardo 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.
### Altera FLEX Family Supported Devices

#### FLEX 6K Family
- Default Speed Grade: 2
- Speed Grades supported: 2, 3

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPF6016</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 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>EPF10K50</td>
</tr>
<tr>
<td>EPF10K70</td>
</tr>
<tr>
<td>EPF10K100A</td>
</tr>
<tr>
<td>EPF10K130V</td>
</tr>
<tr>
<td>EPF10K250A</td>
</tr>
</tbody>
</table>
Leonardo provides synthesis for Altera MAX5000, MAX7000, and MAX9000 families of high density PLDs, and provides synthesis for these families into any of the other supported technologies. This chapter presents the following information:

- Before Beginning
- Altera MAX Architecture
- Design Flow
- Library Mapping File (LMF)
- Synthesis and Optimization Features
- Optimization Style Points
- Altera MAX Reports
- Design I/O
- Additional Options
- Using MAX Designs as Input to Leonardo

**Before Beginning**

The technology information presented in this chapter assumes that you have read the introductory chapters in this guide. Refer also to the HDL Synthesis and Command Reference guides for information.

**Altera MAX Architecture**

The basic Altera MAX structures are:
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, MAX9000) 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 Leonardo 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 5000, 7000, or 9000 devices, optimization in Leonardo 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.

If you do not want Leonardo to lock LCELLs, set the variable `dont_lock_lcells` to TRUE before running optimization. Leonardo will write out SOFT buffers instead of LCELL buffers. By default Leonardo locks LCELLS by using LCELL buffers because in most cases it gives better results after MAX+PLUS II place and route. Refer to Table 10-1 and Screen 10-1.
Table 10-1. Mapping Options

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Option</th>
<th>Command Line**</th>
<th>Available in Families</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>□ Do not lock LCELLS</td>
<td>on</td>
<td>dont_lock_lcells</td>
<td>All</td>
<td>Controls locking of LCELLS</td>
</tr>
<tr>
<td></td>
<td>off*</td>
<td>default (false)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*GUI default options

**Variables, options, constraint attributes, commands

Screen 10-1. Lock LCELLS for Altera MAX 7000
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 Leonardo's 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. Leonardo 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 if it is not already named. You can specify pin location from the Leonardo 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.
Examples for pin location assignment:

```
--library exemplar;
--use exemplar.exemplar_l164.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 inpl
  (edifVersion 2 0 0)
  (edifLevel 0)
  (net A
    (joined
      (portRef A)
      (portRef IN (instanceRef II))
      (property CHIP_PIN_LC (string chip1@4)))
  (design inpl
    (cellRef inpl
      (libraryRef inpl)))
```
**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 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 as Expanders, 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. This is the fastest performing result if it fits.
- **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. Therefore, it is recommended to try and fit the other results as well, even if the first pass seems to be the smallest one.
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 and MAX9000 architectures. Module generation support for MAX7000 can also be used by designers targeting the MAX5000 architecture, although usually with less optimal results.

Sequential Optimization

Leonardo eliminates registers and latches when it is determined that 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.

Leonardo does not “move” registers or change the phase of the signal stored in the registers.

Reporting

Refer to Chapter 6, Reports, in this guide for examples and more 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.
For example:

```
BUFFER_SIG SCLK clock1
BUFFER_SIG GLOBAL oe
```

Design rules for using these primitives (number of primitives, signals connected to primitives, etc.) will not be violated by Leonardo.

**Additional Options**

**Max Fanin and Max Cubes**

There are two important variables that control the optimization. You can adjust them and re-optimize if a fit cannot be achieved for any of the passes.

```
set max_fanin <n>
```

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 defaults 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 \( \text{max}\_\text{pt} \) are:

- max5 technology:30
- max7 technology:20
- max9 technology:20

Appropriate defaults are picked when \( \text{max}\_\text{fanin}/\text{max}\_\text{pt} \) is set to 0.

**Using MAX Designs as Input to Leonardo**

**EDIF 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 will write 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 Leonardo, 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.
This chapter provides guidelines for the synthesize of Lucent ORCA field programmable gate arrays (FPGAs). This chapter is divided as follows:

- Before Beginning
- FPGA Architecture
- Design Flow
- 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
- More New Features for ORCA

**Before Beginning**

The technology information presented in this chapter assumes that you have read the introductory chapters in this guide. Refer also to the HDL Synthesis and Command Reference Guides for information.
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. Leonardo supports the following families of ORCA devices:
- ORCA-1C (ORCA-1c)
- ORCA-2C (ORCA-2c)
- ORCA-2CA (ORCA-2a)
- ORCA-2TA (ORCA-2a)
- ORCA-3C (ORCA-3c)
- ORCA-3T (ORCA-3t)

When targeting ORCA 2CA or ORCA 2TA, Leonardo 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 ORCA 3C.

(2) ORCA3C modgen library does not use PFU gates.

The flow and interface between Leonardo 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

Exemplar Logic developed ORCA-specific synthesis and optimization algorithms to take full 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 (E-1):

\[ X = (A \cdot (B+C)) + (B \cdot D) + (E \cdot F \cdot G \cdot H \cdot I) \]  \hspace{1cm} (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:

\[
\begin{align*}
X &= T1 + T2 + T3 \\
T1 &= A \cdot T4 \\
T2 &= B \cdot D \\
T3 &= E \cdot F \cdot G \cdot H \cdot I \\
T4 &= B + C
\end{align*}
\]  \hspace{1cm} (E-2)

Because T3 has more than four inputs, further decomposition is required:

\[
\begin{align*}
T3 &= E \cdot F \cdot G \cdot T5 \\
T5 &= H \cdot I
\end{align*}
\]  \hspace{1cm} (E-3)
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). Leonardo writes NOMERGE properties in the EDIF output to mark the boundaries of lookup tables.
Consider the following description of a 6-1 multiplexer:

\[
\begin{align*}
T1 &= A \cdot C2' \cdot C1' \cdot C0'; \\
T2 &= B \cdot C2' \cdot C1' \cdot C0; \\
T3 &= C \cdot C2' \cdot C1 \cdot C0'; \\
T4 &= D \cdot C2 \cdot C1' \cdot C0'; \\
T5 &= E \cdot C2 \cdot C1' \cdot C0; \\
T6 &= F \cdot C2 \cdot C1 \cdot C0'; \\
\text{OUT} &= T1 + T2 + T3 + T4 + T5 + T6;
\end{align*}
\]

*Figure 11-3 Example of ORCA Lookup Table Mapping*
Leonardo Variables to Control Optimization and Mapping

The following GUI options and command line switches control the optimization and mapping algorithms:

Table 11-1. Mapping Options

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Option on/off</th>
<th>Command Line**</th>
<th>Available in Technologies</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Look Up Tables</td>
<td>on*</td>
<td>default (true)</td>
<td>All Listed LUT-based FPGAs</td>
<td>Controls LUT mapping.</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>lut_map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Map to 6-input LUTs</td>
<td>on</td>
<td>use_f6_lut</td>
<td>ORCA</td>
<td>Controls mapping to 6-Input LUTs. Note: Use the -k option to the ORCA Foundry Mapper.</td>
</tr>
<tr>
<td></td>
<td>off*</td>
<td>default (false)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not applicable</td>
<td>max_fanin &lt;number&gt;</td>
<td>ORCA</td>
<td></td>
<td>Specifies the maximum number of fanins for a single function. The default value of max_fanin area mode is 4; delay mode is 6.</td>
</tr>
</tbody>
</table>

*GUI default options

**variables, options, constraint attributes, commands
Screen 11-1. ORCA 1C Output Options
Constraint-Driven Timing Optimization

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, in which case it will try to improve the longest path.

```
oopimize_timing -force
```

The optimize_timing command restructuring 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 ORCA architecture. Using the PFU gate, it tries to map a critical signal directly to the input of the PFU-MUX gate, bypassing the LUTs in the PFU. However, this may not always be feasible.

Also the specific end point or instance to improve can be specified to the optimize_timing command with the -through option.

```
oopimize_timing -through
```

Data Path Synthesis (Modgen)

Leonardo 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 2C, 1C, 2CA, and 2TA:

- Logical Operators: and, or, nand, nor, xor, xnor
- Relational Operators: =, /=, <, <=, >, >=
- Arithmetic Operators: +, -, *
- Misc. Functions: incrementer, decrementer, absolute value, unary minus, counters, RAMs (synchronous and asynchronous for 2A only), multiplexers

Using ORCA Architectural Features

Use of Enabled D-Type Flip-Flops

Mapping to enabled D-type flip-flops can be controlled with the use_dffenable variable. This variable allows Leonardo to infer clock enable from HDL description. The default is TRUE.

When reading HDL designs with use_dffenable value set to TRUE, Leonardo infers clock enable logic while parsing HDL code. Detecting the clock enable early in the synthesis flow achieves better results.

```
use_dffenable true
```

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. A STARTUP block is instantiated automatically if global_sr is set to a signal name (my_signal) 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.

```
set global_sr <my_signal>
```
Setting this Tcl variable allows Leonardo to use the specified "Active High Signal" as Global Set or Reset. The signal is disconnected from the registers Set/Reset pin and connected to a startup block.

**Caution** – Specifying the Global Set/Reset on an active low signal will create incorrect logic.

```tcl
set infer_gsr TRUE
```

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 Screen 11-2 and Table 11-2.

**Table 11-2. Global Set/Reset**

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Option on/off</th>
<th>Command Line**</th>
<th>Available in Families</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect the Global Set/Reset Signal</td>
<td>on*</td>
<td>infer_gsr</td>
<td>All</td>
<td>Detects the global set/reset signal.</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>default true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assign Global SR: my_signal</td>
<td></td>
<td>global_sr</td>
<td></td>
<td>Defines an active high signal name as global set/reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>default not set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not applicable</td>
<td></td>
<td>delete_startup</td>
<td></td>
<td>If set to true then Leonardo does not instantiate a startup block.</td>
</tr>
</tbody>
</table>

*GUI default options
** options, variables, constraint attributes, commands

The `delete_startup` variable allows Leonardo to process global set/reset when the `optimize` command is called with the `-macro` option.

```tcl
set delete_startup TRUE
```
Screen 11-2. Global Set/Reset
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
    -- 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;
```
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
    elseif 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
Leonardo 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 Leonardo 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); Leonardo 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
  10: 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);
begin
    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.

**ORCA 1C/2C/2A Modgen Support for RAMs**

The ORCA 1C and ORCA 2C Modgen Libraries support asynchronous 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;
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 mem;

architecture rtl of mem is
  signal mem : MEM_WORD;
begin
  process (we, addr, din)
  begin
    if (we = '1') then
      mem(addr) <= din;
    end if;
  end process;
  dout <= mem (addr);
end rtl;
-- Synchronous Single Port Ram With Bidirectional Data
-- Inference : Leonardo infers RCE16x4 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, 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, Leonardo will infer a synchronous \texttt{ram\_io} cell.

**RAM Instantiation**

You can use \texttt{scuba} (ORCA Foundry tool) to generate RAMs. This tool can be used in the input HDL design.

When using \texttt{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 itself. For example, the following memgen invocation will create a 256x6 RAM and write it 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 Leonardo the resulting EDIF file will contain a symbol myram.

To set the `initval` attribute on the RAM instance in Verilog, use the `set_attribute` command in Leonardo 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;
```
Example 8b. Verilog RAM Usage Example

```
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.edf run:
// scuba -addr_width 8 -data_width 6 -lang vhdl -n myram -type aspram
endmodule
```

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 INITV AL 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, Leonardo only maps to simple registers and registers with multiplexed data inputs. It does not map to Synchronous Set/Reset registers.

