Synthesis and Simulation Design Guide

8.1i

About This Guide

This guide provides a general overview of designing Field Programmable Gate Arrays (FPGA devices) with Hardware Description Languages (HDLs). It includes design hints for the novice HDL user, as well as for the experienced user who is designing FPGA devices for the first time.

The design examples in this guide were:

- created with Verilog and VHSIC Hardware Description Language (VHDL)
- compiled with various synthesis tools
- targeted for Spartan™-II, Spartan-IIE, Spartan-3, Spartan-3E, Virtex™, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X and Virtex-4 devices

Xilinx® equally endorses both Verilog and VHDL. VHDL may be more difficult to learn than Verilog, and usually requires more explanation.

This guide does not address certain topics that are important when creating HDL designs, such as the design environment; verification techniques; constraining in the synthesis tool; test considerations; and system verification. For more information, see your synthesis tool documentation and design methodology notes.

Before using this guide, you should be familiar with the operations that are common to all Xilinx software tools.

Guide Contents

This guide contains the following chapters.

- Chapter 1, "Introduction," provides a general overview of designing Field Programmable Gate Arrays (FPGA devices) with HDLs. This chapter also includes installation requirements and instructions.
- Chapter 2, "Understanding High-Density Design Flow," provides synthesis and Xilinx implementation techniques to increase design performance and utilization.
- Chapter 3, "General HDL Coding Styles," includes HDL coding hints and design examples to help you develop an efficient coding style.
- Chapter 4, "Coding Styles for FPGA Devices," includes coding techniques to help you use the latest Xilinx FPGA devices.
- Chapter 5 "Using SmartModels," describes the special considerations encountered when simulating designs for Virtex-II Pro and Virtex-II Pro X FPGA devices.
- Chapter 6, "Simulating Your Design," describes simulation methods for verifying the function and timing of your designs.
- Chapter 7, "Equivalency Checking." Information on equivalency checking is no longer included in the *Synthesis and Simulation Design Guide*. For information on

running formal verification with Xilinx devices, see
http://www.xilinx.com/xlnx/xil_tt_product.jsp?BV_UseBVCookie=yes&sProduct=formal

Additional Resources

To find additional documentation, see the Xilinx website at:

http://www.xilinx.com/literature.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

http://www.xilinx.com/support.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *`design_name`*
Helvetica bold	Commands that you select from a menu	**File →Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *`design_name`*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** `[`*`option_name`*`]` *`design_name`*
Braces { }	A list of items from which you must choose one or more	**`lowpwr`** **`=`**`{`**`on`**`|`**`off`**`}`

Convention	Meaning or Use	Example
Vertical bar |	Separates items in a list of choices	**`lowpwr`** **`={on|off}`**

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis . . .	Repetitive material that has been omitted	**`allow block`** *`block_name`*
		`loc1 loc2 ... locn;`

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current file or in another file in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details..
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-II Platform FPGA User Guide.*
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

Table of Contents

Chapter 4: Coding Styles for FPGA Devices

XILINX®

Chapter 1

Introduction

This chapter provides a general overview of designing Field Programmable Gate Arrays (FPGA devices) with Hardware Description Languages (HDLs). It includes the following sections.

- "Device Support"
- "Hardware Description Languages"
- "Advantages of Using HDLs to Design FPGA Devices"
- "Designing FPGA Devices with HDLs"

Device Support

The ISE software supports the following devices:

- Virtex™
- Virtex-II
- Virtex-E
- Virtex-II PRO
- Virtex-II PRO X
- Virtex-4 (SX/LX/FX)
- Spartan™
- Spartan-II
- Spartan-IIE
- Spartan-3
- Spartan-3E
- CoolRunner™ XPLA3
- CoolRunner-II
- XC9500™ (XL/XV)

Hardware Description Languages

Designers use Hardware Description Languages (HDLs) to describe the behavior and structure of system and circuit designs. This chapter includes:

- A general overview of designing FPGA devices with HDLs.
- System requirements and installation instructions for designs available from the web.
- A brief description of why FPGA devices are superior to ASICs for your design needs.

Understanding FPGA architecture allows you to create HDL code that effectively uses FPGA system features. To learn more about designing FPGA devices with HDL:

- Enroll in training classes offered by Xilinx and by the vendors of synthesis software.
- Review the sample HDL designs in the later chapters of this guide.
- Download design examples from Xilinx Support.
- Take advantage of the many other resources offered by Xilinx, including documentation, tutorials, Tech Tips, service packs, a telephone hotline, and an answers database. See "Additional Resources" in the Preface of this guide.

Advantages of Using HDLs to Design FPGA Devices

Using HDLs to design high-density FPGA devices has the following advantages:

- "Top-Down Approach for Large Projects"
- "Functional Simulation Early in the Design Flow"
- "Synthesis of HDL Code to Gates"
- "Early Testing of Various Design Implementations"
- "Reuse of RTL Code"

Top-Down Approach for Large Projects

Designers use to create complex designs. The top-down approach to system design supported by HDLs is advantageous for large projects that require many designers working together. After they determine the overall design plan, designers can work independently on separate sections of the code.

Functional Simulation Early in the Design Flow

You can verify the functionality of your design early in the design flow by simulating the HDL description. Testing your design decisions before the design is implemented at the RTL or gate level allows you to make any necessary changes early in the design process.

Synthesis of HDL Code to Gates

You can synthesize your hardware description to target the FPGA implementation. This step:

- Decreases design time by allowing a higher-level specification of the design rather than specifying the design from the FPGA base elements.
- Generally reduces the number of errors that can occur during a manual translation of a hardware description to a schematic design.
- Allows you to apply the automation techniques used by the synthesis tool (such as machine encoding styles and automatic I/O insertion) during the optimization of your design to the original HDL code. This results in greater optimization and efficiency.

Early Testing of Various Design Implementations

HDLs allow you to test different implementations of your design early in the design flow. Use the synthesis tool to perform the logic synthesis and optimization into gates.

Additionally, Xilinx FPGA devices allow you to implement your design at your computer. Since the synthesis time is short, you have more time to explore different architectural possibilities at the Register Transfer Level (RTL). You can reprogram Xilinx FPGA devices to test several implementations of your design.

Reuse of RTL Code

You can retarget RTL code to new FPGA architectures with a minimum of recoding.

Designing FPGA Devices with HDLs

If you are used to schematic design entry, you may find it difficult at first to create HDL designs. You must make the transition from graphical concepts, such as block diagrams, state machines, flow diagrams, and truth tables, to abstract representations of design components. Ease this transition by not losing sight of your overall design plan as you code in HDL.

To effectively use an HDL, you must understand:

- the syntax of the language
- the synthesis and simulator software
- the architecture of your target device
- the implementation tools

This section gives you some design hints to help you create FPGA devices with HDLs.

Designing FPGA Devices with Verilog

Verilog is popular for synthesis designs because:

- Verilog is less verbose than traditional VHDL.
- Verilog is standardized as IEEE-STD-1364-95 and IEEE-STD-1364-2001.

Since Verilog was not originally intended as an input to synthesis, many Verilog constructs are not supported by synthesis software. The Verilog examples in this guide were tested and synthesized with current, commonly-used FPGA synthesis software. The coding strategies presented in the remaining chapters of this guide can help you create HDL descriptions that can be synthesized.

SystemVerilog is a new emerging standard for both synthesis and simulation. It is currently unknown if, or when, this standard will be adopted and supported by the various design tools.

Whether or not you plan to use this new standard, Xilinx recommends that you:

- Review the standard to make sure that your current Verilog code can be readily carried forward as the new standard evolves.
- Review any new keywords specified by the standard.
- Avoid using the new keywords in your current Verilog code.

Designing FPGA Devices with VHDL

VHSIC Hardware Description Language (VHDL) is a hardware description language for designing Integrated Circuits (ICs). It was not originally intended as an input to synthesis, and many VHDL constructs are not supported by synthesis software. However, the high level of abstraction of VHDL makes it easy to describe the system-level components and test benches that are not synthesized. In addition, the various synthesis tools use different subsets of the VHDL language. The examples in this guide work with most commonly used FPGA synthesis software. The coding strategies presented in the remaining chapters of this guide can help you create HDL descriptions that can be synthesized.

Designing FPGA Devices with Synthesis Tools

Most of the commonly-used FPGA synthesis tools have special optimization algorithms for Xilinx FPGA devices. Constraints and compiling options perform differently depending on the target device. Some commands and constraints in ASIC synthesis tools do not apply to FPGA devices. If you use them, they may adversely impact your results.

You should understand how your synthesis tool processes designs before you create FPGA designs. Most FPGA synthesis vendors include information in their guides specifically for Xilinx FPGA devices.

Using FPGA System Features

To improve device performance, area utilization, and power characteristics, create HDL code that uses such FPGA system features as DCM, multipliers, shift registers, and memory. For a description of these and other features, see the FPGA data sheet and user guide. The choice of the size (width and depth) and functional characteristics need to be taken into account by understanding the target FPGA resources and making the proper system choices to best target the underlying architecture.

Designing Hierarchy

Hardware Description Languages (HDLs) give added flexibility in describing the design. However, not all HDL code is optimized the same. How and where the functionality is described can have dramatic effects on end optimization. For example:

- Certain techniques may unnecessarily increase the design size and power while decreasing performance.
- Other techniques can result in more optimal designs in terms of any or all of those same metrics.

This guide will help instruct you in techniques for optional FPGA design methodologies.

Design hierarchy is important in both the implementation of an FPGA and during incremental or interactive changes. Some synthesizers maintain the hierarchical boundaries unless you group modules together. Modules should have registered outputs so their boundaries are not an impediment to optimization. Otherwise, modules should be as large as possible within the limitations of your synthesis tool.

The “5,000 gates per module” rule is no longer valid, and can interfere with optimization. Check with your synthesis vendor for the preferred module size. As a last resort, use the grouping commands of your synthesizer, if available. The size and content of the modules influence synthesis results and design implementation. This guide describes how to create effective design hierarchy.

Specifying Speed Requirements

To meet timing requirements, you should understand how to set timing constraints in both the synthesis tool and the placement and routing tool. If you specify the desired timing at the beginning, the tools can maximize not only performance, but also area, power, and tool runtime. This generally results in a design that better matches the desired performance. It may also result in a design that is smaller, and which consumes less power and requires less time processing in the tools. For more information, see "Setting Constraints" in Chapter 2 of this guide.

Chapter 2

Understanding High-Density Design Flow

This chapter describes the steps in a typical HDL design flow. Although these steps may vary with each design, the information in this chapter is a good starting point for any design. This chapter includes the following sections.

Design Flow

The following figure shows an overview of the design flow steps.

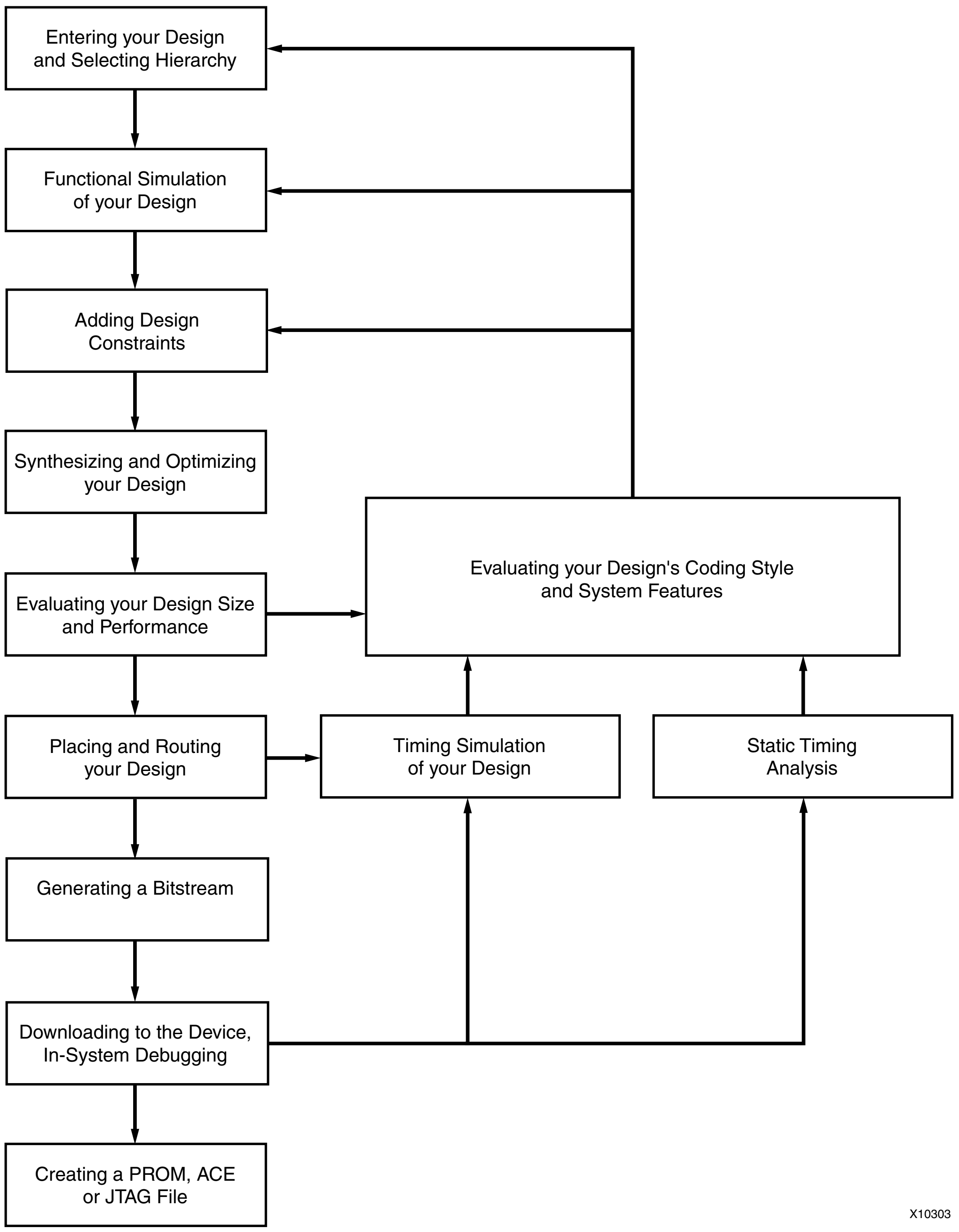
Entering your Design and Selecting Hierarchy
Functional Simulation of your Design
Adding Design Constraints
Synthesizing and Optimizing your Design
Evaluating your Design Size and Performance
Placing and Routing your Design
Generating a Bitstream
Downloading to the Device, In-System Debugging
Creating a PROM, ACE or JTAG File
Evaluating your Design's Coding Style and System Features
Timing Simulation of your Design
Static Timing Analysis
X10303

Figure 2-1: **Design Flow Overview**

Entering Your Design and Selecting Hierarchy

The first step in implementing your design is to create the HDL code based on your design criteria.

Design Entry Recommendations

The following recommendations can help you create effective designs.

Use RTL Code

Use register transfer level (RTL) code, and, when possible, do not instantiate specific components. Following these two practices allows for:

- Readable code
- Ability to use the same code for synthesis and simulation
- Faster and simpler simulation
- Portable code for migration to different device families
- Reusable code for future designs

Note: In some cases instantiating optimized CORE Generator™ modules is beneficial with RTL.

Select the Correct Design Hierarchy

Selecting the correct design hierarchy:

- Improves simulation and synthesis results
- Improves debugging and modifying modular designs
- Allows parallel engineering, in which a team of engineers can work on different parts of the design at the same time
- Improves the placement and routing by reducing routing congestion and improving timing
- Allows for easier code reuse in the current design, as well as in future designs

Architecture Wizard

The Architecture Wizard in Project Navigator lets you configure complicated aspects of some Xilinx® devices. The Architecture Wizard consists of several components for configuring specific device features. Each component is presented as an independent wizard. See "Architecture Wizard Components" below.

The Architecture Wizard can also produce a VHDL, Verilog, or EDIF file, depending on the flow type that is passed to it. The generated HDL output is a module consisting of one or more primitives and the corresponding properties, and not just a code snippet. This allows the output file to be referenced from the HDL Editor. There is no UCF output file, since the necessary attributes are embedded inside the HDL file.

Opening Architecture Wizard

There are two ways to open the Architecture Wizard, from Project Navigator and from the command line.

Opening Architecture Wizard from Project Navigator

To open Architecture Wizard from Project Naviator:

1. Select **Project →New Source.**
2. Select **IP (Coregen & Architecture Wizard)** from the **New Source** window.

Opening Architecture Wizard from the Command Line

To open Architecture Wizard from the command line, type **`arwz`**.

Architecture Wizard Components

The following wizards make up the Architecture Wizard:

- "Clocking Wizard"
- "RocketIO Wizard"
- "ChipSync Wizard"
- "XtremeDSP Slice Wizard"

Clocking Wizard

The Clocking Wizard enables:

- digital clock setup
- DCM and clock buffer viewing
- DRC checking

The Clocking Wizard allows you to:

- view the DCM component
- specify attributes
- generate corresponding components and signals
- execute DRC checks
- display up to eight clock buffers
- set up the Feedback Path information
- set up the Clock Frequency Generator information and execute DRC checks
- view and edit component attributes
- view and edit component constraints
- automatically place one component in the XAW file
- save component settings in a VHDL file
- save component settings in a Verilog file

RocketIO Wizard

The RocketIO Wizard enables serial connectivity between devices, backplanes, and subsystems.

The RocketIO Wizard allows you to:

- specify RocketIO type
- define Channel Bonding options
- specify General Transmitter Settings, including encoding, CRC, and clock

- specify General Receptor Settings, including encoding, CRC, and clock
- provide the ability to specify Synchronization
- specify Equalization, Signal integrity tip (resister, termination mode...)
- view and edit component attributes.
- view and edit component constraints
- automatically place one component in the XAW file
- save component settings to a VHDL file
- save component settings to a Verilog file

ChipSync Wizard

The ChipSync Wizard applies to Virtex-4 only.

The ChipSync Wizard facilitates the implementation of high-speed source synchronous applications. The wizard configures a group of I/O blocks into an interface for use in memory, networking, or any other type of bus interface. The ChipSync Wizard creates HDL code with these features configured according to your input:

- Width and IO standard of data, address, and clocks for the interface
- Additional pins such as reference clocks and control pins
- Adjustable input delay for data and clock pins
- Clock buffers (BUFIO) for input clocks
- ISERDES/OSERDES or IDDR/ODDR blocks to control the width of data, clock enables, and 3-state signals to the fabric

XtremeDSP Slice Wizard

The XtremeDSP Slice Wizard applies to Virtex-4 only. The XtremeDSP Slice Wizard facilitates the implementation of the XtremeDSP Slice. For more information, see the *Virtex-4 Data Sheet* and the *Virtex-4 DSP Design Considerations User Guide*.

CORE Generator

CORE Generator delivers parameterized cores optimized for Xilinx FPGA devices. It provides a catalog of ready-made functions ranging in complexity from simple arithmetic operators such as adders, accumulators, and multipliers, to system-level building blocks such as filters, transforms, FIFOs, and memories.

CORE Generator Templates

When ISE™ generates cores, it adds an instantiation template to the Language Templates. To access the core template:

1. Select **Edit** →**Language Templates** from Project Navigator.
2. Select COREGEN in the Templates window.

CORE Generator Files

For each core it generates, CORE Generator produces the following files:

- "EDN and NGC Files"
- "VHO Files"

- "VEO Files"
- "V and VHD Wrapper Files"

EDN and NGC Files

The Electronic Data Interchange Format (EDIF) netlist (EDN file) and NGC files contain the information required to implement the module in a Xilinx FPGA. Since NGC files are in binary format, ASCII NDF files may also be produced to communicate resource and timing information for NGC files to third party synthesis tools. NGC files are generated for certain cores only.

The NDF file is equivalent to an EDIF file. The ASY and XSF symbol information files allow you to integrate the CORE Generator module into a schematic design for Mentor or ISE tools.

VHO Files

VHDL template (VHO) template files contain code that can be used as a model for instantiating a CORE Generator module in a VHDL design. VHO filed come with a VHDL (VHD) wrapper file.

VEO Files

Verilog template (VEO) files contain code that can be used as a model for instantiating a CORE Generator module in a Verilog design. VEO files come with a VHDL (VHD) wrapper file.

V and VHD Wrapper Files

V and VHD wrapper files support functional simulation. These files contain simulation model customization data that is passed to a parameterized simulation model for the core. In the case of Verilog designs, the V wrapper file also provides the port information required to integrate the core into a Verilog design for synthesis.

The V and VHD wrapper files mainly support simulation and are not synthesizable.

Functional Simulation

Use functional or RTL simulation to verify the syntax and functionality of your design.

Simulation Recommendations

Xilinx recommends that you do the following when you simulate your design:

- "Perform Separate Simulations"
- "Create a Test Bench"

Perform Separate Simulations

With larger hierarchical HDL designs, perform separate simulations on each module before testing your entire design. This makes it easier to debug your code.

Create a Test Bench

Once each module functions as expected, create a test bench to verify that your entire design functions as planned. Use the same test bench again for the final timing simulation to confirm that your design functions as expected under worst-case delay conditions.

ModelSim Simulators

You can use ModelSim simulators with Project Navigator. The appropriate processes appear in Project Navigator when you choose ModelSim as your design simulator, provided you have installed any of the following:

- ModelSim Xilinx Edition-II
- ModelSim PE, EE or SE

You can also use these simulators with third-party synthesis tools in Project Navigator.

For more information about ModelSim support, see the Xilinx *Tech Tips*.

Synthesizing and Optimizing

This section includes recommendations for compiling your designs to improve your results and decrease the run time. For more information, see your synthesis tool documentation.

Creating an Initialization File

Most synthesis tools provide a default initialization with default options. You may modify the initialization file or use the application to change compiler defaults, and to point to the applicable implementation libraries. For more information, see your synthesis tool documentation.

Creating a Compile Run Script

The following tools all support TCL scripting:

- "DCFPGA"
- "LeonardoSpectrum"
- "Precision RTL Synthesis"
- "Synplify"
- "XST"

TCL scripting can make compiling your design easier and faster while achieving shorter compile times. With more advanced scripting, you can run a compile multiple times using different options and write to different directories. You can also invoke and run other command line tools.

You can run the following sample scripts from the command line or from the application.

DCFPGA

DCFPGA needs a setup file to tell the tool where to find all of the libraries. The libraries are installed from the Xilinx installation CDs only on Solaris and Linux platforms.

Once you have determined that the libraries have been installed properly in the `$XILINX/synopsys` directory, you can modify your setup file (.synopsys_dc.setup).

DCFPGA first looks for the setup file in your project directory, then in your home directory. If DCFPGA does not find the file in one of those locations, it relies on a default file located in the DCFPGA installation area. Below are two sample setup scripts for DCFPGA.

Sample DCFPGA Setup Script One

```
#================================================== #
# Template .synopsys_dc.setup file appropriate for    #
# use with DC FPGA when targetting Xilinx VIRTEX™2      #
# architecture.                                       #
# ================================================== #
setenv XILINX /u/pdq_cae/tools/synopsys/dcfpga-lib/2005.03
set XilinxInstall [getenv XILINX]

                # !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  #
                # Ensure that your UNIX environment  #
                # includes the environment variable: #
                # $XILINX (points to the Xilinx      #
                # installation directory)            #
                # !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  #

#set search_path ". $XilinxInstall/VIRTEX2 $search_path"
set search_path ". $XilinxInstall/synopsys/libraries/syn $search_path"
```

Sample DCFPGA Setup Script Two

```
# ================================================= #
# Define a work library in the current project dir  #
# to hold temporary files and keep the project area #
# uncluttered. Note: You must create a subdirectory #
# in your project directory called WORK.            #
# ================================================= #

   define_design_lib WORK -path ./WORK

# ================================================  #
# You may like to copy this file to your project    #
# directory, rename it ".synopsys_dc.setup".        #
# ================================================  #

set target_library "xdcf_virtex2.db"
set synthetic_library "tmg.sldb"
set link_library "* xdcf_virtex2.db tmg.sldb"

set cache_dir_chmod_octal "1777"


set hdlin_enable_non_integer_parameters "true"
set hdlin_translate_off_skip_text "true"
set edifout_ground_name  "GND"
set edifout_ground_pin_name  "G"
set edifout_ground_port_name "G"
set edifout_power_name  "VCC"
set edifout_power_pin_name  "P"
set edifout_power_port_name  "P"
set edifout_netlist_only  "true"
set edifout_power_and_ground_representation  "cell"
set hdlin_enable_vpp  "true"

# set the edifout attributes.
set edifout_write_properties_list " DUTY_CYCLE_CORRECTION INIT_00 INIT_01 INIT_02 INIT_03 \
INIT_04  INIT_05 INIT_06 INIT_07 INIT_08 INIT_09 INIT_0A INIT_0B INIT_0C \
INIT_0D INIT_0E INIT_0F INIT CLKDV_DIVIDE IOB EQN \
lut_function instance_number pad_location part \
INIT_10 INIT_11 INIT_12 INIT_13 INIT_14 INIT_15 \
INIT_16 INIT_17 INIT_18 INIT_19 INIT_1A INIT_1B \
INIT_1C INIT_1D INIT_1E INIT_1F INIT_20 INIT_21 \
INIT_22 INIT_23 INIT_24 INIT_25 INIT_26 INIT_27 \
INIT_28 INIT_29 INIT_2A INIT_2B INIT_2C INIT_2D \
INIT_2E INIT_2F INIT_30 INIT_31 INIT_32 INIT_33 \
INIT_34 INIT_35 INIT_36 INIT_37 INIT_38 INIT_39 \
INIT_3A INIT_3B INIT_3C INIT_3D INIT_3E INIT_3F \
INIT_A INIT_B SRVAL_A SRVAL_B SRVAL WRITE_MODE \
WRITE_MODE_A WRITE_MODE_B INITP_00 INITP_01 INITP_02 \
INITP_03 INITP_04 INITP_05 INITP_06 INITP_07 \
DLL_FREQUENCY_MODE DUTY_CYCLE_CORRECT CLKDV_DIVIDE \
CLK_FEEDBACK CLKOUT_PHASE_SHIFT STARTUP_WAIT FACTORY_JF \
DSS_MODE PHASE_SHIFT CLKFX_MULTIPLY CLKFX_DIVIDE DFS_FREQUENCY_MODE \
DESKEW_ADJUST FACTORY_JF STEPPING "

set post_compile_cost_check  "false"

set_fpga_defaults xilinx_virtex2
```

LeonardoSpectrum

To run the following TCL script from LeonardoSpectrum:

1. Select **File →Run Script**.
2. Type in the Level 3 command line, **`source`** *script_file.tcl*
3. Type in the UNIX/DOS prompt with the EXEMPLAR environment path set up, **`spectrum`** *–file script_file.tcl*
4. Type **`spectrum`** at the UNIX/DOS prompt.
5. At the TCL prompt type **`source`** *script_file.tcl*

You can enter the following TCL commands in LeonardoSpectrum.

Table 2-1: **LeonardoSpectrum TCL Commands**

Function	Command
set part type	**`set part`** *v50ecs144*
read the HDL files	**`read`** *macro1.vhd macro2.vhd top_level.vhd*
set assign buffers	**`PAD`** *IBUF_LVDS data(7:0)*
optimize while preserving hierarchy	**`optimize`** *-ta xcve -hier preserve*
write out the EDIF file	**`auto_write`** *./M1/ff_example.edf*

Precision RTL Synthesis

To run the TCL script from Precision RTL Synthesis:

1. Set up your project in Precision.
2. Synthesize your project.
3. Run the following commands to save and run the TCL script.

Table 2-2: **Precision RTL Synthesis Commands**

Function	Command
save the TCL script	**File →Save Command File**
run the TCL script	**File →Run Script**
run the TCL script from a command line	c:\**`precision`** *-shell -file project.tcl*

complete the synthesis process	**`add_input_file`** *top.vhdl*
	`setup_design` *-manufacturer Xilinx -family Virtex-II -part 2v40cs144 -speed 6*
	`compile`
	`synthesize`

Synplify

To run the following TCL script from Synplify:

- Select **File →Run TCL Script**.

OR

- Type **`synplify`** *-batch script_file.tcl* at a UNIX or DOS command prompt.

Enter the following TCL commands in Synplify.

Table 2-3: **Synplify Commands**

Function	Command
start a new project	**`project`** *-new*
set device options	**`set_option`** *-technology Virtex-E* **`set_option`** *-part XCV50E* **`set_option`** *-package CS144* **`set_option`** *-speed_grade -8*
add file options	**`add_file`** *-constraint "watch.sdc"* **`add_file`** *-vhdl -lib work "macro1.vhd"* **`add_file`** *-vhdl -lib work "macro2.vhd"* **`add_file`** *-vhdl -lib work "top_levle.vhd"*
set compilation/mapping options	**`set_option`** *-default_enum_encoding onehot* **`set_option`** *-symbolic_fsm_compiler true* **`set_option`** *-resource_sharing true*

Table 2-3: **Synplify Commands**

Function	Command
set simulation options	**`set_option`** *-write_verilog false* **`set_option`** *-write_vhdl false*
set automatic place and route (vendor) options	**`set_option`** *-write_apr_constraint true* **`set_option`** *-part XCV50E* **`set_option`** *-package CS144* **`set_option`** *-speed_grade -8*
set result format/file options	**`project`** *-result_format "edif"* **`project`** *-result_file "top_level.edf"* **`project`** *-run* **`project`** *-save "watch.prj"*
exit	**`exit`**

XST

For information and options used with the Xilinx Synthesis Tool (XST), see the Xilinx *XST User Guide*.

Synthesizing Your Design

Xilinx recommends the following to help you successfully synthesize your design.

Modifying Your Design

You may need to modify your code to successfully synthesize your design because certain design constructs that are effective for *simulation* may not be as effective for *synthesis*. The synthesis syntax and code set may differ slightly from the simulator syntax and code set.

Synthesizing Large Designs

Older versions of synthesis tools required incremental design compilations to decrease run times. Some or all levels of hierarchy were compiled with separate compile commands and saved as output or database files. The output netlist or compiled database file for each module was read during synthesis of the top level code. This method is not necessary with new synthesis tools, which can handle large designs from the top down. The 5,000 gates per module rule of thumb no longer applies with the new synthesis tools. For more information, see your synthesis tool documentation.

Saving Compiled Design as EDIF or NGC

After your design is successfully compiled, save it as an EDIF or NGC file for input to the Xilinx software.

Reading Cores

The following tools support the use of CORE Generator EDIF files for timing and area analysis:

- "XST"
- "LeonardoSpectrum"
- "Synplify Pro"
- "Precision RTL Synthesis"

Reading the EDIF files results in better timing optimizations, since the delay through the logic elements inside the CORE file is known.

The procedures for reading in cores in these synthesis tools are as follows.

XST

Invoke XST using the *read_cores* switch. When the switch is set to *on*, the default, XST, reads in EDIF and NGC netlists. For more information, see the Xilinx *XST User Guide*. For more information on doing this in ISE, see the Project Navigator help.

LeonardoSpectrum

Use the **`read_coregen`** TCL command line option. For more information, see Xilinx Answer Record 13159, "*How do I read in a CORE Generator core's EDIF netlist in LeonardoSpectrum?*"

Synplify Pro

EDIF is treated as just another source format, but when reading in EDIF, you must specify the top level VHDL/Verilog in your project. Support for reading in EDIF is included in Synplify Pro version 7.3. For more information, see the Synplify documentation.

Precision RTL Synthesis

Precision RTL Synthesis can add EDIF and NGC files to your project as source files. For more information, see the *Precision RTL Synthesis* help.

Setting Constraints

Setting constraints is an important step in the design process. It allows you to control timing optimization and enables more efficient use of synthesis tools and implementation processes. This efficiency helps minimize runtime and achieve the requirements of your design.

There are many different types of constraints that can be set. Additionally, there can be multiple constraints files used in the design process. The table below outlines the different types of constraints, the methods by which they are most commonly entered, and the files they are found in.

Table 2-4: **Constraints Table**

Type of Constraint	Constraint Entry Method	Where Found
Synthesis	Constraints Editor Text Editor (HDL source)	• Netlist Constraints File (NCF) • XST Constraints File (XCF)
Implementation (Mapping, Placing, Routing)	Constraints Editor Text Editor (HDL source) Floorplanner FPGA Editor	• User Constraints File (UCF) • Netlist Constraints File (NCF) • Physical Constraints File (PCF)
Timing	Text Editor (HDL Source) Text Editor (UCF, NCF, PCF)	• User Constraints File (UCF) • Netlist Constraints File (NCF) • Physical Constraints File (PCF)
Pinout & Area	Pinout and Area Constraints Editor (PACE)	• User Constraints File (UCF)

Setting Constraints Using a Synthesis Tool Constraints Editor

LeonardoSpectrum, Precision RTL Synthesis, and Synplify all have constraints editors that allow you to apply constraints to your HDL design. For more information on how to use your synthesis tool’s constraints editor, see your synthesis tool documentation.

You can add the following constraints:

- Clock frequency or cycle and offset
- Input and Output timing
- Signal Preservation
- Module constraints
- Buffering ports
- Path timing
- Global timing

Generally, the timing constraints are written to an NCF file, and all other constraints are written to the output EDIF file. In XST, all constraints are written to the NGC file.

Setting Constraints in the UCF File

The UCF gives you tight control of the overall specifications by giving you access to more types of constraints; the ability to define precise timing paths; and the ability to prioritize signal constraints. To simplify timing specifications, group signals together. Some synthesis tools translate certain synthesis constraints to Xilinx implementation constraints. The translated constraints are placed in the NCF/EDIF file (NGC file for XST). For more information on timing specifications in the UCF file, see the Xilinx *Constraints Guide*.

Setting Constraints Using the Xilinx Constraints Editor

The Xilinx Constraints Editor:

- enables you to easily enter design constraints in a spreadsheet form
- writes out the constraints to the UCF file
- eliminates the need to know the UCF syntax
- reads the design
- lists all the nets and elements in the design

To run the Xilinx Constraints Editor from Project Navigator:

1. Go to the Processes window.
2. Select **Design Entry Utilities →User Constraints →Create Timing Constraints**.

To run the Xilinx Constraints Editor from the command line, type **`constraints_editor`**.

Some constraints are not available from Constraints Editor. For unavailable constraints:

1. Enter the unavailable constraints directly in the UCF file with a text editor.
2. Use the command line method to re-run the new UCF file through the Translate step or NGDBuild.

Setting Constraints Using PACE

The Xilinx Pinout and Area Constraints Editor (PACE) enables you to:

- assign pin location constraints
- assign certain IO properties such as IO Standards

To run PACE from Project Navigator:

1. Go to the Processes window.
2. Select **Design Entry Utilities →User Constraints →Assign Package Pins OR →Create Area Constraints**.

To run PACE from the command line, type **`pace`**.

Evaluating Design Size and Performance

Your design must:

- function at the specified speed
- fit in the targeted device

After your design is compiled, use your synthesis tool's reporting options to determine preliminary device utilization and performance. After your design is mapped by the Xilinx tools, you can determine the actual device utilization.

At this point in the design flow, you should verify that your chosen device is large enough to accommodate any future changes or additions, and that your design will perform as specified.

Estimating Device Utilization and Performance

Use the area and timing reporting options of your synthesis tool to estimate device utilization and performance. After compiling, use the report area command to obtain a report of device resource utilization. Some synthesis tools provide area reports automatically. For correct command syntax, see your synthesis tool documentation.

The device utilization and performance report lists the compiled cells in your design, as well as information on how your design is mapped in the FPGA. These reports are generally accurate because the synthesis tool creates the logic from your code and maps your design into the FPGA. However, these reports are different for the various synthesis tools. Some reports specify the minimum number of CLBs required, while other reports specify the "unpacked" number of CLBs to make an allowance for routing. For an accurate comparison, compare reports from the Xilinx mapper tool after implementation.

Any instantiated components, such as CORE Generator modules, EDIF files, or other components that your synthesis tool does not recognize during compilation, are not included in the report file. If you include these components, you must include the logic area used by these components when estimating design size. Sections of your design may get trimmed during the mapping process, which may result in a smaller design.

Use the timing report command of your synthesis tool to obtain a report with estimated data path delays. For more information on command syntax, see your synthesis tool documentation.

The timing report is based on the logic level delays from the cell libraries and estimated wire-load models. This report is an estimate of how close you are to your timing goals; however, it is not the actual timing. An accurate timing report is only available after the design is placed and routed.

Determining Actual Device Utilization and Pre-routed Performance

To determine if your design fits the specified device, map it using the Xilinx Map program. The generated report file `design_name.mrp` contains the implemented device utilization information. To read the report file, double-click **Map Report** in the Project Navigator Processes window. Run the Map program from Project Navigator or from the command line.

Using Project Navigator to Map Your Design

To map your design using Project Navigator:

1. Go to the Processes window.
2. Click the "+" symbol in front of **Implement Design.**
3. Double-click **Map.**
4. To view the Map Report, double-click **Map Report**.

   If the report does not currently exist, it is generated at this time. A green check mark in front of the report name indicates that the report is up-to-date, and no processing is performed.

5. If the report is not up-to-date:

   a. Click the report name.

   b. Select **Process** →**Rerun** to update the report.

      The auto-make process automatically runs only the necessary processes to update the report before displaying it.

Alternatively, you may select **Process →Rerun All** to re-run all processes— even those processes that are currently up-to-date— from the top of the design to the stage where the report would be.

6. View the Logic Level Timing Report with the Report Browser. This report shows the performance of your design based on logic levels and best-case routing delays.
7. Run the Timing Analyzer to create a more specific report of design paths (optional).
8. Use the Logic Level Timing Report and any reports generated with the Timing Analyzer or the Map program to evaluate how close you are to your performance and utilization goals.

   Use these reports to decide whether to proceed to the place and route phase of implementation, or to go back and modify your design or implementation options to attain your performance goals. You should have some slack in routing delays to allow the place and route tools to successfully complete your design. Use the verbose option in the Timing Analyzer to see block-by-block delay. The timing report of a mapped design (before place and route) shows block delays, as well as minimum routing delays.

A typical Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, or Virtex-4 design should allow 40% of the delay for logic, and 60% of the delay for routing. If most of your time is taken by logic, the design will probably not meet timing after place and route.

Using the Command Line to Map Your Design

To map your design using the command line:

Note: For available options, enter only the trce command at the command line without any arguments.

1. Run the following command to translate your design:

   `ngdbuild −p` *target_device design_name*`.edf` **(**or **`ngc`)**

2. Run the following command to map your design:

   `map` *design_name*`.ngd`

3. Use a text editor to view the Device Summary section of the `design_name.mrp` Map Report. This section contains the device utilization information.
4. Run a timing analysis of the logic level delays from your mapped design as follows.

   `trce [options]` *design_name*`.ncd`

   Use the Trace reports to evaluate how close you are to your performance goals. Use the report to decide whether to proceed to the place and route phase of implementation, or to go back and modify your design or implementation options to attain your performance goals. You should have some slack in routing delays to allow the place and route tools to successfully complete your design.

   The following is the Design Summary section of a Map Report containing device information.

Evaluating Coding Style and System Features

At this point, if you are not satisfied with your design performance, re-evaluate your code and make any necessary improvements. Modifying your code and selecting different compiler options can dramatically improve device utilization and speed.

This section describes ways to improve design performance by modifying your code and by incorporating FPGA system features. Most of these techniques are described in more detail in this guide.

Modifying Your Code

Improve design performance with the following design modifications.

- Reduce levels of logic to improve timing
  - ♦ use pipelining and retiming techniques
  - ♦ rewrite the HDL descriptions
  - ♦ enable or disable resource sharing
- Redefine hierarchical boundaries to help the compiler optimize design logic
  - ♦ restructure logic
- Reduce critical nets fanout to improve placement and reduce congestion
  - ♦ perform logic replication
- Take advantage of device resource with the CORE Generator modules

Using FPGA System Features

After correcting any coding style problems, use any of the following FPGA system features in your design to improve resource utilization and to enhance the speed of critical paths.

Each device family has a unique set of system features. For more information about the system features available for the device you are targeting, see the device data sheet.

