Exemplar Logic **Xilinx Corporation**

Model Technology

Applications Note

Large Device Design Methodology

July 15, 1998

Revision 2.1

Overview

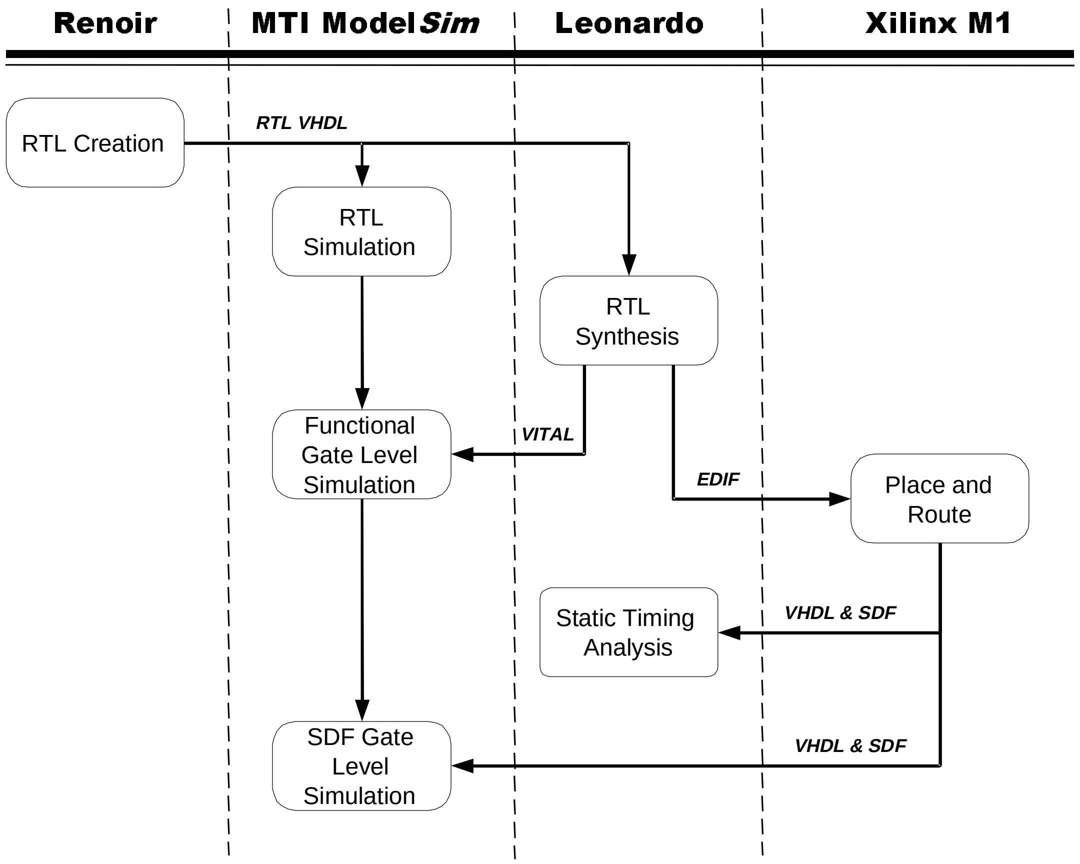
Leonardo performs architecture specific synthesis and optimization for all Xilinx devices. ***Xilinx Alliance Series*** performs placement and routing of the synthesized netlist. ***ModelSim*** performs pre-synthesis RTL simulation and post-place and route gate-level timing simulation with SDF backannotation of timing. This applications note discusses methodology and optimization settings for Leonardo, Alliance Series and ModelSim when targeting all Xilinx devices. The intent of this appnote is not to exhaustively explore all the different options in the Leonardo, Alliance Series and Model*Sim* toolsets but to present a single methodology that works. For information beyond the scope of this document, refer to the following web pages:

www.exemplar.com	***Exemplar Logic***
www.model.com	***Model Technology***
www.xilinx.com	***Xilinx***

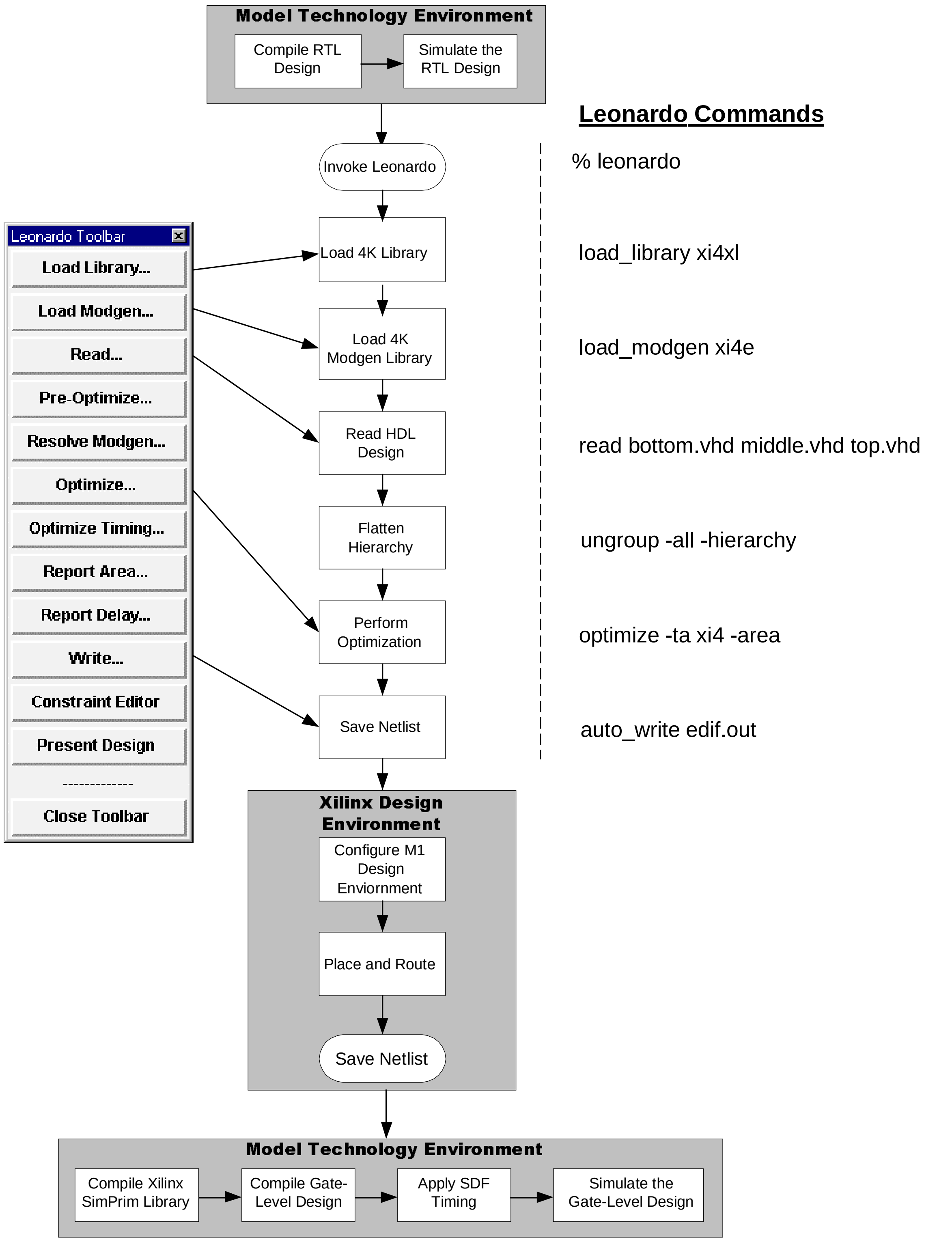
When targeting all Xilinx devices, Leonardo will map the design into Xilinx lookup tables. Both Galileo Extreme and Alliance Series offer configuration options that dictate how this mapping takes place. In some cases both tools can perform the same functions but, as we'll see, with different results.

The Alliance Series allows the creation of VHDL or Verilog of the placed and routed design along with timing information in a SDF file. This gate-level timing design can then be compiled (along with the Xilinx SIMPRIM library) and simulated in Model*Sim*.

Exemplar, Model Technology & Xilinx Toolflow Overview



Synthesis Process Design Flow



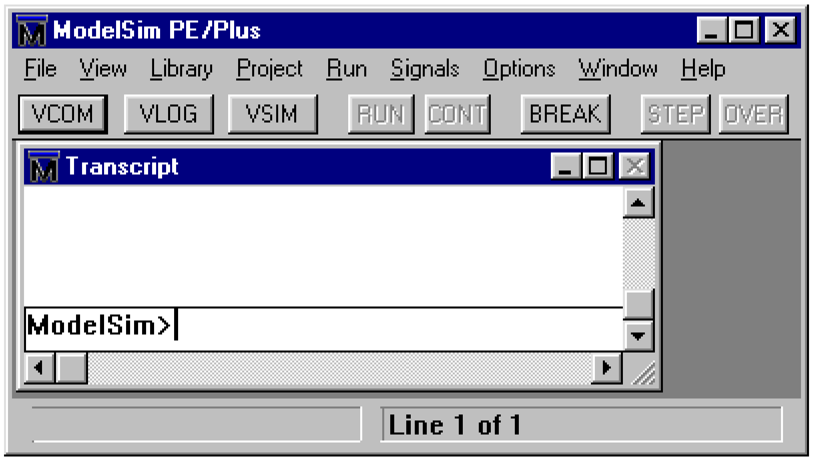
Example Session

Example Model*Sim* RTL Simulation Session

This section describes the basic steps to compiling and simulating the pre-synthesis RTL design. It is in this step that the functionality of the design is verified prior to synthesis.

1) Invoke Model*Sim*

   % vsim   **-or-**   **Programs > Model Tech > ModelSim**


ModelSim PE/Plus
File View Library Project Run Signals Options Window Help
VCOM VLOG VSIM RUN CONT BREAK STEP OVER
Transcript
ModelSim>
Line 1 of 1

2) Set Model*Sim* to the directory where the RTL design resides.

   ModelSim> **cd c:\mydesign**   **-or-**   **File > Directory**

3) Create a working library to store the compiled RTL design.

   ModelSim> **vlib work**   **-or-**   **Library > New**

4) Compile the RTL design.

   ModelSim> **vcom bottom.vhd middle.vhd top.vhd**

   -or-   **VCOM** button

5) Start the Model*Sim* simulator.

   ModelSim> **vsim top**   **-or-**   **VSIM** button

6) View all the Model*Sim* debugging windows.

   VSIM> **view ***   **-or-**   **View > All**

7) Wave and list signals of interest in the design.

   VSIM> **wave /***   -- Adds all top level signals to the wave window
   VSIM> **list /***   -- Adds all top level signals to the list window

8) Unless you have a VHDL testbench which stimulates the RTL design, you will need to force the inputs of the RTL design.

   VSIM> **force /clk 0 @ 0 ns, 1 @ 50 ns –repeat 100 ns**
   VSIM> **force /input 0**

9) Run the simulation and analyze the information in the Model*Sim* debugging windows.

   VSIM> **run –all**   **-or-**   **Run > Run Forever**

Example Leonardo Session

This section provides a detailed example of using Leonardo with the Xilinx Alliance Series toolset targeting the XC4000XL device family. All commands are easily accessible via the toolbar, pulldown menus or the Xilinx specific flow guide. This example will demonstrate Leonardo shell commands.

10) Invoke leonardo

    % leonardo

    LEONARDO{1}

11) Load the Xilinx xi4e library. This will load cell data only

    LEONARDO{1} **load_library xi4xl**

12) Load the Xilinx Modgen library. Modgen is a library of handcrafted implementations for all the inferred design elements. This includes operators, RAMs and counters. There are typically multiple architectures for each element. If this library is not loaded Leonardo will use a generic Modgen library which will not be able to take advantage of Xilinx specific cells.

    LEONARDO{2**} load_modgen xi4e**

13) Read in the HDL files. VHDL design files must be listed in their bottom-up order. Verilog users enjoy "auto-top detection" which means that Leonardo will automatically detect the top-level module from files listed in any order.

Note*: Leonardo uses file suffixes to figure out file formats; VHDL files = .vhd, .vhdl; Verilog files = .v, .ver; EDIF files = .edn, .edf, .edif.*

    LEONARDO{3} **read bottom.vhd middle.vhd top.vhd**

14) Flatten the design. Hierarchical boundaries prevent or limit important optimizations from occuring. Sometimes there are good reasons to preserve hierarchy, i.e., design size or to separate out speed critical blocks. Only a minimum of hierarchy should be kept. It is recommended to have no more than 50K gates per hierarchical block.

    LEONARDO{4} **ungroup -all -hierarchy**

15) Perform optimization. Leonardo can perform both area and timing optimization. In this example we will be performing optimization to achieve the smallest design. Additionally, the effort level can be specified. Quick performs 1 pass and standard performs 4 passes and will take 4 times longer to complete.

    LEONARDO{5} **optimize -ta xi4e -area -effort quick**

16) Generate area and timing reports. The optimization runs will display a single area and worst case timing number. Reports are only necessary if more information is required.

    LEONARDO{6} **report_area**

    LEONARDO{7} **report_delay**

17) Generate an EDIF netlist for Alliance Series. A Netlist pre-processor is built into Leonardo. Because the Xilinx XC4000E technology is specified the correct netlist pre-processing will take place.

Example Alliance Series Place and Route Session

The Xilinx graphical tools are designed to behave, look, and feel like the XACT 6.0 tools. So despite the fact that the core technology algorithms have been redesigned, the graphical tools allow users to run the software in the same way as previous PC versions. For PC customers, the learning curve should be short.

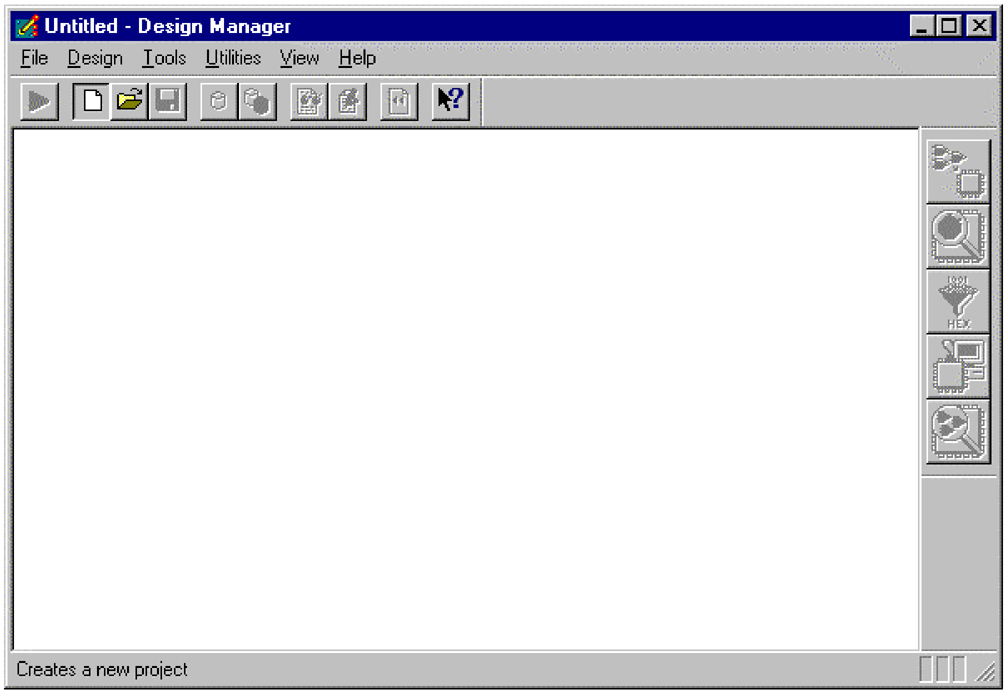
The Design Manger (DM) is the graphical tool that manages the design files that are created during design implementation. The DM also provides push button access to the following Xilinx tools: Flow Engine, Prom File Formatter, Timing Analyzer, Hardware Debugger, and JTAG Programmer.