Use the variable `include_gates`, to allow Leonardo to use the Synchronous Set/Reset registers, as shown in the following example.

```
set include_gates "FD1S3JX_Q FD1S3IX_Q FD1P3IZ_Q FD1P3JZ_Q FD1S1I_Q FD1S1J_Q"
```

Where:

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD1S3JX_Q</td>
<td>Synchronous Set DFF without Clock Enable</td>
</tr>
<tr>
<td>FD1P3IZ_Q</td>
<td>Synchronous Set DFF with Clock Enable</td>
</tr>
<tr>
<td>FD1S3IX_Q</td>
<td>Synchronous Reset DFF without Clock Enable</td>
</tr>
<tr>
<td>FD1P3IZ_Q</td>
<td>Synchronous Reset DFF with Clock Enable</td>
</tr>
<tr>
<td>FD1S1I_Q</td>
<td>Synchronous Reset DLATCH</td>
</tr>
<tr>
<td>FD1S1J_Q</td>
<td>Synchronous Set DLATCH</td>
</tr>
</tbody>
</table>

This command allows Leonardo to use the Synchronous Set/Reset registers selectively. For example, `set include_gates FD1P3IZ_Q FD1S3IX_Q`. This variable should be set before loading the synthesis library.

I/O Mapping

Leonardo 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 Leonardo where appropriate.
All other I/O buffers, as described in the ORCA Library section of the ORCA Foundry Manual, can be inserted from the Leonardo shell using the PAD command or instantiated by component instantiation in VHDL or Verilog HDL.

**ORCA Reporting**

Refer to Chapter 6, Reports, in this guide for examples and information on Leonardo reporting.

**Generation of ORCA Foundry Properties from Leonardo**

**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 Leonardo procedures or 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, Leonardo 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

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 (1C, 2C, 2CA, 2TA) part number for the device into which the design is programmed. Specify the part number in Leonardo by setting a variable called `part` (e.g., in the Leonardo shell set `part att2c04M84`). Leonardo writes out the specified Part number as PICSPEC property in output EDIF in the properties section of top-level view.

```
part att2c04M84
```

Speed Grade - PICSPEED

PICSPEED specifies the speed of the target ORCA device into which the device is programmed. Specify the Speed Grade in Leonardo by setting a variable called `process` (e.g., in the Leonardo shell set `process 2`).

```
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 properties on ports of a design specified in VHDL, or from the Leonardo 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.

ORCA Preference File Writer

The ORCA preference file writer translates timing constraints into an ORCA preference file.

Description

Timing constraints are written to the ORCA preference file. Your specified arrival_time, required_time, clock_cycle, clock_offset, and pulse_width are converted to ORCA preferences.

The absolute time constraints used by Leonardo are translated into the relative constraints used by Foundry, as illustrated in the following examples:

Note: Relative delays are between I/O ports, while periods are written for internal nets.
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;  
  );
end freq_div_4;
architecture Exemplar of freq_div_4 is begin
  div_by_4: 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...

```vhdl
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;
```
Preference File

The gc demo.vhd demo.edf -target=orca2c -modgen=orca2c -preference_file=demo.prf Leonardo option produces the demo.prf preference file.

Only the demo.prf file has preferences on primary I/Os. The preference file corresponds to timing constraints that are specified in VHDL attributes in demo.vhd.

Preference File demo.prf

SCHEMATIC START ;

SCHEMATIC END ;
# USER Defined Preferences...
PERIOD NET clk_int 15.000000 ns HIGH 9.000000 ns;
OFFSET IN COMP en 3.000000 ns BEFORE COMP clk;
OFFSET OUT COMP xmplr_net_20 16.000000 ns AFTER COMP clk;
# End file: demo.prf:

Additional Writing Preference File Features

Leonardo writes timing constraints to the preference file. Your specified arrival_time, required_time, clock_cycle, clock_offset, pulse_width are converted to ORCA preferences. These preferences are described in ORCA Foundry 9.0 Properties and Attributes, Lucent Technologies, 12/2/96. EDIF naming conventions are used for the instance and net names written into the preference file. This ensures that the preference file names match the names in the EDIF netlist.

The absolute timing constraints used by Leonardo are translated into the relative constraints used by Foundry. Refer to the following constraint file example. Note that relative delays are between I/O ports, while periods are written for internal nets.

Constraint File:
CLOCK_OFFSET 10 clock
ARRIVAL_TIME 5 sensor1
CLOCK_CYCLE 30 clock
PULSE_WIDTH 12 clock
REQUIRED_TIME 22 red1

**Resulting Preference File:**

```
SCHEMATIC START;
SCHEMATIC END;

# USER Defined Preferences...
period net CLOCK_int 30.000000 ns high 12.000000 ns;
offset IN comp SENSOR1 5.000000 ns BEFORE comp CLOCK;
offset OUT comp RED1 12.000000 ns AFTER comp CLOCK;
```
## Devices Supported for Lucent ORCA FPGA Families

### Lucent ORCA 1C family

- **Default Speed Grade:** 2
- **Speed Grades supported:** 0, 1, 2

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>att1c03</td>
<td>M84 T100 F132 T144 S208 H225 R225</td>
</tr>
<tr>
<td>att1c05</td>
<td>M84 T100 F132 T144 S208 H225 R225 S240</td>
</tr>
<tr>
<td>att1c07</td>
<td>S208 S240 R280 S304</td>
</tr>
<tr>
<td>att1c09</td>
<td>S208 S240 S304</td>
</tr>
</tbody>
</table>

### Lucent ORCA 2C family

- **Default Speed Grade:** 2
- **Speed Grades supported:** 2, 3

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>att2c04</td>
<td>M84 T100 T144 J160 S208</td>
</tr>
<tr>
<td>att2c06</td>
<td>M84 T100 T144 J160 S208 S240</td>
</tr>
<tr>
<td>att2c08</td>
<td>M84 J160 S208 S240 B256 S304</td>
</tr>
<tr>
<td>att2c10</td>
<td>M84 S208 S240 S304</td>
</tr>
<tr>
<td>att2c12</td>
<td>S208 S240 S304 R364</td>
</tr>
<tr>
<td>att2c15</td>
<td>S208 S240 S304 R364</td>
</tr>
<tr>
<td>att2c26</td>
<td>PS208 PS240 PS304 R428</td>
</tr>
<tr>
<td>att2c40</td>
<td>PS208 PS240 PS304 R428</td>
</tr>
<tr>
<td>att2c15</td>
<td>S208 S240 S304</td>
</tr>
</tbody>
</table>
Lucent ORCA 2CA family

Default Speed Grade: 4

Speed Grades supported: 2, 3, 4, 5

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>or2c04a M84 T100 T144 J160 S208</td>
</tr>
<tr>
<td>or2c06a M84 T100 T144 J160 S208 S240 B256</td>
</tr>
<tr>
<td>or2c08a M84 J160 S208 S240 B25 M84</td>
</tr>
<tr>
<td>or2c10a J160 S208 S240 B256 B352</td>
</tr>
<tr>
<td>or2c12a M84 S208 S240 B256 S304 B352</td>
</tr>
<tr>
<td>or2c15a M84 S208 S240 B256 S304 B352 SB432</td>
</tr>
<tr>
<td>or2c26a PS208 PS240 PS304 B352 SB432 SB600</td>
</tr>
<tr>
<td>or2c40a PS208 PS240 PS304 SB432 SB600</td>
</tr>
</tbody>
</table>

Lucent ORCA 2TA family

Default Speed Grade: 4

Speed Grades supported: 2, 3, 4, 5

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>or2t04a M84 T100 T144 J160 S208</td>
</tr>
<tr>
<td>or2t06a M84 T100 T144 J160 S208 S240 B256</td>
</tr>
<tr>
<td>or2t08a M84 J160 S208 S240 B256</td>
</tr>
<tr>
<td>or2t10a M84 J160 S208 S240 B256 B352</td>
</tr>
<tr>
<td>or2t12a M84 S208 S240 B256 B352</td>
</tr>
<tr>
<td>or2t15a M84 S208 S240 B256 B352 SB432</td>
</tr>
<tr>
<td>or2t26a PS208 PS240 B352 SB432 SB600</td>
</tr>
<tr>
<td>or2t40a PS208 PS240 SB432 SB600</td>
</tr>
</tbody>
</table>
Lucent ORCA ATT3 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

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>or3c55S</td>
</tr>
<tr>
<td>or3c55B</td>
</tr>
<tr>
<td>or3c80S</td>
</tr>
<tr>
<td>or3c80SB</td>
</tr>
</tbody>
</table>
New Features for ORCA

- Neoprim Library: The Neoprim Library was added to support back annotation flow for Lucent ORCA technologies. For successful timing back annotation use VHDL netlist generated by the following invocation:
  
  \texttt{ngd2vhd -n -t <input> <output>}

- RAMs: ORCA 1C, 2C, 2A libraries: RAMs with data size mod 4 = 3
- ORCA 3C and ORCA 3T technologies were added. Refer to the FPGA Architecture section in this chapter.

ORCA Preference File

ORCA logical preference file writer was added to release 4.2. 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.

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>or3t80 S208</td>
</tr>
<tr>
<td>or3t80 S240</td>
</tr>
<tr>
<td>or3t80 B352</td>
</tr>
<tr>
<td>or3t80 SB432</td>
</tr>
</tbody>
</table>

Lucent ORCA 3t family

Default Speed Grade: 5

Speed Grades supported: 5, 6
Translation from logical to physical preference file:

Note: You must use Foundry version 9.2 or higher.

- Run Leonardo targeting ORCA. Leonardo will automatically generate 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 Leonardo will pass it to the ORCA preference file. Leonardo timing optimization and delay reporting are sensitive to multicycle paths.

Example:

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;

Mapping to ORCALUTs

Leonardo maps directly to ORCA Foundry’s ORCALUTs. This improves design results.
Leonardo provides synthesis support for the QuickLogic pASIC 1, pASIC 2 and pASIC3 devices. Designs may be entered using VHDL or Verilog description. Additionally, 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
- New Features for QuickLogic
- Design Flow
- Optimization Style Points
- QuickLogic Reports
- Process Derating Factors
- Design I/O
- Assigning Pin Locations

**Before Beginning**

The technology information presented in this chapter assumes that you have read the introductory chapters in this guide. Refer also to the HDL Synthesis and Command Reference guides for information.
FPGA Architecture

The 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.

New Features for QuickLogic

- Support for pASIC1 and pASIC2 is added. 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 Leonardo 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 Leonardo. The result is an EDIF netlist that is used by downstream SpDE tools.
Leonardo currently does not accept QDIF files as input. However, pASIC designs in EDIF netlist are accepted. The EDIF netlist can be hierarchical as long as all the modules are described in a single EDIF file.

**Optimization Style Points**

**Quality of Optimization**

Certain types of logic will be optimized better than others: arithmetic circuits such as adders and multipliers will not be 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 will be optimized as well as or better than the best hand designs.

**Combinational Logic Loops**

By default, Leonardo 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**

Leonardo eliminates registers and latches if the register or latch will always be latching a constant value, and there is no set or reset used on the register. Leonardo also eliminates registers or latches that have identical inputs; these registers are effectively merged.

Leonardo 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. Thus, all tristate gates must be connected to a pad. If your design has tristates, Leonardo issues the message:

```
Internal tristates not available in this technology.
```

The internal tristates must be converted to combinational logic (by setting the `tristate_map` variable to TRUE) or to a tristate or bidirectional pad (using the -chip option).

**QuickLogic Reports**

Refer to Chapter 6, Reports, in this guide for examples and information.

**Process Derating Factors**

The following table for pASIC1 and pASIC2, list derating factors for these libraries. These values are entered on the Leonardo command line.
Design I/O

Leonardo maps I/O cells automatically. Based on the input description the appropriate I/O cells from the pASIC technology library will be used. Leonardo 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 \texttt{PAD} and \texttt{BUFFER_SIG} command.

Manual Assignment of I/O Buffers

There are two commands that can be used to manually assign I/O pads: \texttt{PAD} and \texttt{BUFFER_SIG}. These commands can be used as attributes from VHDL, or as commands from the Leonardo shell. The \texttt{PAD} command is the recommended way to assign pads since it is the most general, and because it can be used to assign any pad that Leonardo is able to map. The \texttt{PAD} command is limited, however, and only works on inputs and outputs. The \texttt{BUFFER_SIG} command can only be used to assign buffers, but it can be used on internal signals (internal clock buffers) as well.
**PAD Command**

When using this command from the Leonardo shell, the syntax is:

```
PAD IO_gate_name_signal_1 . . . signal_n
```

The following I/O gates can be manually mapped using the `PAD` command:

**Input buffers:**
- INPAD
- HDAD
- HD2PAD
- HD3PAD
- HDIPAD
- HD3PAD

**Output buffers:**
- OUTPAD
- OUTORPAD
- OUTIPAD

**Tristate buffers:**
- TRIPAD
- TRIORPAD
- TRIPAD

**Bidirectional buffers:**
- BIPAD
- BIORPAD
- BIIPAD
BUFFER_SIG Command

It is useful to manually assign clock buffers. For example:

```
BUFFER_SIG clk CKPAD
```

The following clock buffers are available:
- CKPAD
- CKDPAD
- 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 Leonardo’s 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;
```

Here is an example of assigning pin numbers from Leonardo’s shell:

```bash
set_attribute -port -name pin_number -value 6 din (0)
```
Leonardo provides synthesis for all families of Xilinx CPLDs, which includes XC7200A, XC7300, and XC9500. Designs may be entered using standard VHDL and Verilog descriptions or EDIF netlist. Leonardo also provides synthesis for these families into any of the other technologies supported by Leonardo. 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 and Optimization Features
- Other Synthesis Features
- Reporting
- Xilinx CPLD Family Supported Devices

**Before Beginning**

The technology information presented in this chapter assumes that you have read the introductory chapters in this guide. Refer also to the HDL Synthesis and Command Reference guides for information.
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, 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 or T flip-flop with clear and preset inputs.

These 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. Leonardo 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 Leonardo 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.
Leonardo uses dedicated algorithms to optimize logic for the Xilinx CPLD architectures. Leonardo removes redundant logic and prepares the design for the Xilinx CPLD software.

**Special Synthesis Feature**

The following features are for the XC9500 families only.

**TFF/DFF Mapping**

The macrocell register can be configured as a D-type or a T-type flip-flop. Leonardo maps the register so that it uses the appropriate flip-flop. This takes into account the tradeoff involved in implementing a D-type flip-flop as a T-type flip-flop.
**XOR Optimization**

XOR optimization looks for tradeoffs in implementing a macrocell as an XOR function as opposed to an OR gate.

**Other 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 will automatically select the appropriate device that fits your design. Optionally, you can specify the part number and speed grade or process of the target device from either the GUI or command line arguments.

**I/O Buffers and Pads**

Leonardo 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, Leonardo also inserts I/O PADS in the EDIF netlist.

The other I/O Buffers like OBUF_S, IBUF_S, OBUFE_S etc. are available through component instantiation.

**Pin (LOC) Assignment**

Leonardo 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**

Leonardo supports module generation for XC7200A and XC7300 families only.

Leonardo 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: \(+\), \(-\)
- Misc. Functions: incrementer, decremener

**User Options for XC9500 Families Only**

These options are available on the Output Options dialog for XC9500. Options not available on the GUI are entered in the Special Options field. Refer to Table 13-1 and Screen 13-1.

Table 13-1. GUI and Command Line Options

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Command Line*</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max PT</td>
<td><code>set max_pt &lt;integer&gt;</code></td>
<td>Determines the maximum number of product terms in a macrocell. Default values are 15 in area mode and 25 in delay mode. The higher product terms may result in a faster but bigger implementation.</td>
</tr>
<tr>
<td>Max Fanin</td>
<td><code>set max_fanin &lt;integer&gt;</code></td>
<td>Determines the maximum number of inputs to a function block. Default value is 36.</td>
</tr>
</tbody>
</table>

*variables, options, constraint attributes, commands
Screen 13-1. Example - Xilinx 9500 Output Options

**Reporting**

Leonardo reports on estimates of area of the optimized design as follows:

- Reports on the number of AND-OR gates can be issued during each optimization pass.
- Detailed reports on all cells used in the design can be issued after optimization.
  Type: `report_area -cell`

Refer to Reports, Chapter 6, in this guide for more information.
### Xilinx CPLD Family Supported Devices

<table>
<thead>
<tr>
<th>Xilinx CPLD Family - Xilinx XC7200A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 20</td>
<td></td>
</tr>
<tr>
<td>Speed Grades supported: 16, 20, 25</td>
<td></td>
</tr>
<tr>
<td><strong>Devices Supported</strong></td>
<td></td>
</tr>
<tr>
<td>7236 APC44 AWC44</td>
<td></td>
</tr>
<tr>
<td>7272 APC68 AWC68 APC84 AWC84 APG84</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Xilinx CPLD Family - Xilinx XC7300</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Speed Grade: 10</td>
<td></td>
</tr>
<tr>
<td>Speed Grades supported: 5, 7, 10, 12, 15, 20</td>
<td></td>
</tr>
<tr>
<td><strong>Devices Supported</strong></td>
<td></td>
</tr>
<tr>
<td>7336 PC44 WC44 PQ44 QPC44 QC44 QCWC44 QPQ44 QVQ44</td>
<td></td>
</tr>
<tr>
<td>7354 PC68 WC68 WC44 PC44</td>
<td></td>
</tr>
<tr>
<td>7372 PC68 WC68 PC84 WC84 PQ100</td>
<td></td>
</tr>
<tr>
<td>7310 8PC84 8WC84 8PQ100 8PG144 8PQ160 8BG225</td>
<td></td>
</tr>
<tr>
<td>7344 PQ160</td>
<td></td>
</tr>
<tr>
<td>7314 4BG225</td>
<td></td>
</tr>
</tbody>
</table>
## Xilinx CPLD Family - Xilinx XC9500

Default Speed Grade: 7

Speed Grades supported: 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>9510</td>
</tr>
<tr>
<td>9514</td>
</tr>
<tr>
<td>9521</td>
</tr>
<tr>
<td>9528</td>
</tr>
</tbody>
</table>
Leonardo provides full synthesis support for the following Xilinx LCA based architectures:

- XC3000/A/L, XC3100/A
- XC4000/A/E/EX/H/L/XL/XV, XC5200
- CPLD Family: XC7200A, XC7300, XC9500
- Spartan

Designs may be entered using VHDL or Verilog HDL 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 Design Flows
- Synthesis and Optimization Features
- Design I/O
- Using TimeSpecs
- Using Xilinx Attributes
- Additional Xilinx Specific Options
- EDIF and XNF
- Interfacing with XBLOX
- Additional Xilinx Architecture Features
- Xilinx FPGA Devices Supported
Before Beginning

The technology information presented in this chapter assumes that you have read the introductory chapters in this guide. Refer also to the HDL Synthesis and Command Reference guides for information.

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. Leonardo 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 EDIF M1 flow. You may also use XNF as an input to M1 software. This flow will not be supported in the future by Xilinx.
Xilinx LCA Synthesis

Figure 14-1 Simplified Design Flow
Figure 14-2  Simplified Back Annotation
**Design Flow Guidelines**

During Xilinx synthesis, the output options are available on the GUI and can be entered on the command line. Refer to Table 14-1 and Screen 14-1.

**User-Defined Variables**

```
set lut_map false
set write_lut_binding false
set write_clb_packing true
```

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Option on/off</th>
<th>Command Line**</th>
<th>Available in Families</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map Look Up Tables</td>
<td>on*</td>
<td>default TRUE</td>
<td>All except XC7200A, XC7300, XC9500</td>
<td>Controls LUT mapping for accurate reports.</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>lut_map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not applicable</td>
<td>default TRUE</td>
<td></td>
<td>XC9000</td>
<td>Controls packing of LUTs into CLBs.</td>
</tr>
<tr>
<td></td>
<td>write_lut_binding</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not applicable</td>
<td>write_clb_packing</td>
<td></td>
<td></td>
<td>Controls printing of CLB packing (HBLKNM) in the output netlist, as required.</td>
</tr>
<tr>
<td></td>
<td>default FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*GUI default options
** options, variables, constraint attributes, commands

Notes:
- If the variable write_lut_binding is set to FALSE the XNF output will not be in terms of MAP (FMAP/HMAP) symbols.
- 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 Leonardo CLB packing gave superior results over XACT. Also, by packing to CLBs, Leonardo can estimate interconnect delays more accurately. Currently, CLB packing is available for the XC4000 architecture only.
Using LUT mapping is recommended. Using LUT mapping in Leonardo gives significantly better results than using it during place and route. Also, mapping to LUTs gives Leonardo the ability to report area and delays more accurately.