- Use clock enables
- In Virtex family components, modify large multiplexers to use 3-state buffers
- Use one-hot encoding for large or complex state machines
- Use I/O registers when applicable
- In Virtex families, use dedicated shift registers
- In Virtex-II families, use dedicated multipliers

Using Xilinx-Specific Features of Your Synthesis Tool

Your synthesis tool allows better control over the logic generated, the number of logic levels, the architecture elements used, and fanout. The place and route tool (PAR) has advanced its algorithms to make it more efficient to use your synthesis tool to achieve design performance if your design performance is more than a few percentages away from the requirements of your design.

Most synthesis tools have special options for the Xilinx-specific features listed in the previous section. For more information on using Xilinx-specific features, see your synthesis tool documentation.

Incremental Design

Incremental design allows you to make last minute changes to your design and rerun synthesis on only those sections of the design that have changed. This keeps the unchanged sections of the design locked down and unaffected when resynthesizing your design. To use incremental design, you must first define area groups in your design. For more information, see the following.

Table 2-5: **Incremental Design Resources**

Item	Resource
INCREMENTAL_SYNTHESIS constraint	Xilinx *XST User Guide*
AREA_GROUP constraint	Xilinx *Constraints Guide*
General use and concepts of incremental design	Xilinx *Development System Reference Guide*

Modular Design

Modular design allows a team of engineers to independently work on different sections, or modules, of a design and later merge these modules into a single FPGA design. Modular design can help you plan and manage large designs. For more information, see the Xilinx *Development System Reference Guide*.

Placing and Routing

Note: For more information on placing and routing your design, see the Xilinx *Development System Reference Guide*.

The overall goal when placing and routing your design is fast implementation and high-quality results. However, depending on the situation and your design, you may not always accomplish this goal, as described in the following examples.

- Earlier in the design cycle, run time is generally more important than the quality of results, and later in the design cycle, the converse is usually true.
- If the targeted device is highly utilized, the routing may become congested, and your design may be difficult to route. In this case, the placer and router may take longer to meet your timing requirements.
- If design constraints are rigorous, it may take longer to correctly place and route your design, and meet the specified timing.

Decreasing Implementation Time

The placement and routing options you select directly influence run time. Generally, these options decrease the run time at the expense of the best placement and routing for a given device. Select your options based on your required design performance.

To decrease implementation time using Project Navigator:

1. Right click **Place & Route** in theProcesses window.
2. Select **Properties.**
3. Set the following options in the Process Properties dialog box:
   a. Place & Route Effort Level

      Note: Alternatively, run the `-ol` switch at the command line.

      Generally, to reduce placement times, select a less CPU-intensive algorithm for placement. Set the placement level at one of three settings:

      - Standard (fastest run time) (the default)

- Medium (medium run time with equal place and route optimization)
- Highest (longest run time with the best place and route results)

Note: In some cases, poor placement with a lower placement level setting can result in longer route times.

b. Router Options

To limit router iterations to reduce routing times, set the Number of Routing Passes option. However, this may prevent your design from meeting timing requirements, or your design may not completely route. The amount of time spent routing is controlled with router effort level (**`-rl`**).

c. Use Timing Constraints During Place and Route

To improve run times, do not specify some or all timing constraints. This is useful at the beginning of the design cycle during the initial evaluation of the placed and routed circuit. To disable timing constraints in the Project Navigator, uncheck the Use Timing Constraints check box. To disable timing constraints at the command line, run the **`-x`** switch with PAR.

4. Click **OK** to exit the Process Properties dialog box.
5. Double click **Place & Route** in the Processes window to begin placing and routing your design.

Improving Implementation Results

You can select options that *increase* the run time, but produce a better design. These options generally produce a faster design at the cost of a longer run time. These options are useful when you run your designs for an extended period of time (overnight or over the weekend). Use the following options to improve implementation results. For more information, see the Xilinx *Development System Reference Guide.*

Map Timing

Use the Xilinx Map program Timing option to improve timing during the mapping phase. This switch directs the mapper to give priority to timing critical paths during packing. To use this feature at the command line, use the **`-timing`** switch.

Extra Effort Mode in PAR

Use the Xilinx PAR program Extra Effort mode to invoke advanced algorithmic techniques to provide higher quality results. To use this feature at the command line, use the ***`-xe level`*** switch. The level can be a value from 0 to 5; the default is 1.

Turns Engine Option

This Unix and Linux-only feature works with the Multi-Pass Place and Route option to allow parallel processing of placement and routing on several Unix or Linux machines. The only limitation to how many cost tables are concurrently tested is the number of workstations you have available.

To use this option in Project Navigator, see the Project Navigator help for a description of the options that can be set under Multi-Pass Place and Route.

To use this feature at the command line:

- run the **-m** switch to specify a node list
- run the **-n** switch to specify the number of place and route iterations

Reentrant Routing Option

Use the reentrant routing option to further route an already routed design. The router reroutes some connections to improve the timing, or to finish routing unrouted nets. You must specify a placed and routed design (`.ncd`) file for the implementation tools. This option is best used when router iterations are initially limited, or when your design timing goals are close to being achieved.

From Project Navigator

To initiate a reentrant route from Project Navigator:

1. Right click **Place & Route** in the Processes window.
2. Select **Properties.**
3. In the Process Properties dialog box, set the Place And Route Mode option to **Reentrant Route.**
4. Click **OK**.
5. Double click **Place & Route** in the Processes window to place and route your design.

Using PAR and Cost Tables

The PAR module places in two stages: a constructive placement and an optimizing placement. PAR writes the NCD file after constructive placement, and modifies the NCD after optimizing placement.

During constructive placement, PAR places components into sites based on factors such as constraints specified in the input file (for example, certain components must be in certain locations), the length of connections, and the available routing resources. This placement also takes into account "cost tables," which assign weighted values to each of the relevant factors. There are 100 possible cost tables. Constructive placement continues until all components are placed. PAR writes the NCD file after constructive placement.

From the Command Line

To initiate a reentrant route from the command line, run PAR with the **-k** and **-p** options, as well as any other options you want to use for the routing process. You must either specify a unique name for the post reentrant routed design (`.ncd`) file or use the **-w** switch to overwrite the previous design file, as shown in the following examples.

```
par -k -p other_options design_name.ncd new_name.ncd
```

```
par -k -p -w other_options design_name.ncd design.ncd
```

Guide Option

Use a guide file for both Map and PAR to instruct the implementation processes to use existing constraints in an input design file. This is useful if minor incremental changes have been made to an existing design to create a new design. To increase productivity, use your last design iteration as a guide design for the next design iteration.

This option is generally not recommended for synthesis-based designs, except for modular design flows. Re-synthesizing modules can cause the signal and instance names in the resulting netlist to be significantly different from those in earlier synthesis runs. This can occur even if the source level code (Verilog or VHDL) contains only a small change. Because the guide process is dependent on the names of signals and comps, synthesis designs often result in a low match rate during the guiding process. Generally, this option does not improve implementation results.

For more information on the guide option in modular design flows, see:

- Xilinx Modular Design on the Xilinx Support website
- Xilinx *Development System Reference Guide*, "Modular Design"

Timing Simulation

Timing simulation is important in verifying the operation of your circuit after the worst-case placed and routed delays are calculated for your design. In many cases, you can use the same test bench that you used for functional simulation to perform a more accurate simulation with less effort. Compare the results from the two simulations to verify that your design is performing as initially specified. The Xilinx tools create a VHDL or Verilog simulation netlist of your placed and routed design, and provide libraries that work with many common HDL simulators. For more information on design simulation, see Chapter 6, "Simulating Your Design" in this guide.

Timing-driven PAR is based upon TRACE, the Xilinx timing analysis software. TRACE is an integrated static timing analysis, and does not depend on input stimulus to the circuit. Placement and routing are executed according to timing constraints that you specify at the beginning of the design process. TRACE interacts with PAR to make sure that the timing constraints you impose on the design are met.

If you have timing constraints, TRACE generates a report based on your constraints. If there are no constraints, TRACE has an option to write out a timing report containing:

- An analysis that enumerates all clocks and the required OFFSETs for each clock
- An analysis of paths having only combinatorial logic, ordered by delay

For more information on TRACE, see the Xilinx *Development System Reference Guide*. For more information on Timing Analysis, see the Timing Analyzer help.

General HDL Coding Styles

This chapter contains HDL coding styles and design examples to help you develop an efficient coding style. This chapter includes the following sections.

Introduction

HDLs contain many complex constructs that are difficult to understand at first. In addition, the methods and examples included in HDL guides do not always apply to the design of FPGA devices. If you currently use HDLs to design ASIC devices, your established coding style may unnecessarily increase the number of gates or CLB levels in FPGA designs.

HDL synthesis tools implement logic based on the coding style of your design. To learn how to efficiently code with HDLs, you can:

- Attend training classes
- Read reference and methodology notes
- See synthesis guidelines and templates available from Xilinx® and the synthesis vendors

When coding your designs, remember that HDLs are mainly hardware description languages. You should try to find a balance between the quality of the end hardware results and the speed of simulation.

The hints and examples included in this chapter are not intended to teach you every aspect of VHDL or Verilog, but they should help you develop an efficient coding style.

Naming, Labeling, and General Coding Styles

Xilinx recommends that you and your design team agree on a style for your code at the beginning of your project. An established coding style allows you to read and understand code written by your team members. Inefficient coding styles can adversely impact synthesis and simulation, which can result in slow circuits. Because portions of existing HDL designs are often used in new designs, you should follow coding standards that are understood by the majority of HDL designers. This chapter describes recommended coding styles that you should establish before you begin your designs.

Using Xilinx Naming Conventions

Use the Xilinx naming conventions listed in this section for naming signals, variables, and instances that are translated into nets, buses, and symbols.

- VHDL-200x Key words (such as entity, architecture, signal, and component)
- System Verilog 3.1a key word: (such as module, reg, and wire). See Annex B of System Verilog Spec version 3.1a
- A user generated name should not contain a forward slash (/). The forward slash (/) is generally used to denote a hierarchy separator.
- Names must contain at least one non-numeric character.
- Names must not contain a dollar sign ($).
- Names must not use less-than (<)or greater-than signs (>). These signs are sometimes used to denote a bus index.
- The following FPGA resource names are reserved. They should not be used to name nets or components.
  - Device architecture names (such as CLB, IOB, PAD, and Slice)
  - Dedicated pin names (such as CLK and INIT)
  - GND and VCC
  - UNISIM primitive names such as BUFG, DCM, and RAMB16
  - Do not use pin names such as P1 and A4 for component names

For language-specific naming restrictions, see the language reference manual for Verilog or VHDL. Xilinx does not recommend using escape sequences for illegal characters. In addition, if you plan to import schematic, or to use mixed language synthesis or verification, use the most restrictive character set.

Naming Guidelines for Signals and Instances

Xilinx recommends that you follow the naming conventions set forth below in order to help achieve the goals of:

- maximum line length
- coherent and legible code
- allowance for mixed VHDL and Verilog design
- consistent HDL code

General

Xilinx recommends that you observe the following general rules:

- Do not use reserved words for signal or instance names.
- Do not exceed 16 characters for the length of signal and instance names, whenever possible.
- Create signal and instance names that reflect their connection or purpose.
- Do not use mixed case for any particular name or keyword. Use either all capitals, or all lowercase.

Recommendations for VHDL and Verilog Capitalization

Xilinx recommends that you observe the following guidelines when naming signals and instances in VHDL and Verilog.

Table 3-1: **VHDL and Verilog Capitalization**

Lower Case	Upper Case	Mixed Case
library names	USER PORTS	Comments
keywords	INSTANCE NAMES	
module names	UNISIM COMPONENT NAMES	
entity names	PARAMETERS	
user component names	GENERICS	
internal signals		

Matching File Names to Entity and Module Names

Xilinx recommends the following practices in naming your HDL files.

- Make sure that the VHDL or Verilog source code file name matches the designated name of the entity (VHDL) or module (Verilog) specified in your design file. This is less confusing, and generally makes it easier to create a script file for the compilation of your design.
- If your design contains more than one entity or module, put each in a separate file with the appropriate file name. For VHDL design, Xilinx recommends grouping the entity and the associated architecture into the same file.
- It is a good idea to use the same name as your top-level design file for your synthesis script file with either a .do, .scr, .script, or the appropriate default script file extension for your synthesis tool.

Naming Identifiers

Follow these naming practices to make design code easier to debug and reuse:

- Use concise but meaningful identifier names.
- Use meaningful names for wires, regs, signals, variables, types, and any identifier in the code such as CONTROL_REGISTER.
- Use underscores to make the identifiers easier to read.

Guidelines for Instantiation of Sub-Modules

Follow these guidlines when instantiating sub-modules.

Xilinx recommends that you always use named association to prevent incorrect connections for the ports of instantiated components. Never combine positional and named association in the same statement as illustrated in the following examples.

Table 3-2: **Correct and Incorrect VHDL and Verilog Examples**

	VHDL	Verilog

Incorrect	`CLK_1: BUFG`
	`   port map (`
	`      I=>CLOCK_IN,`
	`      CLOCK_OUT`
	`);`
		`BUFG CLK_1 (`
		`   .I(CLOCK_IN),`
		`   CLOCK_OUT`
		`);`

Correct	`CLK_1: BUFG`
	`   port map(`
	`      I=>CLOCK_IN,`
	`      O=>CLOCK_OUT`
	`);`
		`BUFG CLK_1 (`
		`   .I(CLOCK_IN),`
		`   .O(CLOCK_OUT)`
		`);`

Xilinx also recommends using one port mapping per line to improve readability, provide space for a comment, and allow for easier modification.

VHDL Example

```
-- FDCPE: Single Data Rate D Flip-Flop with Asynchronous Clear, Set and
-- Clock Enable (posedge clk).  All families.
-- Xilinx  HDL Language Template

FDCPE_inst : FDCPE
generic map (
   INIT => '0') -- Initial value of register ('0' or '1')
port map (
   Q => Q,       -- Data output
   C => C,       -- Clock input
   CE => CE,     -- Clock enable input
   CLR => CLR,   -- Asynchronous clear input
   D => D,       -- Data input
   PRE => PRE    -- Asynchronous set input
);

-- End of FDCPE_inst instantiation
```

Verilog Example

```
// FDCPE: Single Data Rate D Flip-Flop with Asynchronous Clear, Set and
   //        Clock Enable (posedge clk).  All families.
   // Xilinx HDL Language Template

   FDCPE #(
      .INIT(1'b0) // Initial value of register (1'b0 or 1'b1)
   ) FDCPE_inst (
      .Q(Q),      // Data output
      .C(C),      // Clock input
      .CE(CE),    // Clock enable input
      .CLR(CLR),  // Asynchronous clear input
      .D(D),      // Data input
      .PRE(PRE)   // Asynchronous set input
   );

   // End of FDCPE_inst instantiation
```

Recommended Length of Line

Xilinx recommends that the length of a line of VHDL or Verilog code not exceed 80 characters. If a line needs to exceed this limit, break it with the continuation character, and align the subsequent lines with the proceeding code. Choose signal and instance names carefully in order to not break this limit.

Try not to make too many nests in the code, such as nested *if* and *case* statements. If you have too many *if* statements inside of other *if* statements, it can make the line length too long, as well as inhibit optimization. By following this guideline, code is generally more readable and more portable, and can be more easily formatted for printing.

Using a Common File Header

Xilinx recommends that you use a common file header surrounded by comments at the beginning of each file. A common file header:

- allows for better documentation of the design and code
- improves code revision tracking
- enhances reuse

The contents of the header depend on personal and company standards. Following is an example file header in VHDL.

```
--------------------------------------------------------------------------------
-- Copyright (c) 1996-2003 Xilinx, Inc.
-- All Rights Reserved
--------------------------------------------------------------------------------
--   ____  ____
--  /   /\/   /    Company: Xilinx
-- /___/  \  /   Design Name: MY_CPU
-- \   \   \/    Filename: my_cpu.vhd
--  \   \        Version: 1.1.1
--  /   /        Date Last Modified:  Fri Sep 24 2004
-- /___/   /\    Date Created: Tue Sep 21 2004
-- \   \  /  \
--  \___\/\___\
--
--Device: XC3S1000-5FG676
--Software Used: ISE 8.1i
--Libraries used: UNISIM
--Purpose: CPU design
--Reference:
--    CPU specification found at: http://www.mycpu.com/docs
--Revision History:
--    Rev 1.1.0 - First created, joe_engineer, Tue Sep 21 2004.
--    Rev 1.1.1 - Ran changed architecture name from CPU_FINAL
--        john_engineer, Fri Sep 24 2004.
```

Use of Indentation and Spacing in the Code

Proper indentation in code offers the following benefits:

- More readable and comprehensible code by showing grouped constructs at the same indentation level
- Fewer coding mistakes
- Easier debugging

The indentation style is somewhat arbitrary. But as long as a common theme is used, the benefit listed above are generally achieved.

Following are examples of code indentation in VHDL and Verilog.

VHDL Example

```
entity AND_OR is
  port (
    AND_OUT : out std_logic;
    OR_OUT  : out std_logic;
    I0      : in std_logic;
    I1      : in std_logic;
    CLK     : in std_logic;
    CE      : in std_logic;
    RST     : in std_logic);
end AND_OR;
architecture BEHAVIORAL_ARCHITECTURE of AND_OR is

   signal and_int : std_logic;
   signal or_int  : std_logic;

begin

   AND_OUT <= and_int;
   OR_OUT  <= or_int;
   process (CLK)
   begin
      if (CLK'event and CLK='1') then
         if (RST='1') then
            and_int <= '0';
            or_int  <= '0';
         elsif (CE ='1') then
            and_int <= I0 and I1;
            or_int  <= I0 or I1;
         end if;
      end if;
   end process;

end AND_OR;
```

Verilog Example

```
module AND_OR  (AND_OUT, OR_OUT, I0, I1, CLK, CE, RST);

   output reg AND_OUT, OR_OUT;

   input I0,  I1;
   input CLK, CE,  RST;
```

```
      always @(posedge CLK)
         if (RST) begin
            AND_OUT <= 1'b0;
            OR_OUT <= 1'b0;
         end else (CE) begin
            AND_OUT <= I0 and I1;
            OR_OUT <= I0 or I1;
         end

   endmodule
```

Use of TRANSLATE_OFF and TRANSLATE_ON in Source Code

The synthesis directives TRANSLATE_OFF and TRANSLATE_ON were formerly used when passing generics/parameters for synthesis tools, since most synthesis tools were unable to read generics/parameters. These directives were also used for library declarations such as library UNISIM, since synthesis tools did not understand that library.

More recent synthesis tools can now read generics/parameter, and can understand the use of the UNISIM library. Accordingly, there is no longer any need to use these directives in synthesizable code. The TRANSLATE_OFF and TRANSLATE_ON directives can also be used to embed simulation-only code in synthesizable files. Xilinx recommends that any simulation-only constructs reside in simulation-only files or test benches.

Attributes and Constraints

The terms *attribute* and *constraint* have been used interchangeably by some in the engineering community, while others give different meanings to these terms. In addition, language constructs use the terms *attribute* and *directive* in similar yet different senses. For the purpose of clarification, the Xilinx documentation refers to the terms *attributes* and *constraints* as defined below.

Attributes

An attribute is a property associated with a device architecture primitive component that generally affects an instantiated component's functionality or implementation. Attributes are passed as follows:

- in VHDL, by means of generic maps
- in Verilog, by means of defparams or inline parameter passing while instantiating the primitive component

Examples of attributes are:

- the INIT property on a LUT4 component
- the CLKFX_DIVIDE property on a DCM

All attributes are described in the appropriate Xilinx *Libraries Guide* as a part of the primitive component description.

Synthesis Constraints

Synthesis constraints direct the synthesis tool optimization technique for a particular design or piece of HDL code. They are either embedded within the VHDL or Verilog code, or within a separate synthesis constraints file. Examples of synthesis constraints are:

- USE_DSP48 (XST)
- RAM_STYLE (XST)

Synthesis constraints are documented as follows:

- XST constraints are documented in the Xilinx *XST User Guide*.
- Synthesis constraints for other synthesis tools should be documented in the vendor's documentation. For more information on synthesis constraints, see your synthesis tool documentation.

Implementation Constraints

Implementation constraints are instructions given to the FPGA implementation tools to direct the mapping, placement, timing or other guidelines for the implementation tools to follow while processing an FPGA design. Implementation constraints are generally placed in the UCF file, but may exist in the HDL code, or in a synthesis constraints file. Examples of implementation constraints are:

- LOC (placement) constraints
- PERIOD (timing) constraints

For more information about implementation constraints, see the Xilinx *Constraints Guide*.

Passing Attributes

Attributes are properties that are attached to Xilinx primitive instantiations in order to specify their behavior. They are generally passed by specifying a generic map specifying the attribute or a defparam or inline parameter passing in Verilog. By passing the attribute in this way, you can make sure that it is properly passed to both synthesis and simulation.

VHDL Primitive Attribute Example

The following VHDL code shows an example of setting the INIT primitive attribute for an instantiated RAM16X1S which will specify the initial contents of this RAM symbol to the hexadecimal value of A1B2.

```
-- RAM16X1S: 16 x 1 posedge write distributed  => LUT RAM
--           All FPGA
-- Xilinx HDL Language Template

small_ram_inst : RAM16X1S
generic map (
   INIT => X"A1B2")
port map (
   O => ram_out,       -- RAM output
   A0 => addr(0),      -- RAM address[0] input
   A1 => addr(1),      -- RAM address[1] input
   A2 => addr(2),      -- RAM address[2] input
   A3 => addr(3),      -- RAM address[3] input
   D => data_in,       -- RAM data input
   WCLK => clock,      -- Write clock input
   WE => we            -- Write enable input
);

-- End of small_ram_inst instantiation
```

Verilog Primitive Attribute Example

The following Verilog example shows an instantiated IBUFDS symbol in which the DIFF_TERM and IOSTANDARD are specified as "FALSE" and "LVDS_25" respectively.

```
// IBUFDS: Differential Input Buffer
//         Virtex-II/II-Pro/4, Spartan-3/3E
// Xilinx HDL Language Template

IBUFDS #(
// Differential Termination (Virtex-4 only)
   .DIFF_TERM("FALSE"),
   .IOSTANDARD("LVDS_25")    // Specify the input I/O standard
) serial_data_inst (
   .O(diff_data),   // Clock buffer output
   .I(DATA_P),  // Diff_p clock buffer input
   .IB(DATA_N) // Diff_n clock buffer input
);

// End of serial_data_inst instantiation
```

A constraint can be attached to HDL objects in your design, or specified from a separate constraints file. You can pass constraints to HDL objects in two ways:

- Predefine data that describes an object
- Directly attach an attribute to an HDL object

Predefined attributes can be passed with a COMMAND file or constraints file in your synthesis tool, or you can place attributes directly in your HDL code. This section illustrates passing attributes in HDL code only. For information on passing attributes via the command file, see your synthesis tool documentation.

Most vendors adopt identical syntax for passing attributes in VHDL, but not in Verilog. The following examples illustrate the VHDL syntax

VHDL Synthesis Attribute Examples

The following are examples of VHDL attributes:

- “Attribute Declaration”
- “Attribute Use on a Port or Signal”
- “Attribute Use on an Instance”
- “Attribute Use on a Component”

Attribute Declaration

attribute *attribute_name* **:** *attribute_type;*

Attribute Use on a Port or Signal

attribute *attribute_name* **of** *object_name* **: signal is** *attribute_value*

Example:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity d_register is
  port (
        CLK, DATA: in STD_LOGIC;
        Q: out STD_LOGIC);
  attribute FAST : string;
  attribute FAST of Q : signal is "true";
end d_register;
```

Attribute Use on an Instance

attribute *attribute_name* **of** *object_name* **: label is** *attribute_value*

Example:

```
architecture struct of spblkrams is
attribute LOC: string;
attribute LOC of  SDRAM_CLK_IBUFG: label is "AA27";
Begin

   -- IBUFG: Single-ended global clock input buffer
   --        All FPGA
   -- Xilinx HDL Language Template

   SDRAM_CLK_IBUFG : IBUFG
   generic map (
      IOSTANDARD => "DEFAULT")
   port map (
      O => SDRAM_CLK_o, -- Clock buffer output
      I => SDRAM_CLK_i  -- Clock buffer input
   );

   -- End of IBUFG_inst instantiation
```

Attribute Use on a Component

attribute *attribute_name* **of** *object_name* **: component is** *attribute_value*

Example:

```
architecture xilinx of tenths_ex is
attribute black_box : boolean;
component tenths
  port (
        CLOCK : in STD_LOGIC;
        CLK_EN : in STD_LOGIC;
        Q_OUT : out STD_LOGIC_VECTOR(9 downto 0)
        );
end component;
attribute black_box of tenths : component is true;
begin
```

Verilog Synthesis Attribute Examples

Following are examples of attribute passing in Verilog via a method called meta-comments. Attribute passing in Verilog is synthesis tool specific.

- "Attribute Use in Precision Synthesis Syntax"
- "Synthesis Attribute Use in Synplify Syntax"

Attribute Use in Precision Synthesis Syntax

//pragma **attribute** *object_name attribute_name attribute_value*

Examples:

```
my_fd U1 (
   .D(data),
   .CLK(clock),
   .Q(data_out));
//pragma attribute U1 hierarchy "preserve";
```

Synthesis Attribute Use in Synplify Syntax

```
/* synthesis directive */
/* synthesis attribute_name=value>*/
```

Examples:

```
// FDCE: Single Data Rate D Flip-Flop with Asynchronous Clear and
   //       Clock Enable (posedge clk).  All families.
   // Xilinx HDL Language Template
   FDCE #(
      .INIT(1'b0)    // Initial value of register (1'b0 or 1'b1)
   ) U2 (
      .Q(q1),        // Data output
      .C(clk),       // Clock input
      .CE(ce),       // Clock enable input
      .CLR(rst),     // Asynchronous clear input
      .D(q0)         // Data input
   ) /* synthesis rloc="r1c0" */;

   // End of FDCE_inst instantiation
```

or

```
module MY_BUFFER(I, O) /* synthesis black_box */
  input I;
  output O;
endmodule
```

Verilog 2001 provides a uniform syntax of passing attributes. Since the attribute is declared immediately before the object is declared, the object name is not mentioned during the attribute declaration.

```
(* attribute_name = "attribute_value" *)
Verilog_object;
```

For example:

```
// FDCE: Single Data Rate D Flip-Flop with Asynchronous Clear and
// Clock Enable (posedge clk). All families.
// Xilinx HDL Language Template

(* RLOC = "R1C0.S0" *) FDCE #(
   .INIT(1'b0) // Initial value of register (1'b0 or 1'b1)
) U2 (
   .Q(q1), // Data output
   .C(clk), // Clock input
   .CE(ce), // Clock enable input
   .CLR(rst), // Asynchronous clear input
   .D(q0) // Data input
);

// End of FDCE_inst instantiation
```

This method of attribute passing is not supported by all synthesis tools. For more information, see your synthesis tool documentation.

Synthesis Tool Naming Conventions

Some net and logic names are preserved and some are altered by the synthesis tools during the synthesis process. This may result in a netlist that is hard to read or trace back to the original code.

This section discusses how different synthesis tools generate names from your VHDL/Verilog codes. This helps you determine how nets and component names appearing in the EDIF netlist relate to the original input design. It also helps determine how nets and names during your after-synthesis design view of the VHDL/Verilog source relate to the original input design.

Note: The following naming conventions apply to inferred logic. The names of instantiated components and their connections, and port names are preserved during synthesis.

LeonardoSpectrum and Precision Synthesis Naming Styles

Register instance: reg_*outputsignal*

Output of register: preserved, except if the output is also an external port of the design. In this case, it is *signal*_dup0

Clock buffer/ibuf: *driversignal*_ibuf

Output of clock buffer/ibuf: *driversignal*_int

3-state instance: tri_*outputsignal*

Driver and output of 3-state: preserved

Hierarchy notation: '_'

Other names are machine generated.

Synplify Naming Styles

Register instance: output_signal

Output of register: output_signal

Clock buffer instance/ibuf: *portname*_ibuf

Output of clock buffer: *clkname*_c

Output/inout 3-state instance: *outputsignal*_obuft or *outputsignal*_iobuf

Internal 3-state instance: un*n*_*signalname*_tb (*n* can be any number), or *signalname*_tb

Output of 3-state driving an output/inout : name of port

Output of internal 3-state: *signalname*_tb_*number*

RAM instance and its output

- Dual Port RAM:

  ram instance: *memoryname*_*n*.I_*n*

  ram output : DPO->*memoryname*_*n*.rout_bus, SPO->*memory_name*_*n*.wout_bus

- Single Port RAM:

  ram instance: *memoryname*.I_*n*

  ram output: *memoryname*

- Single Port Block SelectRAM™:

ram_instance: *memoryname*.I_*n*

ram output: *memoryname*

- Dual Port Block SelectRAM:

  ram_instance: *memory_name*.I_*n*

  ram output: *memoryname* [the output that is used]

Hierarchy delimiter is usually a ".", however when syn_hier="hard", the hierarchy delimiter in the edif is "/"

Other names are machine generated.

Specifying Constants

Use constants in your design to substitute numbers to more meaningful names. Constants make a design more readable and portable.

Using Constants and Parameters to Clarify Code

Specifying constants can be a form of in-code documentation that allows for easier understanding of code function. For VHDL, Xilinx generally recommends not to use variables for constants in your code. Define constant numeric values in your code as constants and use them by name. For Verilog, parameters can be used as constants in the code in a similar manner. This coding convention allows you to easily determine if several occurrences of the same literal value have the same meaning. In the following code examples, the OPCODE values are declared as constants/parameters, and the names refer to their function. This method produces readable code that may be easier to understand and modify.

VHDL Example

```
constant ZERO    : STD_LOGIC_VECTOR (1 downto 0):="00";
constant A_AND_B: STD_LOGIC_VECTOR (1 downto 0):="01";
constant A_OR_B : STD_LOGIC_VECTOR (1 downto 0):="10";
constant ONE     : STD_LOGIC_VECTOR (1 downto 0):="11";

process (OPCODE, A, B)
begin
  if (OPCODE = A_AND_B)then OP_OUT <= A and B;
    elsif (OPCODE = A_OR_B) then
        OP_OUT <= A or B;
    elsif (OPCODE = ONE) then
        OP_OUT <= '1';
    else
        OP_OUT <= '0';
  end if;
end process;
```

Verilog Example

```
//Using parameters for OPCODE functions
parameter ZERO = 2'b00;
parameter A_AND_B = 2'b01;
parameter A_OR_B = 2'b10;
parameter ONE = 2'b11;
```

```
always @ (*)
  begin
    if (OPCODE == ZERO)
        OP_OUT = 1'b0;
    else if (OPCODE == A_AND_B)
        OP_OUT=A&B;
    else if (OPCODE == A_OR_B)
        OP_OUT = A|B;
    else
        OP_OUT = 1'b1;
  end
```

Using Generics and Parameters to Specify Dynamic Bus and Array Widths

To specify a dynamic or paramatizable bus width for a VHDL or Verilog design module:

1. Define a generic (VHDL) or parameter (Verilog).
2. Use the generic (VHDL) or parameter (Verilog) to define the bus width of a port or signal.

The generic (VHDL) or parameter (Verilog) can contain a default which can be overridden by the instantiating module. This can make the code easier to reuse, as well as making it more readable.

VHDL Example

```
-- FIFO_WIDTH data width(number of bits)
-- FIFO_DEPTH by number of address bits
-- for the FIFO RAM i.e. 9 -> 2**9 -> 512 words
-- FIFO_RAM_TYPE: BLOCKRAM or DISTRIBUTED_RAM
-- Note: DISTRIBUTED_RAM suggested for FIFO_DEPTH
-- of 5 or less

entity async_fifo is
   generic (FIFO_WIDTH: integer := 16;)
            FIFO_DEPTH: integer := 9; FIFO_RAM_TYPE: string := "BLOCKRAM");  port ( din
: in std_logic_vector(FIFO_WIDTH-1 downto 0);
           rd_clk : in std_logic;
           rd_en  : in std_logic;
           ainit  : in std_logic;
           wr_clk : in std_logic;
           wr_en  : in std_logic;
           dout   : out std_logic_vector(FIFO_WIDTH-1 downto 0) := (others=> '0');
           empty  : out std_logic := '1';
           full   : out std_logic := '0';
           almost_empty : out std_logic := '1';
           almost_full  : out std_logic := '0');
end async_fifo;

architecture BEHAVIORAL of async_fifo is

   type ram_type is array ((2**FIFO_DEPTH)-1 downto 0) of std_logic_vector (FIFO_WIDTH-1
downto 0);
```

Verilog Example

```
-- FIFO_WIDTH data width(number of bits)
-- FIFO_DEPTH by number of address bits
-- for the FIFO RAM i.e. 9 -> 2**9 -> 512 words
-- FIFO_RAM_TYPE: BLOCKRAM or DISTRIBUTED_RAM
-- Note: DISTRIBUTED_RAM suggested for FIFO_DEPTH
-- of 5 or less

module async_fifo (din, rd_clk, rd_en, ainit, wr_clk, wr_en, dout, empty, full,
almost_empty, almost_full, wr_ack);

   parameter FIFO_WIDTH = 16;
   parameter FIFO_DEPTH = 9;
  parameter FIFO_RAM_TYPE = "BLOCKRAM";

   input [FIFO_WIDTH-1:0] din;
   input                  rd_clk;
   input                  rd_en;
   input                  ainit;
   input                  wr_clk;
   input                  wr_en;

   output reg [FIFO_WIDTH-1:0] dout;
   output                      empty;
   output                      full;
   output                      almost_empty;
   output                      almost_full;
   output reg                  wr_ack;

   reg [FIFO_WIDTH-1:0] fifo_ram [(2**FIFO_DEPTH)-1:0];
…
```

Choising Data Type

Note: This section applies to VHDL only.

Use the Std_logic (IEEE 1164) standards for hardware descriptions when coding your design. These standards are recommended for the following reasons.

1. Applies as a wide range of state values

   Std_logic has nine different values that represent most of the states found in digital circuits.

2. Allows indication of all possible logic states within the FPGA

   a. Std_logic not only allows specification of logic high (1) and logic low (0) but also whether a pullup (H) or pulldown (L) is used, or whether an output in high impedance (Z).

   b. Std_logic allows the specification of unknown values (X) due to possible contention, timing violations, or other occurrences, or whether an input or signal is unconnected (U).

   c. Std_logic allows a more realistic representation of the FPGA logic for both synthesis and simulation which many times will allow more accurate results.

3. Easily performs board-level simulation

For example, if you use an integer type for ports for one circuit and standard logic for ports for another circuit, your design can be synthesized; however, you need to perform time-consuming type conversions for a board-level simulation.

The back-annotated netlist from Xilinx implementation is in Std_logic. If you do not use Std_logic type to drive your top-level entity in the test bench, you cannot reuse your functional test bench for timing simulation. Some synthesis tools can create a wrapper for type conversion between the two top-level entities; however, this is not recommended by Xilinx.

Declaring Ports

Xilinx recommends that you use the Std_logic type for all entity port declarations. This type makes it easier to integrate the synthesized netlist back into the design hierarchy without requiring conversion functions for the ports. The following VHDL example uses the Std_logic type for port declarations.

```
Entity alu is
  port(
        A   : in STD_LOGIC_VECTOR(3 downto 0);
        B   : in STD_LOGIC_VECTOR(3 downto 0);
        CLK : in STD_LOGIC;
        C   : out STD_LOGIC_VECTOR(3 downto 0)
        );
end alu;
```

If a top-level port is specified as a type *other* than STD_LOGIC, then software generated simulation models (such as timing simulation) may no longer bind to the test bench. This is due to the following factors:

- Type information can not be stored for a particular design port.
- Simulation of FPGA hardware requires the ability to specify the values of STD_LOGIC such as high-Z (3-state), and X unknown in order to properly display hardware behavior.

Xilinx recommends that you not declare arrays as ports. This information can not be properly represented or re-created. For this reason, Xilinx recommends that you use STD_LOGIC and STD_LOGIC_VECTOR for all top-level port declarations.

Using Arrays in Port Declarations

VHDL allows you to declare a port as an array type. However, Xilinx strongly recommends that you ***never*** do this, for the following reasons.

Incompatibility with Verilog

There is no equivalent way to declare a port as an array type in Verilog. Verilog does not allow ports to be declared as arrays. This limits portability across languages. It also limits as the ability to use the code for mixed-language projects.

Inability to Store and Re-Create Original Declaration of the Array

When you declare a port as an array type in VHDL, the original declaration of the array can not be stored and re-created. The EDIF netlist format, as well as the Xilinx database, are unable to store the original type declaration for the array.

Consequently, when NetGen or another netlister attempts to re-create the design, there is no information as to how the port was originally declared. The resulting netlist generally has mis-matched port declarations and resulting signal names. This is true not only for the top-level port declarations, but also for the lower-level port declarations of a hierarchical design since the KEEP_HIERARCHY mechanism can be used to attempt to preserve those net names.

Mis-Correlation of Software Pin Names

Another effect array port declarations can have is a mis-correlation of the software pin names from the original source code. Since the software must treat each I/O as a separate label, the corresponding name for the broken-out port may not match what is expected by the user. This makes design constraint passing, design analysis and design reporting more difficult to understand.

Minimizing the Use of Ports Declared as Buffers

Do not use buffers when a signal is used internally and as an output port. In the following VHDL example, signal C is used internally and as an output port.

```
Entity alu is
  port(
       A   : in STD_LOGIC_VECTOR(3 downto 0);
       B   : in STD_LOGIC_VECTOR(3 downto 0);
       CLK : in STD_LOGIC;
       C   : buffer STD_LOGIC_VECTOR(3 downto 0) );
end alu;
architecture BEHAVIORAL of alu is
begin
  process begin
    if (CLK'event and CLK='1') then
       C <= UNSIGNED(A) + UNSIGNED(B) UNSIGNED(C);
    end if;
  end process;
end BEHAVIORAL;
```

Because signal C is used both internally and as an output port, every level of hierarchy in your design that connects to port C must be declared as a buffer. However, buffer types are not commonly used in VHDL designs because they can cause problems during synthesis.

To reduce the amount of buffer coding in hierarchical designs, you can insert a dummy signal and declare port C as an output, as shown in the following VHDL example.