Start the Design Manager from the Windows 95 or NT desktop by executing the command:

Selecting Start > Programs > Xilinx > Design Manager

From a shell invoke Design Manager by typing:

dsgnmgr

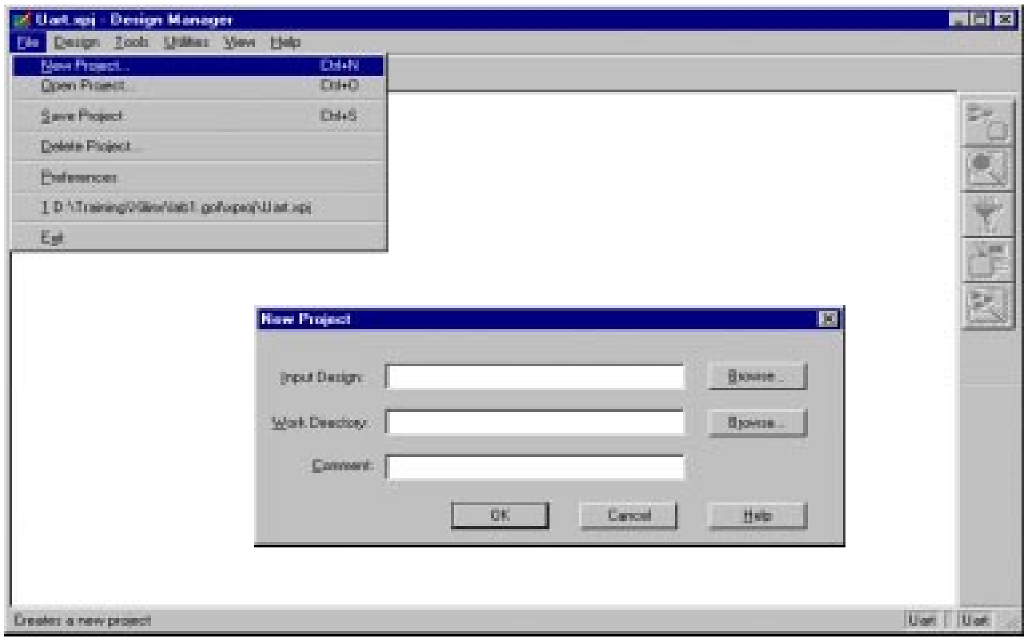

Untitled - Design Manager
File Design Tools Utilities View Help
HEX

Creates a new project

1) Create a New Project in Alliance Series. From the Design Manager toolbar, execute the pulldown menu command

FILE > New Project

Push the ***"input design"*** button and navigate to the EDIF file generated by Leonardo. This file should have the extension, ".edn".



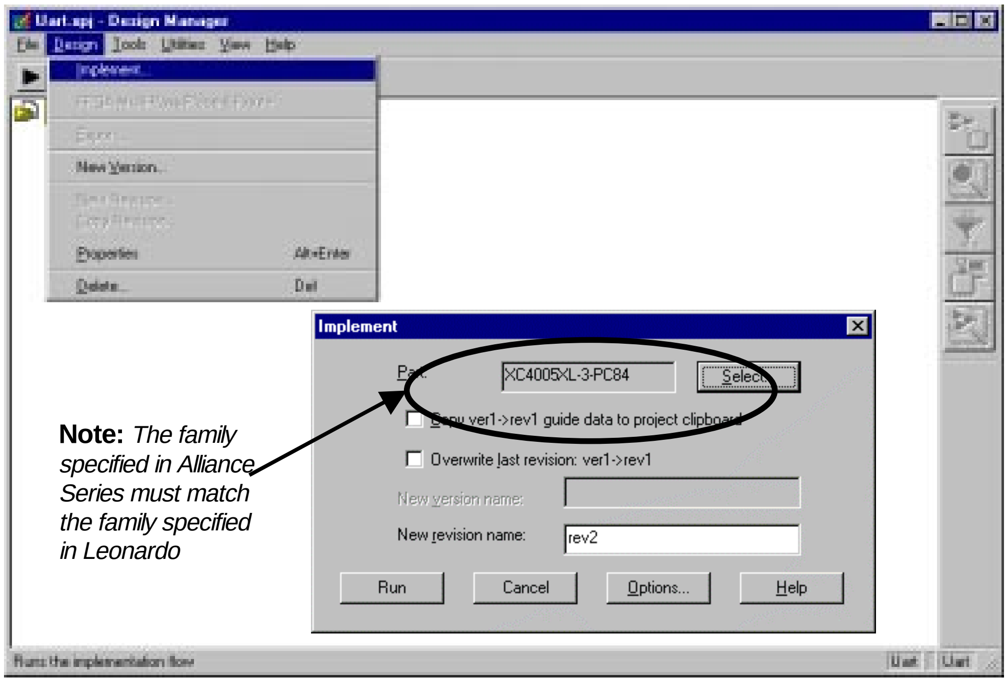
2) Perform "Implement" on the design. From the pulldown menu, execute the command,

Design > Implement

- Select a specific Xilinx part
- Push the *Select* button. In the "Part Selector" dialog and choose the appropriate member, speed and package combination from those available for the XC4000XV family

Note: *If the Leonardo command, "generate_timespec" is issued after optimization and before saving the EDIF netlist then clock frequency timing data is included in the EDIF netlist.*

3) Click "*OK, but do not hit "Run" on the Implement dialog box*

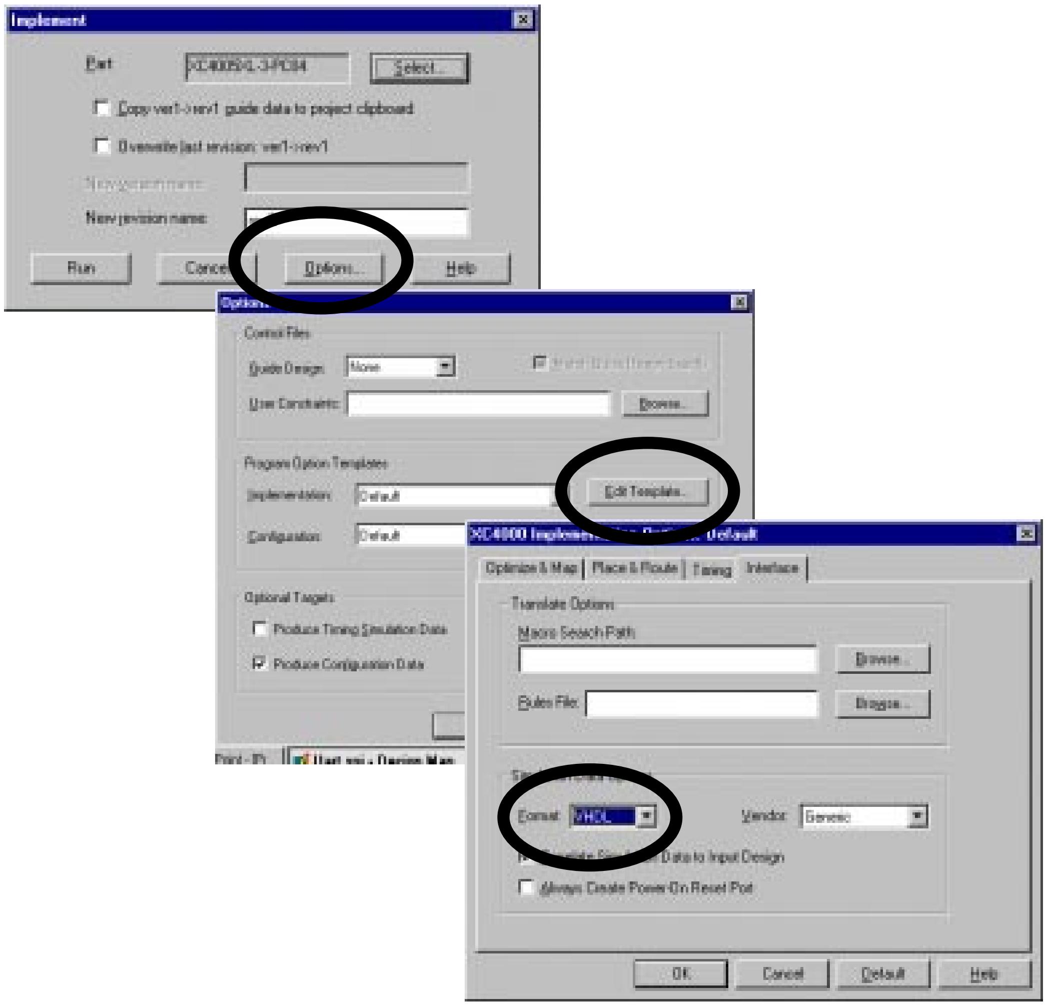

Uart.apj - Design Manager
File Design Tools Utilities View Help
Implement...
New Version...
Properties... Alt+Enter
Delete... Del
Implement
Part: XC4005XL-3-PC84
Select...
Copy ver1->rev1 guide data to project clipboard
Overwrite last revision: ver1->rev1
New version name:
New revision name: rev2
Run
Cancel
Options...
Help
Runs the implementation flow
Uart
Uart

Note: *The family specified in Alliance Series must match the family specified in Leonardo*

Note: *The first step in implementing a design is the selection of a target device. If a valid PART has been specified in Leonardo, it will be pre-selected in the Part selection dialog box. For designs that do not have the PART specified in the netlist (or the Design Manager is unable to detect its presence), the user must identify the target part using this dialog box. Users may define the part in Leonardo by setting the "part" variable, i.e.,* ***"set part xc4005xl-3-PC84".*** *Setting the part variable is a step of convenience and will not effect optimization results*

Setup Alliance Series to generate VITAL VHDL Simulation Model

1) From the Implement dialog box select the ***"Options"*** button. This will bring up the "Options" dialog box.
2) In the "Optional Targets" field, check the box labeled***, "Produce Timing Simulation Data"***
3) In the "Program Option Templates" field, select the ***"Edit Template***" button for "Implementation". This will bring up the "XC4000 Implementation Options: Default" dialog box.
4) Once up Select the ***"Interface"*** tab and Set the simulation data output to "***VHDL"***
5) ***"OK"*** all the dialog boxes


Implement

Part: XC4005XL-3-PC84

Copy ver1->rev1 guide data to project clipboard

Overwrite last revision: ver1->rev1

New version name:

New revision name:

Options

Control Files

Guide Design: None

User Constraints:

Program Option Templates

Implementation: Default

Configuration: Default

Optional Targets

Produce Timing Simulation Data

[✓] Produce Configuration Data

XC4000 Implementation Options: Default

Translate Options

Macro Search Path

Rules File:

Format: VHDL

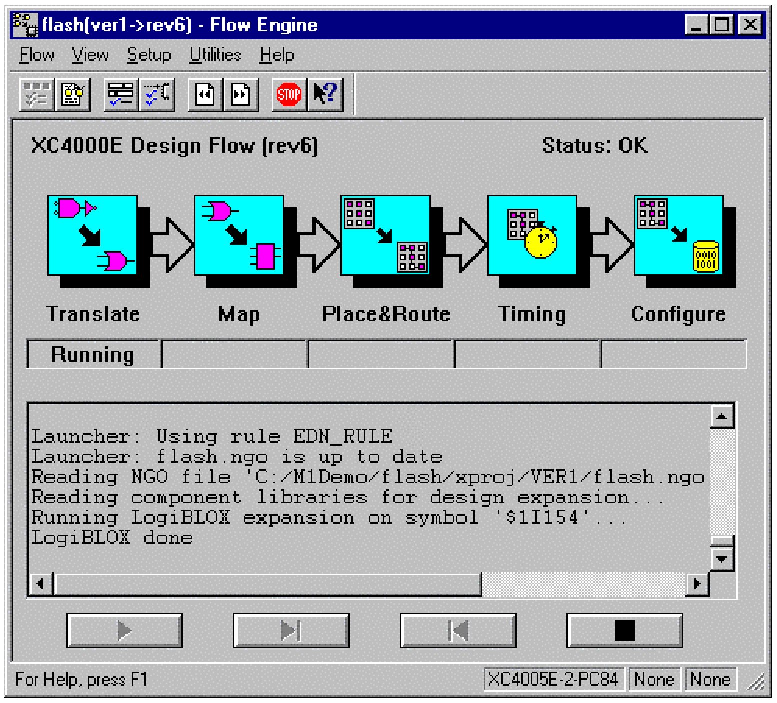
Vendor: Generic

Always Create Power-On Reset Port

Run the "Implement" Command

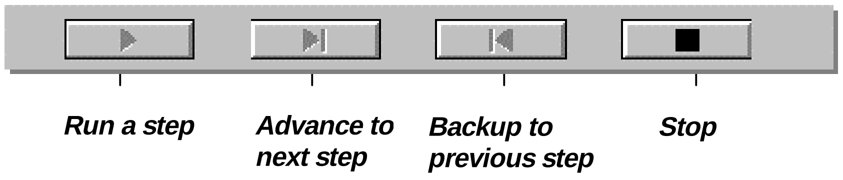
To launch the implementation process, in the “Implement” dialog, push the ***"Run"*** button.

This will cause the "Flow Engine" graphical interface to appear. The design is now processed through a 5 step sequence.

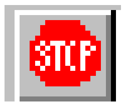


```
Launcher: Using rule EDN_RULE
Launcher: flash.ngo is up to date
Reading NGO file 'C:/M1Demo/flash/xproj/VER1/flash.ngo
Reading component libraries for design expansion...
Running LogiBLOX expansion on symbol '$1I154'...
LogiBLOX done
```

Notice the arrow buttons on the bottom of the window. These look like CD Player buttons and provide a similar function.

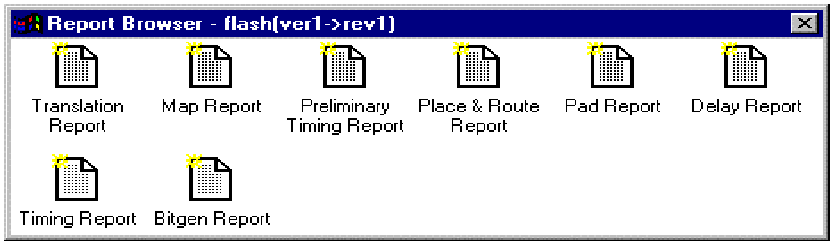


An "Implement" run can be stopped after any step by hitting the "Stop" button which appears in the form of a stop sign



Review the Alliance Series "Implement" Results

Once the Flow Engine has completed the “Implement” process, the “Implement Status” dialog box is posted. To review the processing which has occurred, review the log file. In the “Implement Status” dialog box, push the **“*View Logfile*”** button that will bring up the "Report Browser". Any report can be quickly viewed with a simple "double click" of the mouse

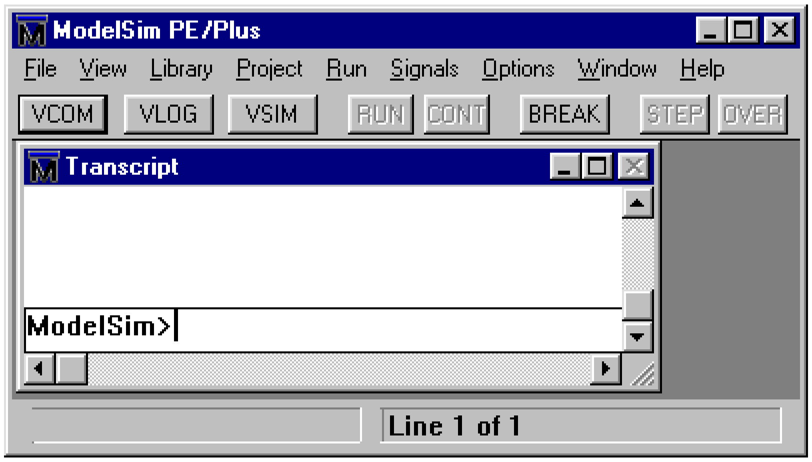


Example Model*Sim* Gate-Level Simulation (with SDF) Session

This section describes the basic steps to compiling and simulating the post-synthesis/post-place&route gate-level design with SDF timing back annotation.