Screen 14-1. Set Optimize Variables for XC4000E
**Synthesis and Optimization Features**

Leonardo 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 limitations of the Xilinx CLB.

- An important feature is the ability to map logic functions into lookup tables. This mapping is available for XC3000, XC4000, XC4000E, 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 have control of whether Leonardo 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 and wide edge decoders.

**Fanin Limited Optimization**

For the purposes of this example, the Xilinx 4000 architecture is used. 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 its 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:

\[
X = T_1 + T_2 + T_3 \\
T_1 = A \cdot T_4 \\
T_2 = B \cdot D \\
T_3 = E \cdot F \cdot G \cdot H \cdot I \\
T_4 = B \cdot C.
\]

Because T3 has more than four inputs, further decomposition is required:

\[
T_3 = E \cdot F \cdot G \cdot T_5 \\
T_5 = H \cdot I
\]

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:

Figure 14-3  CLB Packing After AND-OR Decomposition
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.

\[ H \rightarrow T5 \rightarrow \text{FMAP} (T3) \rightarrow \text{HMAP} (X) \]

When using fanin limited decomposition techniques, the decomposition of the equation yields better results:

\[
\begin{align*}
X &= T1 + (T2 \times E) \\
T1 &= A \times (B + C) + (B \times D) \\
T2 &= F \times G \times H \times I
\end{align*}
\]

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.

---

*Figure 14-4  CLB Packing After Fanin-Limited Decomposition*
Lookup Table (LUT) Mapping

LUT mapping maps fanin limited logic functions to Xilinx function generators. This technology is available for XC3000, XC4000, XC4000E, 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 4-1 multiplexer:

\[
\begin{align*}
T1 &= A \times C1' \times C2' \\
T2 &= B \times C1' \times C2 \\
T3 &= C \times C1 \times C2' \\
T4 &= D \times C1 \times C2 \\
\text{OUT} &= T1 + T2 + T3 + T4
\end{align*}
\]

Figure 14-5  LUT Mapping Targeting XC4000
**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.

![Figure 14-6 LUT Mapping Targeting XC5200](image)

Here the design is mapped using two F function generators and a F5_MUX gate.

**Constraint-Driven Timing Optimization**

The `optimize_timing` command can be used after `optimize` to improve the timing quality of the design. `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. `optimize_timing` may be invoked repeatedly for added improvements if possible.

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

```
optimize_timing -through
```

**CLB Packing for Xilinx XC4000 Technology**

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 is useful for two reasons:

1. You can get an accurate estimate of the design in terms of occupied CLBs.
2. The delay estimate on a post synthesized design (but pre place and route) 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 Leonardo's interactive shell.

```
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, 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.
These are the constraints:

- 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;
```

Leonardo Synthesis and Technology Guide
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, f,
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, q<1>, 3.000000
PIN, D, I, f,
END
```
Data Path Synthesis (Modgen)

Leonardo 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. Leonardo 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:

<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>
<tr>
<td>Misc Functions</td>
<td>counters (up/down, loadable, etc.)</td>
<td>RAMs</td>
<td>incrementer</td>
<td>decrementer</td>
<td>absolute value</td>
<td>unary minus</td>
</tr>
</tbody>
</table>

### Modgen Implementation

Module generators attempt to use dedicated hardware resources whenever 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_MUX`s 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 Variables

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

You can control modgen implementation that is optimized for area or for delay by using this switch. This switch controls modgen resolution for all operators. You can also use the GUI as explained in Table 14-2 and shown in Screen 14-2 to control modgen implementation.

Table 14-2. GUI Modgen Select and Results

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>GUI Option</th>
<th>Command Line**</th>
<th>Available in Families</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modgen Select:</td>
<td>Auto on*</td>
<td>Auto (default)</td>
<td>All</td>
<td>Picks smallest if optimization in area mode; and picks fastest if optimization in delay mode.</td>
</tr>
<tr>
<td></td>
<td>Smallest off</td>
<td>Smallest</td>
<td></td>
<td>Picks the most compact implementation available.</td>
</tr>
<tr>
<td></td>
<td>Small off</td>
<td>Small</td>
<td></td>
<td>Picks a compact implementation.</td>
</tr>
<tr>
<td></td>
<td>Fast off</td>
<td>Fast</td>
<td></td>
<td>Picks a fast implementation.</td>
</tr>
<tr>
<td></td>
<td>Fastest off</td>
<td>Fastest</td>
<td></td>
<td>Picks the fastest implementation available.</td>
</tr>
</tbody>
</table>

* GUI default option
** variables, options, constrained attributes, commands
Using Xilinx Architectural Features

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. Leonardo GSR support is enhanced to handle GSR processing across hierarchy. Global set/reset and local set/reset is detected on most flat or hierarchical designs.

A STARTUP block is instantiated automatically if \texttt{global\_sr} is set to a signal name (\texttt{my\_signal}) in the design, or if \texttt{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.

\begin{verbatim}
set global\_sr <my\_signal>
\end{verbatim}

Setting this Tcl variable allows Leonardo 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.

\textbf{Caution} – Specifying the Global Set/Reset on an active low signal will create incorrect logic.
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 Screen 14-3 and Table 14-3.

Table 14-3. Global Set/Reset - Example Xilinx 4000

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Option on/off</th>
<th>Command Line**</th>
<th>Available in Families</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect the Global Set/Reset Signal</td>
<td>on*</td>
<td>infer_gsr</td>
<td>All except XC7200A, XC7300, XC9500</td>
<td>Detects the global set/reset signal.</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>default true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assign Global SR: my_signal</td>
<td></td>
<td>global_sr</td>
<td></td>
<td>Defines an active high signal name as global set/reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>default not set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>not applicable</td>
<td></td>
<td>delete_startup</td>
<td></td>
<td>If set to true then Leonardo does not instantiate a startup block.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>default false</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*GUI default options
** options, variables, constraint attributes, commands

The delete_startup variable allows Leonardo to process global set/reset when the optimize command is called with the -macro option.

```
set infer_gsr TRUE
```

```
set delete_startup TRUE
```

```
optimize -chip (default) | macro
```
Screen 14-3. Set Optimize Variables for Global Set/Reset - Example Xilinx 4000
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. You can only Set/Reset with local S/R. Preserve the global set signal:

```
set xlx_preserve_gsr (default FALSE)
```

Preserve the global tristate signal:

```
xlx_preserve_gts (default FALSE)
```
**Example 1a: VHDL Example Using GSR 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
  -- 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.

Leonardo 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 Leonardo before instantiation, otherwise Leonardo 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

```
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

```verilog
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 multiplexers may be more efficiently implemented with tristate buffers.
Leonardo 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 Leonardo to either keep the internal tristate buffers or convert them to multiplexers. Refer to Screen 14-4 and Table 14-4 for setting tristate variable on the GUI.

Table 14-4. Tristate Map

<table>
<thead>
<tr>
<th>GUI Option</th>
<th>Option on/off</th>
<th>Command Line**</th>
<th>Available in Families</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑ Allow converting internal tristates</td>
<td>on</td>
<td>tristate_map</td>
<td>All LCA</td>
<td>Converts internal tristates to combinational logic.</td>
</tr>
<tr>
<td></td>
<td>off*</td>
<td>default false</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*GUI default options
** options, variables, constraint attributes, commands

```plaintext
set tristate_map
```
To keep the tristate buffers, use the default settings. To convert tristate buffers to muxes, set the `tristate_map` variable to TRUE.

Screen 14-4. Setting Variables - Example Xilinx 4000
Note – Leonardo does not check for potential bus conflict. You must be sure that only one driver is driving the bus.

Example 4a. VHDL Example Inferring an Internal Tristate Using Both a Component and Tristate Behavior

```
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 : 1’bz;
assign int_bus = ( en_b == 1 ) ? inbus_b : 1’bz;

buft u0 ( .i(inbus_c), .t(en_c), .o(int_bus) );
assign outbus = common ^ int_bus;

endmodule
```
Figure 14-10 and Figure 14-11 are the circuits generated by Leonardo.

![Figure 14-10](image1)

**Figure 14-10** tristate_map = FALSE (Default)

![Figure 14-11](image2)

**Figure 14-11** tristate_map = TRUE

**Using Clock Enable**

All DFFs in the Xilinx CLBs have a clock enable input. This feature can be used when using schematic entry. Leonardo 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 time.
This is when compared to the implementation of logically enabled DFFs by other means. For example, placing a multiplexer on the D-input with the clock enable controlling the mux.

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. Leonardo 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, Leonardo recognizes the logic and implements it in the target technology, if possible.
Using RAMs

Types of Inferencing RAMs

Leonardo 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 Leonardo 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); Leonardo 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;
beg
  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;

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 4000/4000E 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);
begin
    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, XC4000E, Leonardo 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

  raml6: 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.

```
-------------------------------Output xnf-------------------------------
 : 
 : SYM, RAM16, RAMD, SCHNM=RAM16x1D, LINVER=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:

```
---------Input Verilog Description--------------
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:

```
-----------------------Output xnf----------------------------
USER, AREA, 0
USER, PACKED_ESDTiMATE, 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, 0, o<0>_int, ,
PIN, o1, 0, 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 Leonardo 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 Leonardo.
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

Leonardo 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

Leonardo 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.
Leonardo Synthesis and Technology Guide

- 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. Leonardo uses BUFG for mapping to global buffers.
- XC5200: four global buffers are available.

Leonardo 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/UFUFFCLK 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:

```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;
```
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 please 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.

```
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 with this constraint will be

```
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
```

**User Options**

Leonardo tries to map to global buffers whenever possible by default (true). However, you have the option to switch off automatic global buffer mapping by specifying the following command:

```
set insert_global_bufs false
```

You can also click to clear the check box on the Set Optimize Variables GUI to switch off automatic global buffer mapping. Refer again to Screen 14-4 and to the Map to Clock/Global buffers option.

**Design I/O**

Leonardo maps design I/Os to I/O buffer cells (IBUF, OBUF, etc.) and registered I/O cells (OUTFF, INFF, etc.) automatically.

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. Leonardo maps to these registered I/Os when possible. You can click to clear the check box on the Set Optimize Variables GUI to switch off mapping to complex IO cells. Refer again to Screen 14-4 and to the Map to Complex IO cells option.

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;
```
Leonardo output for this design is:

```
---output xnf------------------------
:  
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, 0, 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,
:
:
```
IOB in Xilinx XC4000E architecture is enhanced and contains flip-flops with clock-enable. For XC4000E architecture, Leonardo maps to IOB flip-flops with clock-enable whenever possible.