```
Entity alu is
  port(
       A   : in STD_LOGIC_VECTOR(3 downto 0);
       B   : in STD_LOGIC_VECTOR(3 downto 0);
       CLK : in STD_LOGIC;
       C   : out STD_LOGIC_VECTOR(3 downto 0)
       );
end alu;

architecture BEHAVIORAL of alu is
-- dummy signal
  signal C_INT : STD_LOGIC_VECTOR(3 downto 0);
  begin
    C <= C_INT;
    process begin
```

```
         if (CLK'event and CLK='1') then
            C_INT <= A and B and C_INT;
         end if;
      end process;
end BEHAVIORAL;
```

Comparing Signals and Variables (VHDL only)

You can use signals and variables in your designs. Signals are similar to hardware and are not updated until the end of a process. Variables are immediately updated and, as a result, can affect the functionality of your design. Xilinx recommends using signals for hardware descriptions; however, variables allow quick simulation. The following VHDL examples show a synthesized design that uses signals and variables, respectively. These examples are shown implemented with gates in the "Gate Implementation of XOR_VAR" and "Gate Implementation of XOR_SIG" figures.

Note: If you assign several values to a signal in one process, only the final value is used. When you assign a value to a variable, the assignment takes place immediately. A variable maintains its value until you specify a new value.

Using Signals (VHDL)

```
-- XOR_SIG.VHD
Library IEEE;
use IEEE.std_logic_1164.all;

entity xor_sig is
  port (
        A, B, C: in  STD_LOGIC;
        X, Y: out STD_LOGIC
        );
end xor_sig;

architecture SIG_ARCH of xor_sig is
  signal D: STD_LOGIC;
  begin
    SIG:process (A,B,C)
    begin
      D <= A; -- ignored !!
      X <= C xor D;
      D <= B; -- overrides !!
      Y <= C xor D;
    end process;
end SIG_ARCH;
```

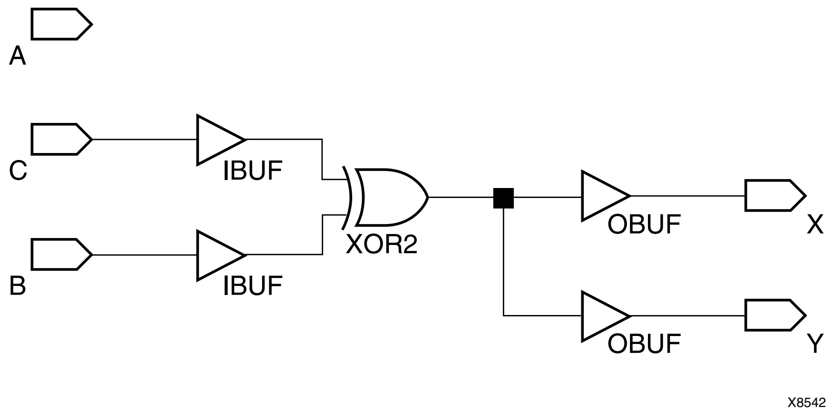

Figure 3-1: **Gate implementation of XOR_SIG**

Using Variables (VHDL)

```
-- XOR_VAR.VHD

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity xor_var is
  port (
        A, B, C: in  STD_LOGIC;
        X, Y:    out STD_LOGIC
        );
end xor_var;

architecture VAR_ARCH of xor_var is
begin
  VAR:process (A,B,C)
    variable D: STD_LOGIC;
    begin
      D := A;
      X <= C xor D;
      D := B;
      Y <= C xor D;
    end process;
end VAR_ARCH;
```

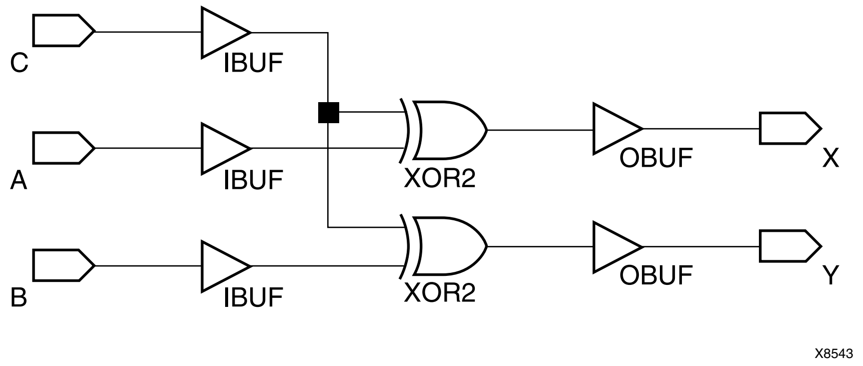

Figure 3-2: **Gate Implementation of XOR_VAR**

Using `timescale

Note: This section applies to Verilog only.

All Verilog test bench and source files should contain a `timescale directive, or reference an include file containing a `timescale directive. Place this near the beginning of the source file, and before any module or other design unit definitions in the source file. Xilinx recommends that you use a `timescale with the resolution of 1 ps. Some Xilinx primitive components such as DCM require a 1ps resolution in order to properly work in either functional or timing simulation. Little or no simulation speed difference should be seen with the use of a 1ps resolution over a more coarse resolution.

The following `timescale directive is a typical default:

```
`timescale 1ns / 1ps
```

Coding for Synthesis

VHDL and Verilog are hardware description and simulation languages that were not originally intended as inputs to synthesis. Therefore, many hardware description and simulation constructs are not supported by synthesis tools. In addition, the various synthesis tools use different subsets of VHDL and Verilog. VHDL and Verilog semantics are well defined for design simulation. The synthesis tools must adhere to these semantics to make sure that designs simulate the same way before and after synthesis. Follow the guidelines in the following sections to create code that simulates the same way before and after synthesis.

Omit the Use of Delays in Synthesis Code

Do not use the **`Wait for XX ns`** (VHDL) or the **`#XX`** (Verilog) statement in your code. **`XX`** specifies the number of nanoseconds that must pass before a condition is executed. This statement does not synthesize to a component. In designs that include this construct, the functionality of the simulated design does not always match the functionality of the synthesized design. VHDL and Verilog examples of the **`Wait for XX ns`** statement are as follows.

- VHDL
  ```
  wait for XX ns;
  ```
- Verilog
  ```
      #XX;
  ```

Also, do not use the ...After XX ns statement in your VHDL code or the Delay assignment in your Verilog code. Examples of these statements are as follows.

- VHDL
  ```
  (Q <=0 after XX ns)
  ```
- Verilog
  ```
  assign #XX Q=0;
  ```

XX specifies the number of nanoseconds that must pass before a condition is executed. This statement is usually ignored by the synthesis tool. In this case, the functionality of the simulated design does not match the functionality of the synthesized design.

Order and Group Arithmetic Functions

The ordering and grouping of arithmetic functions can influence design performance. For example, the following two VHDL statements are not necessarily equivalent.

```
ADD <= A1 + A2 + A3 + A4;
ADD <= (A1 + A2) + (A3 + A4);
```

For Verilog, the following two statements are not necessarily equivalent.

```
ADD = A1 + A2 + A3 + A4;
ADD = (A1 + A2) + (A3 + A4);
```

The first statement cascades three adders in series. The second statement creates two adders in parallel: A1 + A2 and A3 + A4. In the second statement, the two additions are evaluated in parallel and the results are combined with a third adder. RTL simulation results are the same for both statements, however, the second statement results in a faster circuit after synthesis (depending on the bit width of the input signals).

Although the second statement generally results in a faster circuit, in some cases, you may want to use the first statement. For example, if the A4 signal reaches the adder later than the other signals, the first statement produces a faster implementation because the cascaded structure creates fewer logic levels for A4. This structure allows A4 to catch up to the other signals. In this case, A1 is the fastest signal followed by A2 and A3; A4 is the slowest signal.

Most synthesis tools can balance or restructure the arithmetic operator tree if timing constraints require it. However, Xilinx recommends that you code your design for your selected structure.

Use of Resets and Synthesis Optimization

Xilinx FPGA devices have an abundance of flip-flops. All architectures support the use of an asynchronous reset for those registers and latches. Even though this capability exists, Xilinx does not generally recommend that you code for it. The use of asynchronous resets may not only result in more difficult timing analysis, but may also result in a less optimal optimization by the synthesis tool.

The timing hazard an asynchronous reset poses on a synchronous system is generally well known; however, less well known is the optimization trade-off the asynchronous reset poses on a design.

The use of any reset when inferring shift registers will prohibit the inference of the Shift Register LUT component. All current Xilinx FPGA architectures contain LUTs that may be configured to act as a 16-bit shift register called an SRL (Shift Register LUT).

The SRL is a very efficient structure for building static and variable length shift registers; however, a reset (either synchronous or asynchronous) would preclude the use of this component. This generally leads to a less efficient structure using a combination of registers and, sometimes, logic.

The choice between synchronous and asynchronous resets can also change the choices of how registers are used within larger blocks of IP in the FPGA. For instance, the DSP48 in the Virtex-4 family has several registers within the block which, if used, can not only result in a possible substantial area savings, but can also improve the overall performance of the circuit.

The DSP48 has only a synchronous reset. This means if a synchronous reset is inferred in registers around logic that could be packed into a DSP48, the registers can also be packed into the component as well, resulting in a smaller and generally faster design. If, however, an asynchronous reset is used, the register must remain outside of the block, resulting in a less optimal design. Similar optimization applies to the block RAM registers and other components within the FPGA.

The flip-flops within the FPGA are configurable to be either an asynchronous set/reset, or a synchronous set/reset. If an asynchronous reset is described in the code, the synthesis tool must configure the flip-flop to use the asynchronous reset/preset. This precludes the use of any other signals using this resource.

If, however, a synchronous reset (or no reset at all) is described for the flip-flop, the synthesis tool can configure the set/reset as a synchronous operation. Doing so allows the synthesis tool to use this resource as a set/reset from the described code. It may also use this resource to break up the data path. This may result in fewer resources and shorter data paths to the register. Details of these optimizations depend on the code and synthesis tools used.

To illustrate how the use of asynchronous resets can inhibit optimization, see the following code examples.

VHDL Example One

```
process (CLK, RST)
begin
   if (RST = '1') then
      Q <= '0';
   elsif (CLK'event and CLK = '1') then
      Q <= A or (B and C and D and E);
   end if;
end process;
```

Verilog Example One

```
always @(posedge CLK, posedge RST)
   if (RESET)
      Q <= 1'b0;
   else
      Q <= A | (B & C & D & E);
```

To implement the following code, the synthesis tool has no choice but to infer two LUTs for the data path since there are 5 signals used to create this logic. A possible implementation of the above code can now look like:

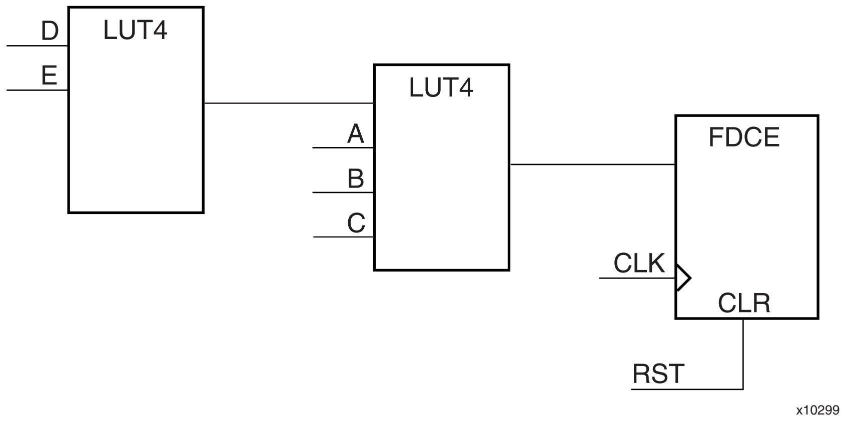

Figure 3-3: **Verilog Example One**

If however, this same code is re-written for a synchronous reset, see the following examples.

VHDL Example Two

```
process (CLK)
begin
   if (CLK'event and CLK = '1') then
      if (RST = '1') then
         Q <= '0';
      else
         Q <= A or (B and C and D and E);
      end if;
   end if;
end process;
```

Verilog Example Two

```
always @(posedge CLK)
   if (RESET)
      Q <= 1'b0;
   else
      Q <= A | (B&C&D&E);
```

The synthesis tool now has more flexibility as to how this function can be represented. A possible implementation of the above code can now look like:

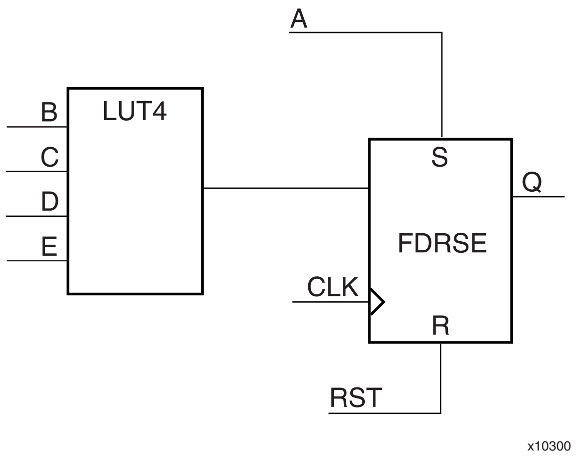

Figure 3-4: **Verilog Example Two**

In the above implementation, the synthesis tool can identify that any time A is active high, Q is always a logic one. With the register now configured with the set/reset as a synchronous operation, the set is now free to be used as part of the synchronous data path. This reduces the amount of logic necessary to implement the function, as well as reducing the data path delays for the D and E signals. Logic could have also been shifted to the reset side as well if the code was written in a way that was a more beneficial implementation.

VHDL Example Three

Now consider the following addition to the above example:

```
process (CLK, RST)
begin
   if (RST = '1') then
      Q <= '0';
   elsif (CLK'event and CLK = '1') then
      Q <= (F or G or H) and (A or (B and C and D and E));
   end if;
end process;
```

Verilog Example Three

```
always @(posedge CLK, posedge RST)
   if (RESET)
      Q <= 1'b0;
   else
      Q <= (F|G|H) & (A | (B&C&D&E));
```

Now that there are eight signals that contribute to the logic function, a minimum of 3 LUTs would be needed to implement this function. A possible implementation of the above code can now look like:

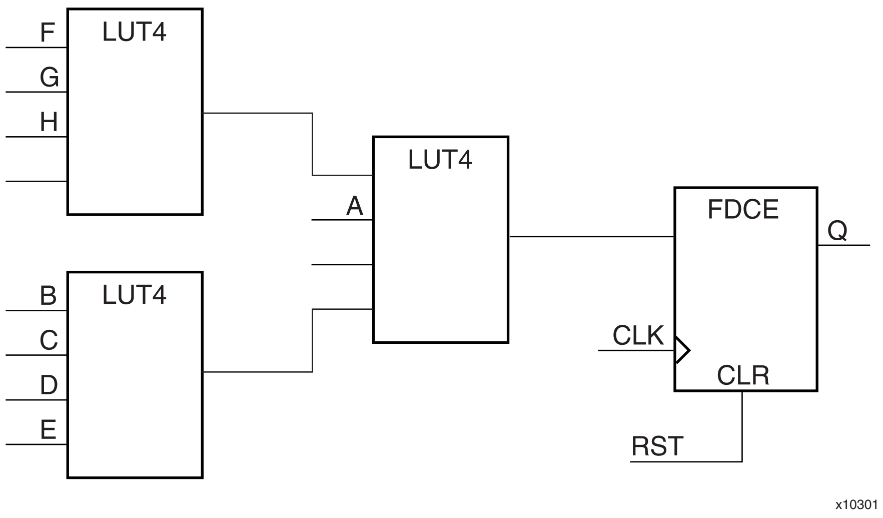

Figure 3-5: **Verilog Example Three**

VHDL Example Four

If the same code is written with a synchronous reset:

```
process (CLK)
begin
   if (CLK'event and CLK = '1') then
      if (RST = '1') then
         Q <= '0';
      else
         Q <= (F or G or H) and (A or (B and C and D and E));
      end if;
   end if;
end process;
```

Verilog Example Four

```
always @(posedge CLK)
   if (RESET)
      Q <= 1'b0;
   else
      Q <= (F|G|H) & (A | (B&C&D&E));
```

A possible implementation of the above code can now look like:

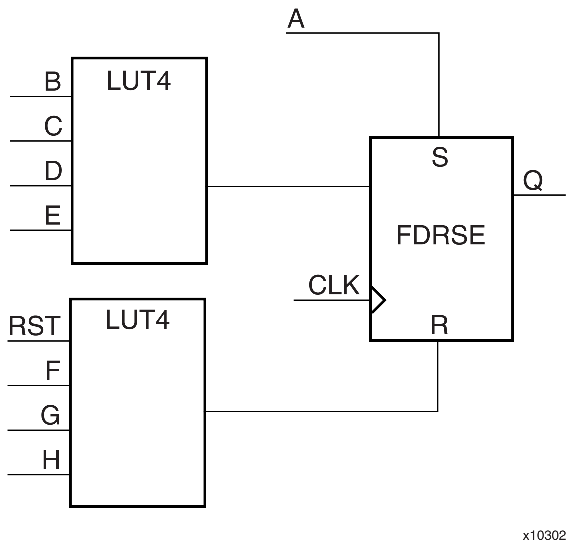

Figure 3-6: **Verilog Example Four**

Again the resulting implementation above not only uses fewer LUTs to implement the same logic function but also could potentially result in a faster design due to the reduction of logic levels for practically every signal that creates this function. These examples are simple in nature but do illustrate the point of how the use of asynchronous resets force all synchronous data signals on the data input to the register thus resulting in a potentially less optimal implementation.

In general, the more signals that fan into a logic function, the more effective the use of synchronous sets/resets (or no resets at all) can be in minimizing logic resources and in maximizing performance of the design.

Considerations When Not Using Asynchronous Resets in a Design

Many users familiar with ASIC designs include a global asynchronous reset signal in the design. This not only allows for proper initialization of the end device, but also aids in simulation of the design. As stated above, however, this practice can have a negative consequence for the end design optimization, and is not necessary.

All Xilinx FPGAs have a dedicated asynchronous reset called GSR (Global Set Reset). GSR is automatically asserted at the end of FPGA configuration, regardless of the coding style. For gate-level simulation, this GSR signal is also inserted to mimic this operation to allow accurate simulation of the initialized design as it happens in the silicon. Adding another asynchronous reset to the actual code only duplicates this dedicated feature. It is not necessary for device initialization or simulation initialization.

Comparing If Statement and Case Statement

The If statement generally produces priority-encoded logic and the Case statement generally creates balanced logic. An If statement can contain a set of different expressions while a Case statement is evaluated against a common controlling expression. In general, use the Case statement for complex decoding and use the If statement for speed critical paths.

Most synthesis tools can determine whether the *if-elsif* conditions are mutually exclusive, and do not create extra logic to build the priority tree. Keep the following in mind when writing *if* statements.

- Make sure that all outputs are defined in all branches of an *if* statement. If not, it can create latches or long equations on the CE signal. A good way to prevent this is to have default values for all outputs before the if statements.
- Remember that limiting the number of input signals into an *if* statement can reduce the number of logic levels. If there are a large number of input signals, see if some of them can be pre-decoded and registered before the *if* statement.
- Avoid bringing the dataflow into a complex *if* statement. Only control signals should be generated in complex *if-else* statements.

4–to–1 Multiplexer Design with If Construct

The following examples use an *if* construct in a 4–to–1 multiplexer design.

VHDL Example

```
-- IF_EX.VHD
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity if_ex is
  port (
        SEL: in STD_LOGIC_VECTOR(1 downto 0);
        A,B,C,D: in STD_LOGIC;
        MUX_OUT: out STD_LOGIC);
end if_ex;

architecture BEHAV of if_ex is
begin

  IF_PRO: process (SEL,A,B,C,D)
    begin
      if (SEL="00") then MUX_OUT <= A;
      elsif (SEL="01") then
          MUX_OUT <= B;
      elsif (SEL="10") then
          MUX_OUT <= C;
      elsif (SEL="11") then
          MUX_OUT <= D;
      else
          MUX_OUT <= '0';
      end if;
    end process; --END IF_PRO
end BEHAV;
```

Verilog Example

```
/////////////////////////////////////////////////
// IF_EX.V                                     //
// Example of a If statement showing a         //
// mux created using priority encoded logic    //
// HDL Synthesis Design Guide for FPGA devices //
/////////////////////////////////////////////////

module if_ex (
input A, B, C, D,
input [1:0] SEL,
output reg MUX_OUT);

always @ (*)
  begin
    if (SEL == 2'b00)
        MUX_OUT = A;
    else if (SEL == 2'b01)
        MUX_OUT = B;
    else if (SEL == 2'b10)
        MUX_OUT = C;
    else if (SEL == 2'b11)
        MUX_OUT = D;
    else
        MUX_OUT = 0;
    end
endmodule
```

4–to–1 Multiplexer Design with Case Construct

The following VHDL and Verilog examples use a Case construct for the same multiplexer. Figure 3-7 shows the implementation of these designs. In these examples, the Case implementation requires only one slice while the If construct requires two slices in some synthesis tools. In this case, design the multiplexer using the Case construct because fewer resources are used and the delay path is shorter.

When writing case statements, make sure all outputs are defined in all branches.

VHDL Example

```
-- CASE_EX.VHD
-- May 2001
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity case_ex is
  port (
        SEL : in STD_LOGIC_VECTOR(1 downto 0);
        A,B,C,D: in STD_LOGIC;
        MUX_OUT: out STD_LOGIC);
end case_ex;

architecture BEHAV of case_ex is
begin
  CASE_PRO: process (SEL,A,B,C,D)
  begin
      case SEL is
```

```
        when “00” => MUX_OUT <= A;
        when “01” => MUX_OUT <= B;
        when “10” => MUX_OUT <= C;
        when “11” => MUX_OUT <= D;
        when others => MUX_OUT <= '0';
      end case;
  end process; --End CASE_PRO
end BEHAV;
```

Verilog Example

```
/////////////////////////////////////////////////
// CASE_EX.V                                   //
// Example of a Case statement showing         //
// A mux created using parallel logic          //
// HDL Synthesis Design Guide for FPGA devices //
/////////////////////////////////////////////////
module case_ex (
input A, B, C, D,
input [1:0] SEL,
output reg MUX_OUT);

always @ (*)
  begin
    case (SEL)
      2'b00: MUX_OUT = A;
      2'b01: MUX_OUT = B;
      2'b10: MUX_OUT = C;
      2'b11: MUX_OUT = D;
      default: MUX_OUT = 0;
    endcase
  end
endmodule
```

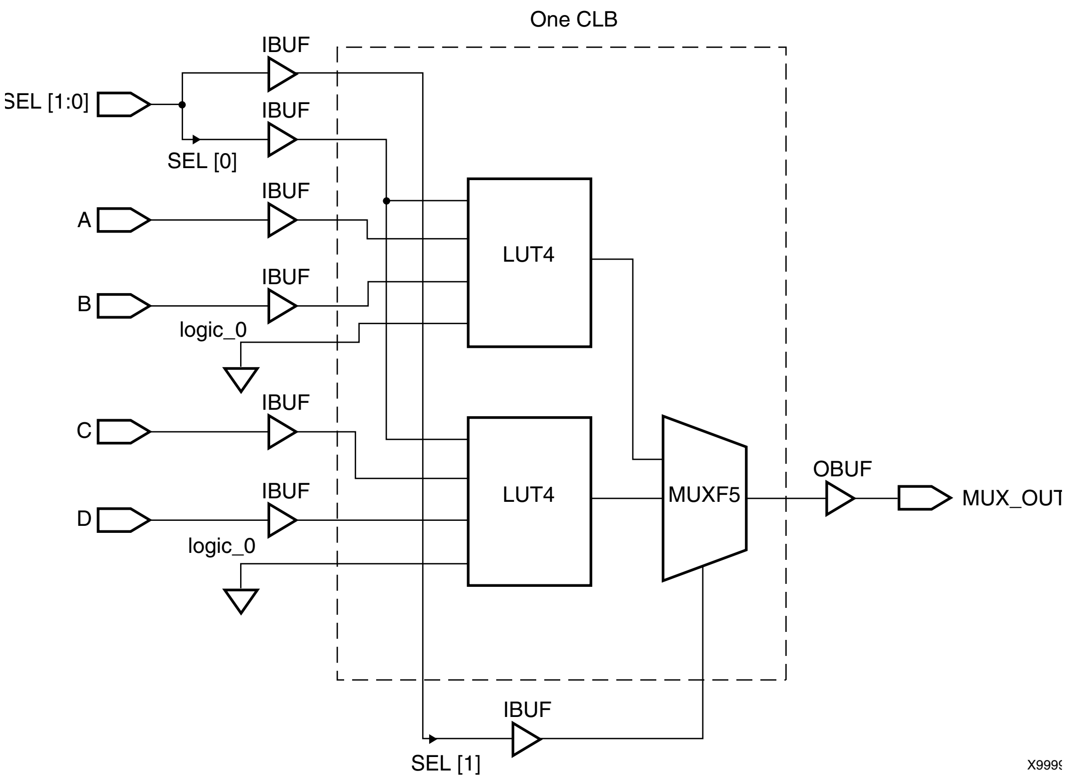

Figure 3-7: **Case_Ex Implementation**

Implementing Latches and Registers

Synthesizers infer latches from incomplete conditional expressions, such as:

- an If statement without an Else clause
- an intended register without a rising edge or falling edge construct

Many times this is done by mistake. However, the design may still appear to function properly in simulation. This can be problematic for FPGA designs, since timing for paths containing latches can be difficult and challenging to analyze. Synthesis tools usually report in the log files when a latch is inferred to alert you to this occurrence.

For FPGA designs, Xilinx generally recommends that you avoid the use of latches, even though they can be properly implemented in the device, due to the more difficult timing analyses that take place when latches are used.

Latch Inference

Following are examples of latch inference.

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity d_latch is
  port (
        GATE, DATA: in STD_LOGIC;
        Q: out STD_LOGIC);
end d_latch;

architecture BEHAV of d_latch is
begin
  LATCH: process (GATE, DATA)
  begin
    if (GATE = '1') then
        Q <= DATA;
    end if;
end process;

end BEHAV;
```

Verilog Example

```
module d_latch (
   input GATE, DATA,
   output reg Q
);

   always @ (*)
      if (GATE)
         Q = DATA;

endmodule
```

Converting Latch to D Register

If your intention is to not infer a latch, but rather to infer a D register, then the following code is the latch code example, modified to infer a D register.

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity d_register is

  port (
        CLK, DATA: in STD_LOGIC;
        Q: out STD_LOGIC
        );
end d_register;

architecture BEHAV of d_register is
begin

MY_D_REG: process (CLK)
  begin
    if (CLK'event and CLK='1') then
        Q <= DATA;
    end if;
  end process;    --End MY_D_REG
end BEHAV;
```

Verilog Example

```
module d_register (
input CLK, DATA,
output reg Q);

  always @ (posedge CLK)
    begin: My_D_Reg
      Q <= DATA;
    end
endmodule
```

Converting Latch to a Logic Gate

If your intention is to not infer a latch, but rather to infer logic for the code, include an *else* clause if you are using an *if* statement, or a default clause if you are using a case statement.

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity latch2gate is
  port (
        GATE, DATA: in STD_LOGIC;
        Q: out STD_LOGIC);
end d_latch;

architecture BEHAV of d_latch is
begin
  GATE: process (GATE, DATA)
  begin
    if (GATE = '1') then
        Q <= DATA;
     else
        Q <= '0';
    end if;
end process;

end BEHAV;
```

Verilog Example

```
module latch2gate (GATE, DATA, Q);

  input GATE;
  input DATA;
  output Q;
  reg Q;

always @ (GATE or DATA)
  begin
    if (GATE == 1'b1)
        Q <= DATA;
     else
        Q <= 1'b0;
  end

endmodule
```

With some synthesis tools you can determine the number of latches that are implemented in your design. Check your software documentation for information on determining the number of latches in your design.

You should convert all If statements without corresponding Else statements and without a clock edge to registers or logic gates. Use the recommended coding styles in the synthesis tool documentation to complete this conversion.

Resource Sharing

Resource sharing is an optimization technique that uses a single functional block (such as an adder or comparator) to implement several operators in the HDL code. Use resource sharing to improve design performance by reducing the gate count and the routing congestion. If you do not use resource sharing, each HDL operation is built with separate circuitry. However, you may want to disable resource sharing for speed critical paths in your design.

The following operators can be shared either with instances of the same operator or with an operator on the same line.

```
*
+ -
> >= < <=
```

For example, a + operator can be shared with instances of other + operators or with – operators. A * operator can be shared only with other * operators.

You can implement arithmetic functions (+, –, magnitude comparators) with gates or with your synthesis tool's module library. The library functions use modules that take advantage of the carry logic in the FPGAs. Carry logic and its dedicated routing increase the speed of arithmetic functions that are larger than 4-bits. To increase speed, use the module library if your design contains arithmetic functions that are larger than 4-bits or if your design contains only one arithmetic function. Resource sharing of the module library automatically occurs in most synthesis tools if the arithmetic functions are in the same process.

Resource sharing adds additional logic levels to multiplex the inputs to implement more than one function. Therefore, you may not want to use it for arithmetic functions that are part of your design's time critical path.

Since resource sharing allows you to reduce the number of design resources, the device area required for your design is also decreased. The area that is used for a shared resource depends on the type and bit width of the shared operation. You should create a shared resource to accommodate the largest bit width and to perform all operations.

If you use resource sharing in your designs, you may want to use multiplexers to transfer values from different sources to a common resource input. In designs that have shared operations with the same output target, the number of multiplexers is reduced as illustrated in the following VHDL and Verilog examples. The HDL example is shown implemented with gates in Figure 3-8.

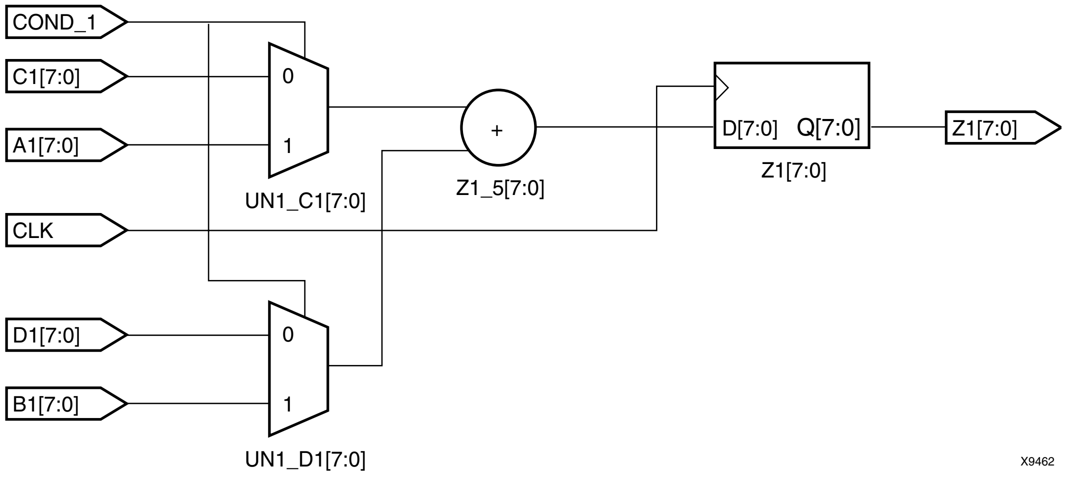

Figure 3-8: **Implementation of Resource Sharing**

VHDL Example

```
-- RES_SHARING.VHD

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity res_sharing is
  port (
        A1,B1,C1,D1 : in STD_LOGIC_VECTOR (7 downto 0);
        COND_1 : in STD_LOGIC;
        Z1 : out STD_LOGIC_VECTOR (7 downto 0));
end res_sharing;

architecture BEHAV of res_sharing is
begin
  P1: process (A1,B1,C1,D1,COND_1)
  begin
    if (COND_1='1') then
        Z1 <= A1 + B1;
    else
        Z1 <= C1 + D1;
    end if;
  end process; -- end P1

end BEHAV;
```

Verilog Example

```
/* Resource Sharing Example
 * RES_SHARING.V
*/

module res_sharing (
input [7:0] A1, B1, C1, D1,
input COND_1,
output reg [7:0] Z1);

always @(*)
  begin
    if (COND_1)
        Z1 <= A1 + B1;
    else
        Z1 <= C1 + D1;
  end
endmodule
```

If you disable resource sharing or if you code the design with the adders in separate processes, the design is implemented using two separate modules as shown in Figure 3-9

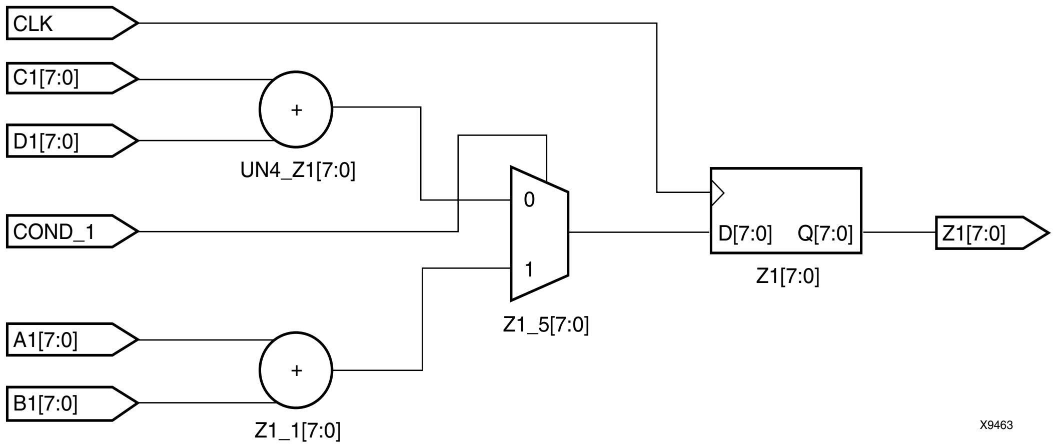

Figure 3-9: **Implementation without Resource Sharing**

For more information on resource sharing, see the appropriate reference guide.

Using Clock Enable Pin Instead of Gated Clocks

Xilinx generally recommends that you use the CLB clock enable pin instead of gated clocks in your designs. Gated clocks can cause glitches, increased clock delay, clock skew, and other undesirable effects. Using **`clock enable`** saves clock resources, and can improve timing characteristic and analysis of the design. If you want to use a gated clock for power reduction, most FPGA devices now have a clock enabled global buffer resource called BUFGCE. However, a clock enable is still the preferred method to reduce or stop the clock to portions of the design. The first two examples in this section (VHDL and Verilog) illustrate a design that uses a gated clock. Following these examples are VHDL and Verilog designs that show how to modify the gated clock design to use the clock enable pin of the CLB.

VHDL Example

```
// The following code is for demonstration purposes only
// Xilinx does not suggest using the following coding style in FPGAs

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity gate_clock is
    port (CLOCK_EN, DATA, CLK : in STD_LOGIC;
         OUT1: out STD_LOGIC);
end gate_clock;

architecture BEHAVIORAL of gate_clock is
signal GATECLK: STD_LOGIC;

begin

   GATECLK <= (CLOCK_EN and CLK);

   GATE_PR: process (GATECLK)
   begin
     if (GATECLK'event and GATECLK='1') then
        OUT1 <= DATA;
     end if;
   end process; -- End GATE_PR

end BEHAVIORAL
```

Verilog Example

```
// The following code is for demonstration purposes only
// Xilinx does not suggest using the following coding style in FPGAs

module gate_clock(
   input CLOCK_EN, DATA, CLK,
   output reg OUT1
);

   wire GATECLK;

   assign GATECLK = (CLOCK_EN & CLK);

   always @(posedge GATECLK)
      OUT1 <= DATA;

endmodule
```

Converting the Gated Clock to a Clock Enable

VHDL Example

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

entity clock_enable is
   port (CLOCK_EN, DATA, CLOCK : in STD_LOGIC;
         OUT1: out STD_LOGIC);
end clock_enable;
architecture BEHAV of clock_enable is
begin

   EN_PR: process (CLOCK)
   begin
     if (CLOCK'event and CLOCK='1') then
       if (CLOCK_EN = '1') then
        OUT1 <= DATA;
```

```
            end if;
         end if;
      end process; -- End EN_PR

   end BEHAV;
```

Verilog Example

```
module clock_enable (
   input CLOCK_EN, DATA, CLK,
   output reg DOUT
);

   always @(posedge CLK)
     if (CLOCK_EN)
       DOUT <= DATA;

endmodule
```

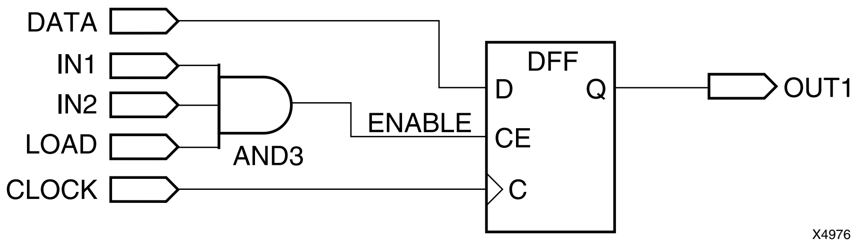

Figure 3-10: **Implementation of Clock Enable**

Chapter 4

Coding Styles for FPGA Devices

This chapter includes coding techniques to help you improve synthesis results. This chapter includes the following sections.

Applicable Architectures

This chapter highlights the features and synthesis techniques in designing with the following Xilinx® FPGA devices:

- Virtex™
- Virtex-E
- Virtex-II
- Virtex-II Pro
- Virtex-II Pro X
- Virtex-4
- Spartan™-II
- Spartan-IIE
- Spartan-3
- Spartan-3E

Unless otherwise stated, the features and examples in this chapter apply to all the FPGA devices listed above.

Virtex-II, Virtex-II Pro, Virtex-II Pro X, Spartan-3,and Spartan-3E provide an architecture that is substantially different from Virtex, Virtex-E and Spartan-II. However, many of the synthesis design techniques apply the same way to all these devices.

For information specific to Virtex-II Pro and Virtex-II Pro X, see the *Virtex-II Pro Platform FPGA User Guide*. For information specific to Virtex-4, see the *Virtex-4 User Guide*.

FPGA HDL Coding Features

This chapter covers the following FPGA HDL coding features:

- Advanced clock management
- On-chip RAM and ROM
- IEEE 1149.1 — compatible boundary scan logic support
- Flexible I/O with Adjustable Slew-rate Control and Pull-up/Pull-down Resistors
- Various drive strength
- Various I/O standards
- Dedicated high-speed carry-propagation circuit

You can use these device characteristics to improve resource utilization and enhance the speed of critical paths in your HDL designs. The examples in this chapter are provided to help you incorporate these system features into your HDL designs.

Instantiating Components

Xilinx provides a set of libraries that your synthesis tool can infer from your HDL code description. However, architecture specific and customized components must be explicitly instantiated as components in your design.

Instantiating FPGA Primitives

Architecture specific components that are built into the implementation software's library are available for instantiation without the need for specifying a definition. These components are marked as primitive in the Xilinx *Libraries Guides*. Components marked as macro in the Xilinx *Libraries Guides* are not built into the implementation software's library so they cannot be instantiated. The macro components in the Xilinx *Libraries Guides* define the schematic symbols. When macros are used, the schematic tool decomposes the macros into their primitive elements when the schematic tool writes out the netlist.

FPGA primitives can be instantiated in VHDL and Verilog.

VHDL Example

The following example shows declaring component and port map.

```
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;
entity flops is port(
        di   : in std_logic;
        ce   : in std_logic;
        clk  : in std_logic;
        qo   : out std_logic;
        rst  : in std_logic);
end flops;
-- remove the following component declarations
-- if using XST or Synplify
architecture inst of flops is
component FDCE port(
        D    : in std_logic;
        CE   : in std_logic;
        C    : in std_logic;
        CLR  : in std_logic;
        Q    : out std_logic);
end component;

begin
U0 : FDCE port map(
        D    => di,
        CE   => ce,
        C    => clk,
        CLR  => rst,
        Q    => qo);
end inst;
```

Verilog Example

```
////////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
////////////////////////////////////////////////////////
module flops (
 input d1, ce, clk, rst,
 output q1);

 FDCE u1 (
       .D (d1),
       .CE (ce),
       .C (clk),
       .CLR (rst),
       .Q (q1));
endmodule
```

Passing Generics and Parameters

Some constraints are properties of primitives. These constraints can be added to the primitive through a variety of methods:

- User Constraints File (UCF)
- VHDL attribute passing
- Verilog attribute passing
- VHDL generic passing
- Verilog parameter passing

To determine which constraints are available to a particular primitive, see the ISE help: Edit > Language Templates > VHDL/Verilog > Device Primitive Instantiation. In each of the primitive instantiation templates, the primitive constraints will be listed in the generics or parameters. Below are some examples of passing constraints in generics and parameters.

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using Synplify:
-- library unisim;
-- use unisim.vcomponents.all;
entity my_ram is port(
       d         : in std_logic;
       clk       : in std_logic;
       we        : in std_logic;
       address   : in std_logic_vector (3 downto 0);
       readout   : out std_logic);
end my_ram;
-- remove the following component declarations
-- if using XST or Synplify
architecture inst of my_ram is
 component RAM16X1S
  generic (INIT : bit_vector
  port (
       D    : in std_logic;
       WE   : in std_logic;
       WCLK: in std_logic;
       A0   : in std_logic;
       A1   : in std_logic;
       A2   : in std_logic;
       A3   : in std_logic;
       O    : out std_logic);
end component;

begin
U0 : RAM16X1S
generic map (INIT => x"FOFO")
port map(
       D      => d,
       WE     => we,
       WCLK   => clk,
       A0     => address (0),
       A1     => address (1),
       A2     => address (2),
       A3     => address (3),
       O      => readout);
end inst;
```

Verilog Example

```
////////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
////////////////////////////////////////////////////////
module my_ram (
 output readout,
 input [3:0] address,
 input d, clk, we);

 RAM16X1S #(.INIT(16'hF0F0))
 U1 (
       .O (readout),
       .A0 (address [0]),
       .A1 (address [1]),
       .A2 (address [2]),
       .A3 (address [3]),
       .D (d),
       .WCLK (clk),
       .WE (we));
endmodule
```

Instantiating CORE Generator Modules

CORE Generator™ allows you to generate complex ready-to-use functions such as:

- FIFO
- Filter
- Divider
- RAM
- ROM

CORE Generator generates:

- an EDIF netlist to describe the functionality
- a component instantiation template for HDL instantiation

To instantiate a CORE Generator module in Project Navigator:

1. Select **Project > New Source**.
2. Select **IP (CoreGen & Architecture Wizard)**.
3. Select the core that you wish to generate.
4. Click **Generate.**

Project Navigator creates an instantiation template of the core and places it in the Project Navigator Language Templates.