1. Invoke Model*Sim*

   % vsim **-or-** **Programs > Model Tech > ModelSim**


ModelSim PE/Plus
File View Library Project Run Signals Options Window Help
VCOM VLOG VSIM RUN CONT BREAK STEP OVER
Transcript
ModelSim>
Line 1 of 1

2. Set Model*Sim* to the directory that contains the gate-level VHDL netlist (from Xilinx Design Manager).

   ModelSim> **cd c:\mygatedesign** **-or-** **File > Directory**

3. Create a working library to store the compiled design.

   ModelSim> **vlib work** **-or-** **Library > New**

4. Create a Simprim library to store the compiled Xilinx Simprim packages.

   ModelSim> **vlib simprim_lib** **-or-** **Library > New**

Note: *A separate library is not required for the Simprim packages. You could compile them into your* work *directory. If you did this, you would map the simprim library to* work *instead of* simprim_lib *as is shown in the next step.*

5. Map the Library name "simprim" to the library simprim_lib that was just created. This will allow ModelSim to know where to look when it encounters a "**library SIMPRIM;"** statement in the VHDL design.

   ModelSim> **vmap simprim simprim_lib**

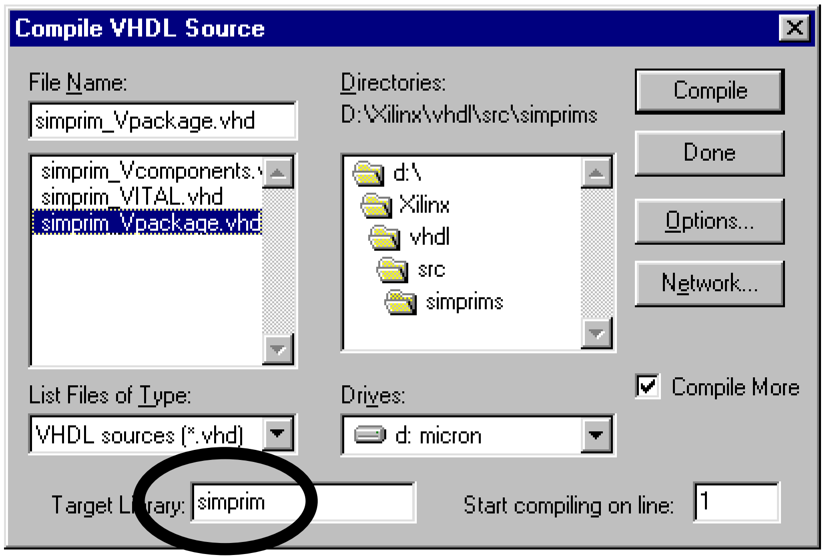
6. Compile the Simprim packages into the simprim_lib library.

ModelSim> **vcom -work simprim –explicit **
<Xilinx dir>\vhdl\src\simprims\simprim_Vpackage.vhd

ModelSim> **vcom -work simprim –explicit **
<Xilinx dir>\vhdl\src\simprims\simprim_VITAL.vhd

ModelSim> **vcom -work simprim –explicit **
<Xilinx dir>\vhdl\src\simprims\simprim_Vcomponents.vhd

This step is easier using the ModelSim VCOM button which brings up the following dialog. Make sure you compile in the proper order (Vpackage, VITAL, Vcomponents). Note the Target Library setting of *simprim* instead of *work*.


Compile VHDL Source

File Name: simprim_Vpackage.vhd

Directories: D:\Xilinx\vhdl\src\simprims

List Files of Type: VHDL sources (*.vhd)

Drives: d: micron

Compile More ✓

Target Library: simprim

Start compiling on line: 1

7. Compile the VHDL netlist created by the Xilinx Design Manager.

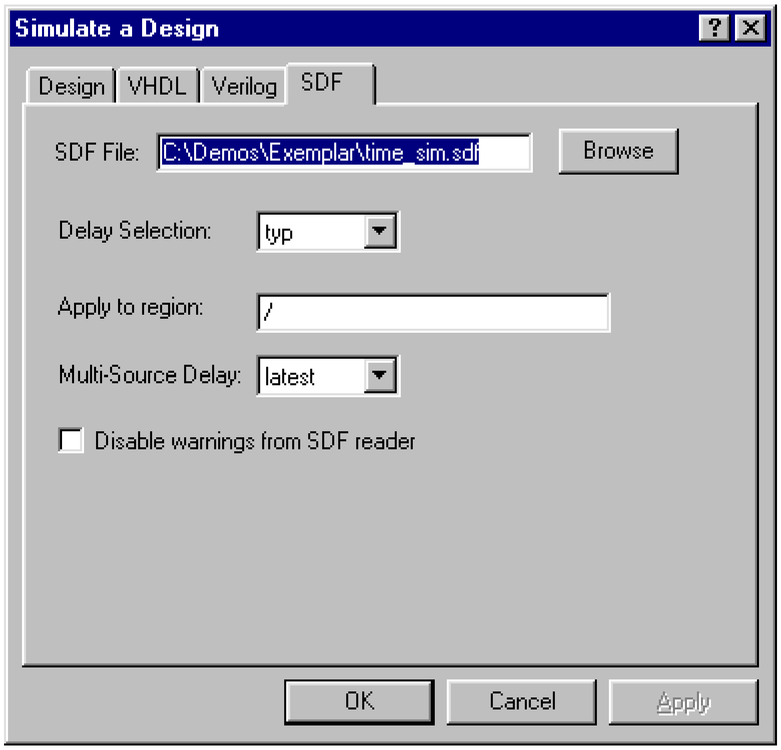
ModelSim> **vcom time_sim.vhd** **-or-** **VCOM** button

Note: *The place and routed* time_sim.vhd *gate-level design uses the* IEEE & SIMPRIM *VHDL libraries. The IEEE library comes pre-built in the ModelSim simulator. The SIMPRIM library was built in step 6 above. If the SIMPRIM libraries were not successfully compiled or if the library name* SIMPRIM *was not properly mapped to the SIMPRIM_LIB library above in step 5, then the ModelSim compiler would issue error messages complaining about "*Library simprim not found.*"

8. Start the Model*Sim* simulator applying the sdf information in the file tim_sim.sdf to the root level of the design (/).

   ModelSim> **vsim –sdftyp /=time_sim.sdf top**

   Or use the ModelSim VSIM button to start the simulator and apply the SDF info:



Simulate a Design

SDF File: C:\Demos\Exemplar\time_sim.sdf

Delay Selection: typ

Apply to region: /

Multi-Source Delay: latest

☐ Disable warnings from SDF reader

9. View all the Model*Sim* debugging windows.

   VSIM> **view ***   **-or-**   **View > All**

10. Wave and list signals of interest in the design.

```
VSIM> wave /*          -- Adds all top level signals to the wave window
VSIM> list /*              -- Adds all top level signals to the list window
```

11. As was done in the RTL simulation, either use a VHDL testbench to stimulate the RTL design, or force the inputs directly in ModelSim.

```
VSIM> force /clk 0 @ 0 ns, 1 @ 50 ns –repeat 100 ns
VSIM> force /input 0
```

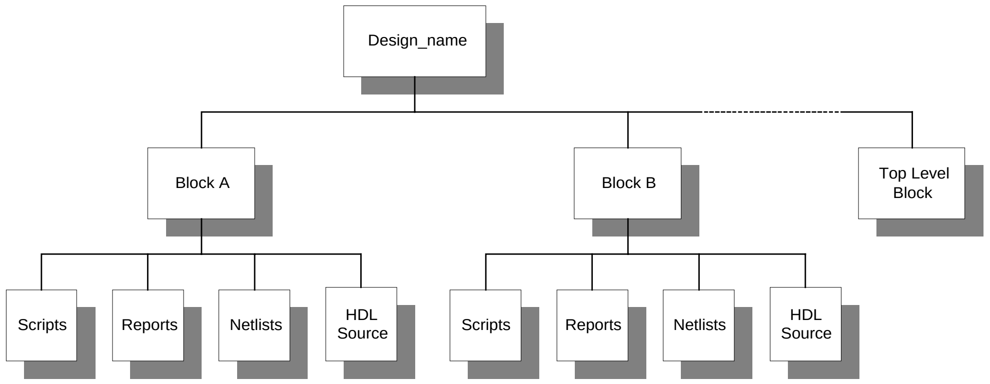
12. Run the simulation and analyze the information in the Model*Sim* debugging windows to verify the results are the same as in the RTL simulation.

   VSIM> **run –all**   **-or-**   **Run > Run Forever**

Environment Setup

Directory Structure

Although the choice of a directory structure is often an individual or team decision, Exemplar Logic provides the following recommendation to consider.



Scripts - Contains all constraint and optimization scripts

Reports - Contains all area, timing, constraint and environment reports

Netlists - Contains all optimized, mapped netlists. Sub-block netlists are saved in the sub-module directories. Once the design is assembled, the complete design netlist would be saved in the top-level netlist directory.

HDL Source - Contains the original VHDL or Verilog source code

Setting Aliases

Leonardo provides the ability to set aliases to rename any Leonardo command. The "exemplar.ini" startup file is the most logical place to define commonly used aliases.

Example Alias Command:

LEONARDO{5} **alias lp list_design -ports**

Leonardo Startup Files

Startup files can be a useful way to pre-configure Leonardo in your day to day convenience.

Example "exemplar.ini" Startup File

```
# Load Xilinx Cell and Modgen Libraries
load_library xi4xl
load_modgen xi4e

# Define common aliases
alias lp list_design -ports
alias reportit {report_area; report_delay}

# Set synthesis working directory - Note directory
# slashes are UNIX style - even for Windows users
cd D:/VHDL/uart_design

# Disable Asynchronous Feedback Loops
set delay_break_loops TRUE
```

Startup files for UNIX

Place the ***"exemplar.ini"*** file in your working directory (the directory that you will be invoking Leonardo from). The commands in the file will be automatically executed when invoking Leonardo.

Startup files for Windows

1. Place the ***"exemplar.ini"*** file in a personal or project folder that is not part of the Exemplar software install directory structure. All Exemplar software files will be deleted and replaced with each new software install.

2. Edit the file **$EXEMPLAR/data/exemplar.ini** directory to add the following line to the bottom of the file. You will have to re-add this line after each new software install.

   ```
   Source d:/<pathname to startup file>/exemplar.ini
   ```

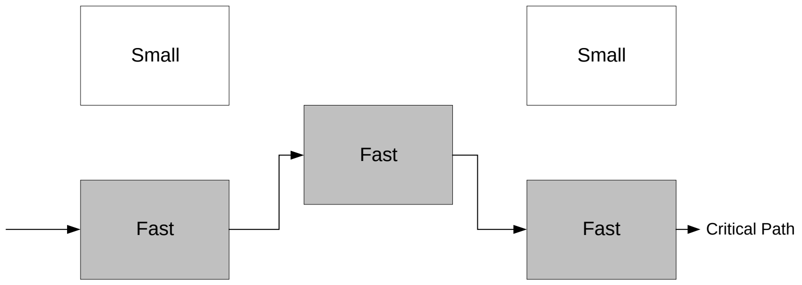
Design Methodology

Design methodology refers to the fundamental process of applying optimizations to an FPGA design. Generally speaking there are two approaches, "top-down" design which refers to applying a single optimization to the top-level of the design, and "bottom-up" which refers to practice of performing individual optimizations to design sub-blocks This section will discuss the merits of both as well as general guidelines to improve the quality of results you obtain.

Design Partitioning

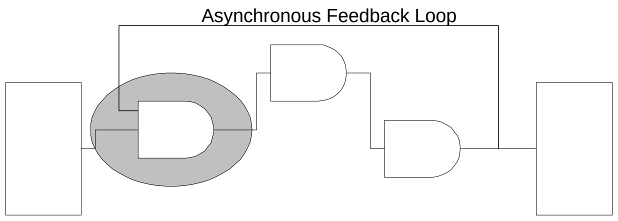
When partitioning a design into leaf blocks designers should take into account the following factors

1) Gate counts in leaf blocks should be between 10K and 50K gates. Optimizations can be performed on blocks much larger provided the sub-hierarchy falls within this guideline
2) Limit clocks to 1 per block. Leonardo supports multiple clocks per block but constraints and timing reports become more complex
3) Group similar logic together, i.e., state machines, data path logic, decoder logic, ROMs. Pay close attention to blocks that may lend themselves to special area or delay optimizations. For example, if you know a particular block is going to contain the critical path, eliminate any non-critical logic from that block.
4) Place state machines into separate blocks of hierarchy. This will help speed optimization and provide greater control over encoding
5) Separate timing critical blocks from non-timing critical blocks. Keep in mind that Leonardo performs area and timing optimizations separately. By separating timing critical logic into one block it may be possible to perform aggressive area optimizations on a greater percentage of the design thus creating a smaller circuit that meets timing.



Asynchronous Feedback Loops

When a timing loop occurs, timing analysis can become extremely slow due to the high iteration limit in Leonardo. Each loop must be evaluated 5000 times before Leonardo concludes that a loop exists and moves on. It is recommended that the ***"delay_break_loops"*** variable be redefined to "TRUE" at the start of each session. This will cause Loops will be automatically broken and a warning message will be written to the timing report file.