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

tenity 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;
```
Leonardo output for this design is:

```
---output xnf---

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 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.

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.
Another way of specifying pin placement is from Leonardo’s shell. Use the shell if you wish to keep pin information separate from the VHDL description, or if you are using Verilog.

Note – Leonardo’s 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**

Leonardo provides estimates of the area and critical path delay for the synthesized designs. These estimates are provided primarily for the purpose of comparing different design implementations, and also for getting a quantitative estimate of design performance. Refer to Chapter 6, Reports, for examples and more information.

**Using Timespecs**

Timespecs are Xilinx-defined primitives that guide Xilinx place and route by specifying timing restrictions on signals in the design. Leonardo can write timespec annotated XNF netlists.

There are two ways in Leonardo to specify TimeSpecs values.

- Using timing constraints commands (i.e., CLOCK_CYCLE)
- Specify attributes in VHDL
You can specify the Timespec design on the Generate TIMESPEC dialog. This dialog comes up by clicking on TIMESPEC on the Optimize menu on the GUI. Refer to Screen 14-5. When you click on Generate TIMESPEC, Leonardo processes the design file and displays the results.

Screen 14-5. Generate TIMESPEC

**User Interaction**

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.** Leonardo 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 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.
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 shell (or using the constraint editor):

```
CLOCK_CYCLE 20 clock
CLOCK_OFFSET 5 clock
PULSE_WIDTH 10 clock
ARRIVAL_TIME 3 inputa
```

**TIMESPECs**

Leonardo 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 \textit{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, Leonardo writes out a TIMESPEC of the form

\[
\text{TS}nn=\text{FROM:} \text{group1: TO:} \text{group}_2=\text{constraint}
\]

If \textit{group}_2 represents a set of D-inputs clocked by CLK, the value of \textit{constraint} is determined by the formula:

\[
\text{constraint} = (\text{CLK}\cdot\text{period} + \text{CLK}\cdot\text{offset} - \text{group_d}\cdot\text{setup}) - \text{group1}\cdot\text{arrival}
\]
For example, read the following VHDL file into Leonardo:

```vhdl
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;
```
This file together with the following constraints, writes out TIMESPEC as shown in
the following XNF output. The constraint file determines your requirements, which are
clock cycle, arrival time on primary inputs and required times on primary outputs.

```
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:

```
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, red1, O, , FAST, TNM=GROUP_3
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
```
Here is another example, read the following VHDL file into Leonardo:

```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
CLOCKCYCLE 20 clk
generate timespec
```
The following XNF file will be generated:

```
LCANET, 5
USER, LIBRARY_NAME, work
USER, CELL_NAME, dff_reg
USER, VIEW_NAME, exemplar
PROG, Leonardo, V4.0.0b (Beta), "Fri Jul 26 16:06:00 1996"
EXT, a<1>, I, , TNM=GROUP_1
EXT, a<0>, I, , TNM=GROUP_1
EXT, clk, I, , TNM=GROUP_1
EXT, o<1>, O,
EXT, o<0>, O,
SYM, XMPLR_INST_3, IBUF, LIBVER=2.0.0, SCHNM=IBUF
PIN, I, I, a<0>,
PIN, O, O, a<0>_int, 3.000000
END
SYM, XMPLR_INST_4, IBUF, LIBVER=2.0.0, SCHNM=IBUF
PIN, I, I, a<1>,
PIN, O, O, a<1>_int, 3.000000
END
SYM, XMPLR_INST_9_O1, OUTF, TNM=GROUP_0, INIT=R, LIBVER=2.0.0,
SCHNM=OFD
PIN, D, I, a<0>_int,
PIN, C, I, clk_int,
PIN, Q, O, o<0>, 7.000000
END
SYM, XMPLR_INST_7_O1, OUTF, TNM=GROUP_0, INIT=R, LIBVER=2.0.0,
SCHNM=OFD
PIN, D, I, a<1>_int,
PIN, C, I, clk_int,
PIN, Q, O, o<1>, 7.000000
END
SYM, XMPLR_INST_2, BUFG, LIBVER=2.0.0
PIN, I, I, clk,
PIN, O, O, clk_int, 6.599999
END
SYM, XMPLR_INST_0, TIMESPEC,
TS0=FROM:GROUP_1:TO:GROUP_0=14.00ns, LIBVER=2.0.0
END
SYM, XMPLR_INST_1, TIMESPEC,
TS1=FROM:GROUP_0:TO:GROUP_0=14.00ns, LIBVER=2.0.0
END
EOF
```
**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 Leonardo’s 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 Leonardo’s shell:

```sh
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>LOC</th>
<th>MEDFAST</th>
</tr>
</thead>
<tbody>
<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>

See Xilinx documentation for further information.
Additional Xilinx-Specific Options

Using Enabled D Type Flip-Flops

You can direct Leonardo to map to enabled D Type flip-flops, by using the following two variables:

```
set use_dffenable true
```

The `use_dffenable` variable directs Leonardo to infer clock enable from the HDL description. If the `use_dffenable` value is set to TRUE when reading in HDL designs, Leonardo infers clock enable logic while parsing the HDL code. Detecting the clock enable that early is powerful and achieves better results.

```
set enable_dff_map true
```

The `enable_dff_map` variable directs Leonardo to infer clock enable from random logic (when input design is a gate level netlist).

Use of Registered Logic in IOBs

```
set complex_ios false
```

Setting the `complex_ios` variable to FALSE inhibits the use of complex I/O primitives (INLAT, INFF, OUTFF) when targeting Xilinx technologies. Leonardo will map I/O to simple input buffers, output buffers, and bidirectional buffers. By default, `complex_ios` is set to TRUE.
Part Number

```
set part <part_number>
```

Specifying the part number when targeting a Xilinx FPGA does not affect the output optimization. The target library is the only criteria used by Leonardo 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 Leonardo without specifying the part. When satisfied with the result from Leonardo, set part variable and then write XNF.

Writing XNF

You can control the flavor of the XNF output through the following variables and commands:

- By default Leonardo will write appropriate FMAP/HMAP symbol for each LUT in the design. You can switch off writing FMAP/HMAP symbols in XNF output by specifying `set write_lut_binding FALSE`. This option may result in worse results and is not recommended.
- By default functionality in the design will be written out as EQN symbols in output XNF. If you want the output in terms of AND/OR gates then run the `decompose_luts` command before writing.
- By default Leonardo 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 `ungroup -all -hier`) before writing out XNF file. You can also selectively flatten the design for writing.
- If you run `pack_clbs` command before writing, then CLB packing information will be written out in XNF using HBLKNM attribute. If you want to do CLB packing (for better area/delay estimates) but do not want to write out CLB packing information in XNF, then specify `set write_clb_packing FALSE`. 
For example, consider the following input design:

```vhdl
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
....
```
(2) Output with 'set write_lut_binding FALSE'

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
....  (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
Using Xilinx Designs as Input to Leonardo

Use the following guidelines to prepare Xilinx designs for input to Leonardo.

Design Restrictions

The input design must not have the following elements:

• BSCAN (IO scan device)
• RDBK, RDCLK
• 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). In case an oscillator macro exists, Leonardo eliminates it, and connects the oscillator’s output to an input pad.
• RAM or ROM (XC4000, XC4000E). These are exported to the target technology as a “black box” module.

XNF File Constructs Readable by Leonardo

Leonardo requires that certain XNF file constructs not be used. For example, CLB records are not allowed, nor are designs with hard macros.

In general a design taken through the entire Xilinx process will be readable by Leonardo. This entails running the design through Placement and Route, and obtaining an LCA representation of the design. This can then be processed by LCA2XNF resulting in a simulatable XNF file.

Preserving Logic in an XNF Netlist

When using Leonardo to read 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**

Leonardo recognizes a NOOPT parameter, which you can add to the SYM definition:

```
SYM, symbol_name, symbol_type, =NOOPT, other_parameters
```

The =NOOPT parameter on a symbol directs Leonardo to preserve the symbol; it 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
```

**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_gsr (default is FALSE)
```

Preserves the global tristate (gts) signal, on XC4000, XC4000E 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.

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 set the `preserve_dangling_net` variable to TRUE (the default value is FALSE).

```
set preserve_dangling_net true
```

`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
```
and the resulting netlist is:

```
LCANET, 5
    EXT, i1, I
    SIG, i2, S
    SIG, out, 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, 0, O, and
    END
    SYM, or, OR
    PIN, 1, I, and
    PIN, 2, I, i2
    PIN, 0, O, out
    END
```

This variable should be used with care as it may sometimes result in unexpected behavior. For example, if `preserve_dangling_nets` 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
```

Leonardo 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:

```
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.

Leonardo 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**

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 library in the Graphical User Interface. Specify the source technology to be xi4hm so Leonardo 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.

**New Xilinx Architecture Features**

These features include:
- Modgen Enhancements
- Fast I/O Buffers
- Quality of Area and Delay Results
- Fast Xilinx Outputs
- M1 Back Annotation Flow
- M1 Support
- Implementing Boundary Scan for Xilinx 4000 FPGAs

**Modgen Enhancements**

**Non-Square Multipliers**

Non-square multipliers were implemented for the following technologies:

XC4000/E/L/EX/XL/XV and XC5200

**Xilinx Libraries**

Support for Xilinx M1 release
**RAMs**

All RAMs: support for arbitrary size RAMs.

**XBLOX3/XBLOX4**

Reduction operators now use XBLOX components.

**New Modgen Libraries**

- XI4E: XC4000E/EX modgen library is adding support for synchronous RAMs.

**Existing Modgen Library Improvement**

- XI5: counter area and delay improvements.
- XI3: reductions in area and delay for adders and subtractors.

**New Components for Modgen Libraries**

- XI4/XI4E: pre-setable, non-loadable up counters
- XI4E: RAMs
- XI5: up/down counters

**Fast I/O Buffers**

Leonardo sets xi4 and xi5 slew rate to Fast for all input and output buffers. This speeds up the design.

**Quality of Area and Delay Results**

**Area and Delay**

Quality of results were improved by 10%-20% for Altera FLEX 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 Leonardo 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

Leonardo 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.

**Description**

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**. Refer to Screen 14-6.

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.
Screen 14-6. Fast Output Buffers
M1 Back Annotation Flow

1. You can generate XNF netlist or EDIF netlist on an output from Leonardo 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 have to 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.

Note: You must purchase a Time Module license to use back annotation flows with Leonardo. Back annotation is supported through VITAL libraries for the following Xilinx families: XC4000/E/EX/XL (through M1 software only), and XC5200.

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 Leonardo commands may be:

```tcl
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, Leonardo writes out Unified library cells, while in XNF it writes out XNF primitives.