Use the instantiation template to instantiate the core in your design:

1. Select **Edit > Language Templates.**
2. Expand the COREGEN folder.
3. Select the VHDL or Verilog instantiation templates for the core that you created.
4. Copy and paste the code into your design.

For more information on CORE Generator, see the CORE Generator help.

Using Boundary Scan

Xilinx FPGA devices contain boundary scan facilities that are compatible with IEEE Standard 1149.1. You can access the built-in boundary scan logic between power-up and the start of configuration.

In a configured device, basic boundary scan operations are always available, and, if desired, can access internal logic. To do so, the proper BSCAN component for the desired target device must be instantiated and connected in the code. This is necessary only if you wish to connect internal logic to the FPGA. All other JTAG commands are still accessible without the BSCAN component existing in the design.

For more information on boundary scan for an architecture, see the Xilinx *Libraries Guides*, and the product data sheet and user guide.

Using Global Clock Buffers

For designs with global signals, use global clock buffers to take advantage of the low-skew, high-drive capabilities of the dedicated global buffer tree of the target device. Your synthesis tool automatically inserts a clock buffer whenever an input signal drives a clock signal, or whenever an internal clock signal reaches a certain fanout.

Most synthesis tools also limit global buffer insertions to match the number of buffers available on the device.

You can instantiate the clock buffers if your design requires a special architecture-specific buffer, or if you want to specify the allocation of the clock buffer resources. Xilinx recommends, however, that you generally let the synthesis tool infer such buffers.

Inserting Global Clock Buffers

Synthesis tools will automatically insert a global buffer (BUFG) when an input port drives a register's clock pin or when an internal clock signal reaches a certain fanout. A BUFGP (an IBUFG-BUFG connection) is inserted for the external clock whereas a BUFG is inserted for an internal clock. Most synthesis tools also allow you to control BUFG insertions manually if you have more clock pins than the available BUFGs resources

Synthesis tools currently insert simple clock buffers (BUFGs) for all FPGA devices. For Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, Spartan-3, and Spartan-3E, some tools provide an attribute to use BUFGMUX as an enabled clock buffer. To use BUFGMUX as a real clock multiplexer, it must be instantiated.

LeonardoSpectrum and Precision Synthesis

LeonardoSpectrum and Precision Synthesis force clock signals to global buffers when the resources are available. The best way to control unnecessary BUFG insertions is to turn off global buffer insertion, then use the BUFFER_SIG attribute to push BUFGs onto the desired signals. By doing this you do not have to instantiate any BUFG components. As long as you use *chip* options to optimize the IBUFs, they are auto-inserted for the input.

The following is a syntax example of the BUFFER_SIG attribute.

```
set_attribute -port clk1 -name buffer_sig -value BUFG
set_attribute -port clk2 -name buffer_sig -value BUFG
```

Synplify

Synplify assigns a BUFG to any input signal that directly drives a clock. Auto-insertion of the BUFG for internal clocks occurs with a fanout threshold of 16 loads. To turn off automatic clock buffers insertion, use the **`syn_noclockbuf`** attribute. This attribute can be applied to the entire module/architecture or a specific signal.

To change the maximum number of global buffer insertion, set an attribute in the SDC file as follows.

```
define_global_attribute xc_global buffers (8)
```

XST

For information on inserting global clock buffers in XST, see the Xilinx *XST User Guide*.

Instantiating Global Clock Buffers

You can instantiate global buffers in your code as described in this section.

Instantiating Buffers Driven from a Port

You can instantiate global buffers and connect them to high-fanout ports in your code rather than inferring them from a synthesis tool script. If you do instantiate global buffers, verify that the Pad parameter is not specified for the buffer.

Synthesis tools insert BUFGP for clock signals which access a dedicated clock pin. To have a regular input pin to a clock buffer connection, you must use an IBUF-BUFG connection. This is done by instantiating BUFG after disabling global buffer insertion.

VHDL Example

```
-----------------------------------------------
-- IBUF_BUFG.VHD Version 1.0
-- This is an example of an instantiation of
-- a global buffer (BUFG)
-----------------------------------------------
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;

library IEEE;
use IEEE.std_logic_1164.all;
entity IBUF_BUFG is
  port (DATA, CLOCK : in STD_LOGIC;
          DOUT : out STD_LOGIC);
end ibuf_bufg;
architecture XILINX of IBUF_BUFG is

signal CLOCK : STD_LOGIC;
signal CLOCK_GBUF : STD_LOGIC;
-- remove the following component declarations
-- if using XST or synplify
component BUFG
  port (
       I : in STD_LOGIC;
       O : out STD_LOGIC
       );
end component;
begin
u0 : BUFG
    begin
      port map (I => CLOCK,
                O => CLOCK_GBUF);
process (CLOCK_GBUF)
```

```
    begin
      if rising_edge(CLOCK_GBUF) then
              DOUT <= DATA;
    end if;
end process;

end XILINX;
```

Verilog Example

```
////////////////////////////////////////////////////////
// IBUF_BUFG.V Version 1.0
// This is an example of an instantiation of
// global buffer (BUFG)
////////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
////////////////////////////////////////////////////////

module ibuf_bufg(
 input DATA, CLOCK,
 output reg DOUT);

wire   CLOCK_GBUF;

    BUFG U0 (.O(CLOCK_GBUF),.I(CLOCK));
    always @ (*) DOUT <= DATA;
endmodule
```

Instantiating Buffers Driven from Internal Logic

Some synthesis tools require you to instantiate a global buffer in your code to use the dedicated routing resource if a high-fanout signal is sourced from internal flip-flops or logic (such as a clock divider or multiplexed clock), or if a clock is driven from a non-dedicated I/O pin. If using Virtex™/E or Spartan-II™ devices, the following VHDL and Verilog examples instantiate a BUFG for an internal multiplexed clock circuit.

Note: Synplify infers a global buffer for a signal that has 16 or greater fanouts.

VHDL Example

```
-----------------------------------------------
-- CLOCK_MUX_BUFG.VHD Version 1.1
-- This is an example of an instantiation of
-- global buffer (BUFG) from an internally
-- driven signal, a multiplexed clock.
-----------------------------------------------
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;

library IEEE;
use IEEE.std_logic_1164.all;
entity clock_mux is
  port (
        DATA, SEL : in STD_LOGIC;
        SLOW_CLOCK, FAST_CLOCK : in STD_LOGIC;
        DOUT : out STD_LOGIC
        );
```

```
end clock_mux;
architecture XILINX of clock_mux is

signal CLOCK : STD_LOGIC;
signal CLOCK_GBUF : STD_LOGIC;
-- remove the following component declarations
-- if using XST or Synplfy
component BUFG
  port (
        I : in STD_LOGIC;
        O : out STD_LOGIC
        );
end component;
begin
Clock_MUX: process (SEL, FAST_CLOCK, SLOW_CLOCK)
    begin
      if (SEL = '1') then
          CLOCK <= FAST_CLOCK;
      else
          CLOCK <= SLOW_CLOCK;
      end if;
    end process;

GBUF_FOR_MUX_CLOCK: BUFG
  port map (
        I => CLOCK,
        O => CLOCK_GBUF
        );

Data_Path: process (CLOCK_GBUF)
  begin
    if (CLOCK_GBUF'event and CLOCK_GBUF='1')then
        DOUT <= DATA;
    end if;
  end process;
end XILINX;
```

Verilog Example

```
////////////////////////////////////////////////////////
// CLOCK_MUX_BUFG.V Version 1.1
// This is an example of an instantiation of
// global buffer (BUFG) from an internally
// driven signal, a multiplied clock.
////////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
////////////////////////////////////////////////////////

module clock_mux(
    input  DATA, SEL, SLOW_CLOCK, FAST_CLOCK;
    output reg DOUT);
    reg    CLOCK;
    wire   CLOCK_GBUF;

    always @ (*)
    begin
      if (SEL == 1'b1)
          CLOCK <= FAST_CLOCK;
```

```
        else
            CLOCK <= SLOW_CLOCK;
      end

      BUFG GBUF_FOR_MUX_CLOCK (.O(CLOCK_GBUF),.I(CLOCK));
      always @ (posedge CLOCK_GBUF)
          DOUT <= DATA;
endmodule
```

For Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex--4, Spartan-3, and Spartan-3E devices, a BUFGMUX can be used to multiplex between clocks.

VHDL Example

```
--------------------------------------------------
-- CLOCK_MUX_BUFG.VHD Version 1.2
-- This is an example of an instantiation of
-- a multiplexing global buffer (BUFGMUX)
-- from an internally driven signal
--------------------------------------------------
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;

library IEEE;
use IEEE.std_logic_1164.all;

entity clock_mux is
  port (DATA, SEL              : in std_logic;
        SLOW_CLOCK, FAST_CLOCK : in std_logic;
        DOUT                   : out std_logic);
end clock_mux;

architecture XILINX of clock_mux is
  signal CLOCK_GBUF : std_logic;
-- remove the following component declarations
-- if using XST or synplify

  component BUFGMUX
    port (
        I0 : in std_logic;
        I1 : in std_logic;
        S  : in std_logic;
        O  : out std_logic);
  end component;
  begin
    GBUF_FOR_MUX_CLOCK : BUFGMUX
      port map(
        I0 => SLOW_CLOCK,
        I1 => FAST_CLOCK,
        S  => SEL,
        O  => CLOCK_GBUF);
    Data_Path : process (CLOCK_GBUF)
      begin
        if (CLOCK_GBUF'event and CLOCK_GBUF='1')then
            DOUT <= DATA;
        end if;
      end process;
end XILINX;
```

Verilog Example

```
////////////////////////////////////////////////////////
// CLOCK_MUX_BUFG.V Version 1.2
// This is an example of an instantiation of
// a multiplexing global buffer (BUFGMUX)
// from an internally driven signal
////////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
////////////////////////////////////////////////////////

module clock_mux(
  input DATA, SEL, SLOW_CLOCK, FAST_CLOCK,
  output reg DOUT);

  reg CLOCK;
  wire CLOCK_GBUF;

  BUFGMUX GBUF_FOR_MUX_CLOCK
    (.O(CLOCK_GBUF),
     .I0(SLOW_CLOCK),
     .I1(FAST_CLOCK),
     .S(SEL));

  always @ (posedge CLOCK_GBUF)
      DOUT <= DATA;

endmodule
```

Using Advanced Clock Management

Virtex, Virtex-E, and Spartan-II devices feature Clock Delay-Locked Loop (CLKDLL) for advanced clock management. The CLKDLL can eliminate skew between the clock input pad and internal clock-input pins throughout the device. CLKDLL also provides four quadrature phases of the source clock. With CLKDLL you can eliminate clock-distribution delay, double the clock, or divide the clock.

The CLKDLL also operates as a clock mirror. By driving the output from a DLL off-chip and then back on again, the CLKDLL can be used to de-skew a board level clock among multiple Virtex, Virtex-E, and Spartan-II devices.

For more information on CLKDLLs, see:

- the Xilinx *Libraries Guides*
- Xilinx Application Note XAPP132, "*Using the Virtex Delay-Locked Loop*"
- Xilinx Application Note XAPP174, "*Using Delay-Locked Loops in Spartan-II FPGAs*"

Virtex-II, Virtex-II Pro, Virtex-II Pro X, and Spartan-3 DCMs

In Virtex-II, Virtex-II Pro, Virtex-II Pro X, and Spartan-3 devices, the Digital Clock Manager (DCM) is available for advanced clock management. The DCM contains the following features.

- *Delay Locked Loop (DLL)* — The DLL feature is very similar to CLKDLL.
- *Digital Phase Shifter (DPS)* — The DPS provides a clock shifted by a fixed or variable phase skew.

- *Digital Frequency Synthesizer (DFS)* — The DFS produces a wide range of possible clock frequencies related to the input clock.

Virtex-4 DCMs

Virtex-4 has 3 different types of DCMs:

- DCM_ADV
- DCM_BASE
- DCM_PS

These new DCMs have the same features as the Virtex-II DCMs, with the addition of a Dynamic Reconfiguration ability. The Dynamic Reconfiguration ability allows the DCM to be reprogrammed without having to reprogram the Virtex-4. DCM_BASE and DCM_PS access a subset of features of DCM_ADV. To access the Virtex-4 DCM, you can instantiate one of the above listed primitives, as well as the Virtex-II DCM.

For more information, see the Xilinx *Libraries Guides* and the product data sheet and user guide.

Using CLKDLL in Virtex, Virtex-E and Spartan-II

There are four attributes available for the CLKDLL:

- CLKDV_DIVIDE
- DUTY_CYCLE
- FACTORY_JF
- STARTUP_WAIT.

To modify these attributes, change the values in the generic map or parameter passing within the instantiation component. For information on how to modify generics or parameters, see "Passing Generics and Parameters" in this chapter.

For information on how to modify attributes on instantiated primitives, see "Passing Generics and Parameters" in this chapter. Instantiation templates for the CLKDLLs are in the Xilinx *Libraries Guides*. For examples on instantiating CLKDLLs, see the Xilinx *Libraries Guides*.

Using the Additional CLKDLL in Virtex-E

There are eight CLKDLLs in each Virtex-E device, with four located at the top and four at the bottom as shown in Figure 4-1. The basic operations of the DLLs in the Virtex-E devices remain the same as in the Virtex and Spartan-II devices, but the connections may have changed for some configurations.

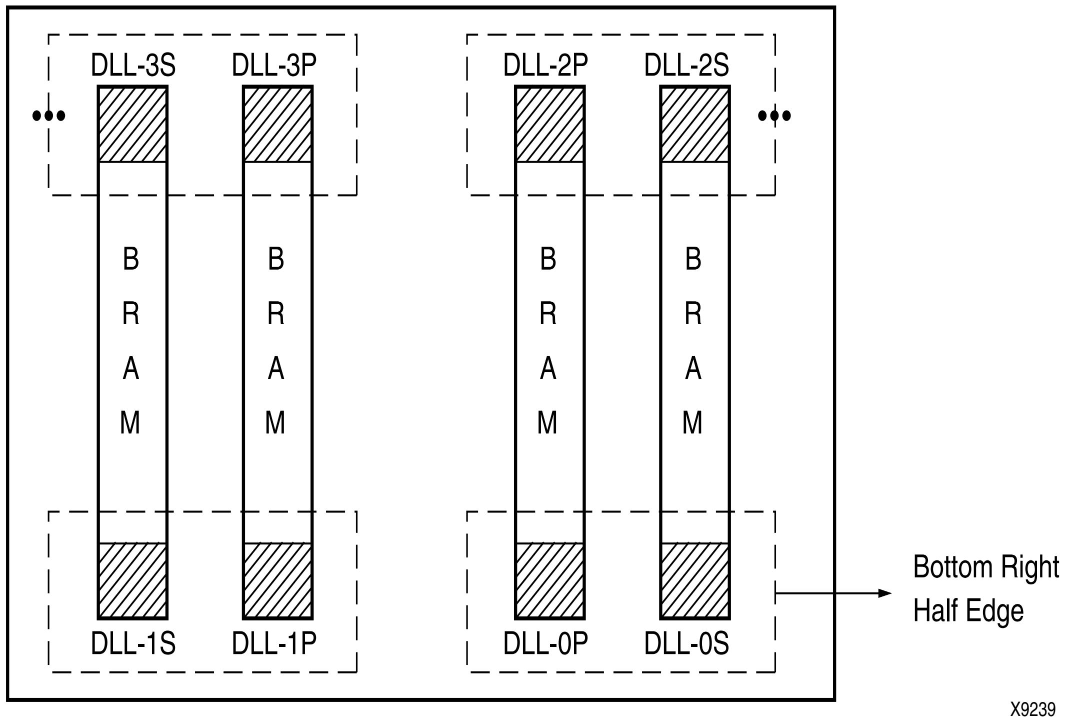

Figure 4-1: **DLLs in Virtex-E Devices**

Two DLLs located in the same half-edge (top-left, top-right, bottom-right, bottom-left) can be connected together, without using a BUFG between the CLKDLLs, to generate a 4x clock as shown in Figure 4-2.

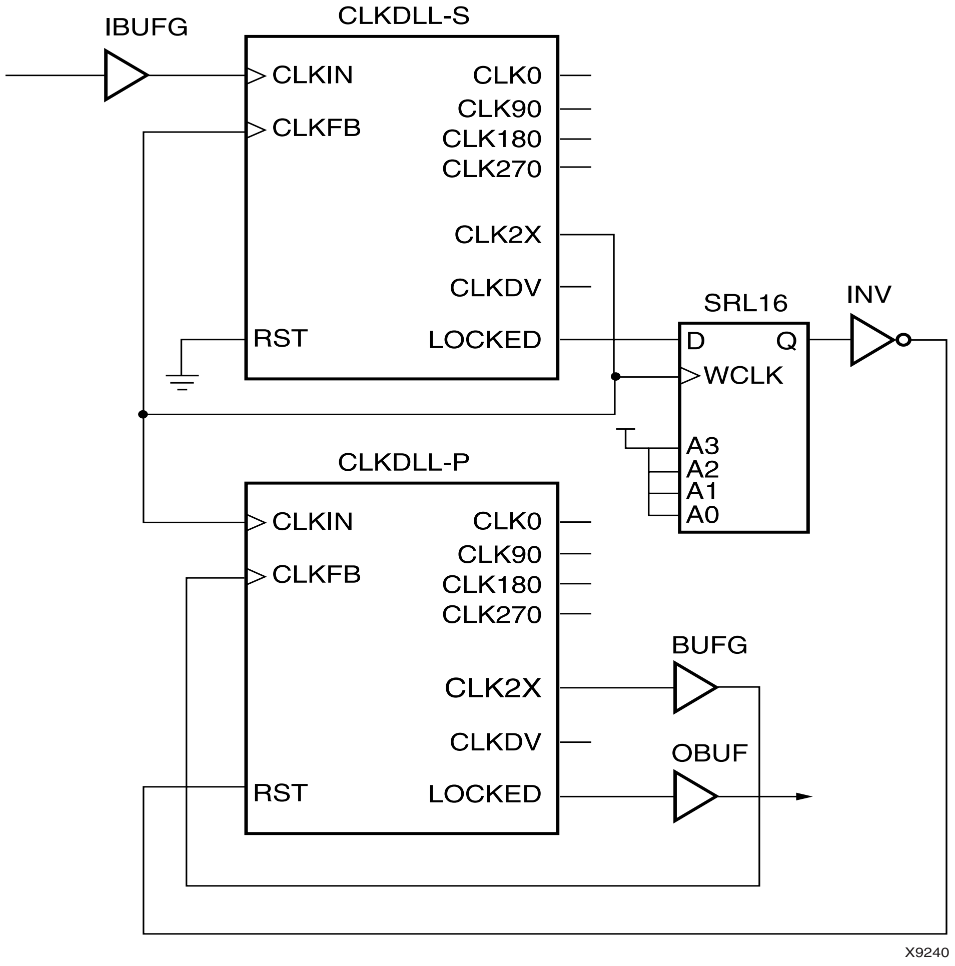

Figure 4-2: **DLL Generation of 4x Clock in Virtex-E™ Devices**

Following are examples of coding a CLKDLL in both VHDL and Verilog.

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;

entity CLOCK_TEST is
  port(
        ACLK   : in std_logic;
        DIN    : in std_logic_vector(1 downto 0);
        RESET  : in std_logic;
        QOUT   : out std_logic_vector (1 downto 0);
    -- CLKDLL lock signal
        BCLK_LOCK : out std_logic
        );
```

```
end CLOCK_TEST;
architecture RTL of CLOCK_TEST is
-- remove the following component declarations
-- if using XST or synplify

  component IBUFG
    port (
        I : in std_logic;
        O : out std_logic);
  end component;
  component BUFG
    port (
        I : in std_logic;
        O : out std_logic);
  end component;
  component CLKDLL
    port (
        CLKIN  : in std_logic;
        CLKFB  : in std_logic;
        RST    : in std_logic;
        CLK0   : out std_logic;
        CLK90  : out std_logic;
        CLK180 : out std_logic;
        CLK270 : out std_logic;
        CLKDV  : out std_logic;
        CLK2X  : out std_logic;
        LOCKED : out std_logic);
  end component;
  -- Clock signals
  signal ACLK_ibufg        : std_logic;
  signal ACLK_2x, BCLK_4x  : std_logic;
  signal BCLK_4x_design    : std_logic;
  signal BCLK_lockin       : std_logic;
begin
  ACLK_ibufginst : IBUFG
    port map (
        I => ACLK,
        O => ACLK_ibufg
        );
  BCLK_bufg: BUFG
    port map (
        I => BCLK_4x,
        O => BCLK_4x_design);
  ACLK_dll : CLKDLL
    port map (
        CLKIN  => ACLK_ibufg,
        CLKFB  => ACLK_2x,
        RST    => '0',
        CLK2X  => ACLK_2x,
        CLK0   => OPEN,
        CLK90  => OPEN,
        CLK180 => OPEN,
        CLK270 => OPEN,
        CLKD   => OPEN,
        LOCKED => OPEN
          );

  BCLK_dll : CLKDLL
    port map (
        CLKIN   => ACLK_2x,
        CLKFB   => BCLK_4x_design,
        RST     => '0',
        CLK2X   => BCLK_4x,
        CLK0    => OPEN,
        CLK90   => OPEN,
        CLK180  => OPEN,
        CLK270  => OPEN,
        CLKDV   => OPEN,
        LOCKED  => BCLK_lockin
          );
process (BCLK_4x_design, RESET)
begin
  if RESET = '1' then
     QOUT <= "00";
  elsif BCLK_4x_design'event and BCLK_4x_design = '1' then
     if BCLK_lockin = '1' then
        QOUT <= DIN;
     end if;
  end if;
end process;
  BCLK_lock <= BCLK_lockin;
END RTL;
```

Verilog Example

```
////////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
////////////////////////////////////////////////////////

module clock_test(
  input  ACLK, RESET,
  input  [1:0] DIN,
  output reg [1:0] QOUT,
  output BCLK_LOCK);

IBUFG CLK_ibufg_A
       (.I (ACLK),
        .O(ACLK_ibufg)
         );
BUFG BCLK_bufg
       (.I (BCLK_4x),
        .O (BCLK_4x_design)
         );
CLKDLL ACLK_dll_2x   // 2x clock
       (.CLKIN(ACLK_ibufg),
        .CLKFB(ACLK_2x),
        .RST(1'b0),
        .CLK2X(ACLK_2x),
        .CLK0(),
        .CLK90(),
        .CLK180(),
        .CLK270(),
        .CLKDV(),
        .LOCKED()
         );
```

```
CLKDLL BCLK_dll_4x    // 4x clock
       (.CLKIN(ACLK_2x),
        .CLKFB(BCLK_4x_design),    // BCLK_4x after bufg
        .RST(1'b0),
        .CLK2X(BCLK_4x),
        .CLK0(),
        .CLK90(),
        .CLK180(),
        .CLK270(),
        .CLKDV(),
        .LOCKED(BCLK_LOCK)
        );
always @(posedge BCLK_4x_design, posedge RESET)
  begin
    if (RESET)
         QOUT <= 2'b00;
    else if (BCLK_LOCK)
         QOUT <= DIN[1:0];
  end
endmodule
```

Using DCM_ADV in Virtex-4

DCM_ADV provides a wide range of clock management features such as phase shifting, clock deskew, and dynamic reconfiguration. The synthesis tools do not infer any of the DCM primitives in the Virtex-4 (DCM_ADV, DCM_BASE, DCM_PS and DCM).To be able to use them, you must instantiate them. Below are two simple templates for instantiating DCM_ADV in Virtex-4. For more information on DCM_ADV, see the Xilinx *Libraries Guides* and the*Virtex-4 User Guide*.

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;
entity clock_block is
  port (
    CLK_PAD : in std_logic;
    RST_DLL : in std_logic;
    DADDR_in : in std_logic_vector (6 downto 0);
    DCLK_in : in std_logic;
    DEN_in : in std_logic;
    DI_in : in std_logic_vector (15 downto 0);
    DWE_in : in std_logic;
    CLK_out : out std_logic;
    DRDY_out: out std_logic;
    DO_out : out std_logic_vector (15 downto 0);
    LOCKED : out std_logic
  );
end clock_block;

architecture STRUCT of clock_block is
  signal CLK, CLK_int, CLK_dcm : std_logic;
  -- remove the following component declarations
  -- if using XST or synplify
  component IBUFG_GTL
```

```
      port (
        I : in std_logic;
        O : out std_logic);
    end component;

    component BUFG
      port (
        I : in std_logic; O : out std_logic);
    end component;

    component DCM_ADV is
      generic (CLKIN_PERIOD : real);
      port (
        CLKFB : in std_logic;
        CLKIN : in std_logic;
        PSCLK : in std_logic;
        PSEN : in std_logic;
        PSINCDEC : in std_logic;
        RST : in std_logic;
        DADDR : in std_logic_vector (6 downto 0);
        DCLK : in std_logic;
        DEN : in std_logic;
        DI : in std_logic_vector (15 downto 0);
        DWE : in std_logic;
        CLK0 : out std_logic;
        CLK90 : out std_logic;
        CLK180 : out std_logic;
        CLK270 : out std_logic;
        CLK2X : out std_logic;
        CLK2X180 : out std_logic;
        CLKDV : out std_logic;
        CLKFX : out std_logic;
        CLKFX180 : out std_logic;
        LOCKED : out std_logic;
        DRDY : out std_logic;
        DO : out std_logic_vector (15 downto 0);
        PSDONE : out std_logic
      );
    end component;

    signal logic_0 : std_logic;

  begin

    logic_0 <= '0';
    U1 : IBUFG_GTL port map ( I => CLK_PAD, O => CLK_int);
    U2 : DCM_ADV generic map (CLKIN_PERIOD => 10.0)
      port map (
        CLKFB => CLK,
        CLKIN => CLK_int,
        PSCLK => logic_0,
        PSEN => logic_0,
        PSINCDEC => logic_0,
        RST => RST_DLL,
        CLK0 => CLK_dcm,
        DADDR => DADDR_in,
        DCLK => DCLK_in,
        DEN => DEN_in,
        DI => DI_in,
        DWE => DWE_in,
        DRDY => DRDY_out,
        DO => DO_out,
        LOCKED => LOCKED
      );

    U3 : BUFG port map (I => CLK_dcm, O => CLK);

    CLK_out <= CLK;

  end architecture STRUCT;
```

Verilog Example

```
// `include "c:\<path_to_synplify>\lib\xilinx\virtex4.v"
module clock_block (
  input CLK_PAD, RST_DLL, DCLK_in, DEN_in, DWE_in,
  input [6:0] DADDR_in,
  input [15:0] DI_in,
  output CLK_out, DRDY_out, LOCKED,
  output [15:0] DO_out);

  wire CLK, CLK_int, CLK_dcm, logic_0;

  assign logic_0 = 1'b0;

  IBUFG_GTL U1 (.I(CLK_PAD), .O(CLK_int));
  DCM_ADV #(.CLKIN_PERIOD(10.0))
    U2 (.CLKFB(CLK),
        .CLKIN(CLK_int),
        .PSCLK(logic_0),
        .PSEN(logic_0),
        .PSINCDEC(logic_0),
        .RST(RST_DLL),
        .CLK0(CLK_dcm),
        .DADDR(DADDR_in),
        .DCLK(DCLK_in),
        .DEN(DEN_in),
        .DI(DI_in),
        .DWE(DWE_in),
        .DRDY(DRDY_out),
        .DO(DO_out),
        .LOCKED(LOCKED));

  BUFG U3 (.I(CLK_dcm), .O(CLK));

endmodule
```

Using DCM in Other Devices

Note: This section applies only to Virtex-II, Virtex-II Pro, Virtex-II Pro X and Spartan-3 devices.

Use the DCM in your Virtex-II, Virtex-II Pro, Virtex-II Pro X or Spartan-3 design to improve routing between clock pads and global buffers. Since synthesis tools do not automatically infer the DCM, you must instantiate the DCM in your VHDL and Verilog designs.

To more easily set up the DCM, use the Clocking Wizard. For more information on the Clocking Wizard, see "Architecture Wizard" in Chapter 2 of this guide.

For more information on the various features in the DCM, see the "Design Considerations" chapters of the *Virtex-II Platform FPGA User Guide* and the *Virtex-II Pro Platform FPGA User Guide*.

The following examples show how to instantiate DCM and apply a DCM attribute in VHDL and Verilog.

For more information on passing attributes in the HDL code to different synthesis vendors, see Chapter 3, "General HDL Coding Styles"in this guide.

VHDL Example

```
-- Using a DCM for Virtex-II (VHDL)
--
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;

entity clock_block is
  port (
    CLK_PAD              : in std_logic;
    RST_DLL              : in std_logic;
    CLK_out              : out std_logic;
    LOCKED               : out std_logic
    );
end clock_block;
architecture STRUCT of clock_block is
  signal CLK, CLK_int, CLK_dcm : std_logic;
-- remove the following component declarations
-- if using XST or synplify
    component IBUFG
    port (
        I : in std_logic;
        O : out std_logic);
  end component;
  component BUFG
    port (
        I : in std_logic;
        O : out std_logic);
  end component;

  component DCM is
  generic (CLKIN_PERIOD : real);
    port (
        CLKFB    : in std_logic;
        CLKIN    : in std_logic;
        DSSEN    : in std_logic;
        PSCLK    : in std_logic;
        PSEN     : in std_logic;
        PSINCDEC : in std_logic;
        RST      : in std_logic;
        CLK0     : out std_logic;
        CLK90    : out std_logic;
        CLK180   : out std_logic;
        CLK270   : out std_logic;
        CLK2X    : out std_logic;
        CLK2X180 : out std_logic;
        CLKDV    : out std_logic;
        CLKFX    : out std_logic;
        CLKFX180 : out std_logic;
        LOCKED   : out std_logic;
        PSDONE   : out std_logic;
        STATUS   : out std_logic_vector (7 downto 0)
        );
  end component;
signal logic_0 : std_logic;

begin
  logic_0 <= '0';

  U1 : IBUFG port map ( I => CLK_PAD, O => CLK_int);
  U2 : DCM generic map (CLKIN_PERIOD => 10.0)
        port map(
        CLKFB    => CLK,
        CLKIN    => CLK_int,
        DSSEN    => logic_0,
        PSCLK    => logic_0,
        PSEN     => logic_0,
        PSINCDEC => logic_0,
        RST      => RST_DLL,
        CLK0     => CLK_dcm,
        LOCKED   => LOCKED
        );
  U3 : BUFG port map (I => CLK_dcm, O => CLK);
  CLK_out <= CLK;
end
end architecture STRUCT;
```

Verilog Example

```
////////////////////////////////////////////////////////
// Using a DCM for Virtex-II (Verilog)
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
////////////////////////////////////////////////////////

module clock_top (
  input clk_pad, rst_dll,
  output clk_out, locked);

  wire   clk, clk_int, clk_dcm;
  IBUFG u1 (.I (clk_pad), .O (clk_int));
  DCM #(.CLKIN_PERIOD (10.0))
         u2(
        .CLKIN    (clk_int),
        .DSSEN    (1'b0),
        .PSCLK    (1'b0),
        .PSEN     (1'b0),
        .PSINCDEC (1'b0),
        .RST      (rst_dll),
        .CLK0     (clk_dcm),
        .LOCKED    (locked));
  BUFG u3(.I (clk_dcm), .O (clk));
  assign clk_out = clk;
endmodule // clock_top
```

Using Dedicated Global Set/Reset Resource

All Xilinx FPGA devices have a dedicated Global Set/Reset (GSR) resource which is routed to the asnynchronous reset of every register in the device. This resource is automatically activated when the FPGA configuration is complete, and can be accessed by the design logic in a configured device.

The use of this resource, however, must be considered carefully. Synthesis tools do not automatically infer GSRs. However, the STARTUP block can be instantiated in your HDL code to access the GSR resources.

Recommendations

Xilinx recommends that you not code a global set/reset into the design unless it is necessary for the design specification or operation. Many times is not.

If a global set/reset is necessary, Xilinx recommends that you:

- Write the high fanout set/reset signal explicitly in the HDL code as a synchronous reset.
- Do not use the STARTUP blocks.

Coding a synchronous reset, as opposed to an asynchronous reset, will likely result in a smaller, more efficient design that is easier to analyze for both timing and functionality. More on the use of asynchronous resets is discussed in Chapter 2, "Understanding High-Density Design Flow."

Advantages to Implicitly Coding

Implicitly coding in the set/reset signal over using the dedicated GSR resource has the following advantages:

- "Faster Speed with Less Skew"
- "TRCE Program Analyzes the Delays"

Faster Speed with Less Skew

Implicitly coding in the set/reset signal gives you a faster speed with less skew. The set/reset signals are routed onto the secondary longlines in the device, which are global lines with minimal skew and less overall delay. Therefore, the reset/set signals on the secondary lines are much faster, and more well behaved in terms of skew than the GSR nets of the STARTUP block. Since the FPGA is rich in routings, placing and routing this signal on the global lines can be easily done by the ISE software.

TRCE Program Analyzes the Delays

By implicitly coding in the set/reset signal, the TRCE program analyzes the delays of the explicitly written set/reset signals. You can read the report file of the TRCE program (the TWR file) to ascertain the exact speed of your design. The TRCE program does not analyze the delays on the GSR net of the STARTUP_architecture. Hence, using an explicit set/reset signal improves your design accountability.

Initial State of the Registers and Latches

The FPGA flip-flops are configured as either preset (asynchronous set) or clear (asynchronous reset) during startup. This is known as the initialization state, or INIT. The initial state of the register can be specified as follows:

- If the register is *instantiated*, it can be specified by setting the INIT attribute on the instantiated register primitive to either a 1 or 0, depending on the desired state.
- If the register is *inferred*, the initial state can be specified by initializing the VHDL signal declaration or the Verilog reg declaration as shown in the following examples.

VHDL Example

```
signal register1 : std_logic := '0'; -- specifying register1 to start as a zero
signal register2 : std_logic := '1'; -- specifying register2 to start as a one
```

Verilog Example

```
reg register1 = 1'b0; // specifying regsiter1 to start as a zero
reg register2 = 1'b1; // specifying register2 to start as a one
```

Not all synthesis tools support this initialization. To determine whether it is supported, see your synthesis tool documentation. If this intialization is not supported, or if it is not specified in the code, the initial value is determined by the presence or absence of an asynchronous preset in the code. If an asynchronous preset is present, the register initializes to a one. If an asynchronous preset is not present, the register initializes to a logic zero.

Implementing Inputs and Outputs

FPGA devices have limited logic resources in the user-configurable input/output blocks (IOB). You can move registers and some logic that is normally implemented with CLBs to IOBs. By moving from CLBs to IOBs, additional logic can be implemented in the available CLBs. Using IOBs can also improve design performance by increasing the number of available routing resources, while decreasing the input setup times and clock-to-out times to and from the FPGA.

All Xilinx FPGAs feature SelectIO™ inputs and outputs that support a wide variety of I/O signaling standards. In addition, each IOB provides three or more storage elements. The following sections discuss IOB features in more detail.

I/O Standards

You can set an IOSTANDARD attribute to a specific I/O standard and attach it to a port, or to an instantiated:

- IBUF
- IBUFG
- IBUFDS
- IBUFGDS
- IOBUF
- IOBUFDS
- OBUF

- OBUFDS
- OBUFT
- OBUFTDS

You can set the IOSTANDARD attribute in the user constraint file (UCF), or it can be set in the netlist by the synthesis tool. Where and how the IOSTANDARD attribute to set is is a matter of user preference. The most common way is to set IOSTANDARD attributes within either the UCF file or a synthesis constraint file.

Some users prefer to write this information into the code itself. They either specify an associated attribute to each specified port in the top level, or instantiate an I/O BUFFER and specify the IOSTANDARD constant on each instance.

For a complete table of I/O standards, see the product data sheet and user guide.

Specifying I/O Standards

This section gives examples of setting the IOSTANDARD attribute in various tools.

LeonardoSpectrum

In LeonardoSpectrum, insert appropriate buffers on selected ports in the constraints editor. Alternatively, you can set the following attribute in TCL script after the **read** but before the **optimize** options.

PAD *IOstandard portname*

The following is an example of setting an I/O standard in LeonardoSpectrum.

```
PAD IBUF_AGP data (7:0)
```

Synplify

In Synplify, you can set the syn_padtype attribute in SCOPE (the Synplify constraints editor), or in HDL code as shown in the following examples.

VHDL Example

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

entity test_padtype is
    port( A : in std_logic_vector(3 downto 0);
          B : in std_logic_vector(3 downto 0);
          CLK, RST, EN : in std_logic;
          BIDIR : inout std_logic_vector(3 downto 0);
          Q : out std_logic_vector(3 downto 0)
    );

  attribute syn_padtype of A : signal is "SSTL_3_CLASS_I";
  attribute syn_padtype of BIDIR : signal is "HSTL_18_CLASS_III";
  attribute syn_padtype of Q : signal is "LVTTL_33";

end entity;
```

Verilog Example

```
module test_padtype (A, B, CLK, RST, EN, BIDIR, Q);

input [3:0] A /* synthesis syn_padtype = "SSTL_3_CLASS_I" */;
input [3:0] B;
input       CLK, RST, EN;
inout [3:0] BIDIR /* synthesis syn_padtype = "HSTL_18_CLASS_III" */;
output [3:0] Q /* synthesis syn_padtype = "LVTTL_33" */;
```

Precision Synthesis, Synplify and XST

In Precision Synthesis, Synplify and XST, IO standards can be passed by the use of a generic constraint on an instantiated I/O buffer component. See the following examples.

VHDL Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;

entity ibuf_attribute is
   port (DATA, CLOCK, RESET : in std_logic;
         DOUT : out std_logic);
end entity;

architecture XILINX of ibuf_attribute is

   signal data_ibuf, reset_ibuf, dout_obuf : std_logic;

-- remove the following component declarations
-- if using XST or synplify

component IBUF
port (I : in std_logic;
       O : out std_logic);
end component;

component OBUF
port (I : in std_logic;
       O : out std_logic);
end component;

begin

   -- IBUF: Single-ended Input Buffer
   --       All devices
   -- Xilinx  HDL Language Template

   IBUF_PCIX_inst : IBUF
   generic map (
      IOSTANDARD => "PCIX")
   port map (
      O => data_ibuf, -- Buffer output
      I => DATA       -- Buffer input (connect directly to top-level port)
   );
   -- End of IBUF_PCIX_inst instantiation

   -- IBUF: Single-ended Input Buffer
   --       All devices
   -- Xilinx  HDL Language Template

   IBUF_LVCMOS33_inst : IBUF
   generic map (
      IOSTANDARD => "LVCMOS33")
   port map (
      O => reset_ibuf, -- Buffer output
      I => RESET       -- Buffer input (connect directly to top-level port)
   );

   -- End of IBUF_LVCMOS33_inst instantiation

   -- OBUF: Single-ended Output Buffer
   --       All devices
   -- Xilinx  HDL Language Template

   OBUF_LVTTL_inst : OBUF
   generic map (
      DRIVE => 12,
      IOSTANDARD => "LVTTL",
      SLEW => "SLOW")
   port map (
      O => DOUT,      -- Buffer output (connect directly to top-level port)
      I => dout_obuf  -- Buffer input
   );

   -- End of OBUF_LVTTL_inst instantiation

   process (CLOCK)
   begin
     if rising_edge(CLOCK) then
       if reset_ibuf= '1' then
          dout_obuf <= '0';
       else
          dout_obuf <= data_ibuf;
       end if;
     end if;
   end process;

end XILINX;
```

Verilog Example

```
module ibuf_attribute(
    input DATA, RESET, CLOCK,
    output DOUT);

    wire data_ibuf, reset_ibuf;
    reg  dout_obuf;

   // IBUF: Single-ended Input Buffer
   //       All devices
   // Xilinx HDL Language Template
```

```
    IBUF #(
       .IOSTANDARD("PCIX")    // Specify the input I/O standard
    )IBUF_PCIX_inst (
       .O(data_ibuf), // Buffer output
       .I(DATA)       // Buffer input (connect directly to top-level port)
    );

    // End of IBUF_PCIX_inst instantiation

    // IBUF: Single-ended Input Buffer
    //       All devices
    // Xilinx HDL Language Template

    IBUF #(
       .IOSTANDARD("LVCMOS33")    // Specify the input I/O standard
    )IBUF_LVCMOS33_inst (
       .O(reset_ibuf),     // Buffer output
       .I(RESET)      // Buffer input (connect directly to top-level port)
    );

    // End of IBUF_LVCMOS33_inst instantiation

    // OBUF: Single-ended Output Buffer
    //       All devices
    // Xilinx HDL Language Template

    OBUF #(
       .DRIVE(12),   // Specify the output drive strength
       .IOSTANDARD("LVTTL"), // Specify the output I/O standard
       .SLEW("SLOW") // Specify the output slew rate
    ) OBUF_LVTTL_inst (
       .O(DOUT),     // Buffer output (connect directly to top-level port)
       .I(dout_obuf)      // Buffer input
    );

    // End of OBUF_LVTTL_inst instantiation

    always@(posedge CLOCK)
      if(reset_ibuf)
         dout_obuf <= 1'b0;
      else
         dout_obuf <= data_ibuf;

endmodule
```

Outputs

FPGA outputs should have an associated IOSTANDARD specified for each output port in the design. To control the slew rate and drive power, add a constraint to the UCF or synthesis constraints file as follows:

- add the attribute to the output port in the design, or
- modify the generic map or parameter in the instance instantiation of the I/O buffer

Using IOB Register and Latch

This section discusses using IOB Register and Latch with various devices and synthesis tools.