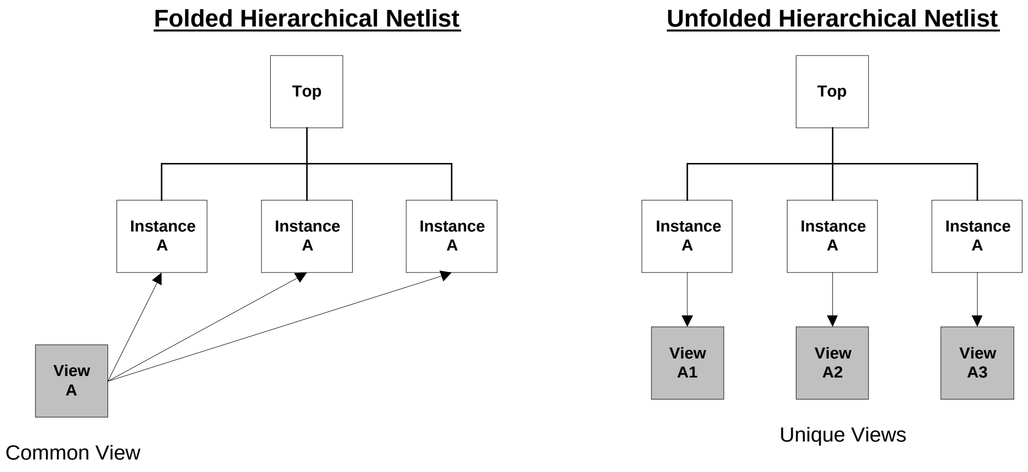


Example Loop Break Command:

LEONARDO{5} **set delay_break_loops TRUE**

Netlist Unfolding

Leonardo, by default, preserves hierarchy in a design. In order to insure the fastest possible run times the netlist is "folded" which means that all common subblocks reference a single view or "netlist". Leonardo only has to optimize this netlist once. In cases where the user wishes to perform two different optimizations on two different instances of common sub-blocks, the netlist must be unfolded. For example, one block may be optimized for area and a second for delay



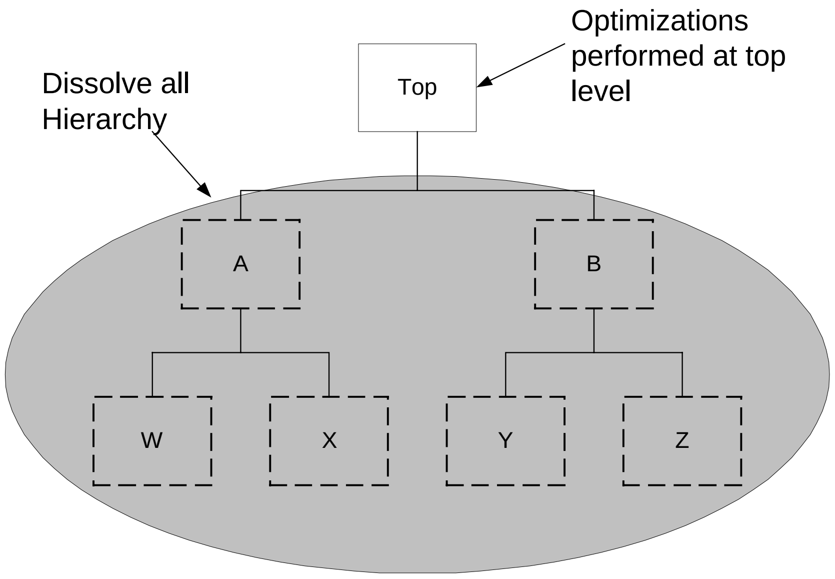
Example Netlist Unfolding Command:

LEONARDO{5} **unfold A**

Top-down Methodology

Block Area Considerations

It is recommended that individual blocks of hierarchy do not exceed 50K gates. Flattening sub-blocks to achieve this block size will generally result in optimal results. There is, however, a trade-off that must be considered. Preserving hierarchy of smaller sub-blocks will allow optimizations to complete much faster, but flattening will generally give better results.



Initial Methodology

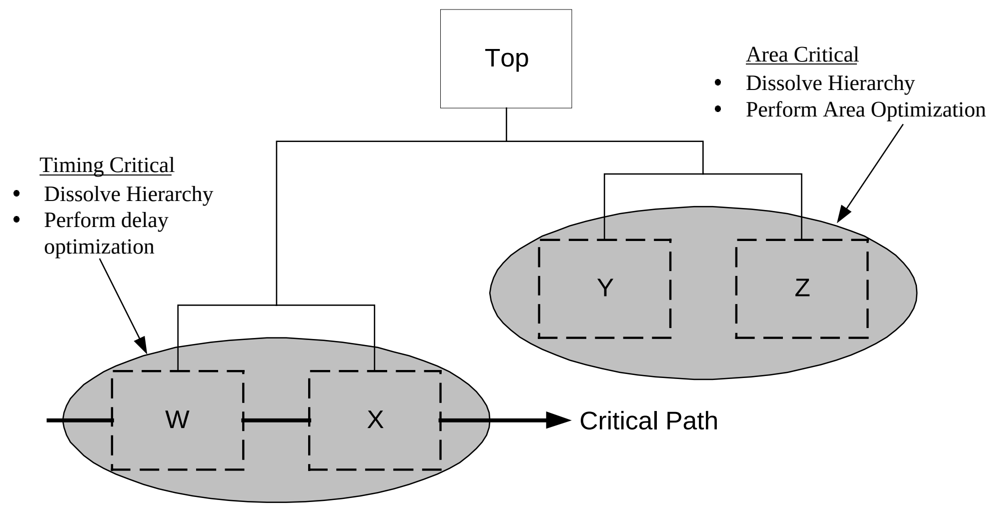
1) Read in the entire design and perform an area optimization with hierarchy preserved
2) Generate area and timing reports. If both are acceptable - you're done!

Area Critical Designs

1) If a smaller design is desired, perform an ***ungroup -all -hierarchy*** on blocks up to 50K gates.
2) Then re-optimize the design for area using the standard effort level. Use the command ***optimize -ta xi4e -area -effort standard***. This could take a while on large designs but will generally provide the best results.

Timing Critical Designs

The goal with timing critical designs is to generate the smallest circuit possible that meets timing. To achieve this we want to limit the use of delay and timing optimization to only the critical blocks.



Procedure

1) After initial, hierarchical optimization, identify critical paths from the timing report

2) Use the group command to combine the timing critical blocks into one hierarchical block and the non-timing critical blocks into a second hierarchical block.

   LEONARDO{8} **Group w x -inst_name timing_critical**

   LEONARDO{9} **Group y z -inst_name area_critical**

3) Use the ungroup command to dissolve all hierarchy beneath the Timing_critical instance and the area_critical instance. Note that the top-level hierarchy is preserved.

   LEONARDO{10} **present_design work.timing_critical**

   LEONARDO{11} **ungroup -all -hierarchy**

4) Perform a delay optimization of this sub-block. Be sure to use the "-macro" switch, this will prevent I/O buffers from being placed in the ports of the sub-blocks

   LEONARDO{12} **optimize -ta xi4e -delay -effort standard -macro**

5) Repeat the same process with the area critical block except perform an area optimization

   LEONARDO{13} **set present_design work.area_critical**

   LEONARDO{14} **ungroup -all -hierarchy**

   LEONARDO{15} **optimize -ta xi4xv -area -effort standard -macro**

   LEONARDO{16} set present_design work.top

6) Save the netlist and you're done

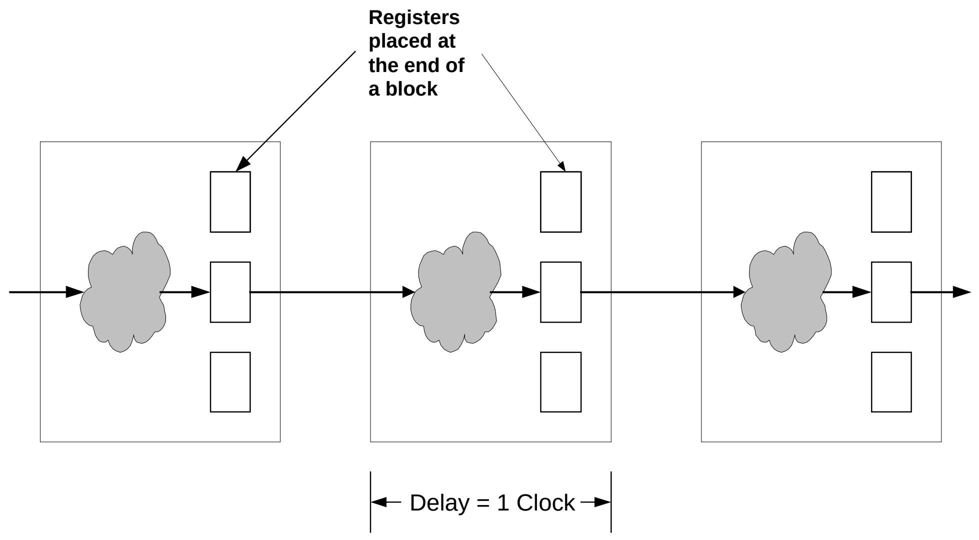
   LEONARDO{17} **auto_write -format edif top.edif**

Bottom-up Methodology

Generally bottom-up design methodologies are employed when doing team design or for extremely large designs. Bottom-up design refers to the technique of performing optimizations on individual sub-blocks of a design then later "stitching" the design together.

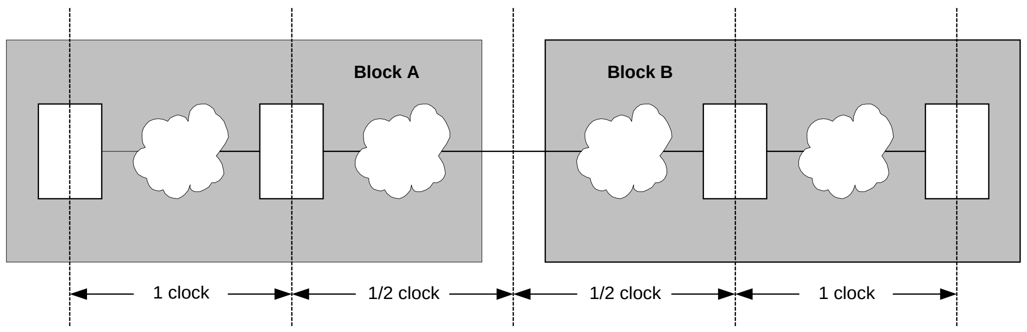
Register Placement within Blocks

There are two "barriers" that constrain optimization, hierarchical boundaries and registers. When designing hierarchically, it is recommended that registers be placed at either the front or back of the hierarchical boundaries (not both!), which essentially combines these two barriers into one single barrier. This will minimize the impact to overall results when performing bottom-up optimizations. If this design practice is followed then preserving hierarchy in a design will have no impact on optimization results and allow faster CPU run times.



Constraining Sub-blocks for Timing

In the ideal world, registers would be placed at hierarchical boundaries, but the world is not ideal. Often random logic must be placed at the hierarchical boundaries forcing the designer to constrain the logic appropriately. Unless more detailed information about the sub-block timing is known, it is recommended that constraints equal to 1/2 the clock period be applied to the boundary. If both sides meet timing, then when the blocks are combined timing will be met



Constraining sub-block Pins for Loading

The nature of Xilinx 4000 device routing channels prevents the need to specify loading and drive constraints on sub-block pins.

Saving Intermediate Netlists

In Leonardo, two commands exist for saving EDIF netlists,

1. ***write -format edif <filename.edif>***
2. ***auto_write -format edif <filename.edif>***

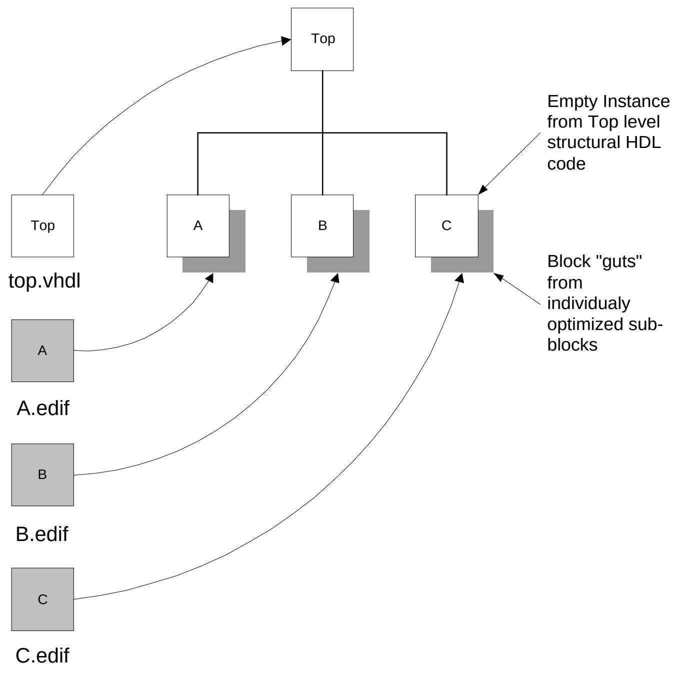
The ***"auto_write"*** command invokes a technology specific netlist post-processor that performs several modifications required for seamless integration into the Xilinx Alliance Series place and route environment. These changes may not be compatible with Leonardo internal netlist formats and libraries. For this reason, ***"auto_write"*** should not be used for saving intermediate netlists. Use ***"write -format edif"*** to save all intermediate netlists. ***Use "auto_write"*** when netlisting to the Alliance Series place and route environment only.

> **Note:** *All "write" commands issued from the toolbar, flow diagram and pulldown menus will issue an "auto_write" command by default. To save intermediate netlists from the user interface, deselect the "do automatic processing for target technology" option*

```
LEONARDO{5} auto_write -format edif top.edif  -- saves netlist for place and route
LEONARDO{6} write -format edif top.edif -- saves netlist in original format
```

Design Stitching

Design stitching refers to the process of building up the entire design after bottom-up optimizations have been performed on individual sub-blocks. Leonardo has the ability to automatically connect sub-blocks with top-level structural netlists - provided all instance names, port names and view names match. The Leonardo design browser can be a useful tool when working through design stitching issues.


Top
Empty Instance from Top level structural HDL code
Top
top.vhdl
A
B
C
Block "guts" from individualy optimized sub-blocks
A
A.edif
B
B.edif
C
C.edif

Designs should be stitched "bottom-up" meaning that all lower level blocks, that have completed optimization, should be read into Leonardo first. The next step is to synthesize the top-level structural VHDL or Verilog file thus connect the sub-blocks together.

LEONARDO {5} **read -format edif A.edif B.edif C.edif**

LEONARDO {6} **read -format vhdl top.vhdl**

Note: *View names between the sub-blocks and the instances contained within the top-level structural code must match exactly for bottom-up design stitching to be successful. Generally problems are easiest to resolve by modifying EDIF, VHDL or Verilog source. Alternatively the Leonardo* ***"add_rename_rule"*** *command may be used. Refer to the Leonardo Command Reference Manual" for more information.*

Final Optimization

Once a design has been stitched together bottom-up the user should generate final area and timing reports. If the design meets specification then 1 final optimization run is required to insert the GSR reset circuitry and add the chip I/O buffers. To perform final optimizations run the following command.

LEONARDO{5} **optimize -ta xi4xv -chip -area**

Note: *Leonardo automatically instantiates "ibufs", "obufs", "bufgp's" and "bufgs" buffers. A maximum of 4 bufgp's and bufgs's will be instantiated on the input ports with the highest capacitance. The "optimize" command must be run with the "-chip" switch to enable I/O buffer insertion*

State Machine Synthesis

Leonardo encodes State Machines during the synthesis process. Once a design has been "encoded" during synthesis it cannot be re-encoded later in optimization. A particular VHDL or Verilog coding style must be followed to allow Leonardo to identify the state machine. Refer to the manual, "HDL Synthesis Guide for Leonardo" for additional information.

Note*: Although not required, it is recommended that state machines are isolated into separate hierarchical blocks. This will speed optimization performance and allow easy modifications to state machine encoding.*

Supported State Machine Styles

Binary - Will generate state machines with the fewest possible flip-flops. Binary state machines are useful for area critical designs when timing is not a concern.

Gray - Will generate state machines where only one flip-flop changes during each transition. Gray encoded state machines tend to be glitchless.

Random - Will generate state machines using random state encoding. Random state machine encoding should only be used when all other implementations are not achieving the desired results. It is basically a shot in the dark and not recommended.

One Hot - Will generate state machines containing one flip-flop for each state. One hot state machines provide the best performance and shortest clock to out delays. One-hot implementations are larger than binary but are the preferred choice for Xilinx FPGA architectures. This is because these devices are rich in Flip-flops.