EDIF flow includes:
- writing out FMAPs/HMAPs
- writing CLB information
- writing Timespecs

XNF flow is similar to XACT. In addition, Leonardo supports functional verification and timing back-annotation using SimPrims.

**Accurate Delay Estimations**

Wire load models are added for the following Xilinx technologies: XC4000/E/EX/XL and XC5200. These models allow Leonardo to accurately predict routing delays. Leonardo’s estimates are within 5%-10% of xdelay estimates.

**Sophisticated CLB packing**

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.

```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 Leonardo much smaller and M1 processing time is faster.
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 --
 ul: 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.

Leonardo 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
    );
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 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;
**Xilinx FPGA Devices Supported**

<table>
<thead>
<tr>
<th>Device</th>
<th>Default Speed Grade</th>
<th>Speed Grades supported:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx XC3000</td>
<td>100</td>
<td>70, 100, 125</td>
</tr>
<tr>
<td>Xilinx XC3100</td>
<td>4</td>
<td>2, 3, 4, 5</td>
</tr>
</tbody>
</table>

**Devices Supported**

<table>
<thead>
<tr>
<th>Speed Grade</th>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>3020</td>
<td>PC68 PC84 PG84 PQ100 CB100</td>
</tr>
<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>
<tr>
<td>3120</td>
<td>PC68 PC84 PG84 PQ100</td>
</tr>
<tr>
<td>3130</td>
<td>PC44 PC68 PC84 PG84 PQ100 TQ100</td>
</tr>
<tr>
<td>3142</td>
<td>PC84 PG84 PQ100 TQ100 PP132 PG132 TQ144</td>
</tr>
<tr>
<td>3164</td>
<td>PC84 PP132 PG132 PQ160</td>
</tr>
<tr>
<td>3190</td>
<td>PC84 PP175 PG175 PQ208 PQ160</td>
</tr>
<tr>
<td>3195</td>
<td>PC84 PQ160 PP175 PG175 PQ208 PG223</td>
</tr>
</tbody>
</table>
### Xilinx XC3000A

**Default Speed Grade:** 6  
**Speed Grades supported:** 6, 7

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>3020A</td>
</tr>
<tr>
<td>3030A</td>
</tr>
<tr>
<td>3042A</td>
</tr>
<tr>
<td>3064A</td>
</tr>
<tr>
<td>3090A</td>
</tr>
</tbody>
</table>

### Xilinx XC3000L

**Default Speed Grade:** 8  
**Speed Grades supported:** 8

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>3020L</td>
</tr>
<tr>
<td>3030L</td>
</tr>
<tr>
<td>3042L</td>
</tr>
<tr>
<td>3064L</td>
</tr>
<tr>
<td>3090L</td>
</tr>
</tbody>
</table>
### Xilinx XC3100A

Default Speed Grade: 4

Speed Grades supported: 2, 3, 4, 5

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>3120A</td>
</tr>
<tr>
<td>3130A</td>
</tr>
<tr>
<td>3142A</td>
</tr>
<tr>
<td>3164A</td>
</tr>
<tr>
<td>3190A</td>
</tr>
<tr>
<td>3195A</td>
</tr>
</tbody>
</table>

### Xilinx XC4000

Default Speed Grade: 5

Speed Grades supported: 4, 5, 6

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>4003</td>
</tr>
<tr>
<td>4005</td>
</tr>
<tr>
<td>4006</td>
</tr>
<tr>
<td>4008</td>
</tr>
<tr>
<td>4010</td>
</tr>
<tr>
<td>4013</td>
</tr>
<tr>
<td>4025</td>
</tr>
</tbody>
</table>
Xilinx XC4000H

Default Speed Grade: 5

Speed Grades supported: 4, 5, 6

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>4003H</td>
</tr>
<tr>
<td>4005H</td>
</tr>
</tbody>
</table>

Xilinx XC4000A

Default Speed Grade: 5

Speed Grades supported: 4, 5, 6

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>4002A</td>
</tr>
<tr>
<td>4003A</td>
</tr>
<tr>
<td>4004A</td>
</tr>
<tr>
<td>4005A</td>
</tr>
</tbody>
</table>
### Xilinx XC4000E/L

**Default Speed Grade:** 3

**Speed Grades supported:** 1, 2, 3, 4

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>4003E</td>
</tr>
<tr>
<td>4005E</td>
</tr>
<tr>
<td>4005E</td>
</tr>
<tr>
<td>4006E</td>
</tr>
<tr>
<td>4008E</td>
</tr>
<tr>
<td>4010E</td>
</tr>
<tr>
<td>4010E</td>
</tr>
<tr>
<td>4010E</td>
</tr>
<tr>
<td>4013E</td>
</tr>
<tr>
<td>4013E</td>
</tr>
<tr>
<td>4013E</td>
</tr>
<tr>
<td>4020E</td>
</tr>
<tr>
<td>4025E</td>
</tr>
</tbody>
</table>

### Xilinx XC4000EX

**Default Speed Grade:** 3

**Speed Grades supported:** 2, 3, 4

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>4028EX</td>
</tr>
<tr>
<td>4036EX</td>
</tr>
</tbody>
</table>
### Xilinx XC4000XL

Default Speed Grade: 3

Speed Grades supported: 1, 2, 3

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>4005x</td>
</tr>
<tr>
<td>4010x</td>
</tr>
<tr>
<td>4013x</td>
</tr>
<tr>
<td>4020x</td>
</tr>
<tr>
<td>4028x</td>
</tr>
<tr>
<td>4036x</td>
</tr>
<tr>
<td>4044x</td>
</tr>
<tr>
<td>4052x</td>
</tr>
<tr>
<td>4062x</td>
</tr>
<tr>
<td>4085x</td>
</tr>
</tbody>
</table>

### Xilinx XC5200

Default Speed Grade: 5

Speed Grades supported: 3, 4, 5, 6

<table>
<thead>
<tr>
<th>Devices Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>5202</td>
</tr>
<tr>
<td>5204</td>
</tr>
<tr>
<td>5206</td>
</tr>
<tr>
<td>5210</td>
</tr>
<tr>
<td>5215</td>
</tr>
</tbody>
</table>
### Xilinx Spartan

- **Default Speed Grade:** 3
- **Speed Grades supported:** 3, 4

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th>S30</th>
<th>VQ100, TQ144, PQ208, PQ240, BG256</th>
</tr>
</thead>
<tbody>
<tr>
<td>S10</td>
<td>VQ100, TQ144, PC84</td>
<td></td>
</tr>
<tr>
<td>S20</td>
<td>VQ100, TQ144, PQ208</td>
<td></td>
</tr>
<tr>
<td>S05</td>
<td>VQ100, PC84</td>
<td></td>
</tr>
<tr>
<td>S40</td>
<td>PQ208, PQ240, BG256</td>
<td></td>
</tr>
</tbody>
</table>

### Xilinx XC4000XV

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

<table>
<thead>
<tr>
<th>Devices Supported</th>
<th>40125XV</th>
<th>BG560, PG559</th>
</tr>
</thead>
<tbody>
<tr>
<td>40150XV</td>
<td>BG560, PG559</td>
<td></td>
</tr>
</tbody>
</table>
New Technologies

The following new FPGA technologies are supported:

- Vantis
- Cypress Flash 370i and Cypress Ultra 3700
- Motorola MPA1000
- Philips PS Cool Runner

Vantis

VANTIS CPLD and FPGA devices are supported in Leonardo. You select the Vantis library as a target library. Leonardo generates an EDIF output netlist based on MINC primitives. The output netlist is used as input to Design Direct place-and-route. When targeting Vantis devices, use auto_write to generate EDIF.

Cypress

Leonardo 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.

Motorola MPA 1000

The Motorola MPA 1000 library was modified. The new library is available through Motorola:

Motorola Programmable Products
**Philips PS Cool Runner**

Philips cool runner is supported. Select Philips technology as a target technology from the GUI. When running in batch mode, you should use `-target=ps` as a command line option. When targeting Philips devices, use `auto_write` to generate EDIF.
Index

A
acf file, 10-5
ACTGen, 8-11
Adobe Acrobat Reader, 1-3
ALSPIN, Actel, 8-21, 8-24
AND/OR representation, 14-7
area
  estimation for ASICs, 7-14
  reporting, Altera FLEX, 6-5
  reporting, Xilinx, 6-1
arithmetic operator, 8-7, 9-10, 11-11, 13-5, 14-17
array_pin_number, attribute, 10-5, 11-28, 14-54
ARRIVAL_TIME, 4-4, 14-58
ASIC
  design flow, 7-1
ASYNC_VAL, attribute, Xilinx, 14-66
attribute
  BUFFER_SIG, 14-49
automatic constraining of designs, 5-14
setup checks, 3-9
TimingChecksOn, 3-9
tipd_in, 3-9
tpd_in_out, 3-9
balloon help, 1-3
BASE, attribute, Xilinx, 14-66
BDBUF, Xilinx, 14-75
BIIPAD, QuickLogic, 12-6
BIORPAD, QuickLogic, 12-6
BIPAD, QuickLogic, 12-6
black box, 9-22, 11-23, 14-27
BLKNM, attribute, Xilinx, 14-65, 14-66
Boolean mapping
  algorithm, 2-2, 8-6, 8-7
boundary scan, 14-85
BSCAN, Xilinx, 14-71, 14-85
BUFFER_SIG, 2-9, 2-10, 8-16, 8-18, 9-25, 10-8, 12-5, 14-49, 14-50