Virtex, Virtex-E, and Spartan-II IOBs

Note: This section applies only to Virtex, Virtex-E, and Spartan-II devices.

Virtex, Virtex-E, and Spartan-II IOBs (Input Output Blocks) contain three storage elements. The three IOB storage elements function either as edge-triggered D-type flip-flops, or as level sensitive latches. Each IOB has a clock (CLK) signal shared by the three flip-flops, and independent clock enable (CE) signals for each flip-flop.

In addition to the CLK and CE control signals, the three flip-flops share a Set/Reset (SR). However, each flip-flop can be independently configured as any of the following:

- synchronous set
- synchronous reset
- asynchronous preset
- an asynchronous clear

FDCP (asynchronous reset and set) and FDRS (synchronous reset and set) register configurations are not available in IOBs.

Virtex-II and Newer IOBs

Note: This section applies only to Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, Spartan-IIE, and Spartan-3 devices.

Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, Spartan-IIE, and Spartan-3 IOBs also contain three storage elements with an option to configure them as FDCP, FDRS, and Dual-Data Rate (DDR) registers. Each register has an independent CE signal. The OTCLK1 and OTCLK2 clock pins are shared between the output and 3-state enable register. A separate clock (ICLK1 and ICLK2) drives the input register. The set and reset signals (SR and REV) are shared by the three registers.

Inferring Usage of Flip-Flops

There are a few ways to infer usage of these flip-flops if the rules for pulling them into the IOB are followed. The following rules apply.

All Devices

All flip-flops that are to be pulled into the IOB must have a fanout of 1. This applies to output and 3-state enable registers. For example, if there is a 32 bit bidirectional bus, the 3-state enable signal must be replicated in the original design so that it has a fanout of 1.

Virtex, Virtex-E, and Spartan-II Devices

In Virtex, Virtex-E, and Spartan-II devices, all flip-flops must share the same clock and reset signal. They can have independent clock enables.

Virtex-II and Newer Devices

In Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, Spartan-IIE, and Spartan-3 devices, output and 3-state enable registers must share the same clock. All flip-flops must share the same set and reset signals.

Virtex-4 Devices

In Virtex-4 devices, the output and 3-state registers share the same clock and set / reset lines. The input registers share the same clock, set / reset and clock enable lines.

Pulling Flip-Flops into the IOB

One way you can pull flip-flops into the IOB is to use the IOB=TRUE setting. Another way is to pull flip-flops into the IOB using the **map –pr** command, which is discussed in a later section. Some synthesis tools apply the IOB=TRUE attribute and allow you to merge a flip-flop to an IOB by setting an attribute. For more information about the correct attribute and settings, see your synthesis tool documentation.

LeonardoSpectrum

LeonardoSpectrum, through ISE, can push registers into IOBs:

1. Right click the **Synthesize** process.
2. Click **Properties**.
3. Click the Architecture Options tab.
4. Enable the **Map to IOB Registers** setting

In standalone LeonardoSpectrum, you can select **Map IOB Registers** from the Technology tab in the application or set the following attribute in your TCL script:

```
set virtex_map_iob_registers TRUE
```

Synplify

In Synplify, attach the SYN_USEIOFF attribute to the module or architecture of the top-level in one of these ways:

- Add the attribute in SCOPE. The constraint file syntax looks like this:

  ```
  define_global_attribute syn_useioff 1
  ```

- Add the attribute in the VHDL or Verilog top-level source code as follows:

  VHDL Example

  ```
  architecture rtl of test is
    attribute syn_useioff : boolean;
    attribute syn_useioff of rtl : architecture is true;
  ```

  Verilog Example

  ```
  module test(d, clk, q) /* synthesis syn_useioff = 1 */;
  ```

Using Dual Data Rate IOB Registers

The following VHDL and Verilog examples demostrate how to infer dual data rate registers for inputs only. For an attribute to enable I/O register inference in your synthesis tool, see "Using IOB Register and Latch" in this chapter. The dual data rate register primitives (the synchronous set/reset with clock enable FDDRRSE, and asynchronous set/reset with clock enable FDDRCPE) must be instantiated in order to utilize the dual data rate registers in the outputs. For information on instantiating primitive, see "Instantiating Components" in this chapter.

VHDL Example

```
library ieee;
use ieee.std_logic_1164.all;
entity ddr_input is
  port (
        clk : in std_logic;
        d   : in std_logic;
        rst : in std_logic;
        q1  : out std_logic;
        q2  : out std_logic
        );
end ddr_input;
architecture behavioral of ddr_input is
begin
  q1reg : process (clk, rst)
  begin
    if rst = '1' then
        q1 <= '0';
    elsif clk'event and clk='1' then
        q1 <= d;
    end if;
  end process;
  q2reg : process (clk, rst)
  begin
    if rst = '1' then
        q2 <= '0';
    elsif clk'event and clk='0' then
        q2 <= d;
    end if;
  end process;
end behavioral;
```

Verilog Example

```
module ddr_input (
input data_in, clk, rst,
output data_out);

reg q1, q2;
always @ (posedge clk, posedge rst)
  begin
    if (rst)
        q1 <=1'b0;
    else
        q1 <= data_in;
  end
always @ (negedge clk, posedge rst)
  begin
    if (rst)
        q2 <=1'b0;
    else
        q2 <= data_in;
  end
assign data_out = q1 & q2;
end module
```

Using Output Enable IOB Register

The following VHDL and Verilog examples illustrate how to infer an output enable register. For an attribute to turn on I/O register inference in synthesis tools, see the above section.

VHDL Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tri_state is
port (
       DATA_IN_P : in std_logic_vector(7 downto 0);
       CLK : in std_logic;
       TRI_STATE_A : in std_logic;
       DATA_OUT : out std_logic_vector(7 downto 0)
       );
end tri_state;

architecture behavioral of tri_state is

   signal data_in_reg : std_logic_vector(7 downto 0);
   signal data_out_reg : std_logic_vector(7 downto 0);
   signal tri_state_bus : std_logic_vector(7 downto 0);
begin

   process (tri_state_bus, data_out_reg)
   begin
      G2:  for J in 0 to 7 loop
         if (tri_state_bus(J) = '0') then  -- 3-state data_out
            DATA_OUT(J) <= data_out_reg(J);
         else
            DATA_OUT(J) <= 'Z';
         end if;
      end loop;
   end process;

   process(CLK)
   begin
      if CLK'event and CLK='1' then
         data_in_reg <= DATA_IN_P;     -- register for input
         data_out_reg <= data_in_reg;  -- register for output
         if (TRI_STATE_A = '0') then   -- register and replicate 3state signal
            tri_state_bus <= "00000000";
         else
            tri_state_bus <= "11111111";
         end if;
      end if;
   end process;

end behavioral;
```

Verilog Example

```
module tri_state (
   input [7:0] DATA_IN_P,
   input CLK, TRI_STATE_A,
   output reg [7:0] DATA_OUT);

   reg[7:0] data_in_reg;
   reg[7:0] data_out_reg;
   reg[7:0] tri_state_bus;

   integer J;

   always @(*)
      for (J = 0; J <= 7; J = J + 1)
      begin : G2
         if (!tri_state_bus[J])
            DATA_OUT[J] <= data_out_reg[J];
         else
            DATA_OUT[J] <= 1'bz;
      end

   always @(posedge clk) begin
      data_in_reg <= DATA_IN_P;           // register for input
      data_out_reg <= data_in_reg;        // register for output
      tri_state_bus <= {8{TRI_STATE_A}};  // register and replicate 3state signal
   end
endmodule
```

Using the Pack Registers Option with Map

Use the pack registers (**-pr**) option when running Map. This option tells the Map program to move registers into IOBs when possible. Use the following syntax.

```
map -pr {i|o|b} input_file_name |output_file_name
```

Example:

```
map -pr b design_name.ngd
```

Within Project Navigator, this option is called “Pack I/O Registers/Latches into IOB." It is defaulted to "For Inputs and Outputs" or **map -pr b**.

Virtex-E and Spartan-IIE IOBs

Note: This section applies only to Virtex-E and Spartan-IIE devices.

Virtex-E and Spartan-IIE devices have the same IOB structure and features as Virtex and Spartan-II devices except for the available I/O standards.

Additional I/O Standards for Virtex-E Devices

Virtex-E devices have two additional I/O standards: LVPECL and LVDS.

Because LVDS and LVPECL require two signal lines to transmit one data bit, it is handled differently from any other I/O standards. A UCF or an NGC file with complete pin LOC information must be created to ensure that the I/O banking rules are not violated. If a UCF or NGC file is not used, PAR issues errors.

The input buffer of these two I/O standards may be placed in a wide number of IOB locations. The exact locations are dependent on the package that is used. The Virtex-E™ package information lists the possible locations as IO_L#P for the P-side and IO_L#N for the N-side where # is the pair number. Only one input buffer is required to be instantiated in the design and placed on the correct IO_L#P location. The N-side of the buffer is reserved and no other IOB is allowed on this location.

The output buffer may be placed in a wide number of IOB locations. The exact locations are dependent on the package that is used. The Virtex-E package information lists the possible locations as IO_L#P for the P-side and IO_L#N for the N-side where # is the pair number. However, both output buffers are required to be instantiated in the design and placed on the correct IO_L#P and IO_L#N locations. In addition, the output (O) pins must be inverted with respect to each other. (one HIGH and one LOW). Failure to follow these rules leads to DRC errors in the software.

Coding Examples for LVDS I/O Standards

The following examples show VHDL and Verilog coding for LVDS I/O standards targeting a V50ECS144 device. An AUCF example is also provided.

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;
entity LVDSIO is
  port (
        CLK, DATA, Tin : in STD_LOGIC;
        IODATA_p, IODATA_n : inout STD_LOGIC;
        Q_p, Q_n : out STD_LOGIC
        );
end LVDSIO;
architecture BEHAV of LVDSIO is
-- remove the following component declarations
-- if using XST or synplify
  component IBUF_LVDS is port (
        I : in STD_LOGIC;
        O : out STD_LOGIC
        );
  end component;
  component OBUF_LVDS is port (
        I : in STD_LOGIC;
        O : out STD_LOGIC
        );
  end component;
  component IOBUF_LVDS is port (
        I : in STD_LOGIC;
        T : in STD_LOGIC;
        IO : inout STD_LOGIC;
        O : out STD_LOGIC
        );
  end component;
  component INV is port (
        I : in STD_LOGIC;
        O : out STD_LOGIC
        );
```

```
end component;
component IBUFG_LVDS is port(
     I : in STD_LOGIC;
     O : out STD_LOGIC
     );
end component;
component BUFG is port(
     I : in STD_LOGIC;
     O : out STD_LOGIC
     );
end component;
signal iodata_in    : std_logic;
signal iodata_n_out : std_logic;
signal iodata_out   : std_logic;
signal DATA_int     : std_logic;
signal Q_p_int      : std_logic;
signal Q_n_int      : std_logic;
signal CLK_int      : std_logic;
signal CLK_ibufgout : std_logic;
signal Tin_int      : std_logic;
begin
  UI1: IBUF_LVDS port map (
     I => DATA,
     O => DATA_int
     );
  UI2: IBUF_LVDS port map (
     I => Tin,
     O => Tin_int
     );
  UO_p: OBUF_LVDS port map (
     I => Q_p_int,
     O => Q_p
     );
  UO_n: OBUF_LVDS port map (
     I => Q_n_int,
     O => Q_n
     );
  UIO_p: IOBUF_LVDS port map (
     I => iodata_out,
     T => Tin_int,IO => iodata_p,
     O => iodata_in
     );
  UIO_n: IOBUF_LVDS port map (
     I => iodata_n_out,
     T => Tin_int,
     IO => iodata_n,
     O => open
     );
  UINV: INV port map (
     I => iodata_out,
     O => iodata_n_out
     );
  UIBUFG: IBUFG_LVDS port map (
     I => CLK,
     O => CLK_ibufgout
     );
```

```
    UBUFG: BUFG port map (
        I => CLK_ibufgout,
        O => CLK_int
        );

  My_D_Reg: process (CLK_int, DATA_int)
  begin
        if (CLK_int'event and CLK_int='1') then
          Q_p_int <= DATA_int;
        end if;
    end process;         -- End My_D_Reg
  iodata_out <= DATA_int and iodata_in;
  Q_n_int <= not Q_p_int;
end BEHAV;
```

Verilog Example

```
/////////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
////////////////////////////////////////////////////////

module LVDSIOinst (
 input CLK, DATA, Tin,
 inout IODATA_p, IODATA_n,
 output  Q_p, Q_n);

wire iodata_in;
wire iodata_n_out;
wire iodata_out;
wire DATA_int;
reg Q_p_int;
wire Q_n_int;
wire CLK_int;
wire CLK_ibufgout;
wire Tin_int;

IBUF_LVDS UI1 (.I(DATA), .O(DATA_int));
IBUF_LVDS UI2 (.I(Tin), .O (Tin_int));
OBUF_LVDS UO_p (.I(Q_p_int), .O(Q_p));
OBUF_LVDS UO_n (.I(Q_n_int), .O(Q_n));
IOBUF_LVDS UIO_p(
        .I(iodata_out),
        .T(Tin_int)
        .IO(IODATA_p),
        .O(iodata_in)
        );
IOBUF_LVDS UIO_n (
        .I (iodata_n_out),
        .T(Tin_int),
        .IO(IODATA_n), .O ()
        );
INV UINV ( .I(iodata_out), .O(iodata_n_out));
IBUFG_LVDS UIBUFG ( .I(CLK), .O(CLK_ibufgout));
BUFG UBUFG ( .I(CLK_ibufgout), .O(CLK_int));

always @ (posedge CLK_int)
  begin
    Q_p_int <= DATA_int;
  end
  assign iodata_out = DATA_int && iodata_in;
  assign Q_n_int = ~Q_p_int;
endmodule
```

UCF Example Targeting V50ECS144

```
NET CLK LOC = A6;        #GCLK3
NET DATA LOC = A4;       #IO_L0P_YY
NET Q_p LOC = A5;        #IO_L1P_YY
NET Q_n LOC = B5;        #IO_L1N_YY
NET iodata_p LOC = D8;   #IO_L3P_yy
NET iodata_n LOC = C8;   #IO_L3N_yy
NET Tin LOC = F13;       #IO_L10P
```

Coding Examples Using the IOSTANDARD Generic or Parameter

The following examples use the IOSTANDARD generic (VHDL) or parameter (Verilog) on I/O buffers as a work around for LVDS buffers. This example can also be used with other synthesis tools to configure I/O standards with the IOSTANDARD generic (VHDL) or parameter (Verilog).

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;

entity flip_flop is
  port(
    d : in std_logic;
    clk : in std_logic;
    q : out std_logic;
    q_n : out std_logic
  );
end flip_flop;

architecture flip_flop_arch of flip_flop is

-- remove the following component declarations
-- if using XST or synplify

  component IBUF
    generic(IOSTANDARD : string);
    port(
      I: in std_logic;
      O: out std_logic);
  end component;

  component OBUF
    generic(IOSTANDARD : string);
    port(
      I: in std_logic;
      O: out std_logic);
  end component;

------------------------------------------------
```

```
-- Pin location A5 on the cs144
-- package represents the
-- 'positive' LVDS pin.
-- Pin location D8 represents the
-- 'positive' LVDS pin.
-- Pin location C8 represents the
-- 'negative' LVDS pin.
-----------------------------------------------
  attribute LOC of u1 : label is "A5";
  attribute LOC of u2 : label is "D8";
  attribute LOC of u3 : label is "C8";

  signal d_lvds, q_lvds, q_lvds_n : std_logic;

begin
  u1 : IBUF generic map("LVDS") port map (d,d_lvds);
  u2 : OBUF generic map("LVDS") port map (q_lvds,q);
  u3 : OBUF generic map("LVDS") port map (q_lvds_n,q_n);

  process (clk) begin
    if clk'event and clk = '1' then
      q_lvds <= d_lvds;
    end if;
  end process;

  q_lvds_n <= not(q_lvds);

end flip_flop_arch;
```

Verilog Example

```
///////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify>\lib\xilinx\unisim.v"
///////////////////////////////////////////////////////
module flip_flop (d, clk, q, q_n);

/////////////////////////////////////
// Pin location A5 on the Virtex-E
// cs144 package represents the
// 'positive' LVDS pin.
// Pin location D8 represents the
// 'positive' LVDS pin.
// Pin location C8 represents the
// 'negative' LVDS pin.
/////////////////////////////////////

  input clk;

(*LOC = "A5" *) input d;
(*LOC = "D8" *) output q;
(*LOC = "C8" *) output q_n;

  wire d, clk, d_lvds, q;
  reg q_lvds;

  IBUF #(.IOSTANDARD("LVDS")) u1 (.I(d),.O(d_lvds));
  OBUF #(.IOSTANDARD("LVDS")) u2 (.I(q_lvds),.O(q));
```

```
    OBUF #(.IOSTANDARD("LVDS")) u3 (.I(q_lvds_n),.O(q_n));

    always @(posedge clk) q_lvds <= d_lvds;
    assign q_lvds_n=~q_lvds;

  endmodule
```

Virtex-II and Newer IOBs

Note: This section applies only to Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Spartan-3 devices.

Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Spartan-3 devices offer more SelectIO configuration than Virtex, Virtex-E, and Spartan-II devices. IOSTANDARD and synthesis tools' specific attributes can be used to configure the SelectIO.

Additionally, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Spartan-3 provide digitally controlled impedance (DCI) I/Os which are useful in improving signal integrity and avoiding the use of external resistors. This option is only available for most of the single ended I/O standards. To access this option, instantiate the 'DCI' suffixed I/Os from the library such as HSTL_IV_DCI.

For low-voltage differential signaling, additional IBUFDS, OBUFDS, OBUFTDS, and IOBUFDS components are available. These components simplify the task of instantiating the differential signaling standard.

Differential Signaling

Differential signaling in Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Spartan-3 devices can be configured using IBUFDS, OBUFDS, and OBUFTDS. The IBUFDS is a two-input one-output buffer. The OBUFDS is a one-input two-output buffer. For the component diagram and description, see the Xilinx *Libraries Guides*. For information on the supported differential I/O standards for the target FPGA architecture, see the product data sheet and user guide.

Note: Virtex-E treats differential signals differently than later architectures. For more information, see Xilinx Answer Record 9174, "*Virtex-E - How do I use LVDS, LVPECL macros (such as IBUFDS_FD_LVDS, OBUFDS_FD_LVDS) in designs?*"

Differential Signaling Coding Examples

The following VHDL and Verilog examples show how to instantiate differential signaling buffers.

VHDL Example

```
--------------------------------------------
-- LVDS_33_IO.VHD Version 1.0
-- Example of a behavioral description of
-- differential signal I/O standard using
-- LeonardoSpectrum attribute.
-- HDL Synthesis Design Guide for FPGA devices
--------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;
entity LVDS_33_IO is
```

```
    port(
        CLK_p,CLK_n,DATA_p,DATA_n, Tin_p,Tin_n : in STD_LOGIC;
        datain2_p, datain2_n : in STD_LOGIC;
        ODATA_p, ODATA_n : out STD_LOGIC;
        Q_p, Q_n : out STD_LOGIC
        );
end LVDS_33_IO;
architecture BEHAV of LVDS_33_IO is

-- remove the following component declarations
-- if using XST or synplify

  component IBUFDS is
    port (
        I : in STD_LOGIC;
        IB : in STD_LOGIC;
        O : out STD_LOGIC
        );
  end component;
  component OBUFDS is
    port (
        I : in STD_LOGIC;
        O : out STD_LOGIC;
        OB : out STD_LOGIC
        );
  end component;
  component OBUFTDS is
    port (
        I : in STD_LOGIC;
        T : in STD_LOGIC;
        O : out STD_LOGIC;
        OB : out STD_LOGIC
        );
  end component;
  component IBUFGDS is
    port(
        I : in STD_LOGIC;
        IB : in STD_LOGIC;
        O : out STD_LOGIC
        );
  end component;
  component BUFG is
    port(
        I : in STD_LOGIC;
        O :out STD_LOGIC
        );
  end component;

  signal datain2 : std_logic;
  signal odata_out : std_logic;
  signal DATA_int : std_logic;
  signal Q_int : std_logic;
  signal CLK_int : std_logic;
  signal CLK_ibufgout : std_logic;
  signal Tin_int : std_logic;
begin
  UI1 : IBUFDS port map (I => DATA_p, IB => DATA_n, O => DATA_int);
  UI2 : IBUFDS port map (I => datain2_p,IB => datain2_n,O => datain2);
  UI3 : IBUFDS port map (I => Tin_p,IB => Tin_n,O => Tin_int);
  UO1 : OBUFDS port map (I => Q_int,O => Q_p,OB => Q_n);
  UO2 : OBUFTDS port map (
       I => odata_out,
       T => Tin_int,
       O => odata_p,
       OB => odata_n
       );
  UIBUFG : IBUFGDS port map (I => CLK_p,IB => CLK_n,O => CLK_ibufgout);
  UBUFG  : BUFG port map (I => CLK_ibufgout,O => CLK_int);
  My_D_Reg: process (CLK_int, DATA_int)
    begin
      if (CLK_int'event and CLK_int='1') then
         Q_int <= DATA_int;
      end if;
  end process;       -- End My_D_Reg
  odata_out <= DATA_int and datain2;
end BEHAV;
```

Verilog Example

```
////////////////////////////////////////////
// LVDS_33_IO.v Version 1.0
// Example of a behavioral description of
// differential signal I/O standard
// HDL Synthesis Design Guide for FPGA devices
////////////////////////////////////////////

//////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
//////////////////////////////////////////////////////

module LVDS_33_IO (
input CLK_p, CLK_n, DATA_p, DATA_n, DATAIN2_p, DATAIN2_n, Tin_p, Tin_n,
output ODATA_p, ODATA_n, Q_p, Q_n);

wire datain2;
wire odata_in;
wire odata_out;
wire DATA_int;
reg Q_int;
wire CLK_int;
wire CLK_ibufgout;
wire Tin_int;
IBUFDS UI1 (
       .I (DATA_p),
       .IB(DATA_n),
       .O (DATA_int)
       );
IBUFDS UI2 (
       .I (Tin_p),
       .IB(Tin_n),
       .O (Tin_int)
       );
IBUFDS UI3 ( .I(DATAIN2_p), .IB(DATAIN2_n), .O(datain2));
OBUFDS UO1 ( .I(Q_int), .O(Q_p), .OB(Q_n));
OBUFTDS UO2 ( .I(odata_out), .T(Tin_int), .O(ODATA_p), .OB(ODATA_n));
IBUFGDS UIBUFG ( .I(CLK_p), .IB(CLK_n), .O(CLK_ibufgout));
BUFG UBUFG ( .I(CLK_ibufgout), .O(CLK_int));
```

```
always @ (posedge CLK_int)
  begin
    Q_int <= DATA_int;
  end
assign odata_out = DATA_int && datain2;
endmodule
```

Encoding State Machines

The traditional methods used to generate state machine logic result in highly-encoded states. State machines with highly-encoded state variables typically have a minimum number of flip-flops and wide combinatorial functions. These characteristics were acceptable for ASIC, PAL and gate array architectures. However, because FPGA devices have many flip-flops and 4-input function generators, highly-encoded state variables can result in inefficient implementation in terms of speed and density.

One-hot encoding allows you to create state machine implementations that are more efficient for FPGA architectures. You can create state machines with one flip-flop per state and decreased width of combinatorial logic for both state-encoding and output encoding. One-hot encoding is usually the preferred method for large FPGA based state machine implementation. For small state machines (fewer than 8 states), binary encoding may be more efficient. To improve design performance, you can divide large (greater than 32 states) state machines into several small state machines and use the appropriate encoding style for each

Three design examples are provided in this section to illustrate the three coding methods (binary, enumerated type, and one-hot) you can use to create state machines. All three examples contain the same Case statement. To conserve space, the complete Case statement is only included in the binary encoded state machine example; refer to this example when reviewing the enumerated type and one-hot examples.

Some synthesis tools allow you to add an attribute, such as TYPE_ENCODING_STYLE, to your VHDL code to set the encoding style. This is a synthesis vendor attribute (not a Xilinx® attribute). For more information on attribute-driven state machine synthesis, see your synthesis tool documentation.

Using Binary Encoding

The state machine bubble diagram in the following figure shows the operation of a seven-state machine that reacts to inputs A through E as well as previous-state conditions. The binary encoded method of coding this state machine is shown in the VHDL and Verilog examples that follow. These design examples show you how to take a design that has been previously encoded (for example, binary encoded) and synthesize it to the appropriate decoding logic and registers. These designs use three flip-flops to implement seven states.

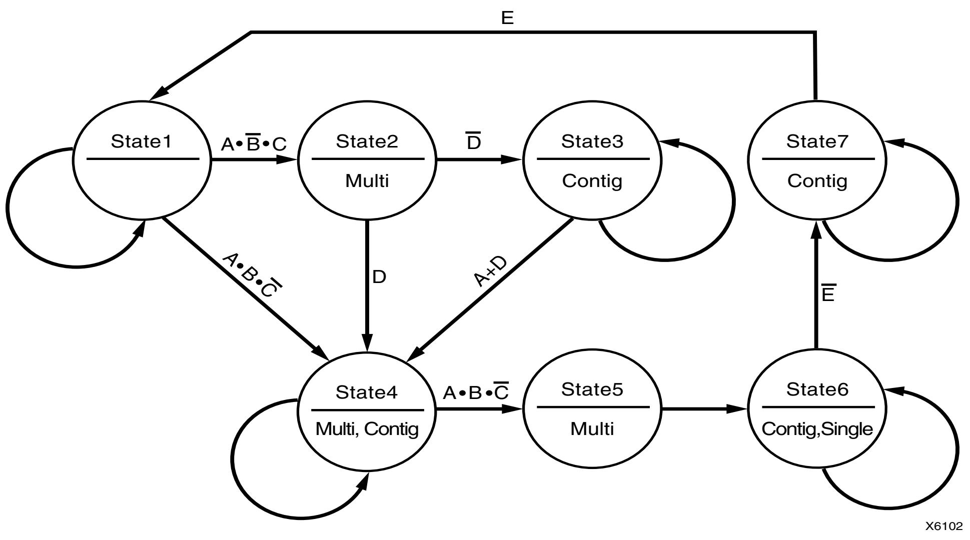

Figure 4-3: **State Machine Bubble Diagram**

Binary Encoded State Machine VHDL Example

The following is a binary encoded state machine VHDL example.

```
-------------------------------------------------
-- BINARY.VHD Version 1.0
-- Example of a binary encoded state machine
-------------------------------------------------
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity binary is
  port (
        CLOCK, RESET : in STD_LOGIC;
        A, B, C, D, E : in BOOLEAN;
        SINGLE, MULTI, CONTIG : out STD_LOGIC
        );
end binary;

architecture BEHV of binary is

  type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
  attribute ENUM_ENCODING: STRING;
  attribute ENUM_ENCODING of STATE_TYPE:type is
        "001 010 011 100 101 110 111";
  signal CS, NS: STATE_TYPE;

  begin
    SYNC_PROC: process (CLOCK, RESET)
    begin
      if (RESET='1') then
         CS <= S1;
      elsif (CLOCK'event and CLOCK = '1') then
         CS <= NS;
      end if;
    end process; --End REG_PROC
```

```
COMB_PROC: process (CS, A, B, C, D, E)
begin
  case CS is
    when S1 =>
        MULTI <= '0';
        CONTIG <= '0';
        SINGLE <= '0';
        if (A and not B and C) then
          NS <= S2;
        elsif (A and B and not C) then
          NS <= S4;
        else
          NS <= S1;
        end if;

    when S2 =>
        MULTI <= '1';
        CONTIG <= '0';
        SINGLE <= '0';
        if (not D) then
            NS <= S3;
        else
            NS <= S4;
        end if;

    when S3 =>
        MULTI <= '0';
        CONTIG <= '1';
        SINGLE <= '0';
        if (A or D) then
            NS <= S4;
        else
            NS <= S3;
        end if;

    when S4 =>
        MULTI <= '1';
        CONTIG <= '1';
        SINGLE <= '0';
        if (A and B and not C) then
            NS <= S5;
        else
            NS <= S4;
        end if;

    when S5 =>
        MULTI <= '1';
        CONTIG <= '0';
        SINGLE <= '0';
        NS <= S6;

    when S6 =>
        MULTI <= '0';
        CONTIG <= '1';
        SINGLE <= '1';
        if (not E) then
            NS <= S7;
        else
```

```
                NS <= S6;
            end if;

        when S7 =>
            MULTI <= '0';
            CONTIG <= '1';
            SINGLE <= '0';
            if (E) then
                NS <= S1;
            else
                NS <= S7;
            end if;
    end case;
  end process; -- End COMB_PROC
end BEHV;
```

Binary Encoded State Machine Verilog Example

```
module binary (
 input  CLOCK, RESET, A, B, C, D, E;
 output reg SINGLE, MULTI, CONTIG);

// Declare the symbolic names for states
parameter
    S1 = 3'b001,
    S2 = 3'b010,
    S3 = 3'b011,
    S4 = 3'b100,
    S5 = 3'b101,
    S6 = 3'b110,
    S7 = 3'b111;

// Declare current state and next state variables
reg [2:0] CS;
reg [2:0] NS;

// state_vector CS
  always @ (posedge CLOCK, posedge RESET)
    begin
      if (RESET == 1'b1)
        CS <= S1;
      else
        CS <= NS;
    end
  always @ (*)
    begin
      case (CS)
        S1 :
          begin
            MULTI = 1'b0;
            CONTIG = 1'b0;
            SINGLE = 1'b0;
            if (A && ~B && C)
                NS = S2;
            else if (A && B && ~C)
                NS = S4;
            else
              NS = S1;
          end
```

```
S2 :
  begin
    MULTI = 1'b1;
    CONTIG = 1'b0;
    SINGLE = 1'b0;
    if (!D)
        NS = S3;
    else
        NS = S4;
  end
S3 :
  begin
    MULTI = 1'b0;
    CONTIG = 1'b1;
    SINGLE = 1'b0;
    if (A || D)
        NS = S4;
    else
      NS = S3;
  end

S4 :
  begin
    MULTI = 1'b1;
    CONTIG = 1'b1;
    SINGLE = 1'b0;
    if (A && B && ~C)
        NS = S5;
    else
        NS = S4;
  end

S5 :
  begin
    MULTI = 1'b1;
    CONTIG = 1'b0;
    SINGLE = 1'b0;
    NS = S6;
  end

S6 :
  begin
    MULTI = 1'b0;
    CONTIG = 1'b1;
    SINGLE = 1'b1;
    if (!E)
        NS = S7;
    else
        NS = S6;
  end

S7 :
  begin
    MULTI = 1'b0;
    CONTIG = 1'b1;
    SINGLE = 1'b0;
    if (E)
        NS = S1;
```

```
                else
                    NS = S7;
              end
          endcase
        end
      endmodule
```

Using Enumerated Type Encoding

The recommended encoding style for state machines depends on which synthesis tool you are using. Some synthesis tools encode better than others depending on the device architecture and the size of the decode logic. You can explicitly declare state vectors or you can allow your synthesis tool to determine the vectors. Xilinx® recommends that you use enumerated type encoding to specify the states and use the Finite State Machine (FSM) extraction commands to extract and encode the state machine as well as to perform state minimization and optimization algorithms. The enumerated type method of encoding the seven-state machine is shown in the following VHDL and Verilog examples. The encoding style is not defined in the code, but can be specified later with the FSM extraction commands. Alternatively, you can allow your compiler to select the encoding style that results in the lowest gate count when the design is synthesized. Some synthesis tools automatically find finite state machines and compile without the need for specification.

For the complete case statement portion of the code, see the previous VHDL and Verilog Binary Encoded State Machine examples.

Enumerated Type Encoded State Machine VHDL Example

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity enum is
  port (
        CLOCK, RESET : in STD_LOGIC;
        A, B, C, D, E : in BOOLEAN;
        SINGLE, MULTI, CONTIG : out STD_LOGIC
        );
end enum;

architecture BEHV of enum is

  type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
  signal CS, NS: STATE_TYPE;

  begin
    SYNC_PROC: process (CLOCK, RESET)
    begin
      if (RESET='1') then
          CS <= S1;
      elsif (CLOCK'event and CLOCK = '1') then
          CS <= NS;
      end if;
    end process; --End SYNC_PROC
    COMB_PROC: process (CS, A, B, C, D, E)
    begin
      case CS is
          when S1 =>
              MULTI <= '0';
              CONTIG <= '0';
```

```
                SINGLE <= '0';
.
.
.
```

Enumerated Type Encoded State Machine Verilog Example

```
/////////////////////////////////////////////////////
// ENUM.V Version 1.0
// Example of an enumerated encoded state machine
/////////////////////////////////////////////////////

module enum (
 input CLOCK, RESET, A, B, C, D, E,
 output reg SINGLE, MULTI, CONTIG);

// Declare the symbolic names for states
parameter
    S1 = 3'b000,
    S2 = 3'b001,
    S3 = 3'b010,
    S4 = 3'b011,
    S5 = 3'b100,
    S6 = 3'b101,
    S7 = 3'b110;

// Declare current state and next state variables
reg [2:0] CS;
reg [2:0] NS;

// state_vector CS
always @ (posedge CLOCK, posedge RESET)
  begin
    if (RESET == 1'b1)
        CS <= S1;
    else
        CS <= NS;
  end

always @ (*)
  begin
    case (CS)
      S1 :
      begin
        MULTI = 1'b0;
        CONTIG = 1'b0;
        SINGLE = 1'b0;
        if (A && ~B && C)
            NS = S2;
        else if (A && B && ~C)
            NS = S4;
        else
            NS = S1;
      end
.
.
.
```

Using One-Hot Encoding

One-hot encoding allows you to create state machine implementations that are more efficient for FPGA architectures. One-hot encoding is usually the preferred method for large FPGA-based state machine implementation.

The following examples show a one-hot encoded state machine. Use this method to control the state vector specification or when you want to specify the names of the state registers. These examples use one flip-flop for each of the seven states.

For the complete case statement portion of the code, see the previous VHDL and Verilog Binary Encoded State Machine examples.

One-Hot Encoded State Machine VHDL Example

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity one_hot is
  port (
        CLOCK, RESET : in STD_LOGIC;
        A, B, C, D, E : in BOOLEAN;
        SINGLE, MULTI, CONTIG : out STD_LOGIC
        );
end one_hot;

architecture BEHV of one_hot is
  type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
  attribute ENUM_ENCODING: STRING;
  attribute ENUM_ENCODING of STATE_TYPE: type is
    "0000001 0000010 0000100 0001000 0010000 0100000 1000000 ";
  signal CS, NS: STATE_TYPE;

  begin
    SYNC_PROC: process (CLOCK, RESET)
    begin
      if (RESET='1') then
         CS <= S1;
    elsif (CLOCK'event and CLOCK = '1') then
         CS <= NS;
    end if;
    end process;     --End SYNC_PROC

    COMB_PROC: process (CS, A, B, C, D, E)
      begin
        case CS is
          when S1 =>
              MULTI <= '0';
              CONTIG <= '0';
              SINGLE <= '0';
              if (A and not B and C) then
                  NS <= S2;
              elsif (A and B and not C) then
                  NS <= S4;
              else
                  NS <= S1;
              end if;
:
```

One-Hot Encoded State Machine Verilog Example

```
/////////////////////////////////////////////////////////
// ONE_HOT.V Version 1.0
// Example of a one-hot encoded state machine
// Xilinx HDL Synthesis Design Guide for FPGA devices
////////////////////////////////////////////////////////

module one_hot (
 input  CLOCK, RESET, A, B, C, D, E,
 output reg SINGLE, MULTI, CONTIG);

// Declare the symbolic names for states
parameter
    S1 = 7'b0000001,
    S2 = 7'b0000010,
    S3 = 7'b0000100,
    S4 = 7'b0001000,
    S5 = 7'b0010000,
    S6 = 7'b0100000,
    S7 = 7'b1000000;

// Declare current state and next state variables
(* signal_encoding = "user" *) // directive for XST
(* fsm_encoding = "user" *)  // directive for XST
reg [2:0] CS /* synthesis syn_encoding="original" */;
//directive for Synplify Pro
(* signal_encoding = "user" *) // directive for XST
reg [2:0] NS;

// state_vector CS

  always @ (posedge CLOCK, posedge RESET)
    begin
      if (RESET == 1'b1)
        CS <= S1;
      else
        CS <= NS;
    end

always @ (*)
  begin
    case (CS)
      S1 :
      begin
        MULTI = 1'b0;
        CONTIG = 1'b0;
        SINGLE = 1'b0;
        if (A && ~B && C)
            NS = S2;
        else if (A && B && ~C)
            NS = S4;
        else
            NS = S1;
      end
:
:
```

Accelerating FPGA Macros with One-Hot Approach

Most synthesis tools provide a setting for finite state machine (FSM) encoding. This setting prompts synthestools to automatically extract state machines in your design and perform special optimizations to achieve better performance. The default option for FSM encoding is "One-Hot" for most synthesis tools. However, this setting can be changed to other encoding such as binary, gray, or sequential.

In LeonardoSpectrum, FSM encoding is set to "Auto" by default, which differs depending on the Bit Width of your state machine. To change this setting to a specific value, select the Input tab. Available options are: Binary, One-Hot, Random, Gray, and Auto.

In Synplify Pro, the Symbolic FSM Compiler option can be accessed from the main application. When set, the FSM Compiler extracts the state machines as symbolic graphs, and then optimizes them by re-encoding the state representations and generating a better logic optimization starting point for the state machines. This usually results in one-hot encoding. However, you may override the default on a register by register basis with the SYN_ENCODING directive/attribute. Available options are: default, onehot, gray, sequential, safe and original.

In XST, FSM encoding is set to Auto by default. Available options are: Auto, One-Hot, Compact, Gray, Johnson, Sequential, and User. The related directives are fsm_encoding and signal_encoding.

Note: XST recognizes enumerated encoding only if the encoding option is set to User.

Summary of Encoding Styles

In the previous examples, the state machine’s possible states are defined by an enumeration type. Use the following syntax to define an enumeration type.

`type` *type_name* **`is`** (*enumeration_literal* {, *enumeration_literal*});

After you have defined an enumeration type, declare the signal representing the states as the enumeration type as follows.