Note: *The encoding default is onehot but can be changed via the* ***set encoding*** *command or from the GUI.*

Setting State Machine Encoding in Leonardo

There are two ways to instruct Leonardo to perform a particular state machine encoding

1) VHDL Attributes or Verilog Pragmas
2) Using the Leonardo command **"set encoding"**

Setting VHDL Attributes:

To set the encoding for a particular state machine insert the following statements into your code.

```
-- Declare the type_encoding_style attribute
type encoding_style is (BINARY, ONEHOT, GRAY, RANDOM);
attribute TYPE_ENCODING_STYLE: ONEHOT;

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

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

Setting Verilog Pragmas:

To set the encoding for a state machine in Verilog insert the following comment text into your Verilog Model above the state machine model

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

Note*: In the first line of the above code example, the state machine encoding specified is "onehot". This is an optional specification that could also be set to "binary", "gray" and "random". If the "enum" pragma is specified but not set to a partito indicate FSM encoding.*

Note: *VHDL Attributes and Verilog pragmas will override the encoding variable*

Setting State Machine Encoding using the "Encoding" Variable

Alternatively, the ***encoding*** variable may be used to set state machine encoding. Once this variable is set all state machines synthesized from that point on will employ the specified encoding until another “set encoding” command issued. Set this variable prior to reading in VHDL or Verilog code

VHDL Example,

LEONARDO{5} **set encoding onehot**

LEONARDO{6} **read uart_control_sm.vhdl**

Verilog Example,

LEONARDO{5} **set encoding binary**

LEONARDO{6} **read -format verilog control.v**

Table 3. Arguments to the Encoding variable

Argument	Description
binary	Sets state machine encoding to binary
onehot	Sets state machine encoding to onehot
gray	Sets state machine encoding to gray
random	Sets state machine encoding

Reading Designs

Design Input Commands

Leonardo provides two methods for reading in designs.

The "Read" Command

The read command analyzes and elaborates the design in one step. Generally speaking this is the most useful command. Read can be used to input structural or RTL designs in VHDL, Verilog, EDIF and XNF. Read supports both single file and multi-file designs. Read cannot be used for RTL VHDL designs that contain user defined packages or if the user wishes to re-define generics during synthesis.

Arguments to the Read Command

Argument	Description
-format	Edif | vhdl | verilog | edif | sdf | xnf
-dont_elaborate	Only analyze, don't elaborate
-design	Specify the top-level designname to be read
-work	Specify library where read design is to be stored

Analyze and Elaborate

Must be used when reading in VHDL design with user defined packages. Analyze and Elaborate is never required for Verilog synthesis. Must be used when re-defining VHDL generics during synthesis.

Arguments to the Analyze Command

Argument	Description
-format	Edif | vhdl | verilog
-work	Specify library where read design is to be

Arguments to the Elaborate Command

Argument	Description
-architecture	Root architecture name
-single_level	Only elaborate the top-level of the design
-generics	Redefine any specified generic
-parameters	Redefine root level generics
-work	Specify library where read design is to be stored

VHDL Synthesis

Follow the procedure outlined below when performing VHDL synthesis

1) Files must be read in bottom-up order, i.e., lower level blocks must be read before the top level blocks

2) If no user defined VHDL packages were used in the design, all files may be read using the ***read*** command.

   LEONARDO{5} **read bottom.vhd middle.vhdl top.vhdl**

3) If the design contains user defined packages then use the analyze / elaborate commands. Analyze the packages first, then the VHDL code in a bottom-up order

   LEONARDO{5} **analyze my_package.vhd**

   LEONARDO{6} **analyze bottom.vhd**

   LEONARDO{7} **analyze middle.vhd**

   LEONARDO{8} **analyze top.vhd**

   LEONARDO{9} **elaborate top -generic data_width = 16**

Verilog Synthesis

Verilog designs can be read into Leonardo in any order. Leonardo supports "auto-top detection" which will automatically locate the top-level module.

LEONARDO{5} **read -format bottom.v top.v middle.v**

Synthesizing Operators - ModGen vs LogiBlox

The Exemplar - Xilinx design methodology offers the user two distinct methods to create logic for operators. The easiest method is to simply infer an operator in Leonardo by using an arithmetic or logic operator symbol in VHDL or Verilog.

```
Sum <= a + b;
```

In the above example Modgen will recognize the "+" and build an optimized circuit for Xilinx

Alternatively, users may choose to instantiate a Xilinx LogiBlox cell directly into their HDL code. This cell will be passed to the Alliance Series design environment in the EDIF netlist generated from Leonardo as an unimplemented black-box. In turn, Alliance Series will generate the optimized logic for the cell.

So which one will generate the better adder? The answer is both! For operators supported by ModGen, Leonardo will generate as good or better an implementation than Alliance Series can for LogiBlox. Additionally, since Leonardo is building the circuit from primitive gates, accurate timing and area reports can be generated for the design. For these reasons plus the added advantage of maintaining portable HDL code it is recommended users limit the use of LogiBlox cells in a design to critical blocks that have no ModGen equivalent circuit.

Synthesizing Designs with Black Boxes

Both the Xilinx IP flow and the instantiated LogiBlox flow will insert black-box elements into the netlist. Leonardo can handle this with one exception. Black-boxes will prevent the automatic GSR insertion from occurring. In this case assign an attribute to the reset net to force GSR insertion as follows

```
Set attribute insert_gsr -net work.top.structural.reset
```

Resource Sharing

Resource sharing is an optimization technique that seeks to identify sharable operators. If successful, the optimization can replace several operators, such as adders or multipliers, with a single operator and a mux. Although this can provide significant area gains, often it is at the expense of timing. Issuing a "pre_optimize" command without setting the **"-common_logic"** switch prior to performing "optimize" will disable resource sharing. Please note that this step is not normally required.

```
LEONARDO{6} pre_optimize .work.pl_counter.rtl -unused_logic -extract
```

RAMs

Inference

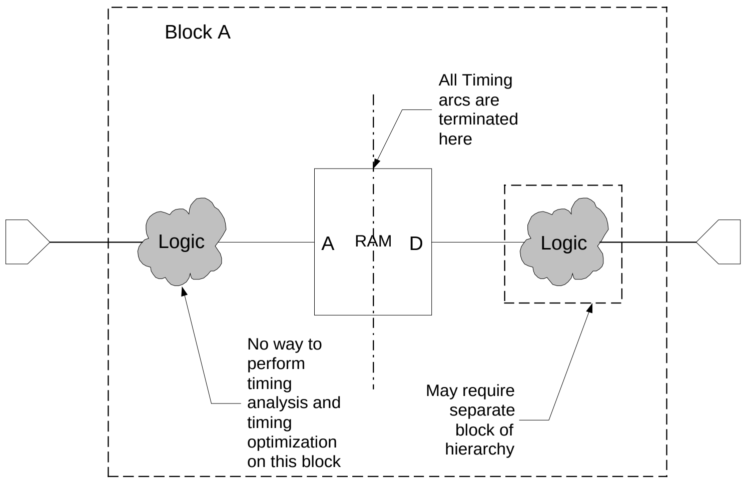
Leonardo has the ability to infer single port, synchronous and asynchronous RAMs from RTL code. When a RAM is modeled in the form of a 2-dimentional array, Leonardo will recognize the function and insert a blackbox into the netlist with attached properties. The Xilinx Alliance Series design environment will recognize the properties and insert the appropriate RAM into the design.

VHDL Example

```
      I0 : process (we,address,mem,data_in)
            begin
              if (we = '1') then
                mem(conv_integer(address)) <= data_in ;
              end if ;
              data_out <= mem(conv_integer(address)) ;
          end process ;
```

Timing

In the current version of Leonardo there is no method to apply timing arcs or timing constraints to the RAM model. This prevents Leonardo from performing timing analysis and timing optimization on logic directly connected to RAMs which often includes the address to dataout path of the ram.

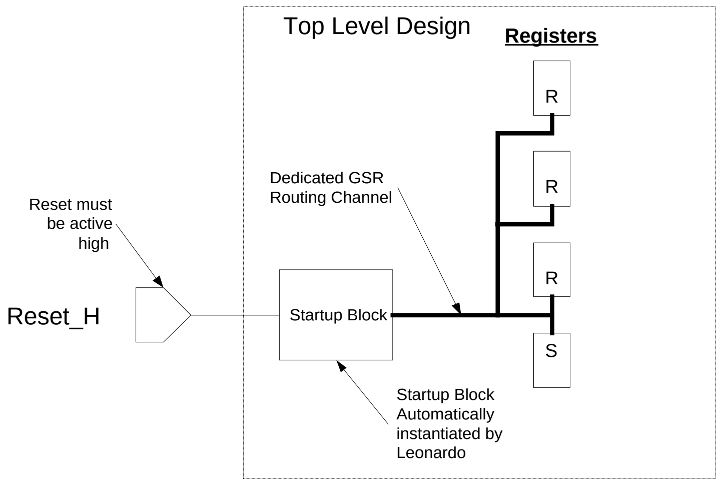

Block A
All Timing arcs are terminated here
Logic
A
RAM
D
Logic
No way to perform timing analysis and timing optimization on this block
May require separate block of hierarchy

It is recommended that users perform timing analysis in the Alliance Series environment to detect timing problems through RAM paths. If a critical path exits and further optimization is required on the circuit follow the following procedure:

1) Re-optimize the circuit with the ***"-delay"*** and the **"*-effort standard*"** options
2) Place critical logic into a separate hierarchical block. This could be done with the ***group*** command or may require code re-write. Leonardo supports the VHDL block statements for hierarchy, which may provide a simple method to achieve this.
3) Set timing constraints and perform optimizations as described in the timing critical optimization strategy section of this document.

GSR Resets

GSR, when asserted, set / resets all the flip-flops in the chip based on the initialization state of the FF (s or r). The GSR uses dedicated routing channels and can be made implicit in Leonardo. For that reason it is the recommended method for initializing design flipflops.


Top Level Design
Registers
R
R
R
S
Dedicated GSR Routing Channel
Reset must be active high
Reset_H
Startup Block
Startup Block Automatically instantiated by Leonardo

The following procedure will correctly insert reset circuitry:

1) Set the variable ***"infer_gsr"*** to TRUE. This will enable the GSR inference in Leonardo.

```
LEONARDO{5} set infer_gsr true
```

2) In your VHDL or Verilog code, implement an active high, asynchronous reset scheme

3) Perform a top-level optimization with the ***"-chip"*** option

```
LEONARDO{6} optimize -ta xi4e -area -chip -effort quick
```

DWANDS

The Xilinx XC4000 and XC4000E families have dedicated decoder circuitry at each edge of each device. Designs that use address decoders 16 bits or greater, can take advantage of this logic to improve chip area and speed. Leonardo does not infer this logic from RTL code, users are required to directly instantiate decoder cells (DWAND) into their HDL models.

Note *When a DWAND cell is used in the design that design will no longer be technology independent. A code re-write would be required to retarget the model to another technology. For that reason they should be used only when necessary.*

Procedure for Using DWANDs

1) Instantiate the DWAND's primitive cells into the VHDL or Verilog source. Refer to the Leonardo Synthesis and Technology guide for a code example

2) Load the xi4 or xi4e technology into Leonardo prior to reading the RTL code. Because DWANDs are technology primitive cells, the library must be loaded for vendor cell instantiation to work properly.

Setting Constraints

Introduction:

Setting constraints within Leonardo is easy and straightforward. Constraints can be as simple as specifying the target design frequency to as powerful as indicating multi-cycle paths between flops. Timing constraints indicate desired target arrival and required times used for setup and hold analysis. Constraints should be applied after the design has been read into Leonardo and before optimization. Setting constraints is simple. Leonardo assumes intuitive defaults. At a minimum, users should define the clock, input port arrival times, and output port required times.

Note: *Designers should not over-constrain the design. Doing so may have several undesirable effects such as increasing the design size and long optimization run times.*

Clocks

Maximum Frequency Constraint:

The easiest way to constrain a design is to specify a global frequency constraint called ***"maxdly".*** For example, if you wish to set the maximum design frequency to 20Mhz. Then the flop to flop, pad to pad, pad to flop and flop to pad constraints = 1000/20 = 50ns and the Leonardo command would be:

```
LEONARDO{6} set maxdly  50
```

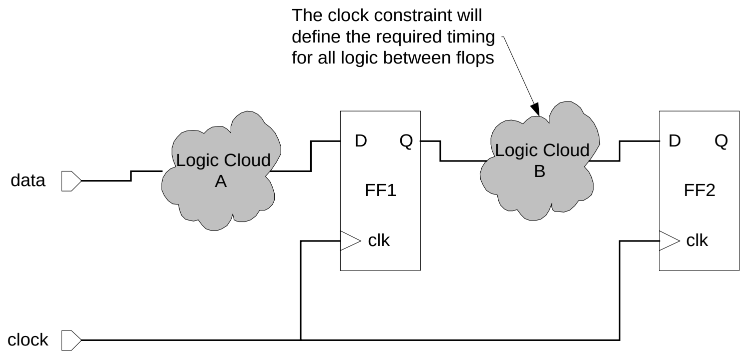
This will constrain all combinational paths throughout the design to 50ns.

Note1: *It is not recommended to set both maxdly and clocks. The user should choose one method of applying constraints*

Note2: *maxdly only constraints the combinatorial logic to, from and between registers. Any library specified setup time for flops is not considered. Therefore, to truly run at target frequency, you should margin maxdly by worst case setup. For example, if he worst case flop setup is 2.5 ns, set maxdly to 47.5 ns*

Clock Constraints:

The "maxdly" command constrains all combinatorial paths in the design to a specific frequency including flop to flop and pad to pad. Setting a clock constraint will only constrain the flop to flop logic. Clocks define timing to and from registers. Without clocks defined, all registers are assumed unconstrained. Therefore all combinational logic between registers is ignored during timing optimization. When you define a clock, you have effectively constrained the combinational logic between all registers to one clock period. Consider the following circuit.


The clock constraint will define the required timing for all logic between flops
data
clock
Logic Cloud A
Logic Cloud B
D
Q
FF1
clk
D
Q
FF2
clk

The logic between FF1 and FF2 is constrained to one clock period. If clock period is 50ns, then the Logic Cloud B has roughly 50ns – setup of FF2 to meet timing.

Leonardo describes clocks by using three basic commands:

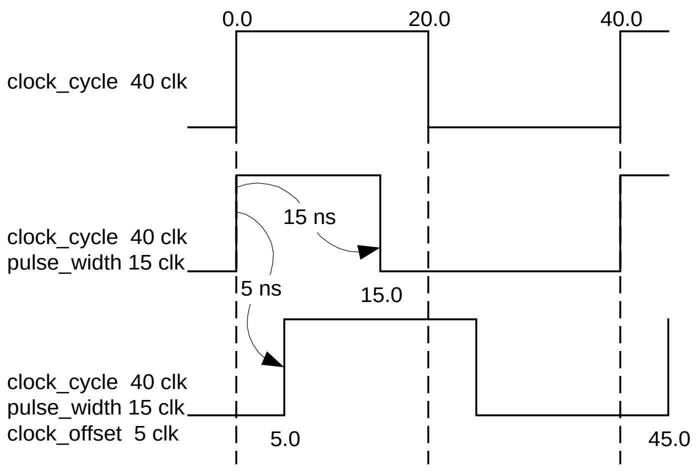
clock_cycle *<clock_period> <primary_input_port>*

pulse_width *<clock_pulse_width> <primary_input_port>*

clock_offset *<clock_offset> <primary_input_port>*

By default, the clock network is assumed to be ideal or in other words, it is assumed to have no delay. So the clock arrives the same time between all flops. To change clock network to propagated delay, set **propagate_clock_delay** variable to true.

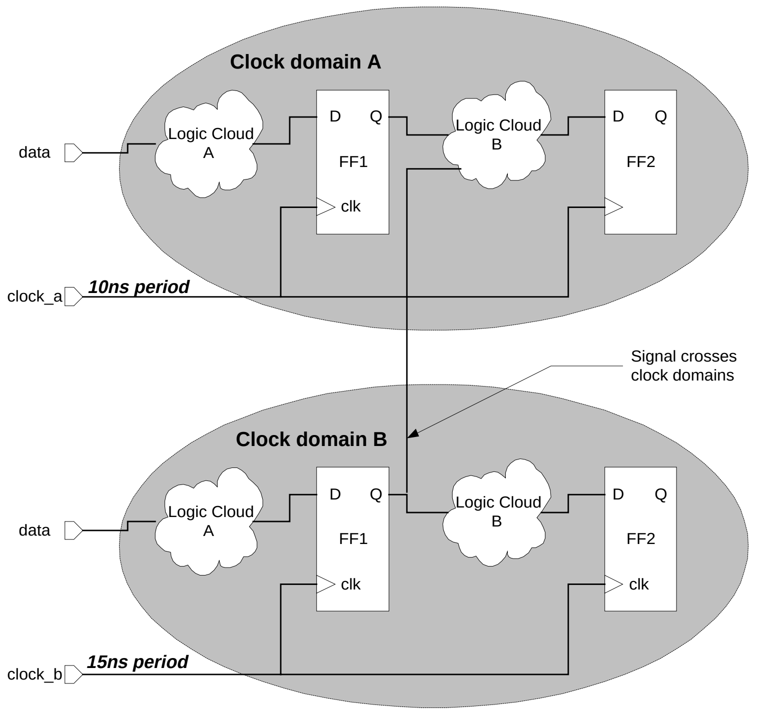
Example of clock constraints:

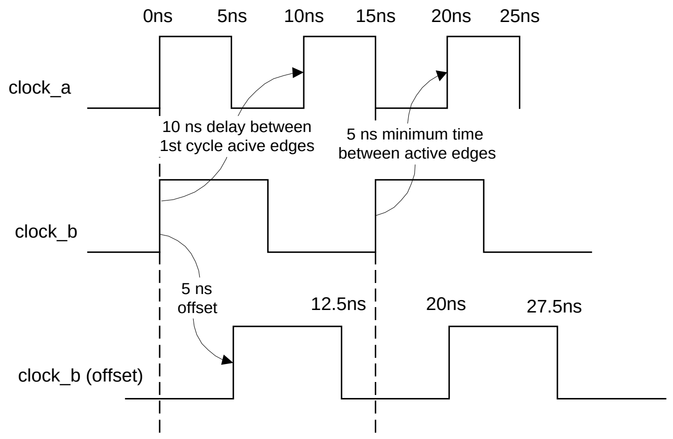


In the first example, clock period is defined as 40 ns and attached to clock port “clk”. The default duty cycle is 50% or in this case a clock pulse width of 20ns. The second example shows how to change the pulse width to 15ns. The final example demonstrates how one can offset the clock. This could be useful for specifying a clock skew relative to zero.

Multiple Synchronous Clocks per Block

The Leonardo 4.2 timing analyzer supports only 1 clock per block for exhaustive timing analysis. Designs with multiple synchronous or asynchronous clocks can be analyzed however, a technique involving setting clock offsets must be employed. To demonstrate how this works consider the following example where Block A and Block B are each driven by different, synchronous clocks.




0ns
5ns
10ns
15ns
20ns
25ns
clock_a
10 ns delay between 1st cycle acive edges
5 ns minimum time between active edges
clock_b
5 ns offset
12.5ns
20ns
27.5ns
clock_b (offset)

Procedure for setting multiple, synchronous clock constraints

1) Manually extrapolate the 2 clocks to determine the minimum time between active edges. This may not happen in the first clock cycle - it all depends on how the active edges come together. In the above example the minimum time is 5 ns.

2) Determine what the delta between active edges is during the first clock cycle. If the above example the delta is 10 ns.

3) Subtract the minimum active edge delta from the 1st cycle active edge delta. This number will become the clock offset for the clock of signal origin. Set the appropriate clock offset

    LEONARDO{5} **clock_offset 5 clock_b**

4) Now the tricky part! Setting the clock offset will alter the input arrival timing. If you had set an input arrival time of 6 for example, the clock offset has effectively turned that number into (6 + offset). You will have to adjust the input arrival time to correct for the offset adding the offset to the input arrival.

Multiple Asynchronous Clocks

Leonardo has no way of analyzing timing for signals that cross between two or more asynchronous clock domains. This is due to the fact that the clocks themselves have no defined relationship. The correct way to handle this is to completely ignore all timing between signals that cross between asynchronous clock boundaries. This can be accomplished by simply assigning a clock offset to the clock of signal origin equal to 2 clock periods or greater. To disable timing between clock boundaries in our example above issue the following command

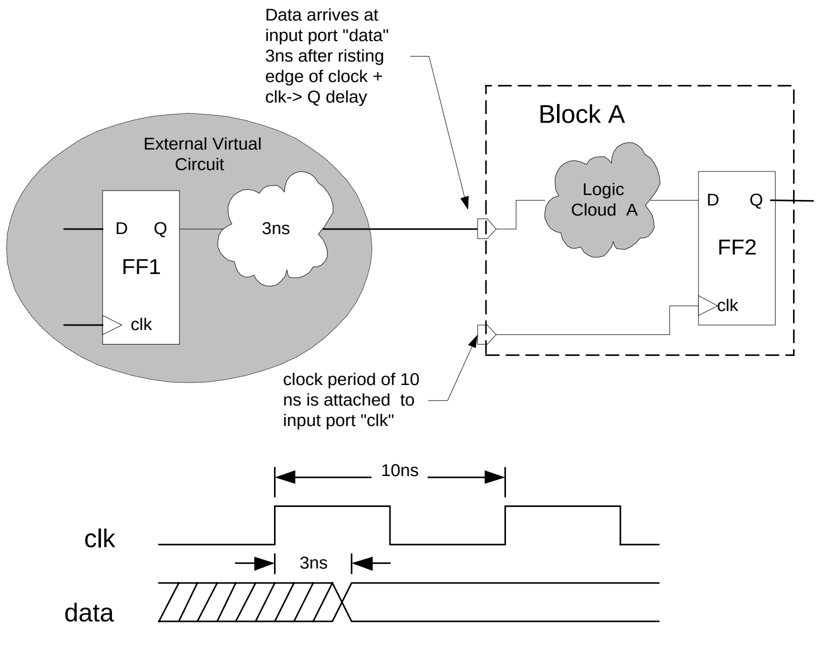
    LEONARDO{6} **clock_offset 30 clock_b**

Any input arrival times would need to be increased by the amount of the clock offset

Input Arrival Time

The input arrival time specifies the maximum delay to the input port through external logic to the synthesized design.

arrival_time *<delay_value> <input_port_list>*


Data arrives at input port "data" 3ns after risting edge of clock + clk-> Q delay
Block A
External Virtual Circuit
D Q
FF1
clk
3ns
Logic Cloud A
D Q
FF2
clk
clock period of 10 ns is attached to input port "clk"
10ns
clk
3ns
data

A clock must be defined prior to specifying an input arrival time. In this case, data arrives roughly 3ns after the rising edge of clk. Therefore, to accurately constrain the input port "data", you should apply the following constraint:

LEONARDO{5} **arrival_time 3 { data }**

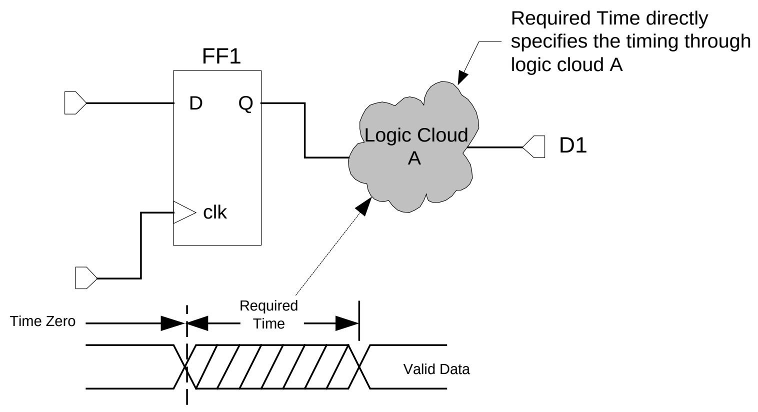
If the clock period were defined as 10ns, then the setup of FF2 added to the combinatorial delay of logic cloud A would need to be 7ns to meet timing.

Note: *All input arrival times start at time zero and cannot be specified relative to a particular clock edge. To adjust for a particular clock edge, users need to add the clock offset to the arrival time.*

Output Required Times:

The output required time specifies the data required time on output ports. Time is always with respect to time zero. In other words, output required time cannot be specified relative to a particular clock edge

LEONARDO{5} **required_time <required_value> <output_port_list>**

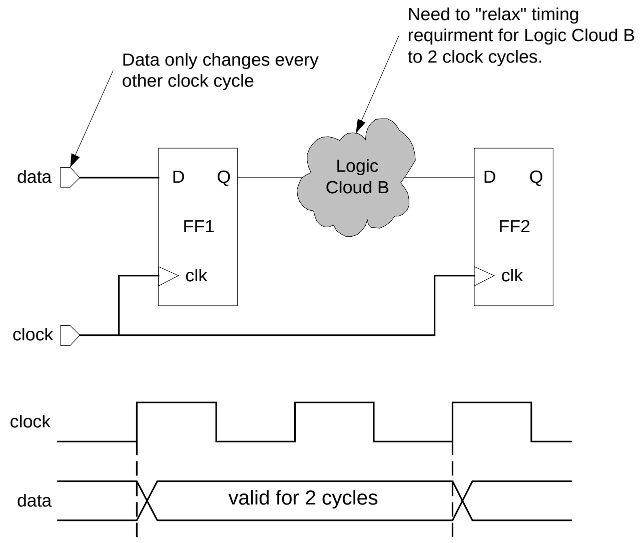

Required Time directly specifies the timing through logic cloud A
FF1
D
Q
clk
Logic Cloud A
D1
Time Zero
Required Time
Valid Data

When specifying required times all constraints are assumed to begin at time zero. This eliminates the need to specify a constraint relative to a particular clock edges. The specified required time will become the time constraint on the output logic cloud show as logic cloud A in the above example

LEONARDO{5} **required_time 7 { d1 }**

Multicycle Path Constraints:

Leonardo versions 4.2 and later offer the ability to constrain multi-cycle paths


Need to "relax" timing requirment for Logic Cloud B to 2 clock cycles.
Data only changes every other clock cycle
data
D
Q
FF1
clk
Logic Cloud B
D
Q
FF2
clk
clock
clock
data
valid for 2 cycles

To appropriately constrain this design above you should apply the following constraints:

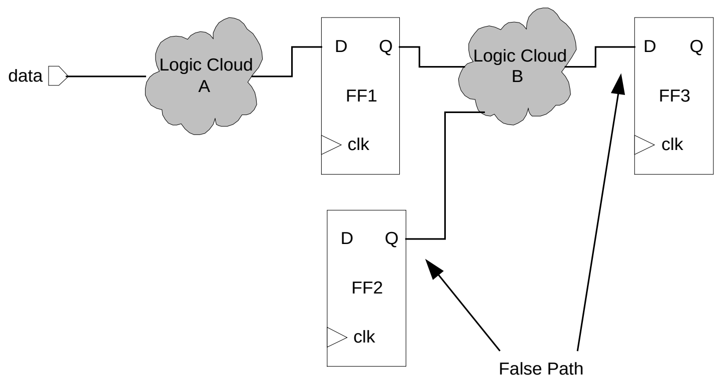
LEONARDO{6} **set_multicycle_path -from { FF1 } -to { FF2 } -value 2**

This constraint has the effect of constraining logic cloud B to 2 clock periods minus the setup of FF2.

Note: *Exercise caution while using multi-cycle constraints since they will slow timing analysis. A few multi-cycle constraints will have little effect, however, many will slow timing optimization dramatically.*

False Path Constraints:

False paths are paths in a design that the user needs Leonardo to ignore for timing optimization. Consider the following design:



By taking advantage of the multi-cycle command, users can specify the path from FF2 to FF3 as false:

LEONARDO{5} **Set_multicycle_path -value 1000 -from { FF2 } -to { FF3 }**

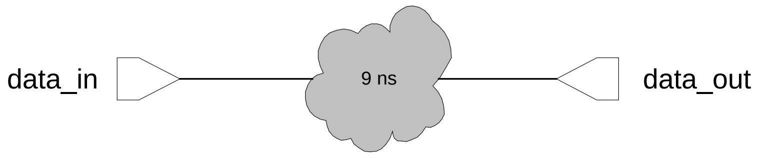
Essentially, the path from FF2 to FF3 has been constrained to a large number. Since it is unlikely that the logic cloud B would ever take more than 1000 cycles, this path has effectively been eliminated from timing optimization and timing analysis.

Note: *As with multi-cycle, exercise caution when specifying too many false paths since it will increase timing analysis run times.*

Constraining Purely Combinatorial Designs

A purely combinatorial design has no clocks. Users can constrain these blocks by simply specifying the ***maxdly*** constraint. In this example, ***maxdly*** is being used to constrain logic beween two ports.

Example:

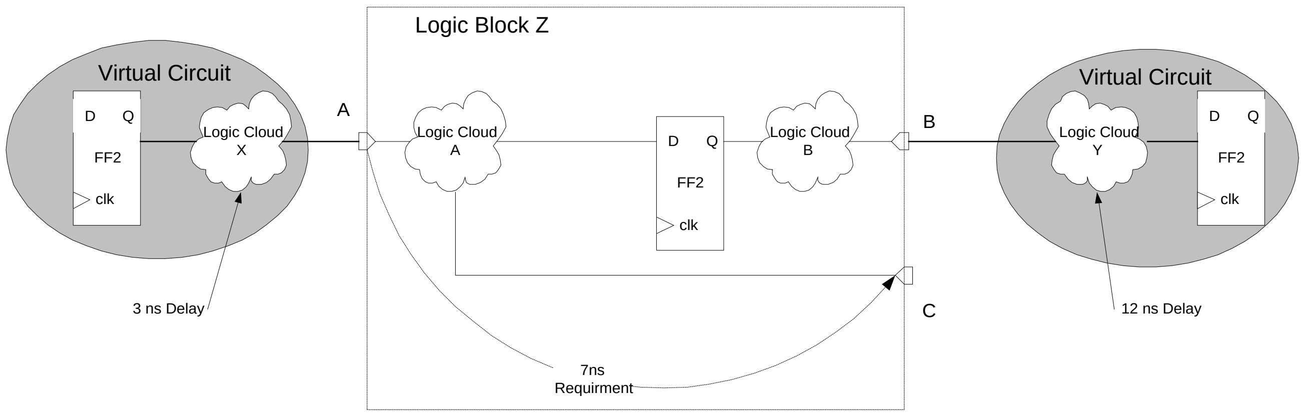


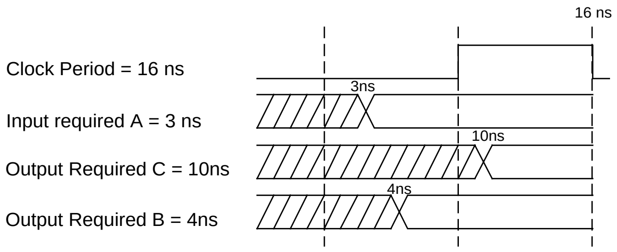
LEONARDO{5} **set maxdly 9**

Note: *The command will constrain any combination path including flop to flop, pad to flop, flop to pad and pad to pad.*

Constraining Mixed Synchronous and Asynchronous Designs

Some blocks have both synchronous and purely combinatorial paths through the circuit. A mealy state machine is a good example of this. To constrain these cases we will apply synchronous constraints to the ports of the synchronous paths and asynchronous constraints to the ports of the asynchronous paths.





Procedure for Setting Constraints on Mixed Designs

1) Define the clock constraints

   LEONARDO{5} **clock_cycle 16 clk**

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

   LEONARDO{6} **arrival_time 3 A**

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

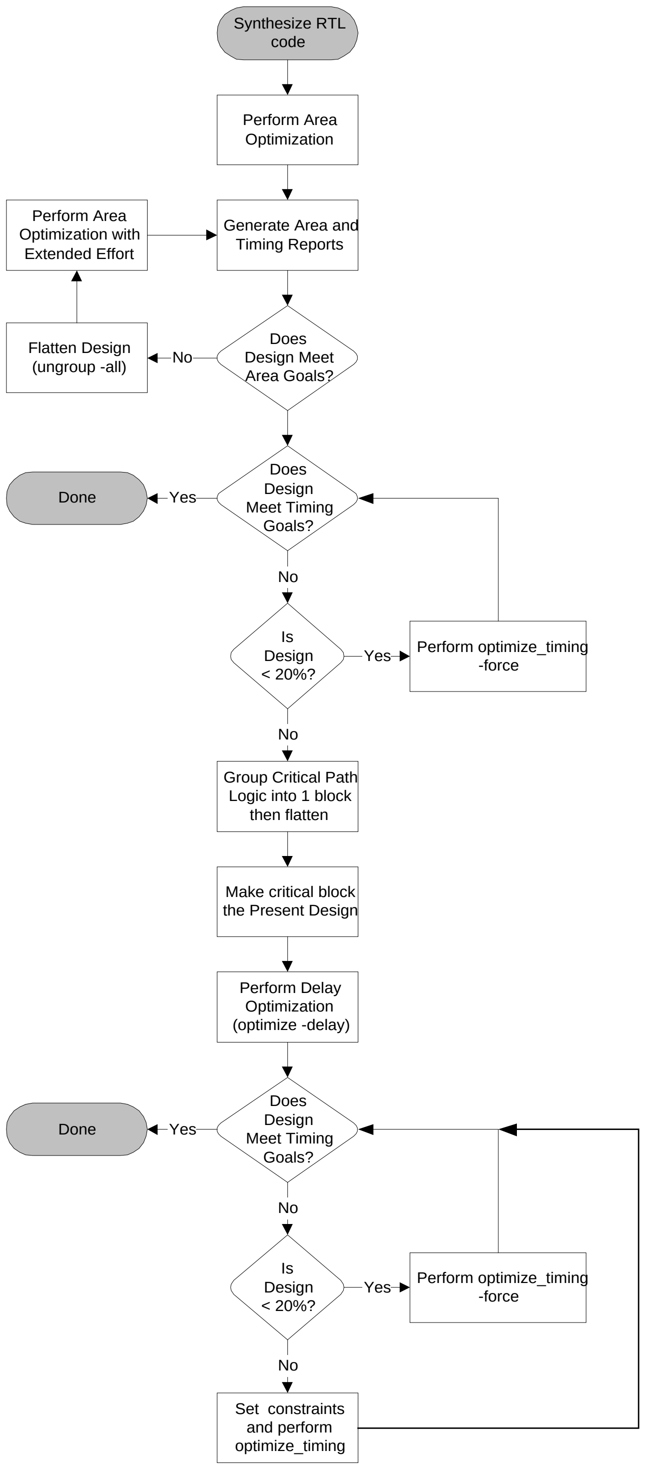
   LEONARDO{7} **required_time 4 B**

4) Apply an output required time to the combinatorial output paths. The maximum delay constraint applied to these paths will be the window created by the difference between the input arrival time and the output-required time. In this example we set an input arrival of 3 and we want to have a maximum delay through the combinatorial path of 7ns therefore the output required time must be 10ns (10ns - 3ns = 7ns).

   LEONARDO{8} **required_time 10 C**

Optimization Strategy

Optimization Flow diagram



Synthesize RTL code
Perform Area Optimization
Perform Area Optimization with Extended Effort
Generate Area and Timing Reports
Flatten Design (ungroup -all)
No
Does Design Meet Area Goals?
Done
Yes
Does Design Meet Timing Goals?
No
Is Design < 20%?
Yes
Perform optimize_timing -force
No
Group Critical Path Logic into 1 block then flatten
Make critical block the Present Design
Perform Delay Optimization (optimize -delay)
Done
Yes
Does Design Meet Timing Goals?
No
Is Design < 20%?
Yes
Perform optimize_timing -force
No
Set constraints and perform optimize_timing

Leonardo can perform two types of optimizations on a design - area and delay. Due to the routing dominated nature of FPGAs (almost 80% of the path delay is due to routing and not cell delays) often the smallest design is the fastest. For this reason, it is recommended that users perform area optimization first - take the design through Alliance Series place and route, identify timing critical areas and focus on just those areas.

Table 4. Arguments to the Optimize command

Argument	Description
-target	Specify the target technology for this design
-single_level	Perform optimization only on top level of hierarchy
-effort	Optimization effort : remap | quick | standard
-nopass <list>	Explicitly avoid an optimization pass
-chip | -macro	"-chip" will insert i/o buffers, for top; "-macro" will not, for sub-blocks
-pass <list>	Explicitly run a pass
-area | -delay	Optimize to obtain minimum delay or area (default)

Area Optimization Strategy

If a design comfortably meets timing and you're trying to achieve the smallest possible circuit then follow the procedure outlined below

1) Flatten hierarchical blocks up to 50K gate "chunks". Use the ***"present_design"*** and ***"ungroup -all -hierarchy"*** commands to do this.

2) Set the area_weight variable to 1 and the delay_weight variable to 0. This will redefine the optimization cost function to favor area over delay.

   LEONARDO{5} **set area_weight 1**

   LEONARDO{6} **set delay_weight 0**

3) Perform a standard effort area optimization

   LEONARDO{7} **optimize -ta xi4e -area -effort standard**

Timing Critical Optimization Strategy

Generally designs have a combination of critical blocks and non-critical blocks. The goal of timing optimization is to create the smallest design possible that meets timing. Leonardo has 2 timing optimization commands; ***optimize -delay*** which creates fast structures during the mapping process and runs algorithms designed to reduce levels of logic and ***optimize_timing*** which performs full constraint-based timing optimization.