buffering of logic, 4-3, 7-11, 8-15

B
back-annotation, 3-7
Altera, 9-2
hold checks, 3-9

C
cae2adl, Actel, 8-12
CAP, attribute, Xilinx, 14-65, 14-66
capacitance interconnect, 3-3
carry
  chain, Altera FLEX, 9-5, 9-10
chain, Xilinx, 14-18
logic, ORCA, 11-2
lookahead generator, Xilinx, 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, 9-25
CKDPAD, QuickLogic, 12-7
CKPAD, QuickLogic, 12-7
CKTPAD, QuickLogic, 12-7

CLB
architecture, 14-2, 14-12
occupied, 14-12
packing, 14-5, 14-7, 14-9, 14-12, 14-68

CLB packing, 14-84
CLKBUF, Actel, 8-15

clock
common zero, 7-12
cycle, 14-57
delay, 7-12
edge, 7-12
enable, 14-13
enable detection, Altera FLEX, 9-5
enable logic, 14-34
enable recognition, 14-35
ideal, 7-12
leading edge, 4-5
manual mapping, Xilinx, 14-49
offset, 14-57
pulse, 4-5
pulse width, 14-57
reference, 4-5
setup time, 4-5, 14-34
clock buffer, 2-10
clock buffer, Actel, 8-6, 8-15
clock buffer, Xilinx
ACLK, 14-47
BUFG, 14-47
BUFGP, 14-48
BUFGS, 14-48
GCLK, 14-47
clock timing constraints, 14-58
CLOCK_CYCLE, 4-5, 14-58
CLOCK_OFFSET, 4-5, 7-12, 14-58
CMOS, attribute, Xilinx, 14-66
combinational logic loops, 12-4
combinational module, Actel, 8-2
command line
connect_path, 4-3
disconnect_path, 4-3
optimize, 14-21
optimize Timing, 14-11
pack_clbs, 14-12
command line options
-area, 4-2
-auto, 9-11, 14-19
-chip, 2-7, 14-21
-delay, 5-2
-fast, 9-11, 14-19
-fastest, 9-11, 14-19
-force, 5-5, 5-14, 8-11, 11-10
-macro, 2-7, 14-21
-nolut_map, 9-36
-small, 9-11, 14-19
-smallest, 9-11, 14-19
-through, 14-12
complex I/Os
Actel, 8-2
checker, 8-18
complex_ios, variable, 14-67
CONFIG, attribute, Xilinx, 14-66
configurable logic block, 14-2
counter
Altera FLEX, 9-10
ORCA, 11-11
Xilinx, 14-18
CPLD Architecture, 13-2
CPLD Family Supported Devices, 13-7
critical path
delay, Xilinx, 6-1, 14-56
critical path report, 3-1
critical path reporting, 3-2, 5-10, 7-12
formatting, 3-7
CYMODE, attribute, Xilinx, 14-66
**D**
data path synthesis
   Actel, 8-7
   Altera FLEX, 9-9
   ORCA, 11-10
   Xilinx, 14-17
DECODE, attribute, Xilinx, 14-66
decompose_luts, command, 14-68
decrementer, 8-7
   Altera FLEX, 9-10
   ORCA, 11-11
   Xilinx, 13-5
DEF, attribute, Xilinx, 14-66
default_max_transition, attribute, 7-11
Defparms, Verilog, 9-13
delay
   estimate, ASICs, 7-18
   reporting, Altera FLEX, 6-5
delay model
   connect, 3-2
   input ramp, 3-2
   intrinsic, 3-2
   linear, 3-2, 7-12
   nonlinear, 3-2, 7-12
   piece wise linear, 3-2
   transition, 3-2
DELAYMODE, ORCA, 11-31
DEPTH, attribute, Xilinx, 14-66
derating factors, 7-13
design
   present, 5-3
   rule checker, 8-7
   rule violation, Actel, 8-18
designer, Actel, 8-15
Devices Supported for Lucent ORCA, 11-39
DFF, gate, 14-13
DIN/DOUT Attributes, 11-29
dont_lock_lcells, variable, 10-2
DOUBLE, attribute, Xilinx, 14-66
  drive
     capability, 4-2
     default, 4-3
input, 4-3
   requirement, 4-2
   resistance, 3-3
DRIVE, ORCA, 11-31
DUTY_CYCLE, attribute, Xilinx, 14-66
DWAND, component, 14-27

**E**
edf extension, 10-2
edge decoders, Xilinx, 14-27
EDIF
   2.0 specification, 9-11
   bus, 9-27
   file, MAX+PLUS II, 10-10
   format, QuickLogic, 12-3
   netlist, 9-2, 9-27, 10-2, 11-23
   output, 8-11
EDIF Input, 9-28
EDIF M1 flow, 14-2
EDIF netlist, 14-1
EDIF_INPUT_LMF1, altera, 10-5
EDIF_INPUT_USE_LMF1, Altera, 10-5
Embedded Array Block (EAB), 9-22
embedded array block, Altera, 9-2
exmpler.lmf file, 9-28
expander, Altera, 10-2, 10-7

**F**
F1_LUT, Altera, 6-9
F2_LUT, Altera, 6-9
F3_LUT
   Altera, 6-9
F4_LUT
   Altera, 6-9
fanin limit, Altera MAX, 10-9
fanin limited optimization, 9-2, 14-7
fanout violations, 7-9, 8-15
fast Xilinx outputs, 14-81
FAST, attribute, Xilinx, 14-65, 14-66
FIFO, Altera FLEX, 9-2
FILE, attribute, Xilinx, 14-66
FLEX architecture, 9-5
flex_use_cascades, variable, 9-8
flip-flop
  D-type, 14-67
  enable, Altera, 10-7
FLOAT_VAL, attribute, Xilinx, 14-66
FMAP, 5-9, 6-4, 14-5, 14-7, 14-13, 14-68
foundry
  ORCA, 11-23, 11-26
FREQ, attribute, Xilinx, 14-65
function generator, 14-10, 14-13
functional block, 14-12

G

gate sizing, 7-8, 7-11
generate_timespec, command, 14-63
genmem
  Altera, 9-20
  support, 9-20
global signals, 9-25
GLOBAL, Altera, 9-25, 10-8
global_sr, variable, 11-11
GLOBALRESET, Xilinx, 14-74
GND, pin, 9-29
GSR
  Xilinx, 11-11, 14-20
GTS
  Xilinx, 14-74
GUI options
  assign global SR, 11-12, 14-21
  chip, 2-7
  detect the global set/reset signal, 11-12, 14-21
  do not lock LCells, 9-8
  macro, 2-7
  map cascades, 9-36
  map look up tables, 2-3, 9-8, 9-36, 11-8
  map to 6-input LUTs, 2-3, 11-8
  map to CASCADEs during LUT mapping, 2-3, 9-8
  max fanin, 13-5
  max PT, 13-5
modgen select, 9-11, 14-19
process, 3-6
temperature, 3-6
use F5MAP symbols, 2-3
use FAST output buffers, 14-81
write look up tables, 14-5
guides
  Command Reference, 1-3
  HDL Synthesis Guide, 1-3
  Installation Guide, 1-2
  Synthesis and Technology Guide, 1-3

H

hard macro, file (.HM), 14-79
HBLKMN, attribute, Xilinx, 14-12, 14-16, 14-65, 14-66, 14-68
HCLKBUF, Actel, 8-16
HD2PAD, QuickLogic, 12-6
HD3PAD, QuickLogic, 12-6
HDIPAD, QuickLogic, 12-6
hdl_array_name_style, variable, 9-22, 9-27
HDPAD, QuickLogic, 12-6
Help, 1-3
HMAP, 5-9, 14-5, 14-7, 14-12, 14-13, 14-68
HU_SET, attribute, Xilinx, 14-66

I

I/O

assign automatically, 2-1, 14-50
assign manually, 2-1, 2-11, 12-5, 14-50
buffers, Actel, 8-2
buffers, ORCA, 11-27
cell, Altera, 10-2
cells, 8-18
cells, Xilinx, 14-50
complex, 2-1
port, 4-4
registered, 8-18, 14-50
set/reset, 8-15
size of, 2-1
Index

I/O Mapping, 2-8, 8-18
  ORCA, 11-27
IBUF, Xilinx, 14-75
IBUFLEVEL, ORCA, 11-31
implementation
  multi-level, 10-7
  one level, 10-7
include_gates, variable, 11-27
incrementer, 8-7
  Altera FLEX, 9-10
  ORCA, 11-11
  Xilinx, 13-5
infer_gsr, variable, 11-12
INFF, Xilinx, 14-67, 14-75
INIT, attribute, Xilinx, 14-16, 14-46
initial, attribute, ORCA, 11-24, 11-26
INLAT, Xilinx, 14-67, 14-75
INPAD, QuickLogic, 12-6
input/output block, 14-2
INPUT_DRIVE, 4-3
INPUT_MAX_FANOUT, 4-3
INPUT_MAX_LOAD, 4-3
inputs
  primary, 4-3
  register, 4-3
INREG, Xilinx, 14-75
insert_global.bufs, variable, 8-17
interconnect
  delay, Xilinx, 6-4
  load, 3-3
  modeling, 7-13
  resistance, 3-3, 7-14
INTERNAL, attribute, Xilinx, 14-65, 14-66
intrinsic delay
  Xilinx, 6-4
IOCLK, 8-21
  Actel, 8-18
IOPCL, 8-21

L
LABs, Altera, 10-2
latch
  identical, 10-8
LCA architecture, 14-1, 14-2
LCA2XNF, Xilinx, 14-71
LCELL
  Altera, 10-2
  primitive, 9-7
Leonardo’s manuals, 1-2
libraries
  synthesis, 1-3
  VITAL, 1-3
Library Mapping File, 10-1
Library Mapping File (LMF), 10-5
LIBVER, attribute, Xilinx, 14-16, 14-66
LMF file, 10-5
load
  capacitive, 3-3, 7-14
  external, 4-2
  fanout, 4-3
  interconnect, 3-2, 3-3
  maximum, 4-3
  maximum fanout, 4-3
  output, 4-2
  requirements, 4-2
  unit, 4-2
  violation, 7-9, 7-11, 8-6
LOAD, ORCA, 11-31
LOC, attribute, Xilinx, 14-65, 14-66
LOC, ORCA, 11-28
logic
  array block, Altera, 9-2, 10-9
  buffering, 4-3, 7-11, 8-15
  cell, pASIC, 12-2
  combinability, Actel, 8-14
  element, Altera, 9-2
  module, Actel, 8-2
  replication, 4-2, 7-11
  logical operator, 8-7, 11-11
lookup table, 14-84
  Altera FLEX, 9-7
  ORCA, 11-2
lookup table mapping, 2-2
  Altera, 2-2, 9-2, 9-7

Index-5
F3_LUT, gate, 6-5
F4_LUT, gate, 6-5
F5_MUX, gate, 14-11
H3_LUT, gate, 6-5
ORCA, 2-2, 11-6
Xilinx, 2-2, 14-6, 14-10
LPM
   conventions, 9-5
   instantiation, 9-13
LPM Support for FLEX 10K, 9-11
LPM_RAM_DQ, component, 9-12
LPM_RAM_IO, component, 9-12
M
   M1 back annotation flow, 14-83
   M1 support, 14-84
   macrocell, 10-7
      Altera, 10-2
   MAP, attribute, Xilinx, 14-66