```
type STATE_TYPE is (S1, S2, S3, S4, S5, S6, S7);
signal CS, NS: STATE_TYPE;
```

The state machine described in the previous examples has seven states. The possible values of the signals CS (Current_State) and NS (Next_State) are S1, S2, ... , S6, S7.

To select an encoding style for a state machine, specify the state vectors. Alternatively, you can specify the encoding style when the state machine is compiled. Xilinx® recommends that you specify an encoding style. If you do not specify a style, your compiler selects a style that minimizes the gate count. For the state machine shown in the three previous examples, the compiler selected the binary encoded style: S1=“000”, S2=”001”, S3=”010”, S4=”011”, S5=”100”, S6=”101”, and S7=”110”.

You can use the FSM extraction tool to change the encoding style of a state machine. For example, use this tool to convert a binary-encoded state machine to a one-hot encoded state machine.

For instructions on how to extract the state machine and change the encoding style, see your synthesis tool documentation.

Initializing the State Machine

When creating a state machine, especially when you use one-hot encoding, add the following lines of code to your design to ensure that the FPGA is initialized to a Set state.

Initializing the State Machine VHDL Example

```
SYNC_PROC: process (CLOCK, RESET)
begin
  if (RESET='1') then
      CS <= s1;
```

Initializing the State Machine Verilog Example

```
always @ (posedge CLOCK, posedge RESET)
  begin
    if (RESET == 1'b1)
        CS <= S1;
```

Alternatively, you can assign an INIT=S attribute to the initial state register to specify the initial state. For information on assigning this attribute, see your synthesis tool documentation.

In the Binary Encode State Machine example, the RESET signal forces the S1 flip-flop to be preset (initialized to 1) while the other flip-flops are cleared (initialized to 0).

Implementing Operators and Generating Modules

Xilinx FPGA devices feature carry logic elements that can be used for optimal implementation of operators and to generate modules. Synthesis tools infer the carry logic automatically when a specific coding style or operator is used.

Using the DSP48 Block

With the release of the Virtex-4 architecture, Xilinx introduced a new primitive called the DSP48. This element allows you to create numerous functions, including multipliers, adders, counters, barrel shifters, comparators, accumulators, multiply accumulate, complex multipliers, and others. For more information about the DSP48 slice, see the *XtremeDSP Design Considerations User Guide*. Currently tools can map multipliers, adders, multiply adds, multiply accumulates and some form of FIR filters. The synthesis tools also take advantage of the internal registers available in the DSP48 as well as the dynamic OPMODE port. Future enhancements to the synthesis tools will improve retiming and cascading of DSP48 resources.

Resources

The following application notes and white papers provide information on DSP48 support from Mentor Graphics Precision Synthesis and Synplicity's Synplify and Synplify Pro.

Synplicity

Using Virtex4 DSP48 Components with the Synplify Pro Software at
http://www.synplicity.com/literature/pdf/dsp48.pdf

Mentor Graphics

Using Precision Synthesis to Design with the XtremeDSP Slice in Virtex-4, available from http://www.mentor.com/products/fpga_pld/techpubs/index.cfm

To obtain this paper:

1. Go to the Mentor Graphics website.
2. Follow the steps specified on the website.
3. Select *Using Precision Synthesis to Design with the XtremeDSP Slice in Virtex-4.*
4. Fill out the provided form to request the document from Mentor Graphics.

For a list of known synthesis issues, see Xilinx Answer Record 21493, "*Where can I find a list of synthesis Known Issues for the DSP48/XtremeDSP Slice?*"

VHDL Code Examples

Following are examples for inferring the DSP48 slice in VHDL:

- "VHDL Code Example 1: 16x16 Multiplier Input and Output Registers"
- "VHDL Code Example 2: 18x18 Multiplier Fully Pipelined"
- "VHDL Code Example 3: Multiply Add"
- "VHDL Code Example 4: 16 Bit Adder"
- "VHDL Code Example 5: 16 Bit Adder, One Input Added Twice"
- "VHDL Code Example 6: Loadable Multiply Accumulate"
- "VHDL Code Example 7: MACC FIR Inferred"

VHDL Code Example 1: 16x16 Multiplier Input and Output Registers

Precision Synthesis, Synplify, and XST infer the DSP48 slice.

```
--------------------------------------------------------------------
-- Example 1: 16x16 Multiplier, inputs and outputs registered once
--         Matches 1 DSP48 slice
--         OpMode(Z,Y,X):Subtract
--           (000,01,01):0'
-- Expected register mapping:
--         AREG: yes
--         BREG: yes
--         CREG: no
--         MREG: yes
--         PREG: no
--------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity mult16_2reg is
 port (
      a   : in std_logic_vector (15 downto 0);
      b   : in std_logic_vector (15 downto 0);
      clk : in std_logic;
      rst : in std_logic;
      ce  : in std_logic;
      p   : out std_logic_vector (31 downto 0)
      );
end entity;
```

```
architecture mult16_2reg_arch of mult16_2reg is
 signal a1 : std_logic_vector (15 downto 0);
 signal b1 : std_logic_vector (15 downto 0);
 signal p1 : std_logic_vector (31 downto 0);

begin

 p1 <= a1*b1;

 process (clk) is
 begin
   if clk'event and clk = '1' then
     if rst = '1' then
       a1 <= (others => '0');
       b1 <= (others => '0');
       p  <= (others => '0');
     elsif ce = '1' then
       a1 <= a;
       b1 <= b;
       p  <= p1;
     end if;
   end if;
 end process;
end architecture;
```

VHDL Code Example 2: 18x18 Multiplier Fully Pipelined

XST infers the DSP48 slice.

```
------------------------------------------------------------------
-- Example 2: 18x18 Multiplier, fully piplelined
--           Matches 1 DSP48 slice
--           OpMode(Z,Y,X):Subtract
--             (000,01,01):0'
-- Expected register mapping:
--            AREG: yes
--            BREG: yes
--            CREG: no
--            MREG: yes
--            PREG: yes
------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity pipelined_mult is
  generic (
    data_width : integer := 18);
 port (
      a   : in std_logic_vector (data_width-1 downto 0);
      b   : in std_logic_vector (data_width-1 downto 0);
      clk : in std_logic;
      rst : in std_logic;
      ce  : in std_logic;
      p   : out std_logic_vector (2*data_width-1 downto 0)
      );
end entity;

architecture pipelined_mult_arch of pipelined_mult is
```

```
 signal a_reg : std_logic_vector (data_width-1 downto 0);
 signal b_reg : std_logic_vector (data_width-1 downto 0);
 signal m_reg : std_logic_vector (2*data_width-1 downto 0);
 signal p_reg : std_logic_vector (2*data_width-1 downto 0);

begin
  p <= p_reg;

 process (clk) is
 begin
   if clk'event and clk = '1' then
     if rst = '1' then
       a_reg <= (others => '0');
       b_reg <= (others => '0');
    m_reg <= (others => '0');
       p_reg <= (others => '0');
     elsif ce = '1' then
       a_reg <= a;
       b_reg <= b;
    m_reg <= a_reg*b_reg;
       p_reg <= m_reg;
     end if;
   end if;
 end process;
end architecture;
```

VHDL Code Example 3: Multiply Add

Precision Synthesis, Synplify, and XST infer the DSP48 slice.

```
------------------------------------------------------------
-- Example 3: Multiply add function, single level of register
--          Matches 1 DSP48 slice
--          OpMode(Z,Y,X):Subtract
--            (011,01,01):0
-- Expected register mapping:
--          AREG: no
--          BREG: no
--          CREG: no
--          MREG: no
--          PREG: yes
------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity mult_add_1reg is
 port (
      a   : in std_logic_vector (15 downto 0);
      b   : in std_logic_vector (15 downto 0);
      c   : in std_logic_vector (31 downto 0);
      clk : in std_logic;
      rst : in std_logic;
      ce  : in std_logic;
      p   : out std_logic_vector (31 downto 0)
      );
end entity;
```

```
architecture mult_add_1reg_arch of mult_add_1reg is
 signal p1 : std_logic_vector (31 downto 0);

begin

 p1 <= a*b + c;

 process (clk) is
 begin
   if clk'event and clk = '1' then
     if rst = '1' then
       p <= (others => '0');
     elsif ce = '1' then
       p <= p1;
     end if;
   end if;
 end process;

end architecture;
```

VHDL Code Example 4: 16 Bit Adder

XST infers the DSP48 slice if the `--use_dsp48` switch is used

```
----------------------------------------------------------------------
-- Example 4: 16 bit adder 2 inputs, input and output registered once
--   Mapping to DSP48 should be driven by timing as DSP48 are limited
--   resources.  The -use_dsp48 XST switch must be set to YES
--   Matches 1 DSP48 slice
--          OpMode(Z,Y,X):Subtract
--            (000,11,11):0 or
--            (011,00,11):0
-- Expected register mapping:
--          AREG: yes
--          BREG: yes
--          CREG: no
--          MREG: no
--          PREG: yes
----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity add16_2reg is
 port (
      a   : in std_logic_vector (15 downto 0);
      b   : in std_logic_vector (15 downto 0);
      clk : in std_logic;
      rst : in std_logic;
      ce  : in std_logic;
      p   : out std_logic_vector (15 downto 0)
      );
end entity;

architecture add16_2reg_arch of add16_2reg is
 signal a1 : std_logic_vector (15 downto 0);
 signal b1 : std_logic_vector (15 downto 0);
 signal p1 : std_logic_vector (15 downto 0);
```

```
begin

 p1 <= a1 + b1;

 process (clk) is
 begin
   if clk'event and clk = '1' then
     if rst = '1' then
       p  <= (others => '0');
       a1 <= (others => '0');
       b1 <= (others => '0');
     elsif ce = '1' then
       a1 <= a;
       b1 <= b;
       p <= p1;
     end if;
   end if;
 end process;

end architecture;
```

VHDL Code Example 5: 16 Bit Adder, One Input Added Twice

XST infers the DSP48 slice if the `--use_dsp48` switch is used.

```
----------------------------------------------------------------------
-- Example 5: 16 bit adder 2 inputs, one input added twice
--   input and output registered once
--   Mapping to DSP48 should be driven by timing as DSP48 are limited
--   resources.  The -use_dsp48 XST switch must be set to YES
--   Matches 1 DSP48 slice
--          OpMode(Z,Y,X):Subtract
--            (000,11,11):0 or
--            (011,00,11):0
-- Expected register mapping:
--          AREG: yes
--          BREG: yes
--          CREG: no
--          MREG: no
--          PREG: yes
----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity add16_multx2_2reg is
 port (
      a   : in std_logic_vector (15 downto 0);
      b   : in std_logic_vector (15 downto 0);
      clk : in std_logic;
      rst : in std_logic;
      ce  : in std_logic;
      p   : out std_logic_vector (15 downto 0)
      );
end entity;

architecture add16_multx2_2reg_arch of add16_multx2_2reg is
 signal a1 : std_logic_vector (15 downto 0);
 signal b1 : std_logic_vector (15 downto 0);
 signal p1 : std_logic_vector (15 downto 0);

begin

 p1 <= a1 + a1 + b1;

 process (clk) is
 begin
   if clk'event and clk = '1' then
     if rst = '1' then
       p  <= (others => '0');
       a1 <= (others => '0');
       b1 <= (others => '0');
     elsif ce = '1' then
       a1 <= a;
       b1 <= b;
       p <= p1;
     end if;
   end if;
 end process;

end architecture;
```

VHDL Code Example 6: Loadable Multiply Accumulate

Precision Synthesis, Synplify, and XST infer the DSP48 resources.

```
----------------------------------------------------------------------
-- Example 6: Loadable Multiply Accumulate with one level of registers
--         Map into 1 DSP48 slice
--         Function: OpMode(Z,Y,X):Subtract
--         - load     (011,00,00):0
--         - mult_acc (010,01,01):0
-- Restriction: Since C input of DSP48 slice is used, then adjacent
--   DSP cannot use a different c input (c input are shared between 2
--   adjacent DSP48 slices)
--
-- Expected mapping:
--         AREG: no
--         BREG: no
--         CREG: no
--         MREG: no
--         PREG: yes
----------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity load_mult_accum_1reg is
port (
     a     : in std_logic_vector (15 downto 0);
     b     : in std_logic_vector (15 downto 0);
     c     : in std_logic_vector (31 downto 0);
     p_rst : in std_logic;
     p_ce  : in std_logic;
     clk   : in std_logic;
     load  : in std_logic;
     p     : out std_logic_vector (31 downto 0)
    );
```

```
end entity;

architecture load_mult_accum_1reg_arch of load_mult_accum_1reg is

 signal a1    : signed (15 downto 0);
 signal b1    : signed (15 downto 0);
 signal p_tmp : signed (31 downto 0);
 signal p_reg : signed (31 downto 0);

begin

 with load select p_tmp <= signed(c) when '1' ,
                           p_reg + a1*b1 when others;


 process(clk)
 begin
   if clk'event and clk = '1' then
     if p_rst = '1' then
       p_reg <= (others => '0');
       a1 <= (others => '0');
       b1 <= (others => '0');
     elsif p_ce = '1' then
       p_reg <= p_tmp;
       a1 <= signed(a);
       b1 <= signed(b);
     end if;
   end if;
 end process;

 p <= std_logic_vector(p_reg);

end architecture;
```

VHDL Code Example 7: MACC FIR Inferred

Precision Synthesis, Synplify, and XST infer the DSP48 resources.

```
-----------------------------------------------------------------------
-- Example 7: Fully pipelined resetable Multiply Accumulate FIR Filter
--          modeled after Figure 3-1 in the XtremeDSP Design
--          Considerations User Guide.  This example does not contain
--          the control block listed in the figure.
--          Maps into 1 DSP48 slice
--          Function: OpMode(Z,Y,X):Subtract
--          - mult_acc (000,01,01):0
--          Pipelined dual port write first on 'a' port no change on
--          'b' port block RAM inferred.
--
-- Expected register mapping:
--          AREG: yes
--          BREG: yes
--          CREG: no
--          MREG: yes
--          PREG: yes
-----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity macc_fir is
generic (
  data_width    : integer := 18;
  address_width : integer := 8;
  mem_depth     : integer := 256);
port (
  clka      : in std_logic;
  clkb      : in std_logic;
  we        : in std_logic;
  en        : in std_logic;
  out_en    : in std_logic;
  macc_rst  : in std_logic;
  address_a : in std_logic_vector (address_width - 1 downto 0);
  address_b : in std_logic_vector (address_width - 1 downto 0);
  di        : in std_logic_vector (data_width - 1 downto 0);
  p_out     : out std_logic_vector (2*data_width-1 downto 0));
end entity;

architecture macc_fir_arch of macc_fir is

  type ram_arry is array (mem_depth-1 downto 0) of std_logic_vector (data_width-1 downto 0);

  signal ram          : ram_arry;
  signal doa_aux      : std_logic_vector (data_width-1 downto 0);
  signal dob_aux      : std_logic_vector (data_width-1 downto 0);
  signal m_reg, p_reg : signed (2*data_width-1 downto 0);
  signal a_in, a_reg  : signed (data_width-1 downto 0);
  signal b_in, b_reg  : signed (data_width-1 downto 0);
  signal we_del       : std_logic_vector (3 downto 0);
  signal macc_load    : std_logic;

begin

  process (clka) is
  begin
    if clka'event and clka = '1' then
      if en = '1' then
        if we = '1' then
          ram(to_integer(unsigned(address_a))) <= di;
        else
          doa_aux <= ram(to_integer(unsigned(address_a)));
        end if;
      end if;
    end if;
  end process;

  process (clkb) is
  begin
    if clkb'event and clkb = '1' then
      if en = '1' then
        dob_aux <= ram(to_integer(unsigned(address_b)));
      end if;
    end if;
  end process;

-- The following process blocks will infer the
-- optional output register that exists in
-- the Virtex-4 block RAM

  process (clka) is
  begin
  -- the output clock is the same as the input clock
  -- the output clock may also be inverted with respect
  -- to the input clock
  -- if clk'event and clk = '0'
    if clka'event and clka = '1' then
      if out_en = '1' then
      -- independent output register clock enable

        a_in <= signed(doa_aux);
      end if;
    end if;
  end process;

  process (clkb) is
  begin
  -- the output clock is the same as the input clock
  -- the output clock may also be inverted with respect
  -- to the input clock
  -- if clk'event and clk = '0'
    if clkb'event and clkb = '1' then
      if out_en = '1' then
      -- independent output register clock enable

        b_in <= signed(dob_aux);
      end if;
    end if;
  end process;

  -- infer the 4 delay SRL
  process(clka) is
  begin
    if clka'event and clka = '1' then
      we_del <= we_del(2 downto 0) & we;
      macc_load <= we_del(3);
    end if;
  end process;

  process(clka)
  begin
    if clka'event and clka = '1' then
      if macc_rst = '1' then
        a_reg <= (others => '0');
        b_reg <= (others => '0');
        m_reg <= (others => '0');
        p_reg <= (others => '0');
        p_out <= (others => '0');
      else
        a_reg <= a_in;
        b_reg <= b_in;
        m_reg <= a_reg * b_reg;
        if macc_load = '1' then
          p_reg <= p_reg + m_reg;
        else
          p_reg <= m_reg;
        end if;
```

```
          p_out <= std_logic_vector(p_reg);
        end if;
      end if;
    end process;
  end architecture;
```

Verilog Code Examples

Following are examples for inferring the DSP48 slice in Verilog.

- "Verilog Code Example 1: 16x16 Multiplier Input and Output Registers"
- "Verilog Code Example 2: 18x18 Multiplier Fully Pipelined"
- "Verilog Code Example 3: Multiply Add"
- "Verilog Code Example 4: 16 Bit Adder"
- "Verilog Code Example 5: 16 Bit Adder, One Input Added Twice"
- "Verilog Code Example 6: Loadable Multiply Accumulate"
- "Verilog Code Example 7: MACC FIR Inferred"

Verilog Code Example 1: 16x16 Multiplier Input and Output Registers

```
//////////////////////////////////////////////////////////////////////
// Example 1: 16x16 Multiplier, inputs and outputs registered once
//          Matches 1 DSP48 slice
//          OpMode(Z,Y,X):Subtract
//            (000,01,01):0
// Expected register mapping:
//          AREG: yes
//          BREG: yes
//          CREG: no
//          MREG: yes
//          PREG: no
//////////////////////////////////////////////////////////////////////

module mult16_2reg (a, b, p, rst, ce, clk);
 input signed [15:0] a;
 input signed [15:0] b;
 input  clk;
 input  rst;
 input  ce;
 output [31:0] p;

 reg  [31:0] p;
 reg  [15:0] a1;
 reg  [15:0] b1;
 wire [31:0] p1;

 assign p1 = a1*b1;

 always @(posedge clk)
   if (rst == 1'b1)
    begin
      a1 <= 0;
      b1 <= 0;
      p <= 0;
    end
   else if (ce == 1'b1)
```

```
    begin
      a1 <= a;
      b1 <= b;
      p  <= p1;
    end
endmodule
```

Verilog Code Example 2: 18x18 Multiplier Fully Pipelined

```
//////////////////////////////////////////////////////////////////////
// Example 2: 18x18 Multiplier, fully pipelined
//          Matches 1 DSP48 slice
//          OpMode(Z,Y,X):Subtract
//             (000,01,01):0
// Expected register mapping:
//            AREG: yes
//            BREG: yes
//            CREG: no
//            MREG: yes
//            PREG: yes
//////////////////////////////////////////////////////////////////////

module pipelined_mult (a, b, p, rst, ce, clk);
 parameter data_width = 18;

 input signed [data_width-1:0] a;
 input signed [data_width-1:0] b;
 input  clk;
 input  rst;
 input  ce;
 output signed [2*data_width-1:0] p;

 //reg [2*data_width-1:0] p;
 reg signed [data_width-1:0] a_reg;
 reg signed [data_width-1:0] b_reg;
 reg signed [2*data_width-1:0] m_reg;
 reg signed [2*data_width-1:0] p_reg;

 assign p = p_reg;

 always @(posedge clk)
   if (rst == 1'b1)
    begin
      a_reg <= 0;
      b_reg <= 0;
      m_reg <= 0;
      p_reg <= 0;
    end
   else if (ce == 1'b1)
    begin
      a_reg <= a;
      b_reg <= b;
      m_reg <= a_reg*b_reg;
      p_reg <= m_reg;
    end
endmodule
```

Verilog Code Example 3: Multiply Add

```
//////////////////////////////////////////////////////////////////
// Example 3: Multiply add function, single level of register
//         Matches 1 DSP48 slice
//         OpMode(Z,Y,X):Subtract
//           (011,01,01):0
// Expected register mapping:
//         AREG: no
//         BREG: no
//         CREG: no
//         MREG: no
//         PREG: yes
/////////////////////////////////////////////////////////////////

module mult_add_1reg (a, b, c, p, rst, ce, clk);
 input signed [15:0] a;
 input signed [15:0] b;
 input signed [31:0] c;
 input  clk;
 input  rst;
 input  ce;
 output [31:0] p;
 reg  [31:0] p;
 wire [31:0] p1;

 assign p1 = a*b + c;

 always @(negedge clk)
   if (rst == 1'b1)
     p <= 0;
   else if (ce == 1'b1) begin
     p <= p1;
   end
endmodule
```

Verilog Code Example 4: 16 Bit Adder

```
////////////////////////////////////////////////////////////////////////
// Example 4: 16 bit adder 2 inputs, input and output registered once
//   Mapping to DSP48 should be driven by timing as DSP48 are limited
//   resources.  The -use_dsp48 XST switch must be set to YES
//   Matches 1 DSP48 slice
//         OpMode(Z,Y,X):Subtract
//           (000,11,11):0 or
//           (011,00,11):0
// Expected register mapping:
//         AREG: yes
//         BREG: yes
//         CREG: no
//         MREG: no
//         PREG: yes
////////////////////////////////////////////////////////////////////////

module add16_2reg (a, b, p, rst, ce, clk);
 input signed [15:0] a;
 input signed [15:0] b;
 input  clk;
 input  rst;
 input  ce;
 output [15:0] p;
 reg    [15:0] a1;
 reg    [15:0] b1;
 reg    [15:0] p;
 wire   [15:0] p1;

 assign p1 = a1 + b1;

 always @(posedge clk)
    if (rst == 1'b1)
    begin
      p  <= 0;
      a1 <= 0;
      b1 <= 0;
    end
   else if (ce == 1'b1)
    begin
      a1 <= a;
      b1 <= b;
      p <= p1;
   end
endmodule
```

Verilog Code Example 5: 16 Bit Adder, One Input Added Twice

```
//////////////////////////////////////////////////////////////////////
// Example 5: 16 bit adder 2 inputs, one input added twice
//   input and output registered once
//   Mapping to DSP48 should be driven by timing as DSP48 are limited
//   resources.  The -use_dsp48 XST switch must be set to YES
//   Matches 1 DSP48 slice
//          OpMode(Z,Y,X):Subtract
//            (000,11,11):0 or
//            (011,00,11):0
// Expected register mapping:
//          AREG: yes
//          BREG: yes
//          CREG: no
//          MREG: no
//          PREG: yes
//////////////////////////////////////////////////////////////////////

module add16_multx2_2reg (a, b, p, rst, ce, clk);
 input signed [15:0] a;
 input signed [15:0] b;
 input  clk;
 input  rst;
 input  ce;
 output [15:0] p;
 reg    [15:0] a1;
 reg    [15:0] b1;
 reg    [15:0] p;
 wire   [15:0] p1;

 assign p1 = a1 + a1 + b1;
 always @(posedge clk)
    if (rst == 1'b1)
    begin
      p  <= 0;
      a1 <= 0;
      b1 <= 0;
    end
   else if (ce == 1'b1)
    begin
      a1 <= a;
      b1 <= b;
      p <= p1;
   end
endmodule
```

Verilog Code Example 6: Loadable Multiply Accumulate

```
//////////////////////////////////////////////////////////////////////
// Example 6: Loadable Multiply Accumulate with one level of registers
//          Map into 1 DSP48 slice
//          Function: OpMode(Z,Y,X):Subtract
//          - load     (011,00,00):0
//          - mult_acc (010,01,01):0
// Restriction: Since C input of DSP48 slice is used, then adjacent
//   DSP cannot use a different c input (c input are shared between 2
//   adjacent DSP48 slices)
//
// Expected mapping:
//          AREG: no
//          BREG: no
//          CREG: no
//          MREG: no
//          PREG: yes
//////////////////////////////////////////////////////////////////////

module load_mult_accum_1reg (a, b, c, p, p_rst, p_ce, clk, load);
 input  signed [15:0] a;
 input  signed [15:0] b;
 input  signed [31:0] c;
 input  p_rst;
 input  p_ce;
 input  clk;
 input  load;
 output [31:0] p;
 reg    [31:0] p;
 reg    [15:0] a1;
 reg    [15:0] b1;
 wire   [31:0] p_tmp;

 assign p_tmp = load ? c:p + a1*b1;
 always @(posedge clk)
    if (p_rst == 1'b1) begin
     p <= 0;
     a1 <=0;
     b1 <=0;
   end
   else if (p_ce == 1'b1) begin
```

```
      p <= p_tmp;
      a1 <=a;
      b1 <= b;
    end
endmodule
```

Verilog Code Example 7: MACC FIR Inferred

```
//////////////////////////////////////////////////////////////////////
// Example 7: Fully pipelined resetable Multiply Accumulate FIR Filter
//          modeled after Figure 3-1 in the XtremeDSP Design
//          Considerations User Guide.  This example does not contain
//          the control block.
//          Maps into 1 DSP48 slice
//          Function: OpMode(Z,Y,X):Subtract
//          - mult_acc (000,01,01):0
//          Pipelined dual port write first on 'a' port no change on
//          'b' port block RAM inferred.
//
// Expected register mapping:
//          AREG: yes
//          BREG: yes
//          CREG: no
//          MREG: yes
//          PREG: yes
//////////////////////////////////////////////////////////////////////

module macc_fir (clka, clkb, we, en, out_en, macc_rst, di, address_a, address_b, p_out);

  parameter data_width = 18;
  parameter address_width = 8;
  parameter mem_depth = 256;  // 2**address_width

  input  clka, clkb, we, en, out_en, macc_rst;
  input [data_width - 1:0] di;
  input [address_width-1 : 0] address_a;
  input [address_width-1 : 0] address_b;
  output reg signed [2*data_width-1:0] p_out;

  reg [data_width-1:0] ram [mem_depth-1:0];
  reg [data_width-1 : 0] doa_aux;
  reg [data_width-1 : 0] dob_aux;
  reg signed [2*data_width-1:0] m_reg, p_reg;
  reg signed [17:0] a_in, a_reg, b_in, b_reg;
  reg [3:0] we_del;
  reg macc_load;

  always @(posedge clka)
  begin
    if (en) begin
      if (we) ram[address_a] <= di;
      else doa_aux <= ram[address_a];
    end //if (en)
  end //always

  always @(posedge clkb)
  begin
    if (en) dob_aux <= ram[address_b];
  end //always
```

```
// The following always blocks will infer the
// optional output register that exists in
// the Virtex-4 block RAM

  always @(posedge clka)
  // the output clock is the same as the input clock
  // the output clock may also be inverted with respect
  // to the input clock
  // always @(negedge clk)

  begin
    if (out_en) begin
    // independent output register clock enable

      a_in <= doa_aux;
    end //if out_en
  end //always

  always @(posedge clkb)
  // the output clock is the same as the input clock
  // the output clock may also be inverted with respect
  // to the input clock
  // always @(negedge clk)

  begin
    if (out_en) begin
    // independent output register clock enable

      b_in <= dob_aux;
    end //if out_en
  end //always

  // infer the 4 delay SRL
  always @(posedge clka)
  begin
    we_del <= {we_del[2:0],we};
    macc_load <= we_del[3];
  end //always

  always @(posedge clka)
  begin
    if (macc_rst == 1'b1) begin
      a_reg <= 0;
      b_reg <= 0;
      m_reg <= 0;
      p_reg <= 0;
      p_out <= 0;
    end // if macc_rst == 1
    else begin
      a_reg <= a_in;
      b_reg <= b_in;
      m_reg <= a_reg * b_reg;
      p_reg <= macc_load ? (p_reg + m_reg) : m_reg;
      p_out <= p_reg;
    end // else
  end // always

endmodule
```

Adder and Subtractor

Synthesis tools infer carry logic in Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Spartan-II, and Spartan-3 devices when an adder (+ operator) or subtractor (- operator) is described.

Multiplier

Synthesis tools utilize carry logic by inferring XORCY, MUXCY, and MULT_AND for Virtex, Virtex-E, and Spartan-II when a multiplier is described.

When a Virtex-II or Virtex-II Pro part is targeted, an embedded 18x18 two's complement multiplier primitive called a MULT18X18 is inferred by the synthesis tools. For synchronous multiplications, LeonardoSpectrum, Synplify and XST infer a MULT18X18S primitive.

LeonardoSpectrum also features a pipeline multiplier that involves putting levels of registers in the logic to introduce parallelism and, as a result, improve speed. A certain construct in the input RTL source code description is required to allow the pipelined multiplier feature to take effect.

This construct infers XORCY, MUXCY, and MULT_AND primitives for Virtex, Virtex-E, Spartan-II, Spartan-3,Virtex-II, Virtex-II Pro and Virtex-II Pro X. The following example shows this construct.

VHDL Example One: Pipelined Multiplier

Following is a VHDL example of LeonardoSpectrum and Precision Synthesis Pipelined Multiplier.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiply is
generic (size : integer := 16; level : integer := 4);
  port (
      clk : in std_logic;
      Ain : in std_logic_vector (size-1 downto 0);
      Bin : in std_logic_vector (size-1 downto 0);
      Qout : out std_logic_vector (2*size-1 downto 0)
      );
end multiply;
architecture RTL of multiply is
  type levels_of_registers is array (level-1 downto 0)
    of unsigned (2*size-1 downto 0);
  signal reg_bank :levels_of_registers;
  signal a, b : unsigned (size-1 downto 0);
  begin
    Qout <= std_logic_vector (reg_bank (level-1));
    process
    begin
      wait until clk'event and clk = '1';
        a <= unsigned(Ain);
        b <= unsigned(Bin);
        reg_bank (0) <= a * b;
        for i in 1 to level-1 loop
            reg_bank (i) <= reg_bank (i-1);
      end loop;
   end process;
end architecture RTL;
```

VHDL Example Two: Synchronous Multiplier

Following is a synchronous multiplier VHDL example coded for LeonardoSpectrum, Precision Synthesis, Synplify, and XST

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

entity xcv2_mult18x18s is
  Port (
       a : in std_logic_vector(7 downto 0);
       b : in std_logic_vector(7 downto 0);
       clk : in std_logic;
       prod : out std_logic_vector(15 downto 0)
       );
end xcv2_mult18x18s;

architecture arch_xcv2_mult18x18s of xcv2_mult18x18s is

  begin
  process(clk) is begin
    if clk'event and clk = '1' then
      prod <= a*b;
    end if;
  end process;
  end arch_xcv2_mult18x18s;
```

Verilog Example One: Pipelined Multiplier

Following is a pipelined multiplier Verilog example coded for LeonardoSpectrum, Precision Synthesis, Synplify, and XST

```
module multiply (clk, ain, bin, q);
  parameter size = 16;
  parameter level = 4;
  input clk;
  input [size-1:0] ain, bin;
  output [2*size-1:0] q;
  reg [size-1:0] a, b;
  reg [2*size-1:0] reg_bank [level-1:0];
  integer i;
  always @(posedge clk)
    begin
      a <= ain;
      b <= bin;
    end
  always @(posedge clk)
    reg_bank[0] <= a * b;
  always @(posedge clk)
    for (i = 1;i < level; i=i+1)
      reg_bank[i] <= reg_bank[i-1];
    assign q = reg_bank[level-1];
endmodule // multiply
```

Verilog Example Two: Synchronous Multiplier

Following is a synchronous multiplier Verilog example coded for LeonardoSpectrum, Precision Synthesis, Synplify, and XST.

```
module mult_sync(
  input [7:0] a, b,
  input clk,
  output reg [15:0] prod);

  always @(posedge clk)
    prod <= a*b;
endmodule
```

Counters

When describing a counter in HDL, the arithmetic operator '+' infers the carry chain. The synthesis tools then infers the dedicated carry components for the counter.

```
count <= count + 1; -- This infers MUXCY
```

This implementation provides a very effective solution, especially for all purpose counters.

VHDL Example: Loadable Binary Counter

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

entity counter is
  port (
        D : in std_logic_vector (7 downto 0);
        LD, CE, CLK, RST : in std_logic;
        Q : out std_logic_vector (7 downto 0)
        );
end counter;

architecture behave of counter is

   signal count : std_logic_vector (7 downto 0);

begin
   process (CLK) begin
      if rising_edge(CLK) then
         if RST = '1' then
            count <= (others => '0');
             elsif CE = '1' then
                if LD = '1' then
               count <= D;
             else
               count <= count + '1';
            end if;
         end if;
      end if;
   end process;

   Q <= count;

end behave;
```

Verilog Example: Loadable Binary Counter

Following is a Verilog example of a loadable binary counter.

```
module counter(
     input [7:0] dD,
     input ldLD, ceCE, clkCLK, rstRST,
     output reg [7:0] qQ
);

  reg [7:0] count;
     always @(posedge clkCLK, posedge rst)
    begin
              if (rstRST)
                   count Q <= 08'h00;
              else if (ldCE)
          if (LD)
                      count Q <= dD;
                 else if (ce)
                      count Q <= count Q + 1;
    end
  assign q = count;

endmodule
```

For applications that require faster counters, LFSR can implement high performance and area efficient counters. LFSR requires very minimal logic (only an XOR or XNOR feedback). For more information, see "Implementing LFSR" in this chapter.

For smaller counters it is also effective to use the Johnson encoded counters. This type of counter does not use the carry chain but provides a fast performance.

The following is an example of a sequence for a 3 bit Johnson counter.

000

001

011

111

110

100

Comparator

Magnitude comparators and equality comparators can infer LUTs for smaller comparators, or use carry chain logic for larger bit-width comparators. This results in fast and efficient implementations in Xilinx devices. For each architecture, synthesis tools choose the best underlying resource for a given comparator description in RTL.

Table 4-1: **Comparator Symbols**

Magnitude Comparators	Equality Comparators
>	==
<	

Table 4-1: **Comparator Symbols**

Magnitude Comparators	Equality Comparators
>=	
<=	

VHDL Example: Unsigned 16-Bit Greater or Equal Comparator

```
-- Unsigned 16-bit greater or equal comparator.
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity compar is
   port(A, B : in std_logic_vector(7 downto 0);
         CMP : out std_logic);
end compar;

architecture archi of compar is
begin
   CMP <= '1' when (A >= B) else '0';
end archi
```

Verilog Example: Unsigned 8-Bit Greater Or Equal Comparator

```
// Unsigned 8-bit greater or equal comparator.
module compar(
   input [7:0] A, B,
   output CMP);

   assign CMP = (A >= B);

endmodule
```

Encoder and Decoders

Synthesis tools might infer the MUXF5 and MUXF6 resources for encoder and decoder in Xilinx FPGA devices. Virtex-II, Virtex-II Pro, Virtex-II Pro X, and Spartan-3 devices feature additional components, MUXF7 and MUXF8, to use with the encoder and decoder.

LeonardoSpectrum infers MUXCY when an if-then-else priority encoder is described in the code. This results in a faster encoder.

Following are VHDL and Verilog examples of LeonardoSpectrum Priority Encoding.

VHDL Example: LeonardoSpectrum Priority Encoding

```
library IEEE;
use IEEE.std_logic_1164.all;

entity prior is
   generic (size: integer := 8);
   port(
      CLK: in std_logic;
      COND1 : in std_logic_vector(size-1 downto 0);
      COND2 : in std_logic_vector(size-1 downto 0);
      DATA  : in std_logic_vector(size-1 downto 0);
```

```
       Q      : out std_logic);
end prior;

architecture RTL of prior is

   signal   data_ff, cond1_ff, cond2_ff: std_logic_vector(size-1 downto 0);

begin
   process(CLK)
   begin
      if CLK'event and CLK = '1' then
         data_ff <= DATA;
         cond1_ff <= COND1;
         cond2_ff <= COND2;
      end if;
   end process;

   process(CLK)
   begin
      if (CLK'event and CLK = '1') then
         if (cond1_ff(1) = '1' and cond2_ff(1) = '1') then
            Q <= data_ff(1);
         elsif (cond1_ff(2) = '1' and cond2_ff(2) = '1') then
            Q <= data_ff(2);
         elsif (cond1_ff(3) = '1' and cond2_ff(3) = '1') then
            Q <= data_ff(3);
         elsif (cond1_ff(4) = '1' and cond2_ff(4) = '1') then
            Q <= data_ff(4);
         elsif (cond1_ff(5)= '1' and cond2_ff(5) = '1' ) then
            Q <= data_ff(5);
         elsif (cond1_ff(6) = '1' and cond2_ff(6) = '1') then
            Q <= data_ff(6);
         elsif (cond1_ff(7) = '1' and cond2_ff(7) = '1') then
            Q <= data_ff(7);
         elsif (cond1_ff(8) = '1' and cond2_ff(8) = '1') then
            Q <= data_ff(8);
         else
            Q <= '0';
         end if;
      end if;
   end process;

end RTL;
```

Verilog Example: LeonardoSpectrum Priority Encoding

```
module prior(CLK, COND1, COND2, DATA, Q);

   parameter size = 8;
   input CLK;
   input [size-1:0] DATA, COND1, COND2;
   output reg Q;

   reg [size-1:0] data_ff, cond1_ff, cond2_ff;

   always @(posedge CLK)
   begin
      data_ff <= DATA;
      cond1_ff <= COND1;
      cond2_ff <= COND2;
   end

   always @(posedge CLK)
      if (cond1_ff[1] && cond2_ff[1])
         Q <= data_ff[1];
      else if (cond1_ff[2] && cond2_ff[2])
         Q <= data_ff[2];
      else if (cond1_ff[3] && cond2_ff[3])
         Q <= data_ff[3];
      else if (cond1_ff[4] && cond2_ff[4])
         Q <= data_ff[4];
      else if (cond1_ff[5] && cond2_ff[5])
         Q <= data_ff[5];
      else if (cond1_ff[6] && cond2_ff[6])
         Q <= data_ff[6];
      else if (cond1_ff[7] && cond2_ff[7])
         Q <= data_ff[7];
      else if (cond1_ff[8] && cond2_ff[8])
         Q <= data_ff[8];
      else
         Q <= 1'b0;

endmodule
```

Implementing Memory

Xilinx FPGA devices provide:

- distributed on-chip RAM (SelectRAM)
- dedicated block memory (Block SelectRAM)

Both memories offer synchronous write capabilities. However, the distributed RAM can be configured for either asynchronous or synchronous reads.

The edge-triggered write capability simplifies system timing, and provides better performance for RAM-based designs. In general, synchronous read capability is also desired. However, distributed RAM offers asynchronous write capability. This can provide more flexibility when latency is not tolerable, or if you are using the RAM as a look-up table or other logical type of operation.

In general, the selection of distributed RAM versus block RAM depends on the size of the RAM. If the RAM is not very deep (16 to 32 bits deep), it is generally advantageous to use the distributed RAM. If you requre a deeper RAM, it is generally more advantageous to use the block memory.

Since all Xilinx RAMs have the ability to be initialized, the RAMs may also be configured either as a ROM (Read Only Memory), or as a RAM with pre-defined contents. The Virtex-4 device adds even more capabilities to the block memory, including:

- asynchronous FIFO logic
- error correction circuitry
- more flexible configurations

Implementing Block RAM

FPGA devices incorporate several large Block SelectRAM memories. These complement the distributed SelectRAM that provide shallow RAM structures implemented in CLBs. The Block SelectRAM is a True Dual-Port RAM which allows for large, discrete blocks of memory.

RAMs may be incorporated into the design in three primary ways:

- inference
- CORE Generator creation
- direct instantiation

Each of these methods has its advantages and disadvantages. Which method is best for you depends on your own goals and objectives. For a side-by-side comparison of the advantages and disadvatages of the three methods of incorporating RAMs into the design. see the following table.

Table 4-2: **Advantages and Disadvantages of the Three Methods of Incorporating RAMs into the Design**

	Inference	**Coregen**	**Instantiation**
Advantages	Most generic means of incorporating RAMs	Allows good control over the creation process of the RAM	Offers the most control
	Simulates the fastest		
Disadvantages	Leaves the user with the least amount of control over the underlying RAM created by the tools	Can complicate portability to different device architectures	Can limit the portability of the design
	Requires specific coding styles to ensure proper inferrence	Can require running a separate tool to generate and regenerate the RAM	Can require multiple instantiations to properly create the right size RAM[a]

a. for data paths that require more than one RAM

Instantiating Block SelectRAM

Below are VHDL and Verilog coding examples for instantiating a BlockRAM for the following architectures:

- Virtex-II
- Virtex-II Pro
- Spartan-3
- Spartan-3E

You can modify the generic maps or inline parameters to change the initialization or other characteristics of the RAM. For more information on instantiating and using this component or other BlockRAM components, see the Xilinx *Libraries Guides*.

The following coding examples provide VHDL and Verilog coding styles for LeonardoSpectrum, Synplify, and XST.