1) Perform an area optimization with quick effort and hierarchy preserved. We will use the results of this optimization to identify our critical blocks

   LEONARDO{8} **optimize -ta xi4e -area -effort quick**

2) Generate a timing report and use this report to identify the timing critical blocks.

3) Use the ***"group"*** command to combine all critical blocks into 1 block. Once combined, use the ***"present_design"*** and ***"ungroup -all -hierarchy"*** commands to dissolve hierarchy within that block

   LEONARDO{9} **group a b -inst_name ab_instance**

   LEONARDO{9} **present_design work.ab_instance**

   LEONARDO{10} **ungroup -all -hierarchy**

4) Perform a delay optimization with standard effort on the timing critical sub block and generate a timing report

5) If timing is still not met, try performing a second standard effort delay optimization. Continue until the results no longer change.

6) If timing is still not met, set timing constraints and perform an optimize_timing command. This invokes a second optimization engine that performs constraint based timing optimization.

7) Once timing has been met in the timing critical sub-block, set the top_level instance to be the present design. Verify that the top-level design meets timing. If not, repeat the above process on other timing critical blocks. If all blocks meet timing than continue

8) Group all non-critical blocks into a single level of hierarchy and follow the procedure outlined in ***"Area Critical Optimization".***

Hint: *The optimize_timing command has a -force option which will cause Leonardo to perform a static timing analysis, subtract 20% from the critical path, set constraints then perform timing optimization. If an unconstrained timing analysis selects the correct critical path, this is the easiest way to perform timing optimization.*

Note: *If you wish to perform timing optimization on a hierarchical design, the -force option is the only way to do this today. Normal timing optimization does not support hierarchical designs in the current version of Leonardo.*

Table 5. Arguments to the Optimize_timing command

Argument	Description
-through <list>	Specify explicit list of end points to optimize
-single_level	Perform optimization only on top level of hierarchy
-force	Force timing constraints on longest path

Example:

LEONARDO{6} **optimize_timing -through data_out_port**

Xilinx Optimization

Decompose LUTs

EDIF netlists written from Leonardo to Alliance Series contain mapped LUTs. Users may decompose LUTs by issuing the command, ***"decompose_luts".***

> **Note:** *Experience has shown that decomposing LUTs generally degrades area and performance. For this reason It is recommended that users do not decompose LUTs*

Generate Timespecs

Users must issue the command ***generate_timespec*** for Leonardo to convert timing constraint data from Leonardo into Xilinx timespec information. This information is included within the generated EDIF netlist as properties. Only clock information is passed to Alliance Series at this time. Any input arrival and output required time constraints are ignored. Users may wish to augment the timespec information from Leonardo with an additional Xilinx UCF file.

> **Note**: *It is recommended that users, who wish to constrain register performance, do so through Leonardo generated timespec information. This will insure that register names match timespec constraints every time.*
>
> **Note:** *False path and multi-cycle path constraints are not passed to place and route through timespec information generated by Leonardo.*

Pack CLBs

Leonardo automatically packs CLBs when generating area reports. Users may wish to preserve this packing and impose it on the Alliance Series placement software by issuing the command ***"pack_clbs"***.

> **Note:** *Experience has shown that Xilinx does a better job of CLB packing than Exemplar. For this reason, it is recommended that users do not pack CLBs in Leonardo.*

Assigning Pin Numbers

Device pin assignments can be made 3 ways

1) Use of the VHDL "attribute" ***pin_number***
2) Use of the "set_attribute" command in Leonardo

   LEONARDO{5} **set_attribute -port enable -name pin_number -value P10**

3) In a Xilinx UCF file

It is recommended option #2 be used. VHDL or Verilog code should remain as technology independent as possible to facilitate design re-use. Although the command is verbose it can easily be scripted.

Place and Route with Alliance Series

Here are some suggestions for compilation flows and strategies when implementing large designs in Alliance Series so that you can avoid excessively long runtimes while still realizing the bulk of a design's performance potential.

Some strategies that can be used to more efficiently place and route designs are:

1) Non Timing-Driven Placement and Routing, + Delay-based Cleanup Routing Pass
2) Non Timing-Driven Placement + "Restricted" Timing Driven Routing
3) Timing Driven Routing, Timing Driven Placement + "Restricted" Timing Driven Routing

These strategies are ordered to progress from quickest runtime (good for initial design evaluation and non-timing critical designs) towards better design performance results (for those designs with timespecs that are increasingly tougher to meet.)

Getting Quick Results

Non-Timing Driven Placement & Routing + Delay-based Cleanup Pass. This strategy will typically produce circuit performance that is anywhere from 55% to 75% of that which could be ultimately obtained with a "full-out" timing-driven run, with a runtime that is usually many times faster. To run PAR in this manner from the command line, use:

par -x -d 1 <design.ncd> <design_r.ncd>

When running non-timing driven PAR, it is important to also run at least one pass of a cleanup router - preferably the Delay-Based cleanup router (hence the –d 1 in the above command line). This will work to minimize route delays.

From the Flow Engine GUI, the same thing can be accomplished by de-selecting the "Use Timing Constraints During Place and Route" check-box, and setting "Run 1 Delay-based Cleanup Passes" under the Place and Route tab of the Implementation Options dialog box.

The Balance between Run Times and Circuit Performance

Non-Timing Driven Placement + "Restricted" Timing Driven Routing This strategy will typically produce a result that has significantly better circuit performance than a fully non-timing driven run (Strategy 1), but have faster runtime than a fully timing driven run. To do non-timing driven placement, use the command line:

par -x -r <design.ncd> <design_r.ncd>

From the GUI, the same thing can be accomplished by specifying the par -r option from template customization form in the Design Manager (***use Utilities -> Template Manager -> Customize*** to access this form), and de-selecting ***the "Use Timing Constraints During Place and Route"*** check-box in the PAR implementation options dialog box.

The next step would be to run a re-entrant routing phase in timing-driven mode to route the design. Testing has shown that 75% to 85% of a given design's ultimate circuit performance is achieved relatively early in the PAR routing process. Therefore it is recommended that you initially limit the number of router iterations to three or four passes.

To run the timing-driven re-entrant routing phase with one delay-based cleanup pass, use the command line:

par -k -i 3 -d 1 <design_r.ncd> <design_r.ncd>

From the Flow Engine GUI select the ***Setup -> Advanced*** pulldown menus. Select the ***"Allow Re-entrant Routing"*** check-box, set ***"Run 3 Re-entrant Route Passes"***, set ***"Run 1 Delay-based Cleanup Passes"***, and check the box labeled ***"Use Timespecs During Re-entrant Place and Route".***

Getting the Fastest Circuits

Timing Driven Placement + "Restricted" Routing. By invoking the timing analysis engine during the placement phase, a better result will be produced than would be obtained using timing-driven routing alone. Again, we limit the runtime by explicitly setting number of attempted router iterations to a relatively low count (three or four.) For timing driven placement, with a limited number of router iterations and one delay-based cleanup pass, the command line usage is:

par -i 3 –d 1 <design.ncd> <design_r.ncd> <design.pcf>

From the Flow Engine GUI, the same thing can be accomplished by selecting ***the "Use Timing Constraints during Place and Route"*** check-box and ***setting "Run 3 Router Iterations"*** under the Place and Route tab of the Implementation Options dialog box.

As with the earlier strategies, further re-entrant routing passes can be used to obtain incremental gains in performance.

Tips for Obtaining Faster Circuits

The above strategies are geared towards minimizing compilation runtimes, while still achieving the bulk of a design's performance potential. If an incremental gain (e.g. 3% to 10%) in circuit performance is required after initially using one of the above strategies, one following PAR usage techniques can be used.

1) Run one or two more delay-based cleanup passes if running non-timing driven PAR (Strategy 1)
2) Run more iterations of the re-entrant route process for the timing-driven runs (Strategies 2, 3).
3) Use higher effort levels in placement and routing

Reduce Levels of Logic:

Often times it is best to change the source design when significant gains in circuit performance are required. Reducing the numbers of levels of logic between synchronous elements will most effectively increase circuit performance - as well as often resulting in a faster runtimes. Performing more aggressive timing optimizations with Leonardo can often remove the necessary levels of logic. Refer the section titled, "Timing critical optimization strategy"

Mulit-Pass Place and Route

As always, the Multi-Pass Place & Route (MPPR) capability can be used with any of the above three strategies if still more performance is required. Testing has shown 15% to 20% differences in ultimate performance using different placement seeds/cost tables. Typically, running MPPR is a task that's best suited for overnight or weekend runs.

Understand both Design and Device.

Though the XL family is quite good at maintaining circuit performance throughout the entire device range, the large (e.g. 4085XL) devices, because they are bigger and have correspondingly longer routing segments, can exhibit longer delays than would be found in a smaller (e.g. 4020XL) device.

With the large FPGA devices, it is much more important to take advantage of running pre-PAR timing analysis runs

Routing vs. Block Delay Estimation

Many designers have traditionally used the "50/50" measure as a rough rule of thumb to evaluate pre-route timing reports. Here, the designer will allocate roughly 50% of the timing budget to the logic blocks (LUTs, TBUFs, IOBs, etc) and the rule was that the remaining 50% should be sufficient to account for general routing delays. Of course, this is only a rough estimate; carry chains have much faster routing, high-fanout nets can cause routing delays to become more dominant. In reality, the 50/50 rule is perhaps more realistically a "30/70 - 70/30" range.

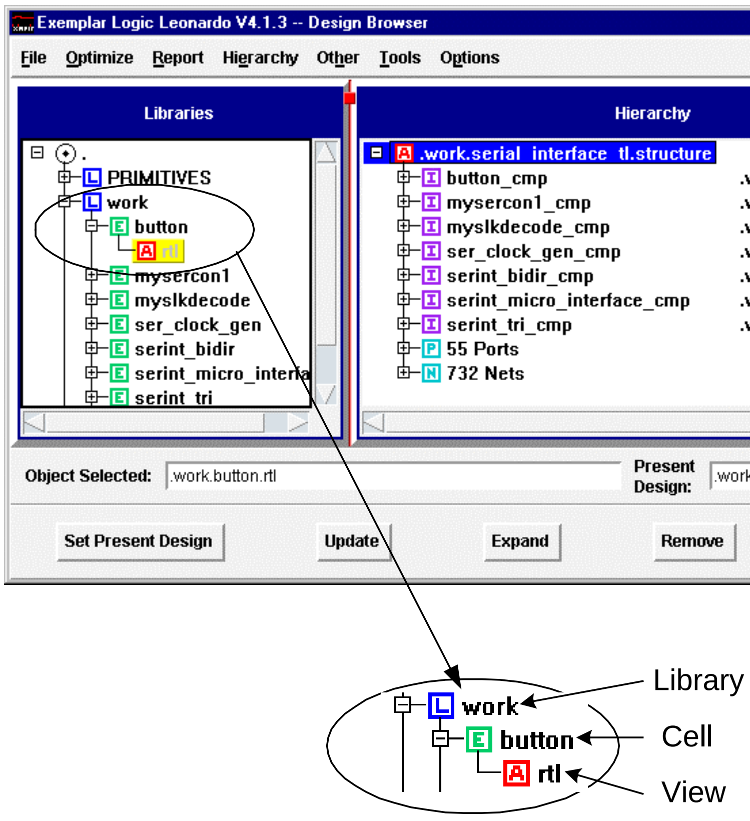
When designing with the new, large FPGAs, one needs to be even more aware that the "50/50" rule is probably too simple of a model on which to base a timing budget. As designs target increasingly larger devices, it is realistic to expect that more delays will fall towards the extremes of the broader "30/70 - 70/30" range.

Appendix A - Referencing Netlist Objects

Several design manipulation commands reference "design objects". The Leonardo database is actually quite simple but understanding the elements will help use these commands effectively

Library Objects

The design browser contains 2 columns of data, the Libraries and Hierarchy. The Libraries column displays all the design objects in memory including "Libraries, Cells and Views.. Any view of a cell may be selected and made the "present design". Once a design has become the present design the "Hierarchy" column displays all hierarchical design information about that design including "Instances", "Nets", "Ports" and "Technology Cells".



Exemplar Logic Leonardo V4.1.3 -- Design Browser
File Optimize Report Hierarchy Other Tools Options Help
Libraries
Hierarchy
Set Present Design
Update
Expand
Remove
Dismiss
work
button
rtl
Library
Cell
View

Libraries:

```
.
  PRIMITIVES
  work
    button
      rtl
    mysercon1
    myslkdecode
    ser_clock_gen
    serint_bidir
    serint_micro_interfa
    serint_tri
```

Hierarchy:

```
.work.serial_interface_tl.structure
  button_cmp                  .work.button.rtl
  mysercon1_cmp               .work.mysercon1.rtl
  myslkdecode_cmp             .work.myslkdecode.rtl
  ser_clock_gen_cmp           .work.ser_clock_gen.rtl
  serint_bidir_cmp            .work.serint_bidir.rtl
  serint_micro_interface_cmp  .work.serint_micro_inte
  serint_tri_cmp              .work.serint_tri.rtl
  55 Ports
  732 Nets
```

Object Selected: .work.button.rtl

Present Design: .work.serial_interface_tl.structur

Library

All design objects (VHDL entities or Verilog Modules) must reside in a library. If no library is specified then the library "work" is used. If a VHDL library references a specific library then that library is used. In the above example the library name is "work"

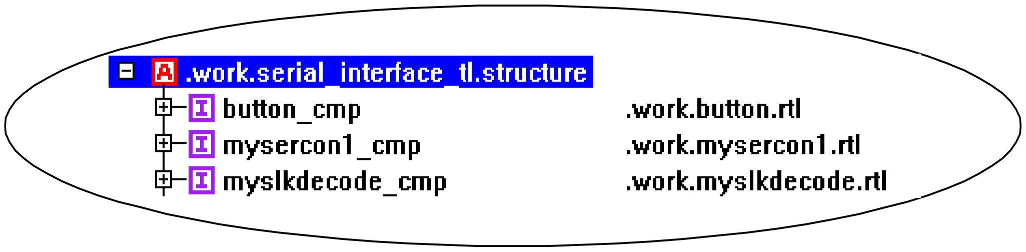
Cells

A cell is basically the instance name that is derived from the VHDL entity name or the Verilog module name. In the above example the cellname is **"button"** and would be referenced by its complete pathname **"work.button"**

View

A "view" is a netlist. In VHDL the viewname is derived from the VHDL architecture name. Because Verilog has no equivalent concept the viewname "INTERFACE" is always used. A cell may have multiple views similar to the way a VHDL entity can have multiple architectures. In the above example the viewname is **"rtl"** and would be referenced by its complete pathname **"work.button.rtl"**

Hierarchy Objects



```
⊟ A .work.serial_interface_tl.structure
  ⊞ I button_cmp                    .work.button.rtl
  ⊞ I mysercon1_cmp                 .work.mysercon1.rtl
  ⊞ I myslkdecode_cmp               .work.myslkdecode.rtl
```

Instances

Instances are design objects attached to the "view" (netlist) of design and represent hierarchical sub-blocks. A selected instance may be grouped or ungrouped. In the above example the instance name is **"button_cmp"** and would be referenced by its complete pathname **"work.serial_interface_tl.structure.button_cmp"**

Ports

Ports are design objects attached to a "view" (netlist) of a design and represent the primary I/O of a block. Timing constraints are attached to ports. In the above example the ports can be referenced by **"work.serial_interface_tl.structure.portname"**

Nets

Nets are design objects attached to a "view" (netlist) of a design and represent interconnect between instances. In the above example the ports can be referenced by **"work.serial_interface_tl.structure.netname"**

Table 1. Arguments to the List_design command

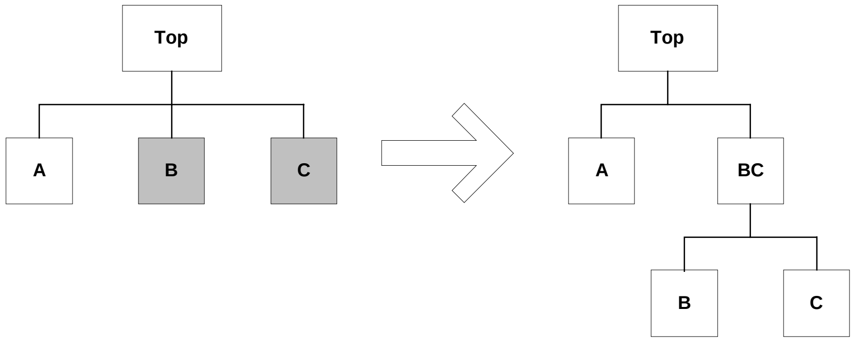
Argument	Description
-ports | -nets | -instances | -references	Lists specified elements of the present_design
-direction <string>	For ports only; Directions in|out|inout
-hdl	Instance name of the group
-short	Generate "short" list of design objects

Appendix B - Hierarchy Manipulation

Leonardo can perform all common hierarchy manipulations on design instances including grouping, ungrouping and unfolding.

Grouping

Hierarchical grouping refers to the creation of a new hierarchical block from 2 or more selected instances. The original blocks still exist after grouping beneath the new level of hierarchy.



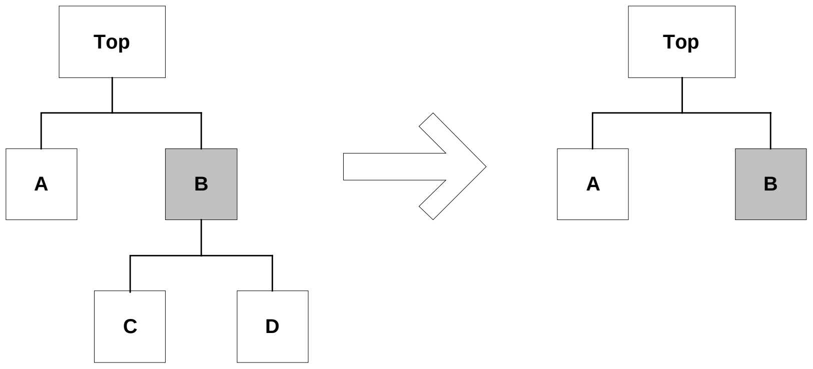
LEONARDO{5} **group B C -inst_name BC**

Table 1. Arguments to the Group command

Argument	Description
-cell_name	Group a list of instances into one instance of a new view
-view_name	View name of the group
-inst_name	Instance name of the group
-except <list>	Exclude these cells from grouping

Ungrouping

Hierarchical ungrouping refers to the dissolving of all hierarchy below a selected instance



LEONARDO{6} **ungroup B**

Note: *The* ***ungroup*** *command will dissolve 1 level of hierarchy. Use the* ***"-all -hierarchy"*** *switchs to dissolve all hierarchy beneath a specified block of hierarchy*

Table 2. Arguments to the Ungroup command

Argument	Description
-all	Ungroups all instances
-hierarchy	Recursively ungroup all hierarchy levels under the selected instance
-simple_names	Use original simple instance names for ungrouped instances
-except <list>	Don't ungroup these instances

Appendix C - Command Reference

Table 2. Leonardo Command Reference

Command	Options
Add_rename_rule <name>	-type <string> -find_word <string> -find_substring <string> -find_character <string> -find_first_character <string> -find_last_character <string> -replace <string> -prepend_word <string> -append_word <string> -escape_word <string>
Alias [alias_name {script_expansion}]	
Analyze <file_name>	-work -format
Apply_rename_rules <design>	-ruleset <string> -single_level -test
Balance_loads <design>	-single_level
Connect	-port <string> -of <string>
Connect_path <port_names>	-instance <string> | -gate <string> | -all
Copy <object> <object> | <object> <object container>	
Create <object_name>	-port | -net | -instance | -dir <string> -of <string>
Create_rename_ruleset <name>	-no_collision_between <list> -case_insensitive
Create_wrapper <entity name>	-architecture -work <string> -file <string> -wrap_name <string> -parameters <list> | -generics <list>
Decompose_luts <design>	-group_luts -single_level
Disconnect	-port <string> -net <string>
Disconnect_path <port_names>	-instance <string> | -gate <string> | -all
elaborate	-architecture -work -parameters <list> -generitcs <list>
Generate_timespec <design>	-single_level
Group <list of instances>	-cell_name <string> -view_name <string> -inst_name <string>
Help <search_string>	-variables

List_attributes <list_of_objects>	-port | -net | -instance
List_connection <list_of_objects>	-port | -net | instance -hierarchical
List_design <list_of_designs>	-ports | -nets | -instances | -references -direction <string> -hdl -short
List_technologies	-single_level
Load_library <technology>	
Load_modgen <library name>	
Move <object> <object>	-port | -net | -instance
Optimize <design>	-target <string> -single_level -effort <string> -chip | -macro -area | -delay -pass <list> | -nopass <list>
Optimze_timing <design>	-through <list> -force -single_level
Pack_clbs <design>	-single_level
Present_design <new_present_design>	
Puts_log <string>	-nonewline
Read <file_name>	-format -don't_elaborate -work
Remove <object_name>	-port | -net | -instance | -hdl
Remove_attribute <object_list>	-port | -net | -instance -name
Remove_rename_ruleset <ruleset_name>	
Report_area <file_name>	-cell_usage -hierarchy -all_leafs
Report_constraints <design>	-port -net -hierarchy
Report_delay <file_name>	-num_paths -longest_paths <integer> -no_io_terminals -show_input_pins -show_nets -through <list> -from <list> -to <list> -not_through <list> -highlight_file <list>
Report_rename_rules <name>	

set_attribute <object_list>	-name <string> -value < string> -type <string> -port | -net | -instance
Unalias <alias_name>	
Unfold <design>	
Ungroup <instance_list>	-all -hierarchy -simple_names -execpt <list>
Write	-format <string> -bottom_library <string> -silent -single_level

Appendix D - Variable Reference

Table 6. Leonardo Variable Reference

Variable	Default	Variable	Default
annotate_packing	true	optimize_cpu_limit	0
area_weight	1.000000	optimize_timing_cpu_limit	0
check_complex_ios	true	optimize_timing_num_paths	5
complex_ios	true	package	not set
delay_break_loops	false	parallel_case	false
delay_weight	1.000000	part	not set
delete_startup	false	preserve_dangling_net	false
dont_lock_lcells	false	preserve_z	false
edif_function_property	eqn	process	not set
edif_write_internal_properties	false	propagate_clock_delay	false
enable_dff_map	false	report_area_format_style	%6.0f
encoding	binary	report_delay_analysis_mode	maximum
exclude_gates	not set	report_delay_arrival_threshold	0.000000
extract_cin_cout	true	report_delay_detail	full
extract_counter	true	report_delay_format_style	%4.2f
extract_decoder	true	report_delay_slack_threshold	0.000000
extract_ram	true	resolve_mux_stat	not set
flex_use_cascades	true	sdf_hier_separator	/
force_user_load_values	false	sdf_hierarchical_names	true
full_case	false	sdf_names_style	vhdl
gate_sizing	true	sdf_type	maximum
global_sr	Not set	temp	not set
hdl_array_name_style	%s(%d)	transformations	false
hdl_integer_name_style	%s(%d)	tristate_map	false
hdl_record_name_style	%s.%s	ungroup_hier_separator	_
include_gates	not set	use_dffenable	true
infer_gsr	false	u se_f5map	false
insert_global_bufs	true	use_f6_lut	false
list_design_object_separator	.	use_qclk_bufs	false
load_library_file_extension	syn	verilog_parameter_to_attribute	true
lut_cell_name	lut_cell	vhdl_87	false
lut_map	true	vhdl_write_87	false
max_cap_load	0.000000	vhdl_write_bit	std_logic
max_fanin	0	voltage	not set
max_fanout_load	0.000000	wire_table	not set
max_pt	0	wire_tree	not set
max_transition	0.000000	xlx_preserve_gsr	false
modgen_select	auto	xlx_preserve_gts	false
names_collision_extension	_rename	xlx_preserve_pins	true
nl_use_approx	true	xnf_write_clb_packing	true
nologic_rep	false	xnf_write_lut_binding	true