   MAX architecture, Altera, 10-1, 10-8
   max cubes, Altera MAX, 10-9
   max fanin, Altera MAX, 10-9
   MAX+PLUS II, 9-29
   MAX+PLUS II, Altera, 10-2, 10-5, 10-9
   max_fanin, variable, 10-9
   max_transition, variable, 7-11
   MD0, input port, Xilinx, 14-71
   MD1, output port, Xilinx, 14-71
   MD2, input port, Xilinx, 14-71
   MEDFAST, attribute, Xilinx, 14-65, 14-66
   MEDSLOW, attribute, Xilinx, 14-65, 14-66
   MEMFILE, attribute, Xilinx, 14-66
   memgen, Xilinx, 14-40
   module generator
      Altera, 10-8
      library, 14-17
      Xilinx, 13-4
   modules
      Leonardo, 1-2
      Model Technology V-System/QuickHDL
         simulator, 1-2
      Time Explorer, 1-2
move command, 9-27
multiplexer
   ORCA, 11-11
mux-based, optimization, 8-6
N
   negative slack, 5-6
   Neoprim library, 11-42
   New Features for Altera FLEX, 9-35
   New Features for QuickLogic, 12-2
   NOBUFF, 4-3
   NODELAY, attribute, Xilinx, 14-65, 14-66
   nlogic_rep, variable, 8-15
   NOMERGE, ORCA, 11-6
   NOOPT, Xilinx, 14-72
   nowire_table, variable, 7-14
O
   OBUF, Xilinx, 14-75
   OBUFLEVEL, ORCA, 11-31
   online guides, 1-3
   operating conditions, 7-13
   operator
      arithmetic, 14-18
      logical, 14-18
      relational, 14-18
   optimization
      flow, 5-1
      passes, Altera MAX, 10-7
      sequential, 9-24, 10-8, 12-4
   optimize command
      -target option, 10-2
      optimize_timing command, 5-5, 8-11, 11-10
         -force option, 5-5, 8-11, 14-11
         -through option, 5-5
      optimize_timing -force, 8-11
      oscillator, Xilinx, 14-71
      OUTFF, Xilinx, 14-67, 14-75
      OUTFFT, Xilinx, 14-75
      OUTIPAD, QuickLogic, 12-6
      OUTORPAD, QuickLogic, 12-6
OUTPAD, QuickLogic, 12-6
OUTPUT_FANOUT, 4-2
OUTPUT_LOAD, 4-2

P
pack_clbs, command, 6-3, 14-14
PAD, 2-9, 8-18, 14-50
PAD Command, 12-6
part number, ORCA, 11-31
part, variable, 14-68
pASIC architecture, QuickLogic, 12-2
pASIC1 and pASIC2 Derating Factors, 12-5
path analysis, 4-3
PERIOD, attribute, Xilinx, 14-65
PFU-MUX, ORCA, 11-6
PFU-NAND, ORCA, 11-6
PFU-XOR, ORCA, 11-6
PICSPEC, ORCA, 11-31
PICSPEED, ORCA, 11-31
pin assignment, Actel, 8-21
pin location
   Altera, 9-25, 10-5, 10-6
   ORCA, 11-28
   QuickLogic, 12-7
   Xilinx, 14-54
PIN_NUMBER, 10-5
pin_number, attribute, 8-22, 9-25, 11-28, 14-54
place-and-route
   Altera, 10-2
PLDs architecture, Altera, 10-1
pop_design, command, 5-4
POWERUP, 11-12, 14-21
PPR, Xilinx, 14-40
preference file writer, 11-32
preserve_dangling_nets, variable, 14-75, 14-77
primary inputs, 4-4
Process Derating Factors, 8-25
process, variable, 7-13, 11-31
product terms, 10-2, 10-10
programmable
   function unit, ORCA, 11-2
   input/output cells, ORCA, 11-2
   logic cells, ORCA, 11-2
propagate_clock_delay, variable, 7-12
propagation delay, 4-4
PULSE_WIDTH, 4-5, 14-58
push_design, command, 5-4

Q
QCLKBUF, Actel, 8-16
QDIF format, 12-3

R
RAM
   Altera FLEX, 9-2, 9-10
dual-port, 9-2, 14-43
dual-port, ORCA, 11-26
inferencing, RTL, 9-15
instantiation, Altera FLEX, 9-5, 9-12
instantiation, ORCA, 11-23
instantiation, Xilinx, 14-40
ORCA, 11-11
Xilinx, 14-71
RAM_DQ
   component, 9-12
ram_example1, 8-9, 9-18, 11-18, 14-37
ram_example2, 8-10, 9-19, 11-19, 14-38
RAM_IO
   component, 9-12
RDBK, Xilinx, 14-71
RDCLK, Xilinx, 14-71
registers, 4-4
   identical, 10-8
relational operator, 8-7, 9-10, 11-11, 13-5, 14-17
replicate logic, 8-15
report_area command, 6-2, 6-7, 7-16
   -cell option, 7-17
report_delay command, 6-4, 7-12, 7-18
reports
   Altera FLEX area/delay example, 6-1
Xilinx area/delay example, 6-1
REQUIRED_TIME, 4-4, 4-6
RES, attribute, Xilinx, 14-65, 14-66
reset
  active high, 11-14, 14-24
  active low, 11-15, 14-25
  signal, 14-13
RLOC, attribute, in Xilinx, 14-66
RLOC_ORIGIN, attribute, Xilinx, 14-66
RLOC_RANGE, attribute, Xilinx, 14-66
ROM
  Altera FLEX, 9-2
  ORCA, 11-26
  Xilinx, 14-46, 14-71
rule constraints, 7-9
runtime requirements, 1-5

S
schematic
  viewer, 3-8
SCHNM, attribute, Xilinx, 14-16, 14-66
SCLK, Altera, 10-8
scuba, ORCA, 11-23, 11-26
SDF
  file, 3-7, 3-9
  HOLD, 3-11
  INTERCONNECT delays, 3-11
  IOPATH delays, 3-11
  PORT, 3-11
  SETUP, 3-11
sequential
  module, Actel, 8-2, 8-16
set, 9-10, 14-5, 14-19
set modgen_select, 9-10, 14-5, 14-19
set, signal, 14-13
shift operator, 8-7
simulation
  Actel, 8-21
  functionality, 3-9
  timing, 3-9
slew input, 3-2
slew property, ORCA, 11-29
SLOW, attribute, Xilinx, 14-65, 14-66
SpDE, QuickLogic, 12-2
speed grade, 11-31
STARTUP, Xilinx, 11-11, 14-20
static timing analysis, 3-1
SYNC_VAL, attribute, Xilinx, 14-66
system requirements, 1-4
SYSTEM, attribute, Xilinx, 14-66

T
TCK, input port, Xilinx, 14-71
TDI, input port, Xilinx, 14-71
TDO, output port, Xilinx, 14-71
temperature, variable, 7-13
THI, attribute, Xilinx, 14-65
time
  arrival, 3-2
  required, 3-2
Time Module, 3-7
Time Module license, 14-83
TIMEGRP, 14-58, 14-59
TIMESPEC, 14-57, 14-58, 14-59
timing
  constraints, 6-5
  required, 4-4
timing analysis, 3-2, 7-12
  annotated XNF netlist, 3-8
  arrival time, 4-4, 5-2
  constraint, 4-4, 5-2
  critical path, 5-2
  end point, 5-2, 5-6
  longest path, 3-2
  negative slack, 3-2
  path analysis, 4-3
  post place-and-route, 3-5
  routing delay, 3-5
  start point, 4-3
timing optimization
  constraint-driven, 11-10
timing optimization, reporting
  area, 5-6
  longest path, 5-6
most critical slack, 5-6
sum of negative slacks, 5-6
TLO, attribute, Xilinx, 14-65
TMS, input port, Xilinx, 14-71
transition time
violations, 7-11
TRIOPAD, QuickLogic, 12-6
tristatable output, 10-2
tristate, internal
QuickLogic, 12-4
Xilinx, 14-29, 14-74
tristate_map, variable, 14-31
TTL, attribute, Xilinx, 14-66

U
U_SET, attribute, Xilinx, 14-66
unconstrained path, 6-5
ungroup command
-all option, 14-68
-hier option, 14-68
unified library, Xilinx, 6-5
use_qclk_buhs, variable, 8-17
USE_RLOC, attribute, in Xilinx, 14-66

V
variables
complex_ios, 2-7
delete_startup, 11-12, 14-21
dont_lock_lcells, 9-8
enable_dff_map, 14-67
flex_use_cascades, 2-3, 9-8
global_sr, 11-12, 14-20
hd1_array_name_style, 8-11
infer_gsr, 11-12, 14-20
insert_global_bufs, 14-50
lut_map, 2-3, 9-8, 11-8, 14-5
max_fanin, 11-8
max_fanin <integer>, 13-5
max_fanout_load, 8-15
max_pt <integer>, 13-5
maxdly, 4-1
modgen_select, 14-19
nologic_rep, 8-15
tristate_map, 14-30
use_dffenable, 14-67
use_f5map, 2-3
use_f6_lut, 2-3, 11-8
write_clb_packing, 14-5
write_lut_binding, 14-5
x1x_preserve_gsr, 14-23
x1x_preserve_gts, 14-23
VCC, pin, 9-29
Verilog, 14-17
Verilog HDL designs, 14-1
VHDL, 14-17
VHDL attributes, 4-5
VHDL designs, 14-1
VITAL library, 3-9
level 0 compliant, 3-9
level 1 compliant, 3-9
voltage, variable, 7-13

W
web site, 1-3
wir2xnf, 14-78
wire load models, 14-84
wire resistance
balanced case, 3-3, 7-14
best case, 3-3, 7-14
worst case, 3-3, 7-14
wire_table, variable, 7-14
wire_tree, variable, 7-14
wireload
model, 6-4
table, 7-14, 7-18
write command, 10-2
write_altera, script, 9-28
write_clb_packing, variable, 14-68
write_lut_binding, variable, 14-68

X
XACT (XNF) flow, 14-2
XACT, software, 14-5
XACT-CPLD tools, 13-2
XBLOX, 14-78, 14-79
   components, 14-18
   output file (.XG), 14-79
   xblox3 library, 14-79
   xblox4 library, 14-79
XBLOX program, 14-79
x1x_preserve_gsr, variable, 14-23, 14-73
x1x_preserve_gts, variable, 14-23, 14-73
XNF, 14-54, 14-57
   EXT record, 14-78
   hierarchical, 14-68
   mapped, 14-10
   NOOPT, 14-72
   output netlist, 14-68
   output, EQN symbols, 14-69
   preserving logic, 14-71
   reader, 14-23
   SIG record, 14-78
   simulatable, 14-71
XNFMERGE, Xilinx, 14-40
XOR
   optimization, Xilinx, 13-4
   synthesis, Altera, 10-2