Instantiating Block SelectRAM VHDL Example

LeonardoSpectrum, and XST

With LeonardoSpectrum and XST you can instantiate a RAMB* cell as a blackbox. The INIT_** attribute can be passed as a string in the HDL file as well as the script file. The following VHDL code example shows how to pass INIT in the VHDL file.

LeonardoSpectrum and Precision Synthesis

With LeonardoSpectrum and Precision Synthesis, in addition to passing the INIT string in HDL, you can pass an INIT string in a LeonardoSpectrum and Precision Synthesis command script. The following code sample illustrates this method.

```
set_attribute -instance "inst_ramb4_s4" -name
INIT_00 -type string -value
"1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09
80706050403020100"

library IEEE;
use IEEE.std_logic_1164.all;

entity spblkrams is
  port(
       CLK  : in std_logic;
       EN   : in std_logic;
       RST  : in std_logic;
       WE   : in std_logic;
       ADDR : in std_logic_vector(11 downto 0);
       DI   : in std_logic_vector(15 downto 0);
       DORAMB4_S4 : out std_logic_vector(3 downto 0);
       DORAMB4_S8 : out std_logic_vector(7 downto 0)
       );
end;
architecture struct of spblkrams is
component RAMB4_S4
  port (
       DI : in STD_LOGIC_VECTOR (3 downto 0);
       EN : in STD_ULOGIC;
       WE : in STD_ULOGIC;
       RST : in STD_ULOGIC;
       CLK : in STD_ULOGIC;
       ADDR : in STD_LOGIC_VECTOR (9 downto 0);
       DO : out STD_LOGIC_VECTOR (3 downto 0)
       );
end component;
component RAMB4_S8
  port (
       DI : in STD_LOGIC_VECTOR (7 downto 0);
       EN : in STD_ULOGIC;
       WE : in STD_ULOGIC;
       RST : in STD_ULOGIC;
       CLK : in STD_ULOGIC;
       ADDR : in STD_LOGIC_VECTOR (8 downto 0);
       DO : out STD_LOGIC_VECTOR (7 downto 0)
       );
end component;
attribute INIT_00: string;
attribute INIT_00 of INST_RAMB4_S4: label is
```

```
  "1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100";
attribute INIT_00 of INST_RAMB4_S8: label is
  "1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100";
begin
    INST_RAMB4_S4 : RAMB4_S4 port map (
        DI => DI(3 downto 0),
        EN => EN,
        WE => WE,
        RST => RST,
        CLK => CLK,
        ADDR => ADDR(9 downto 0),
        DO => DORAMB4_S4
        );
    INST_RAMB4_S8 : RAMB4_S8 port map (
        DI => DI(7 downto 0),
        EN => EN,
        WE => WE,
        RST => RST,
        CLK => CLK,
        ADDR => ADDR(8 downto 0),
        DO => DORAMB4_S8
        );
end struct;
```

XST and Synplify

```
library IEEE;
use IEEE.std_logic_1164.all;
entity spblkrams is
  port(
        CLK : in std_logic;
        EN : in std_logic;
        RST : in std_logic;
        WE : in std_logic;
        ADDR : in std_logic_vector(11 downto 0);
        DI : in std_logic_vector(15 downto 0);
        DORAMB4_S4 : out std_logic_vector(3 downto 0);
        DORAMB4_S8 : out std_logic_vector(7 downto 0)
        );
end;

architecture struct of spblkrams is
  component RAMB4_S4
    generic( INIT_00 : bit_vector :=
x"0000000000000000000000000000000000000000000000000000000000000000");
    port (
        DI : in STD_LOGIC_VECTOR (3 downto 0);
        EN : in STD_ULOGIC;
        WE : in STD_ULOGIC;
        RST : in STD_ULOGIC;
        CLK : in STD_ULOGIC;
        ADDR : in STD_LOGIC_VECTOR (9 downto 0);
        DO : out STD_LOGIC_VECTOR (3 downto 0)
        );
  end component;

  component RAMB4_S8
    generic( INIT_00 : bit_vector :=
x"0000000000000000000000000000000000000000000000000000000000000000");
```

```
    port (
        DI : in STD_LOGIC_VECTOR (7 downto 0);
        EN : in STD_ULOGIC;
        WE : in STD_ULOGIC;
        RST : in STD_ULOGIC;
        CLK : in STD_ULOGIC;
        ADDR : in STD_LOGIC_VECTOR (8 downto 0);
        DO : out STD_LOGIC_VECTOR (7 downto 0)
        );
  end component;

begin
INST_RAMB4_S4 : RAMB4_S4
  generic map (INIT_00 =>
x"1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100")
  port map (
        DI => DI(3 downto 0),
        EN => EN,
        WE => WE,
        RST => RST,
        CLK => CLK,
        ADDR => ADDR(9 downto 0),
        DO => DORAMB4_S4
        );

INST_RAMB4_S8 : RAMB4_S8
  generic map (INIT_00 =>
x"1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100")
port map (
        DI => DI(7 downto 0),
        EN => EN,
        WE => WE,
        RST => RST,
        CLK => CLK,
        ADDR => ADDR(8 downto 0),
        DO => DORAMB4_S8
        );
end struct;
```

Instantiating Block SelectRAM Verilog Examples

The following Verilog examples show Block SelectRAM™ instantiation.

LeonardoSpectrum

With LeonardoSpectrum the INIT attribute can be set in the HDL code or in the command line. See the following example.

```
set_attribute -instance "inst_ramb4_s4" -name
INIT_00 -type string value
    "1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080060504030201000"
```

LeonardoSpectrum and Precision Synthesis block_ram_ex Verilog Example

```
module block_ram_ex (CLK, WE, ADDR, DIN, DOUT);
  input CLK, WE;
  input [8:0] ADDR;
  input [7:0] DIN;
  output [7:0] DOUT;
  RAMB4_S8 U0 (
        .WE(WE),
        .EN(1’b1),
        .RST(1’b0),
        .CLK(CLK),
        .ADDR(ADDR),
        .DI(DIN),
        .DO(DOUT));
//exemplar attribute U0 INIT_00
  1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100
//pragma attribute U0 INIT_00
  1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100
endmodule
```

Synplicity and XST block_ram_ex Verilog Example

```
//////////////////////////////////////////////////////
// BLOCK_RAM_EX.V Version 2.0
// This is an example of an instantiation of
// a Block RAM with an INIT value passed via
// a local parameter
//////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v”
//////////////////////////////////////////////////////

module spblkrams (CLK, WE, ADDR, DIN, DOUT);

input CLK, WE;
input [8:0] ADDR;
input [7:0] DIN;
output [7:0] DOUT;

RAMB4_S8
#(.INIT_00(256'h1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100))
U0 (.WE(WE), .EN(1'b1), .RST(1'b0), .CLK(CLK), .ADDR(ADDR), .DI(DIN), .DO(DOUT));

endmodule
```

Inferring Block SelectRAM VHDL Examples

The following coding examples provide VHDL coding styles for inferring BlockRAMs for most supported synthesis tools. Most also support the initialization of block RAM via signal initialization. This is supported in VHDL only.

The basic syntax for this feature is:

```
  type mem_array is array (255 downto 0) of
       std_logic_vector (7 downto 0);
  signal mem : mem_array :=
 (X"0A", X"00", X"01", X"00", X"01", X"3A",
X"00", X"08", X"02", X"02", X"00", X"02",
X"08", X"00", X"01", X"02", X"40", X"41",
:
:
```

For more RAM inference examples, see your synthesis tool documentation.

Block SelectRAM can be inferred by some synthesis tools. Inferred RAM must be initialized in the UCF file. Not all Block SelectRAM features can be inferred. Those features are pointed out in this section.

LeonardoSpectrum

LeonardoSpectrum can map your memory statements in Verilog or VHDL to the Block SelectRAM on all Virtex devices. The following is a list of the details for Block SelectRAM in LeonardoSpectrum.

- Virtex Block SelectRAM is completely synchronous — both read and write operations are synchronous.
- LeonardoSpectrum infers single port RAMs (RAMs with both read and write on the same address), and dual port RAMs (RAMs with separate read and write addresses).
- Virtex Block SelectRAM supports RST (reset) and ENA (enable) pins. Currently, LeonardoSpectrum does not infer RAMs which use the functionality of the RST and ENA pins.
- By default, RAMs are mapped to Block SelectRAM if possible. To disable mapping to Block SelectRAM, set the attribute BLOCK_RAM to false.

VHDL Example One

Following is a LeonardoSpectrum and Precision Synthesis VHDL example.

```
library ieee, exemplar;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ram_example1 is
  generic(data_width : integer := 8;
  address_width : integer := 8;
  mem_depth : integer : = 256);
  port (
        data : in std_logic_vector(data_width-1 downto 0);
        address : in unsigned(address_width-1 downto 0);
        we, clk : in std_logic;
        q : out std_logic_vector(data_width-1 downto 0)
        );
end ram_example1;

architecture ex1 of ram_example1 is

  type mem_type is array (mem_depth-1 downto 0)
      of std_logic_vector (data_width-1 downto 0);
  signal mem : mem_type;
  signal raddress : unsigned(address_width-1 downto 0);

  begin
  l0: process (clk, we, address)
  begin
    if (clk = '1' and clk'event) then
        raddress <= address;
        if (we = '1') then
            mem(to_integer(raddress)) <= data;
        end if;
    end if;
  end process;
  l1: process (clk, address)
  begin
    if (clk = '1' and clk'event) then
        q <= mem(to_integer(address));
    end if;
  end process;
end ex1;
```

VHDL ExampleTwo

Following is a LeonardoSpectrum and Precision Synthesis VHDL example (Dual Port Block SelectRAM).

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

entity dualport_ram is
  port (
        clka : in std_logic;
        clkb : in std_logic;
        wea : in std_logic;
        addra : in std_logic_vector(4 downto 0);
        addrb : in std_logic_vector(4 downto 0);
        dia : in std_logic_vector(3 downto 0);
        dob : out std_logic_vector(3 downto 0));
end dualport_ram;

architecture dualport_ram_arch of dualport_ram is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal ram : ram_type;

  attribute block_ram : boolean;
  attribute block_ram of RAM : signal is TRUE;

  begin
    write: process (clka)
    begin
      if (clka'event and clka = '1') then
          if (wea = '1') then
              ram(conv_integer(addra)) <= dia;
          end if;
      end if;
    end process write;

    read: process (clkb)
    begin
      if (clkb'event and clkb = '1') then
          dob <= ram(conv_integer(addrb));
      end if;
    end process read;

end dualport_ram_arch;
```

Synplify

To enable the usage of Block SelectRAM, set the attribute syn_ramstyle to **`block_ram`**. Place the attribute on the output signal driven by the inferred RAM. Remember to include the range of the output signal (bus) as part of the name.

For example,

```
define_attribute {a|dout[3:0]} syn_ramstyle "block_ram"
```

The following are limitations of inferring Block SelectRAM:

- ENA/ENB pins currently are inaccessible. They are always tied to “1.”
- RSTA/RSTB pins currently are inaccessible. They are always inactive.
- Automatic inference is not yet supported. The syn_ramstyle attribute is required for inferring Block SelectRAM.
- Initialization of RAMs is not supported.
- Dual port with Read-Write on a port is not supported.

VHDL Example

Following is a Synplify VHDL example.

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

entity ram_example1 is
  generic(
        data_width : integer := 8;
        address_width : integer := 8;
        mem_depth : integer:= 256
        );
  port (
        data : in std_logic_vector(data_width-1 downto 0);
        address : in std_logic_vector(address_width-1 downto 0);
        we, clk : in std_logic;
        q : out std_logic_vector(data_width-1 downto 0)
        );
end ram_example1;

architecture rtl of ram_example1 is type mem_array is array
    (mem_depth-1 downto 0) of std_logic_vector (data_width-1 downto 0);
  signal mem : mem_array;
  attribute syn_ramstyle : string;
  attribute syn_ramstyle of mem : signal is "block_ram";
  signal raddress : std_logic_vector(address_width-1 downto 0);
    begin
    l0: process (clk)
    begin
      if (clk = ’1’ and clk’event) then
          raddress <= address;
          if (we = ’1’) then
              mem(CONV_INTEGER(address)) <= data;
          end if;
      end if;
    end process;
    q <= mem(CONV_INTEGER(raddress));
end rtl;

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

entity ram_example1 is
```

```
  generic(
        data_width : integer := 8;
        address_width : integer := 8;
        mem_depth : integer := 256);
  port(
        data : in std_logic_vector(data_width-1 downto 0);
        address : in std_logic_vector(address_width-1 downto 0);
        en, we, clk : in std_logic;
        q : out std_logic_vector(data_width-1 downto 0));
end ram_example1;

architecture rtl of ram_example1 is

  type mem_array is array (mem_depth-1 downto 0) of
        std_logic_vector (data_width-1 downto 0);
  signal mem : mem_array;
  attribute syn_ramstyle : string;
  attribute syn_ramstyle of mem : signal is "block_ram";
  signal raddress : std_logic_vector(address_width-1 downto 0);

  begin
    l0: process (clk)
    begin
      if (clk = '1' and clk'event) then
          if (we = '1') then
              mem(CONV_INTEGER(address)) <= data;
              q <= mem(CONV_INTEGER(address));
          end if;
      end if;
    end process;
end rtl;
```

XST

For information about inferring Block SelectRAM using XST, see the Xilinx *XST User Guide.*

Inferring Block SelectRAM Verilog Examples

The following coding examples provide VHDL coding styles for inferring BlockRAMs for LeonardoSpectrum, Synplify, and XST.

LeonardoSpectrum

The following coding examples provide VHDL coding styles for inferring BlockRAMs for LeonardoSpectrum.

VHDL Example

For restrictions in inferring Block SelectRAM, see the VHDL example in the section above.

Verilog Example

```
module dualport_ram (clka, clkb, wea, addra, addrb, dia, dob);
  input clka, clkb, wea;
  input [4:0] addra, addrb;
  input [3:0] dia;
  output [3:0] dob /* synthesis syn_ramstyle="block_ram" */;
  reg [3:0] ram [31:0];
  reg [4:0] read_dpra;
  reg [3:0] dob;
// exemplar attribute ram block_ram TRUE
  always @ (posedge clka)
    begin
      if (wea) ram[addra] = dia;
    end
  always @ (posedge clkb)
    begin
      dob = ram[addrb];
    end
endmodule // dualport_ram
```

Synplify

The following coding examples show VHDL coding styles for inferring BlockRAMs for Synplify.

Synplify Verilog Example One

```
module sp_ram(din, addr, we, clk, dout);
  parameter data_width=16, address_width=10, mem_elements=600;
  input [data_width-1:0] din;
  input [address_width-1:0] addr;
  input we, clk;
  output [data_width-1:0] dout;
  reg [data_width-1:0] mem[mem_elements-1:0]
        /*synthesis syn_ramstyle = "block_ram" */;
  reg [address_width - 1:0] addr_reg;
  always @(posedge clk)
    begin
        addr_reg <= addr;
          if (we)
            mem[addr] <= din;
    end
  assign dout = mem[addr_reg];
endmodule
```

Synplify Verilog Example Two

```
module sp_ram(din, addr, we, clk, dout);
  parameter data_width=16, address_width=10, mem_elements=600;
  input [data_width-1:0] din;
  input [address_width-1:0] addr;
  input rst, we, clk;
  output [data_width-1:0] dout;

reg [data_width-1:0] mem[mem_elements-1:0]
        /*synthesis syn_ramstyle = "block_ram" */;
reg [data_width-1:0] dout;

always @(posedge clk)
  begin
    if (we)
      mem[addr] <= din;
    dout <= mem[addr];
  end
endmodule
```

XST

For information about inferring Block SelectRAM using XST, see the Xilinx *XST User Guide*.

Block SelectRAM in Virtex-4

The Block SelectRAM in Virtex-4 has been enhanced from the Virtex and Virtex-II Block SelectRAM. Similar to the Virtex-II and Spartan-3 Block SelectRAM, each Virtex-4 Block SelectRAM can:

- store 18 Kb
- read and write are synchronous operations
- true dual port in that only the stored data is shared
- data available on the outputs is determined by three Block SelectRAM operation modes of *read first, write first*, and *no change*.

Some of the enhancements to the Virtex-4 Block SelectRAM are:

- Cascadable Block SelectRAMs creating a fast 32Kb x 1 block memory
- Pipelined output registers

To infer cascaded Block SelectRAM, create a 32K x 1 Block SelectRAM as shown in the following examples.

VHDL Example

```
---------------------------------------------------
-- cascade_bram.vhd
-- version 1.0
--
-- Inference of Virtex-4 cascaded Block SelectRAM
---------------------------------------------------

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

entity cascade_bram is
generic(data_width: integer:= 1;
        address_width:integer := 15;
        mem_depth: integer:= 32768);  -- 2**address_width
port (data: in std_logic_vector(data_width-1 downto 0);
      address: in std_logic_vector(address_width-1 downto 0);
      we, en, clk: in std_logic;
      do: out std_logic_vector(data_width-1 downto 0));
end cascade_bram;

architecture rtl of cascade_bram is

  type mem_array is array (mem_depth-1 downto 0) of
    std_logic_vector (data_width-1 downto 0);

  signal mem: mem_array;
  signal raddress : std_logic_vector(address_width-1 downto 0);

begin
```

```
  process (clk)
  begin
    if (clk = '1' and clk'event) then
      if (en = '1') then
        raddress <= address;
        if (we = '1') then
          mem(conv_integer(address)) <= data;
        end if;
      end if;
    end if;
  end process;

  do <= mem(conv_integer(raddress));

end architecture;
```

Verilog Example

```
///////////////////////////////////////////////////
// cascade_bram.vhd
// version 1.0
//
// Inference of Virtex-4 cascaded Block SelectRAM
///////////////////////////////////////////////////

module cascade_bram (data, address, we, en, clk, do);

  parameter data_width = 1;
  parameter [3:0] address_width = 15;
  parameter [15:0] mem_depth = 32768; //2**address_width

  input [data_width-1:0] data;
  input [address_width-1:0] address;
  input we, en, clk;

  output [data_width-1:0] do;

  reg [data_width-1:0] mem [mem_depth-1:0];
  reg [address_width-1:0] raddress;

  always @(posedge clk)
  begin
    if (en) begin
      raddress <= address;
      if (we)
        mem[address] <= data;
    end //if (en)
  end //always

  assign do = mem[raddress];
endmodule
```

Single Port VHDL Examples

The following examples show how to infer the Block SelectRAM with the pipelined output registers.

In order to automatically infer Block SelectRAM in Synplify and Synplify-Pro, the address of the RAM has to be greater that 2K. Otherwise, the synthesis directive "syn_ramstyle" set "to block_ram" will have to be set.

VHDL Example One

```
---------------------------------------------------
-- pipeline_bram_ex1.vhd
-- version 1.0
--
-- Inference of Virtex-4 'read after write' block
-- RAM with optional output register inferred
---------------------------------------------------

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

entity pipeline_bram_ex1 is
  generic(
    data_width: integer:= 8;
    address_width:integer := 8;
    mem_depth: integer:= 256);  -- 2**address_width
  port (
    data: in std_logic_vector(data_width-1 downto 0);
    address: in std_logic_vector(address_width-1 downto 0);
    we, en, out_en, clk: in std_logic;
    do: out std_logic_vector(data_width-1 downto 0));
end pipeline_bram_ex1;

architecture rtl of pipeline_bram_ex1 is

  type mem_array is array (mem_depth-1 downto 0) of
    std_logic_vector(data_width-1 downto 0);

  signal mem: mem_array;
  signal do_aux : std_logic_vector(data_width-1 downto 0);

begin

  process (clk)
  begin
    if (clk = '1' and clk'event) then
      if (en = '1') then
        if (we = '1') then
          mem(conv_integer(address)) <= data;
          do_aux <= data;
        else
          do_aux <= mem(conv_integer(address));
        end if;
      end if;
    end if;
  end process;
```

```
-- The following process block will infer the
-- optional output register that exists in
-- the Virtex-4 Block SelectRAM

  process (clk)
  begin
    if clk'event and clk = '1' then

    -- the output clock is the same as the input clock
    -- the output clock may also be inverted with respect
    -- to the input clock
    -- if clk'event and clk = '0' then

      if out_en = '1' then
      -- independent output register clock enable

        do <= do_aux;
      end if;
    end if;
  end process;
end architecture;
```

VHDL Example Two

```
---------------------------------------------------
-- pipeline_bram_ex2.vhd
-- version 1.0
--
-- Inference of Virtex-4 'read first' block
-- RAM with optional output register inferred
---------------------------------------------------

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

entity pipeline_bram_ex2 is
  generic(
    data_width: integer:= 8;
    address_width:integer := 8;
    mem_depth: integer:= 256);  -- 2**address_width
  port (
    data: in std_logic_vector(data_width-1 downto 0);
    address: in std_logic_vector(address_width-1 downto 0);
    we, en, out_en, clk: in std_logic;
    do: out std_logic_vector(data_width-1 downto 0));
end pipeline_bram_ex2;

architecture rtl of pipeline_bram_ex2 is

  type mem_array is array (mem_depth-1 downto 0) of
    std_logic_vector (data_width-1 downto 0);

  signal mem: mem_array;
  signal do_aux : std_logic_vector(data_width-1 downto 0);

begin

  process (clk)
  begin
```

```
    if (clk = '1' and clk'event) then
      if (en = '1') then
        if (we = '1') then
          mem(conv_integer(address)) <= data;
        end if;
        do_aux <= mem(conv_integer(address));
      end if;
    end if;
  end process;

-- The following process block will infer the
-- optional output register that exists in
-- the Virtex-4 Block SelectRAM

  process (clk)
  begin
    if clk'event and clk = '1' then

    -- the output clock is the same as the input clock
    -- the output clock may also be inverted with respect
    -- to the input clock
    -- if clk'event and clk = '0' then

      if out_en = '1' then
      -- independent output register clock enable

        do <= do_aux;
      end if;
    end if;
  end process;
end architecture;
```

VHDL Example Three

```
---------------------------------------------------
-- pipeline_bram_ex3.vhd
-- version 1.0
--
-- Inference of Virtex-4 'no change' block
-- RAM with optional output register inferred
---------------------------------------------------

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

entity pipeline_bram_ex3 is
  generic(
    data_width: integer:= 8;
    address_width:integer := 8;
    mem_depth: integer:= 256);  -- 2**address_width
  port (
    data: in std_logic_vector(data_width-1 downto 0);
    address: in std_logic_vector(address_width-1 downto 0);
    we, en, out_en, clk: in std_logic;
    do: out std_logic_vector(data_width-1 downto 0));
end pipeline_bram_ex3;

architecture rtl of pipeline_bram_ex3 is
```

```
  type mem_array is array (mem_depth-1 downto 0) of
    std_logic_vector (data_width-1 downto 0);

  signal mem: mem_array;
  signal do_aux : std_logic_vector(data_width-1 downto 0);

begin

  process (clk)
  begin
    if (clk = '1' and clk'event) then
      if (en = '1') then
        if (we = '1') then
          mem(conv_integer(address)) <= data;
        else
          do_aux <= mem(conv_integer(address));
        end if;
      end if;
    end if;
  end process;

-- The following process block will infer the
-- optional output register that exists in
-- the Virtex-4 Block SelectRAM

  process (clk)
  begin
    if clk'event and clk = '1' then

    -- the output clock is the same as the input clock
    -- the output clock may also be inverted with respect
    -- to the input clock
    -- if clk'event and clk = '0' then

      if out_en = '1' then
      -- independent output register clock enable

        do <= do_aux;
      end if;
    end if;
  end process;
end architecture;
```

Single Port Verilog Examples

Following are single port Verilog examples.

Verilog Example One

```
///////////////////////////////////////////////////
// pipeline_bram_ex1.v
// version 1.0
//
// Inference of Virtex-4 'read after write' block
// RAM with optional output register inferred
///////////////////////////////////////////////////

module pipeline_bram_ex1 (data, address, we, en,
                          out_en, clk, do);

  parameter [3:0] data_width = 8;
  parameter [3:0] address_width = 8;
  parameter [8:0] mem_depth = 256; // 2**address_width

  input [data_width-1 : 0] data;
  input [address_width-1 : 0] address;
  input we, en, out_en, clk;

  output reg [data_width-1 : 0] do;

  reg [data_width-1:0] mem [mem_depth-1:0];
  reg [data_width-1:0] do_aux;

  always @(posedge clk)
  begin
    if (en) begin
      if (we)  begin
        mem[address] <= data;
        do_aux <= mem[address];
      end // if (we)
      else do_aux <= mem[address];
    end // if (en)
  end //always

// The following always block will infer the
// optional output register that exists in
// the Virtex-4 Block SelectRAM

  always @(posedge clk)
  // the output clock is the same as the input clock
  // the output clock may also be inverted with respect
  // to the input clock
  // always @(negedge clk)

  begin
    if (out_en) do <= do_aux;
    // independent output register clock enable

  end //always

endmodule
```

Verilog Example Two

```
///////////////////////////////////////////////////
// pipeline_bram_ex2.v
// version 1.0
//
// Inference of Virtex-4 'read first' block
// RAM with optional output register inferred
///////////////////////////////////////////////////

module pipeline_bram_ex2 (data, address, we, en,
                          out_en, clk, do);
```

```
  parameter [3:0] data_width = 8;
  parameter [3:0] address_width = 8;
  parameter [8:0] mem_depth = 256; // 2**address_width

  input [data_width-1 : 0] data;
  input [address_width-1 : 0] address;
  input we, en, out_en, clk;

  output reg [data_width-1 : 0] do;

  reg [data_width-1:0] mem [mem_depth-1:0];
  reg [address_width-1:0] raddress;
  reg [data_width-1:0] do_aux;

  always @(posedge clk)
  begin
    if (en) begin
      if (we)  mem[address] <= data;
      do_aux <= mem[raddress];
    end // if (en)
  end //always

// The following always block will infer the
// optional output register that exists in
// the Virtex-4 Block SelectRAM

  always @(posedge clk)
  // the output clock is the same as the input clock
  // the output clock may also be inverted with respect
  // to the input clock
  // always @(negedge clk)

  begin
    if (out_en) do <= do_aux;
    // independent output register clock enable

  end //always

endmodule
```

Verilog Example Three

```
///////////////////////////////////////////////////
// pipeline_bram_ex3.v
// version 1.0
//
// Inference of Virtex-4 'no change' block
// RAM with optional output register inferred
///////////////////////////////////////////////////

module pipeline_bram_ex3 (data, address, we, en,
                          out_en, clk, do);

  parameter [3:0] data_width = 8;
  parameter [3:0] address_width = 8;
  parameter [8:0] mem_depth = 256; // 2**address_width
  input [data_width-1 : 0] data;
  input [address_width-1 : 0] address;
  input we, en, out_en, clk;

  output reg [data_width-1 : 0] do;

  reg [data_width-1:0] do_aux;
  reg [data_width-1:0] mem [mem_depth-1:0];
  reg [address_width-1:0] raddress;

  always @(posedge clk)
  begin
    if (en) begin
      if (we)  mem[address] <= data;
      else do_aux <= mem[address];
    end // if (en)
  end //always


// The following always block will infer the
// optional output register that exists in
// the Virtex-4 Block SelectRAM

  always @(posedge clk)
  // the output clock is the same as the input clock
  // the output clock may also be inverted with respect
  // to the input clock
  // always @(negedge clk)

  begin
    if (out_en) do <= do_aux;
    // independent output register clock enable

  end //always

endmodule
```

Dual Port Block SelectRAM VHDL Examples

Following are dual port Block SelectRAM VHDL examples.

VHDL Example One

Following is the first dual port Block SelectRAM VHDL example.

```
---------------------------------------------------
-- dpbram_ex1.vhd
-- version 1.0
--
-- Inference of Virtex-4 'read after write' dual
-- port Block SelectRAM with optional output
-- registers inferred
-- Synplify will infer distributed RAM along with
-- Block SelectRAM in this example
---------------------------------------------------

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

```
entity dpram_ex1 is
  generic(
    data_width: integer:= 8;
    address_width:integer := 8;
    mem_depth: integer:= 256);  -- 2**address_width
  port (
    clk : in std_logic;
    we, en, out_en : in std_logic;
    address_a : in std_logic_vector(address_width - 1 downto 0);
    address_b : in std_logic_vector(address_width - 1 downto 0);
    di : in std_logic_vector(data_width - 1 downto 0);
    doa : out std_logic_vector(data_width - 1 downto 0);
    dob : out std_logic_vector(data_width - 1 downto 0)
  );
end dpram_ex1;

architecture syn of dpram_ex1 is

  type ram_type is array (mem_depth - 1 downto 0) of
    std_logic_vector (data_width - 1 downto 0);
  signal RAM : ram_type;
  signal doa_aux : std_logic_vector(data_width - 1 downto 0);
  signal dob_aux : std_logic_vector(data_width - 1 downto 0);


--  attribute syn_ramstyle : string;
--  attribute syn_ramstyle of ram : signal is "block_ram";

begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (en = '1') then
        if (we = '1') then
          RAM(conv_integer(address_a)) <= di;
          doa_aux <= di;
          dob_aux <= di;
        else
          doa_aux <= RAM(conv_integer(address_a));
          dob_aux <= RAM(conv_integer(address_b));
        end if;
      end if;
    end if;
  end process;

  process (clk)
  begin
    if clk'event and clk = '1' then

    -- the output clock is the same as the input clock
    -- the output clock may also be inverted with respect
    -- to the input clock
    -- if clk'event and clk = '0' then

      if out_en = '1' then
      -- independent output register clock enable

        doa <= doa_aux;
        dob <= dob_aux;
      end if;
    end if;
  end process;
end syn;
```

VHDL Example Two

```
Following is the second dual port Block SelectRAM VHDL example.
---------------------------------------------------
-- dpbram_ex2.vhd
-- version 1.0
--
-- Inference of Virtex-4 'read first' dual port
-- Block SelectRAM with optional output registers
-- inferred
-- Synplify - 'write first' port 'a'
--            'read first' port 'b'
---------------------------------------------------

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

entity dpram_ex2 is
  generic(
    data_width: integer:= 8;
    address_width:integer := 8;
    mem_depth: integer:= 256);  -- 2**address_width
  port (
    clk : in std_logic;
    we, en, out_en : in std_logic;
    address_a : in std_logic_vector(address_width - 1 downto 0);
    address_b : in std_logic_vector(address_width - 1 downto 0);
    di : in std_logic_vector(data_width - 1 downto 0);
    doa : out std_logic_vector(data_width - 1 downto 0);
    dob : out std_logic_vector(data_width - 1 downto 0)
  );
end dpram_ex2;

architecture syn of dpram_ex2 is

  type ram_type is array (mem_depth - 1 downto 0) of
    std_logic_vector (data_width - 1 downto 0);
  signal RAM : ram_type;
  signal  doa_aux : std_logic_vector(data_width - 1 downto 0);
  signal  dob_aux : std_logic_vector(data_width - 1 downto 0);

begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (en = '1') then
        if (we = '1') then
          RAM(conv_integer(address_a)) <= di;
        end if;
        doa_aux <= RAM(conv_integer(address_a));
        dob_aux <= RAM(conv_integer(address_b));
      end if;
    end if;
  end process;

  process (clk)
  begin
    if clk'event and clk = '1' then

    -- the output clock is the same as the input clock
    -- the output clock may also be inverted with respect
    -- to the input clock
    -- if clk'event and clk = '0' then

      if out_en = '1' then
      -- independent output register clock enable

        doa <= doa_aux;
        dob <= dob_aux;
      end if;
    end if;
  end process;
end syn;
```

VHDL Example Three

Following is the third dual port Block SelectRAM VHDL example.

```
---------------------------------------------------
-- dpbram_ex3.vhd
-- version 1.0
--
-- Inference of Virtex-4 'no change' on port 'a'
-- and 'read first' on port 'b' dual port Block
-- SelectRAM with two clocks and optional output
-- registers inferred
---------------------------------------------------

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

entity dpram_ex3 is
generic(
    data_width: integer:= 8;
    address_width:integer := 11;
    mem_depth: integer:= 2048);  -- 2**address_width
  port (
    clka, clkb : in std_logic;
    we, en, out_en : in std_logic;
    address_a : in std_logic_vector(address_width - 1 downto 0);
    address_b : in std_logic_vector(address_width - 1 downto 0);
    di : in std_logic_vector(data_width - 1 downto 0);
    doa : out std_logic_vector(data_width - 1 downto 0);
    dob : out std_logic_vector(data_width - 1 downto 0)
  );
end dpram_ex3;

architecture syn of dpram_ex3 is

  type ram_type is array (mem_depth - 1 downto 0) of
```

```
      std_logic_vector (data_width - 1 downto 0);
    signal RAM : ram_type;
    signal  doa_aux : std_logic_vector(data_width - 1 downto 0);
    signal  dob_aux : std_logic_vector(data_width - 1 downto 0);

  begin
    process (clka)
    begin
      if (clka'event and clka = '1') then
        if (en = '1') then
          if (we = '1') then
            RAM(conv_integer(address_a)) <= di;
          else
            doa_aux <= RAM(conv_integer(address_a));
          end if;
        end if;
      end if;
    end process;

    process (clkb)
    begin
      if (clkb'event and clkb = '1') then
        if (en = '1') then
          dob_aux <= RAM(conv_integer(address_b));
        end if;
      end if;
    end process;

    process (clka)
    begin
      if clka'event and clka = '1' then

      -- the output clock is the same as the input clock
      -- the output clock may also be inverted with respect
      -- to the input clock
      -- if clk'event and clk = '0' then

        if out_en = '1' then
        -- independent output register clock enable

          doa <= doa_aux;
        end if;
      end if;
    end process;

    process (clkb)
    begin
      if clkb'event and clkb = '1' then

      -- the output clock is the same as the input clock
      -- the output clock may also be inverted with respect
      -- to the input clock
      -- if clk'event and clk = '0' then

        if out_en = '1' then
        -- independent output register clock enable
          dob <= dob_aux;
        end if;
      end if;
    end process;
  end syn;
```

Dual Port Verilog Examples

Following are dual port Verilog examples.

Verilog Example One

Following is the first dual port Verilog example.

```
///////////////////////////////////////////////////
// dpbram_ex1.vhd
// version 1.0
//
// Inference of Virtex-4 'read after write' dual
// port Block SelectRAM with optional output
// registers inferred
// Synplify will infer distributed RAM along with
// Block SelectRAM in this example
///////////////////////////////////////////////////

module dpram_ex1 (clk, we, en, out_en, address_a,
                  address_b, di, doa, dob);

  parameter [3:0] data_width = 8;
  parameter [3:0] address_width = 8;
  parameter [8:0] mem_depth = 256;  // 2**address_width

  input clk, we, en, out_en;
  input [address_width-1 : 0] address_a;
  input [address_width-1 : 0] address_b;
  input [data_width-1 : 0] di;

  output reg [data_width-1 : 0] doa;
  output reg [data_width-1 : 0] dob;

  reg [data_width-1:0] ram [mem_depth-1:0];
  reg [data_width-1 : 0] doa_aux;
  reg [data_width-1 : 0] dob_aux;

  always @(posedge clk)
  begin
    if (en) begin
      if (we) begin
        ram[address_a] <= di;
        doa_aux <= di;
        dob_aux <= di;
      end //if (we)
      else begin
        doa_aux <= ram[address_a];
        dob_aux <= ram[address_b];
      end //else (we)
    end //if (en)
  end //always


// The following always block will infer the
// optional output register that exists in
// the Virtex-4 Block SelectRAM

  always @(posedge clk)
  // the output clock is the same as the input clock
  // the output clock may also be inverted with respect
  // to the input clock
  // always @(negedge clk)

  begin
    if (out_en) begin
    // independent output register clock enable

      doa <= doa_aux;
      dob <= dob_aux;
    end //if out_en
  end //always

endmodule
```

Verilog Example Two

Following is the second dual port Verilog example.

```
///////////////////////////////////////////////////
// dpbram_ex2.vhd
// version 1.0
//
// Inference of Virtex-4 'read first' dual port
// Block SelectRAM with optional output registers
// inferred
// Synplify - 'write first' port 'a'
//            'read first' port 'b'
///////////////////////////////////////////////////

module dpram_ex2 (clk, we, en, out_en, address_a,
                  address_b, di, doa, dob);

  parameter [3:0] data_width = 8;
  parameter [3:0] address_width = 8;
  parameter [8:0] mem_depth = 256;  // 2**address_width

  input clk, we, en, out_en;
  input [address_width-1 : 0] address_a;
  input [address_width-1 : 0] address_b;
  input [data_width-1 : 0] di;

  output reg [data_width-1 : 0] doa;
  output reg [data_width-1 : 0] dob;

  reg [data_width-1:0] ram [mem_depth-1:0];
  reg [data_width-1 : 0] doa_aux;
  reg [data_width-1 : 0] dob_aux;

  always @(posedge clk)
  begin
    if (en) begin
```

```
      if (we) begin
        ram[address_a] <= di;
      end //if (we)
      doa_aux <= ram[address_a];
      dob_aux <= ram[address_b];
    end //if (en)
  end //always

// The following always block will infer the
// optional output register that exists in
// the Virtex-4 Block SelectRAM

  always @(posedge clk)
  // the output clock is the same as the input clock
  // the output clock may also be inverted with respect
  // to the input clock
  // always @(negedge clk)

  begin
    if (out_en) begin
    // independent output register clock enable

      doa <= doa_aux;
      dob <= dob_aux;
    end //if out_en
  end //always

endmodule
```

Verilog Example Three

Following is the third dual port Verilog example.

```
///////////////////////////////////////////////////
// dpbram_ex3.v
// version 1.0
//
// Inference of Virtex-4 'no change' on port 'a'
// and 'read first' on port 'b' dual port Block
// SelectRAM with two clocks and optional output
// registers inferred
///////////////////////////////////////////////////
module dpram_ex3 (clka, clkb, we, en, out_en,
                  address_a, address_b, di, doa, dob);

  parameter [3:0] data_width = 8;
  parameter [3:0] address_width = 8;
  parameter [8:0] mem_depth = 256;  // 2**address_width

  input clka, clkb, we, en, out_en;
  input [address_width-1 : 0] address_a;
  input [address_width-1 : 0] address_b;
  input [data_width-1 : 0] di;

  output reg [data_width-1 : 0] doa;
  output reg [data_width-1 : 0] dob;
  reg [data_width-1:0] ram [mem_depth-1:0];
  reg [data_width-1 : 0] doa_aux;
  reg [data_width-1 : 0] dob_aux;

  always @(posedge clka)
  begin
    if (en) begin
      if (we) ram[address_a] <= di;
      else doa_aux <= ram[address_a];
    end //if (en)
  end //always

  always @(posedge clkb)
  begin
    if (en) dob_aux <= ram[address_b];
  end //always

// The following always blocks will infer the
// optional output register that exists in
// the Virtex-4 Block SelectRAM

  always @(posedge clka)
  // the output clock is the same as the input clock
  // the output clock may also be inverted with respect
  // to the input clock
  // always @(negedge clk)

  begin
    if (out_en) begin
    // independent output register clock enable

      doa <= doa_aux;
    end //if out_en
  end //always

  always @(posedge clkb)
  // the output clock is the same as the input clock
  // the output clock may also be inverted with respect
  // to the input clock
  // always @(negedge clk)

  begin
    if (out_en) begin
    // independent output register clock enable

      dob <= dob_aux;
    end //if out_en
  end //always

endmodule
```

Implementing Distributed SelectRAM

Distributed SelectRAM can be either instantiated or inferred. The following sections describe and give examples of both instantiating and inferring distributed SelectRAM.

The following RAM Primitives are available for instantiation.

- Static synchronous single-port RAM (RAM16x1S, RAM32x1S)

Additional single-port RAM available for Virtex-II,. Virtex-II Pro, Virtex-II Pro X. and Spartan-3 devices only: RAM16X2S, RAM16X4S, RAM16X8S, RAM32X1S, RAM32X2S, RAM32X4S, RAM32X8S, RAM64X1S, RAM64X2S, and RAM128X1S.

- Static synchronous dual-port RAM (RAM16x1D, RAM32x1D)

  Additional dual-port RAM is available for Virtex-II, Virtex-II Pro, Virtex-II Pro X, or Spartan-3 devices only: RAM64X1D.

For more information on distributed SelectRAM, see the Xilinx *Libraries Guides*.

Instantiating Distributed SelectRAM in VHDL

The following examples provide VHDL coding hints for LeonardoSpectrum, Synplify and XST.

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.
library IEEE;
use IEEE.std_logic_1164.all;

-- Add the following two lines if using XST and Synplify:
-- library unisim;
-- use unisim.vcomponents.all;

entity ram_16x4s is
  port (
       o   : out std_logic_vector(3 downto 0);
       we  : in std_logic;
       clk : in std_logic;
       d   : in std_logic_vector(3 downto 0);
       a   : in std_logic_vector(3 downto 0)
       );
end ram_16x4s;

architecture xilinx of ram_16x4s is

-- remove the following component declarations
-- if using XST or Synplify

  component RAM16x1S is
    generic (INIT : bit_vector :=x"0000");
    port (
       O : out std_logic;
       D : in std_logic;
       A3, A2, A1, A0 : in std_logic;
       WE, WCLK : in std_logic
       );
  end component;

  begin
    U0 : RAM16x1S
    generic map (INIT =>x"FFFF")
    port map (O => o(0), WE => we, WCLK => clk, D => d(0),
              A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));

    U1 : RAM16x1S
    generic map (INIT =>x"ABCD")
    port map (O => o(1), WE => we, WCLK => clk, D => d(1),
              A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
    U2 : RAM16x1S
    generic map (INIT =>x"BCDE")
    port map (O => o(2), WE => we, WCLK => clk, D => d(2),
              A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));

    U3 : RAM16x1S
    generic map (INIT =>x"CDEF")
    port map (O => o(3), WE => we, WCLK => clk, D => d(3),
              A0 => a(0), A1 => a(1), A2 => a(2), A3 => a(3));
end xilinx;
```

Instantiating Distributed SelectRAM in Verilog

The following coding provides Verilog coding hints for LeonardoSpectrum, Synplify, and XST.

LeonardoSpectrum

```
// This example shows how to create a
// 16x4 RAM using Xilinx RAM16X1S component.
module RAM_INIT_EX1 (DATA_BUS, ADDR, WE, CLK);
  input [3:0] ADDR;
  inout [3:0] DATA_BUS;
  input WE, CLK;
  wire [3:0] DATA_OUT;
// Only for Simulation
// -- the defparam will not synthesize
// Use the defparam for RTL simulation.
// There is no defparam needed for
// Post P&R simulation.
// exemplar translate_off
  defparam RAM0.INIT="0101", RAM1.INIT="AAAA", RAM2.INIT="FFFF",
        RAM3.INIT="5555";
// exemplar translate_on
  assign DATA_BUS = !WE ? DATA_OUT : 4'hz;
// Instantiation of 4 16X1 Synchronous RAMs
  RAM16X1S RAM3 (
      .O (DATA_OUT[3]),.D (DATA_BUS[3]),.A3 (ADDR[3]),.A2 (ADDR[2]),
        .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* exemplar attribute RAM3 INIT 5555 */;
  RAM16X1S RAM2 (
      .O (DATA_OUT[2]), .D (DATA_BUS[2]), .A3 (ADDR[3]) ,.A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* exemplar attribute RAM2 INIT FFFF */;
  RAM16X1S RAM1 (
      .O (DATA_OUT[1]), .D (DATA_BUS[1]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* exemplar attribute RAM1 INIT AAAA */;
  RAM16X1S RAM0 (
      .O (DATA_OUT[0]), .D (DATA_BUS[0]), .A3 (ADDR[3]), .A2 (ADDR[2]),
      .A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK))
  /* exemplar attribute RAM0 INIT 0101 */;
endmodule
module RAM16X1S (O,D,A3, A2, A1, A0, WE, WCLK);
  output O;
  input D;
  input A3;
  input A2;
  input A1;
  input A0;
  input WE;
  input WCLK;
endmodule
```

Synplify and XST

```
/////////////////////////////////////////////////////////
// RAM_INIT_EX.V Version 2.0
// This is an example of an instantiation of
// a RAM16X1S with an INIT passed through a
// local parameter
/////////////////////////////////////////////////////////
// add the following line if using Synplify:
// `include "<path_to_synplify> \lib\xilinx\unisim.v"
/////////////////////////////////////////////////////////

module RAM_INIT_EX1 (DATA_BUS, ADDR, WE, CLK);
input [3:0] ADDR;
inout [3:0] DATA_BUS;
input WE, CLK;
wire [3:0] DATA_OUT;

assign DATA_BUS = !WE ? DATA_OUT : 4'hz;

RAM16X1S
#(.INIT(16'hFFFF)) RAM3
(.O (DATA_OUT[3]), .D (DATA_BUS[3]), .A3 (ADDR[3]), .A2 (ADDR[2]),
.A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK));

RAM16X1S
#(.INIT(16'hAAAA)) RAM2
(.O (DATA_OUT[2]), .D (DATA_BUS[2]), .A3 (ADDR[3]), .A2 (ADDR[2]),
.A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK));

RAM16X1S
#(.INIT(16'h7777)) RAM1
(.O (DATA_OUT[1]), .D (DATA_BUS[1]), .A3 (ADDR[3]), .A2 (ADDR[2]),
.A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK));

RAM16X1S
#(.INIT(16'h0101)) RAM0
(.O (DATA_OUT[0]), .D (DATA_BUS[0]), .A3 (ADDR[3]), .A2 (ADDR[2]),
.A1 (ADDR[1]), .A0 (ADDR[0]), .WE (WE), .WCLK (CLK));

endmodule
```

Inferring Distributed SelectRAM in VHDL

Precision Synthesis and Synplify Pro support the initialization of block RAM via signal initialization. This is supported in VHDL only. The basic syntax for this feature is:

```
  type mem_array is array (31 downto 0) of
       std_logic_vector (7 downto 0);
  signal mem : mem_array :=
 (X"0A", X"00", X"01", X"00", X"01", X"3A",
 X"00", X"08", X"02", X"02", X"00", X"02",
 X"08", X"00", X"01", X"02", X"40", X"41",
```

:
:

LeonardoSpectrum, Precision Synthesis, Synplify, and XST

The following coding examples provide VHDL and Verilog coding styles for LeonardoSpectrum, Precision Synthesis, Synplify, and XST.

- VHDL Example One

The following is a 32x8 (32 words by 8 bits per word) synchronous, dual-port RAM example.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity ram_32x8d_infer is
  generic(
        d_width : integer := 8;
        addr_width : integer := 5;
        mem_depth : integer := 32
        );
  port (
        o : out STD_LOGIC_VECTOR(d_width - 1 downto 0);
        we, clk : in STD_LOGIC;
        d : in STD_LOGIC_VECTOR(d_width - 1 downto 0);
        raddr, waddr : in STD_LOGIC_VECTOR(addr_width - 1 downto 0));
end ram_32x8d_infer;

architecture xilinx of ram_32x8d_infer is
  type mem_type is array (
        mem_depth - 1 downto 0) of
        STD_LOGIC_VECTOR (d_width - 1 downto 0);
  signal mem : mem_type;
begin
  process(clk, we, waddr)
  begin
      if (rising_edge(clk)) then
          if (we = '1') then
              mem(conv_integer(waddr)) <= d;
          end if;
      end if;
  end process;
  process(raddr)
  begin
    o <= mem(conv_integer(raddr));
  end process;
end xilinx;
```

- VHDL Example Two

The following is a 32x8 (32 words by 8 bits per word) synchronous, single-port RAM example.

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

entity ram_32x8s_infer is
  generic(
        d_width : integer := 8;
        addr_width : integer := 5;
        mem_depth : integer := 32
        );
  port (
        o : out STD_LOGIC_VECTOR(d_width - 1 downto 0);
        we, wclk : in STD_LOGIC;
        d : in STD_LOGIC_VECTOR(d_width - 1 downto 0);
        addr : in STD_LOGIC_VECTOR(addr_width - 1 downto 0));
end ram_32x8s_infer;

architecture xilinx of ram_32x8s_infer is
        type mem_type is array (mem_depth - 1 downto 0)
        of STD_LOGIC_VECTOR (d_width - 1 downto 0);
  signal mem : mem_type;
begin
  process(wclk, we, addr)
  begin
    if (rising_edge(wclk)) then
        if (we = '1') then
            mem(conv_integer(addr)) <= d;
        end if;
    end if;
  end process;
  o <= mem(conv_integer(addr));
end xilinx;
```

Inferring Distributed SelectRAM in Verilog

The following coding examples provide Verilog coding hints for Synplify, LeonardoSpectrum, Precision Synthesis, and XST.

LeonardoSpectrum, Precision Synthesis, Synplify, and XST

The following is a 32x8 (32 words by 8 bits per word) synchronous, dual-port RAM example.

```
module ram_32x8d_infer (o, we, d, raddr, waddr, clk);
  parameter d_width = 8, addr_width = 5;
  output [d_width - 1:0] o;
  input we, clk;
  input [d_width - 1:0] d;
  input [addr_width - 1:0] raddr, waddr;

  reg [d_width - 1:0] o;
  reg [d_width - 1:0] mem [(2 ** addr_width) - 1:0];

  always @(posedge clk)
    if (we)
        mem[waddr] = d;

  always @(raddr)
    o = mem[raddr];
endmodule
```

The following is a 32x8 (32 words by 8 bits per word) synchronous, single-port RAM example.

```
module ram_32x8s_infer (o, we, d, addr, wclk);
  parameter d_width = 8, addr_width = 5;
  output [d_width - 1:0] o;
  input we, wclk;
  input [d_width - 1:0] d;
  input [addr_width - 1:0] addr;

  reg [d_width - 1:0] mem [(2 ** addr_width) - 1:0];
  always @(posedge wclk)
    if (we)
        mem[addr] = d;
  assign o = mem[addr];
endmodule
```

Implementing ROMs

ROMs can be implemented as follows.

- Use RTL descriptions of ROMs
- Instantiate 16x1 and 32x1 ROM primitives

Following are RTL VHDL and Verilog ROM coding examples.

RTL Description of a Distributed ROM VHDL Example

Use the following coding example for LeonardoSpectrum, Precision Synthesis, Synplify and XST.

```
--
--  Behavioral 16x4 ROM Example
--           rom_rtl.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;

entity rom_rtl is
  port (
      ADDR : in INTEGER range 0 to 15;
      DATA : out STD_LOGIC_VECTOR (3 downto 0)
      );
end rom_rtl;

architecture XILINX of rom_rtl is
  subtype ROM_WORD is STD_LOGIC_VECTOR (3 downto 0);
  type ROM_TABLE is array (0 to 15) of ROM_WORD;
  constant ROM : ROM_TABLE := ROM_TABLE'(
      ROM_WORD'("0000"),
      ROM_WORD'("0001"),
      ROM_WORD'("0010"),
      ROM_WORD'("0100"),
      ROM_WORD'("1000"),
      ROM_WORD'("1100"),
      ROM_WORD'("1010"),
      ROM_WORD'("1001"),
      ROM_WORD'("1001"),
      ROM_WORD'("1010"),
      ROM_WORD'("1100"),
      ROM_WORD'("1001"),
      ROM_WORD'("1001"),
      ROM_WORD'("1101"),
      ROM_WORD'("1011"),
      ROM_WORD'("1111")
```

```
        );
begin
  DATA <= ROM(ADDR);     -- Read from the ROM
end XILINX;
```

RTL Description of a Distributed ROM Verilog Example

Use the following coding example for LeonardoSpectrum, Precision Synthesis, Synplify, and XST.

```
/*
 * ROM_RTL.V
 * Behavioral Example of 16x4 ROM
 */

module rom_rtl(ADDR, DATA);
  input [3:0] ADDR;
  output [3:0] DATA;
  reg [3:0] DATA;

// A memory is implemented
// using a case statement

  always @(ADDR)
  begin
    case (ADDR)
      4'b0000 : DATA = 4'b0000 ;
      4'b0001 : DATA = 4'b0001 ;
      4'b0010 : DATA = 4'b0010 ;
      4'b0011 : DATA = 4'b0100 ;
      4'b0100 : DATA = 4'b1000 ;
      4'b0101 : DATA = 4'b1000 ;
      4'b0110 : DATA = 4'b1100 ;
      4'b0111 : DATA = 4'b1010 ;
      4'b1000 : DATA = 4'b1001 ;
      4'b1001 : DATA = 4'b1001 ;
      4'b1010 : DATA = 4'b1010 ;
      4'b1011 : DATA = 4'b1100 ;
      4'b1100 : DATA = 4'b1001 ;
      4'b1101 : DATA = 4'b1001 ;
      4'b1110 : DATA = 4'b1101 ;
      4'b1111 : DATA = 4'b1111 ;
    endcase
  end
endmodule
```

With the VHDL and Verilog examples above, synthesis tools create ROMs using function generators (LUTs and MUXFs) or the ROM primitives.

Another method for implementing ROMs is to instantiate the 16x1 or 32x1 ROM primitives. To define the ROM value, use the Set Attribute or equivalent command to set the INIT property on the ROM component.

For more information on the correct syntax, see your synthesis tool documentation.

This type of command writes the ROM contents to the netlist file so the Xilinx tools can initialize the ROM. The INIT value should be specified in hexadecimal values. For examples of this property using a RAM primitive, see the VHDL and Verilog RAM examples in the following section.

Implementing ROMs Using Block SelectRAM

LeonardoSpectrum and Synplify can infer ROM using Block SelectRAM.

LeonardoSpectrum

- In LeonardoSpectrum, synchronous ROMs with address widths greater than eight bits are automatically mapped to Block SelectRAM.
- Asynchronous ROMs and synchronous ROMs (with address widths less than eight bits) are automatically mapped to distributed SelectRAM.

Synplify

Synplify can infer ROMs using Block SelectRAM instead of LUTs for Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4 and Spartan-3 in the following cases:

- For Virtex and Virtex-E, the address line must be between 8 and 12 bits.
- For Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, or Spartan-3, the address line must be between 9 and 14 bits.
- The address lines must be registered with a simple flip-flop (no resets or enables) or the ROM output can be registered with enables or sets/resets. However, you cannot use both sets/resets and enables. The flip-flops' sets/resets can be either synchronous or asynchronous. In the case where asynchronous sets/resets are used, Synplify creates registers with the sets/resets and then either AND or OR these registers with the output of the block RAM.

RTL Description of a ROM VHDL Example Using Block SelectRAM

Following is some incomplete VHDL that demonstrates the above inference rules.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity rom_rtl is
  port (
        ADDR : in INTEGER range 0 to 1023;
        CLK  : in std_logic;
        DATA : out STD_LOGIC_VECTOR (3 downto 0)
        );
end rom_rtl;

architecture XILINX of rom_rtl is

  subtype ROM_WORD is STD_LOGIC_VECTOR (3 downto 0);
  type ROM_TABLE is array (0 to 1023) of ROM_WORD;
  constant ROM : ROM_TABLE := ROM_TABLE'(
        ROM_WORD'("0000"),
        ROM_WORD'("0001"),
        ROM_WORD'("0010"),
        ROM_WORD'("0100"),
        ROM_WORD'("1000"),
        ROM_WORD'("1100"),
        ROM_WORD'("1010"),
        ROM_WORD'("1001"),
        ROM_WORD'("1001"),
        ROM_WORD'("1010"),
        ROM_WORD'("1100"),
        ROM_WORD'("1001"),
        ROM_WORD'("1001"),
        ROM_WORD'("1101"),
        ROM_WORD'("1011"),
        ROM_WORD'("1111")
    :
    :
    :
        );
  begin
  process (CLK) begin
    if clk'event and clk = '1' then
        DATA <= ROM(ADDR);      -- Read from the ROM
    end if;
  end process;
end XILINX;
```

RTL Description of a ROM Verilog Example using Block SelectRAM

Following is an incomplete Verilog example that demonstrates the above inference rules:

```
/*
* This code is incomplete but demonstrates the
* rules for inferring block RAM for ROMs
* ROM_RTL.V
* block RAM ROM Example
*/
module rom_rtl(ADDR, CLK, DATA) ;
  input [9:0] ADDR ;
  input CLK ;
  output [3:0] DATA ;
  reg [3:0] DATA ;

// A memory is implemented
// using a case statement

  always @(posedge CLK)
  begin
    case (ADDR)
        9'b000000000 : DATA = 4'b0000 ;
        9'b000000001 : DATA = 4'b0001 ;
        9'b000000010 : DATA = 4'b0010 ;
        9'b000000011 : DATA = 4'b0100 ;
        9'b000000100 : DATA = 4'b1000 ;
        9'b000000101 : DATA = 4'b1000 ;
        9'b000000110 : DATA = 4'b1100 ;
        9'b000000111 : DATA = 4'b1010 ;
        9'b000001000 : DATA = 4'b1001 ;
        9'b000001001 : DATA = 4'b1001 ;
        9'b000001010 : DATA = 4'b1010 ;
        9'b000001011 : DATA = 4'b1100 ;
        9'b000001100 : DATA = 4'b1001 ;
        9'b000001101 : DATA = 4'b1001 ;
        9'b000001110 : DATA = 4'b1101 ;
        9'b000001111 : DATA = 4'b1111 ;
                :
                :
                :
    endcase
  end
endmodule
```

Implementing FIFOs

FIFOs are generally implemented in one of three ways:

- Use Core Generator to generate a FIFO implementation which is instantiated in the end design.
- Instantiate the Virtex-4 FIFO primitive into the code.
- Describe the FIFO logic behaviorally described; the synthesis tool infers the FIFO function.

The most common method is to use Core Generato to create the FIFO. For more information on using Core Generator for FIFO generation and implementation, see the Core Generato documentation.

RTL inference of the FIFO is left to the individual to code. There are many examples and application notes available on the subject. For more information on instantiating the Virtex-4 BlockRAM FIFO, see the Xilinx *Virtex-4 HDL Libraries Guide.*

Implementing CAM

Content Addressable Memory (CAM) or associative memory is a storage device which can be addressed by its own contents. For more information on CAM designs in Virtex FPGA devices, see:

- Xilinx Application Note XAPP201, *"An Overview of Multiple CAM Designs in Virtex Family Devices"*
- Xilinx Application Note XAPP202, *"Content Addressable Memory (CAM) in ATM Applications"*
- Xilinx Application Note XAPP203, *"Designing Flexible, Fast CAMs with Virtex Family FPGA devices"*
- Xilinx Application Note XAPP204, *"Using Block RAM for High Performance Read/Write CAMs"*

Using CORE Generator to Implement Memory

Implementing memory with the CORE Generator is similar to implementing any module with CORE Generator except for defining the memory initialization file. For more information on the initialization file, see the memory module data sheets that come with every CORE Generator module.

Implementing Shift Registers

This section applies to the following devices:

- Virtex
- Virtex E
- Virtex II
- Virtex II Pro
- Virtex II Pro X

- Virtex-4
- Spartan-II
- Spartan-IIE
- Spartan-3

The SRL16 is a very efficient way to create shift registers without using up flip-flop resources. You can create shift registers that vary in length from one to sixteen bits. The SRL16 is a shift register look up table (LUT) whose inputs (A3, A2, A1,A0) determine the length of the shift register. The shift register may be of a fixed, static length, or it may be dynamically adjusted. The shift register LUT contents are initialized by assigning a four-digit hexadecimal number to an INIT attribute. The first, or the left-most, hexadecimal digit is the most significant bit. If an INIT value is not specified, it defaults to a value of four zeros (0000) so that the shift register LUT is cleared during configuration.

The data (D) is loaded into the first bit of the shift register during the Low-to-High clock (CLK) transition. During subsequent Low-to-High clock transitions data is shifted to the next highest bit position as new data is loaded. The data appears on the Q output when the shift register length determined by the address inputs is reached.

The Static Length Mode of SRL16 implements any shift register length from 1 to 16 bits in one LUT. Shift register length is (N+1) where N is the input address. Synthesis tools implement longer shift registers with multiple SRL16 and additional combinatorial logic for multiplexing.

In Virtex-II, Virtex-II Pro, Virtex-II Pro X, and Spartan-3 devices, additional cascading shift register LUTs (SRLC16) are available. SRLC16 supports synchronous shift-out output of the last (16th) bit. This output has a dedicated connection to the input of the next SRLC16 inside the CLB. With four slices and dedicated multiplexers (such as MUXF5 and MUXF6) available in one Virtex-II, Virtex-II Pro, Virtex-II Pro X, or Spartan-3 CLB, up to a 128-bit shift register can be implemented effectively using SRLC16. Synthesis tools, Synplify 7.1, LeonardoSpectrum 2002a, and XST can infer the SRLC16. For more information, see the product data sheet and user guide.

Dynamic Length Mode can be implemented using SRL16 or SRLC16. Each time a new address is applied to the 4-input address pins, the new bit position value is available on the Q output after the time delay to access the LUT. LeonardoSpectrum, Synplify, and XST can infer a shift register component. A coding example for a dynamic SRL is included following the SRL16 inference example.

Inferring SRL16 in VHDL

```
-- VHDL example design of SRL16
-- inference for Virtex
-- This design infer 16 SRL16
-- with 16 pipeline delay
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity pipeline_delay is
  generic (
        cycle : integer := 16;
        width :integer := 16
        );
  port (
        DATA_IN :in std_logic_vector(width - 1 downto 0);
```

```
      CLK :in std_logic;
      RESULT :out std_logic_vector(width - 1 downto 0)
      );
end pipeline_delay;

architecture behav of pipeline_delay is

  type my_type is array (0 to cycle -1) of
      std_logic_vector(width -1 downto 0);
  signal int_sig :my_type;

begin
  main : process (CLK)
    begin
      if CLK'event and CLK = '1' then
        int_sig <= DATA_IN & int_sig(0 to cycle - 2);
      end if;
  end process main;

  RESULT <= int_sig(cycle -1);

end behav;
```

Inferring SRL16 in Verilog

```
// Verilog Example SRL
//This design infer 3 SRL16 with 4 pipeline delay

module srle_example (CLK, ENABLE, DATA_IN, RESULT);

   parameter cycle=4;
   parameter width = 3;

   input CLK, ENABLE;
   input [0:width] DATA_IN;
   output [0:width] RESULT;

   reg [0:width-1] shift [cycle-1:0];
   integer i;

   always @(posedge CLK)
      if (ENABLE) begin
         for (i = (cycle-1);i >0; i=i-1)
            shift[i] = shift[i-1];
         shift[0] = DATA_IN;
      end

   assign RESULT = shift[cycle-1];

endmodule
```

Inferring Dynamic SRL16 in VHDL

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

entity srltest is
  port (
        DATAIN : std_logic_vector(7 downto 0);
        CLK, ENABLE : in std_logic;
        ADDR : in integer range 3 downto 0;
        RESULT : out std_logic_vector(7 downto 0));
end srltest;

architecture rtl of srltest is

   type dataAryType is array(3 downto 0) of std_logic_vector(7 downto 0);
   signal srldata : dataAryType;

begin

   RESULT <= srldata(CONV_INTEGER(ADDR));

   process(CLK) begin
      if (CLK'event and CLK = '1') then
         if (ENABLE='1') then
            srldata <= (srldata(2 downto 0) & DATAIN);
         end if;
      end if;
   end process;

end rtl;
```

Inferring Dynamic SRL16 in Verilog

```
module test_srl(CLK, ENABLE, DATAIN, RESULT, ADDR);

  input CLK, ENABLE;
  input [3:0] DATAIN;
  input [3:0] ADDR;
  output [3:0] RESULT;

  reg [3:0] srldata[15:0];

  integer i;
  always @(posedge CLK)
    if (ENABLE)
   begin
        for (i=15; i>0; i=i-1)
             srldata[i] <= srldata[i-1];
        srldata[0] <= dataIn;
   end

  assign RESULT = srldata[ADDR];

endmodule
```

Implementing LFSR

The SRL (Shift Register LUT) implements very efficient shift registers and can be used to implement Linear Feedback Shift Registers. For a description of the implementation of Linear Feedback Shift Registers (LFSR) using the Virtex SRL macro, see Xilinx Application Note XAPP210, "*Linear Feedback Shift Registers in Virtex Devices*." One half of a CLB can be configured to implement a 15-bit LFSR, one CLB can implement a 52-bit LFSR, and with two CLBs a 118-bit LFSR is implemented.

Implementing Multiplexers

A 4-to-1 multiplexer can be efficiently implemented in a single family slice by using dedicated components called MUXF's. The six input signals (four inputs, two select lines) use a combination of two LUTs and MUXF5 available in every slice. Up to 9 input functions can be implemented with this configuration.

Virtex, Virtex-E, and Spartan-II Families

In the Virtex, Virtex-E, and Spartan-II families, larger multiplexers can be implemented using two adjacent slices in one CLB with its dedicated MUXF5s and a MUXF6.

Virtex-II Parts and Newer

The slices in Virtex-II parts and newer contain dedicated two-input multiplexers (one MUXF5 and one MUXFX per slice). MUXF5 is used to combine two LUTs. MUXFX can be used as MUXF6, MUXF7, and MUXF8 to combine 4, 8, and 16 LUTs, respectively. For more information on designing large multiplexers in Virtex-II parts and newer, see the *Virtex-II Platform FPGA User Guide*.

In addition, you can use internal 3-state buffers (BUFTs) to implement large multiplexers. Large multiplexers built with BUFTs have the following advantages.

- Can vary in width with only minimal impact on area and delay
- Can have as many inputs as there are 3-state buffers per horizontal longline in the target device
- Have one-hot encoded selector inputs

This last point is illustrated in the following VHDL and Verilog designs of a 5-to-1 multiplexer built with gates. Typically, the gate version of this multiplexer has binary encoded selector inputs and requires three select inputs (SEL<2:0>). The schematic representation of this design is shown in Figure 4-4.

Some synthesis tools include commands that allow you to switch between multiplexers with gates or with 3-states. For more information, see your synthesis tool documentation.

The VHDL and Verilog designs provided at the end of this section show a 5-to-1 multiplexer built with 3-state buffers. The 3-state buffer version of this multiplexer has one-hot encoded selector inputs and requires five select inputs (SEL<4:0>).

Mux Implemented with Gates VHDL Example

The following example shows a MUX implemented with Gates.

```
-- MUX_GATE.VHD
-- 5-to-1 Mux Implemented in Gates

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity mux_gate is
  port (
        SEL: in STD_LOGIC_VECTOR (2 downto 0);
        A,B,C,D,E: in STD_LOGIC;
        SIG: out STD_LOGIC
        );
end mux_gate;
```

```
architecture RTL of mux_gate is begin
    SEL_PROCESS: process (SEL,A,B,C,D,E)
    begin
      case SEL is
        when "000" => SIG <= A;
        when "001" => SIG <= B;
        when "010" => SIG <= C;
        when "011" => SIG <= D;
        when others => SIG <= E;
      end case;
  end process SEL_PROCESS;
end RTL;
```

MUX Implemented with Gates Verilog Example

The following example shows a MUX implemented with Gates.

```
/// mux_gate.v
// 5-to-1 Mux Implemented in Gates

module mux_gate(
  input [2:0] SEL,
  input A, B, C, D, E,
  output reg SIG);

  always @(*)
    begin
      case (SEL)
        3'b000 : SIG = A;
        3'b001 : SIG = B;
        3'b010 : SIG = C;
        3'b011 : SIG = D;
        default : SIG = E;
      endcase
    end
```

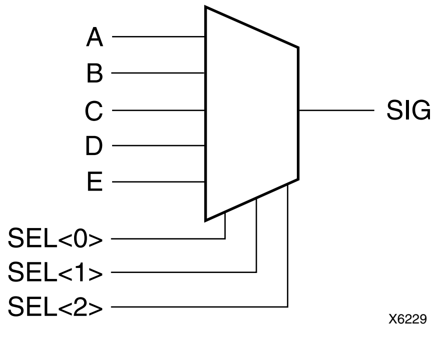

Figure 4-4: **5-to-1 MUX Implemented with Gates**

Wide MUX Mapped to MUXFs

Synthesis tools use MUXF5 and MUXF6, and for Virtex-II, Virtex-II Pro, Virtex-II Pro X, and Spartan-3 use MUXF7 and MUXF8 to implement wide multiplexers. These MUXes can, respectively, be used to create a 5, 6, 7 or 8 input function or a 4-to-1, 8-to-1, 16-to-1 or a 32-to-1 multiplexer.

Using Pipelining

You can use pipelining to dramatically improve device performance at the cost of added latency (more clock cycles to process the data). Pipelining increases performance by restructuring long data paths with several levels of logic and breaking it up over multiple clock cycles.

This method allows a faster clock cycle and, as a result, an increased data throughput at the expense of added data latency. Because the Xilinx FPGA devices are register-rich, this is usually an advantageous structure for FPGA designs, since the pipeline is created at no cost in terms of device resources. Because data is now on a multi-cycle path, special considerations must be used for the rest of your design to account for the added path latency. You must also be careful when defining timing specifications for these paths.

Some synthesis tools have limited capability for constraining multi-cycle paths or translating these constraints to Xilinx implementation constraints. For more information on multi-cycle paths, see your synthesis tool documentation. If your tool cannot translate the constraint, but can synthesize to a multi-cycle path, you can add the constraint to the UCF file.

Before Pipelining

In the following example, the clock speed is limited by:

- the clock-to out-time of the source flip-flop
- the logic delay through four levels of logic
- the routing associated with the four function generators
- the setup time of the destination register

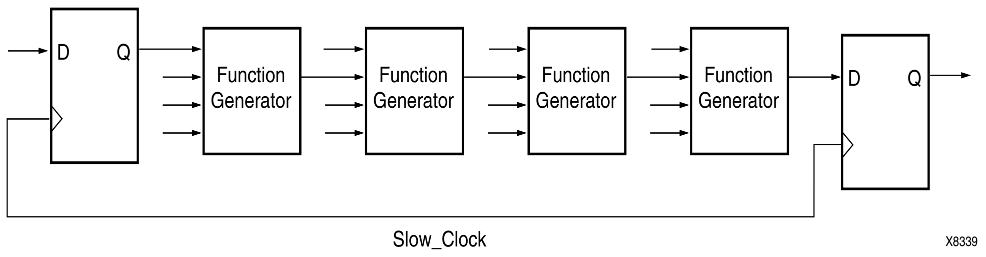

Figure 4-5: **Before Pipelining**

After Pipelining

This is an example of the same data path in the previous example after pipelining. Because the flip-flop is contained in the same CLB as the function generator, the clock speed is limited by:

- the clock-to-out time of the source flip-flop
- the logic delay through one level of logic; one routing delay
- the setup time of the destination register

In this example, the system clock runs much faster than in the previous example.

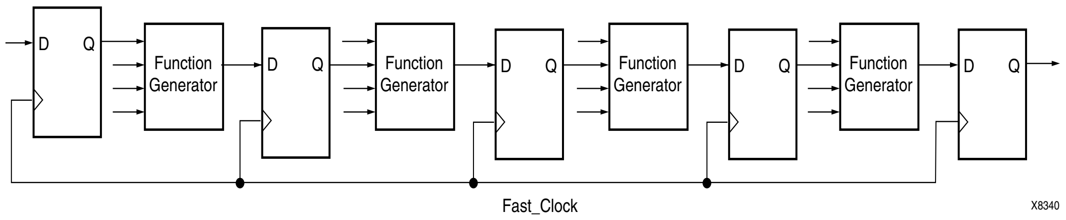

Figure 4-6: **After Pipelining**

Design Hierarchy

HDL designs can either be synthesized as a large flat module, or as many small modules. Each methodology has its advantages and disadvantages, but as higher density FPGA devices are created, the advantages of hierarchical designs outweigh many of the disadvantages.

Advantages of Hierarchical Designs

Hierarchical designs:

- Provide easier and faster verification and simulation
- Allow several engineers to work on one design at the same time
- Speed up design compilation
- Produce designs that are easier to understand
- Manage the design flow efficiently

Disadvantages of Hierarchical Designs

Some disadvantages of hierarchical designs are:

- Design mapping into the FPGA may not be as optimal across hierarchical boundaries; this can cause lesser device utilization and decreased design performance. If special care is taken, the effect of this can be minimized.
- Design file revision control becomes more difficult.
- Designs become more verbose.

You can overcome most of these disadvantages with careful design consideration when you choose the design hierarchy.

Using Synthesis Tools with Hierarchical Designs

By effectively partitioning your designs, you can significantly reduce compile time and improve synthesis results. Here are some recommendations for partitioning your designs:

- "Restrict Shared Resources to the Same Hierarchy Level"
- "Compile Multiple Instances Together"
- "Restrict Related Combinatorial Logic to the Same Hierarchy Level"
- "Separate Speed Critical Paths from Non-Critical Paths"
- "Restrict Combinatorial Logic that Drives a Register to the Same Hierarchy Level"

- "Restrict Module Size"
- "Register All Outputs"
- "Restrict One Clock to Each Module or to Entire Design"

Restrict Shared Resources to the Same Hierarchy Level

Resources that can be shared should be on the same level of hierarchy. If these resources are not on the same level of hierarchy, the synthesis tool cannot determine if these resources should be shared.

Compile Multiple Instances Together

You may want to compile multiple occurrences of the same instance together to reduce the gate count. However, to increase design speed, do not compile a module in a critical path with other instances.

Restrict Related Combinatorial Logic to the Same Hierarchy Level

Keep related combinatorial logic in the same hierarchical level to allow the synthesis tool to optimize an entire critical path in a single operation. Boolean optimization does not operate across hierarchical boundaries. Therefore, if a critical path is partitioned across boundaries, logic optimization is restricted. In addition, constraining modules is difficult if combinatorial logic is not restricted to the same level of hierarchy.

Separate Speed Critical Paths from Non-Critical Paths

To achieve satisfactory synthesis results, locate design modules with different functions at different levels of the hierarchy. Design speed is the first priority of optimization algorithms. To achieve a design that efficiently utilizes device area, remove timing constraints from design modules.

Restrict Combinatorial Logic that Drives a Register to the Same Hierarchy Level

To reduce the number of CLBs used, restrict combinatorial logic that drives a register to the same hierarchical block.

Restrict Module Size

Restrict module size to 100 - 200 CLBs. This range varies based on:

- your computer configuration
- whether the design is worked on by a design team
- the target FPGA routing resources

Although smaller blocks give you more control, you may not always obtain the most efficient design. For the final compilation of your design, you may want to compile fully from the top down. For guidelines, see your synthesis tool documentation.

Register All Outputs

Arrange your design hierarchy so that registers drive the module output in each hierarchical block. Registering outputs makes your design easier to constrain because you only need to constrain the clock period and the ClockToSetup of the previous module. If you have multiple combinatorial blocks at different levels of the hierarchy, you must manually calculate the delay for each module. Also, registering the outputs of your design hierarchy can eliminate any possible problems with logic optimization across hierarchical boundaries.

Restrict One Clock to Each Module or to Entire Design

By restricting one clock to each module, you only need to describe the relationship between the clock at the top level of the design hierarchy and each module clock. By restricting one clock to the entire design, you only need to describe the clock at the top level of the design hierarchy. For more information on optimizing logic across hierarchical boundaries and compiling hierarchical designs, see your synthesis tool documentation.

Chapter 5

Using SmartModels

This chapter describes the special considerations that should be taken into account when simulating designs for Virtex™-II Pro, Virtex-II Pro X, and Virtex-4 FPGA devices. The Virtex-II Pro and Virtex-4 families are platform FPGA devices for designs that are based on IP cores and customized modules. The family incorporates RocketIO™ and PowerPC™ CPU and Ethernet MAC cores in the FPGA architecture. This chapter includes the following sections.

- "Using SmartModels to Simulate Designs"
- "SmartModel Simulation Flow"
- "About SmartModels"
- "Supported Simulators"
- "Installing SmartModels"
- "Setting Up and Running Simulation"

Using SmartModels to Simulate Designs

SmartModels are an encrypted version of the actual HDL code. These models allow you to simulate the actual functionality without having access to the code itself. Simulation of these new features requires the use of Synopsys SmartModels along with the user design. This section describes the SmartModel simulation flow. It assumes that the reader is familiar with the Xilinx® FPGA simulation flow.

Table 5-1: **Architecture Specific SmartModels**

SmartModel	Virtex-II Pro	Virtex-II Pro X	Virtex-4	FPGACore
DCC_FPGACORE	N/A	N/A	N/A	√
EMAC	N/A	N/A	√	N/A
GT	√	N/A	N/A	N/A
GT10	N/A	√	N/A	N/A
GT11	N/A	N/A	√	N/A
PPC405	√	√	N/A	N/A
PPC405_ADV	N/A	N/A	√	N/A

SmartModel Simulation Flow

The HDL simulation flow using Synopsys SmartModels consists of two steps:

1. Instantiate the SmartModel wrapper used for simulation and synthesis. During synthesis, the SmartModels are treated as black box components. This requires that a wrapper be used that describes the modules port.
2. Use the SmartModels along with your design in an HDL simulator that supports the SWIFT interface.

The wrapper files for the SmartModels are automatically referenced when using Architecture Wizard or EDK.

About SmartModels

The Xilinx SmartModels are simulator-independent models that are derived from the actual design and are therefore accurate evaluation models. To simulate these models, you must use a simulator that supports the SWIFT interface.

Synopsys Logic Modeling uses the SWIFT interface to deliver models. SWIFT is a simulator- and platform-independent API developed by Synopsys. SWIFT has been adopted by all major simulator vendors, including Synopsys, Cadence, and Mentor Graphics, as a way of linking simulation models to design tools.

When running a back-annotated simulation, the precompiled SmartModels support:

- gate-level timing
- pin-to-pin timing
- back-annotation timing

Gate-Level Timing

Gate-level timing distributes the delays throughout the design. All internal paths are accurately distributed. Multiple timing versions can be provided for different speed parts.

Pin-to-Pin Timing

Pin-to-pin timing is less accurate, but is faster since only a few top-level delays must be processed.

Back-Annotation Timing

Back-annotation timing allows the model to accurately process the interconnect delays between the model and the rest of the design. It can be used with either gate-level or pin-to-pin timing, or by itself.

Supported Simulators

A simulator with SmartModel capability is required to use the SmartModels. While any HDL simulator that supports the Synopsys SWIFT interface should be able to handle the SmartModel simulation flow, the following HDL simulators are officially supported by Xilinx for SmartModel simulation.

Table 5-2: **Supported Simulators and Operating Systems**

Simulator	Linux	Linux-64	Windows	Solaris	Solaris-64	HP Unix
MTI ModelSim SE (6.0 and newer)	√	N/ A	√	√	√	N/ A
MTI Modelsim PE SWIFT enabled (6.0 and newer) [a]	N/ A	N/ A	√	N/ A	N/ A	N/ A
Cadence NC-Verilog (5.3 and newer)	√	N/ A	√	√	√	N/ A
Cadence NC-VHDL (5.3 and newer)	√	N/ A	√	√	√	N/ A
Synopsys VCS-MX (Verilog only. 7.1.2 and newer)	√	N/ A	N/ A	√	√	N/ A
Synopsys VCS-MXi (Verilog only. 7.1.2 and newer)	√	N/ A	N/ A	√	√	N/ A

a. The SWIFT interface is not enabled by default on ModelSim PE (5.7 or later). Contact MTI to enable this option.

Installing SmartModels

The following software is required to install and run SmartModels:

- the Xilinx implementation tools
- an HDL Simulator that can simulate both VHDL and Verilog, andthe SWIFT interface

SmartModels are installed with the Xilinx implementation tools, but they are not immediately ready for use. There are two ways to use them:

- In "Method One," use the precompiled models. Use this method if your design does not use any other vendors' SmartModels.
- In "Method Two," install the SmartModels with additional SmartModels incorporated in the design. Compile all SmartModels into a common library for the simulator to use.

Method One

The Xilinx ISE™ installer sets the correct environment to work with SmartModels by default. If this should fail, you must make the settings shown below for the SmartModels to function correctly.

- "Method One on Linux"
- "Method One on Windows"
- "Method One on Solaris"

Method One on Linux

To use the SmartModels on Linux, set the following variables:

```
setenv LMC_HOME $XILINX/smartmodel/lin/installed_lin
setenv LMC_CONFIG $LMC_HOME/data/x86_linux.lmc
setenv LD_LIBRARY_PATH $LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
```

Method One on Windows

To use the SmartModels on Windows, set the following variable:

```
LMC_HOME = %XILINX%\smartmodel\nt\installed_nt
```

Method One on Solaris

To use the SmartModels on Solaris, set the following variables:

```
setenv LMC_HOME $XILINX/smartmodel/sol/installed_sol
setenv LMC_CONFIG $LMC_HOME/data/solaris.lmc
setenv LD_LIBRARY_PATH $LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
```

The SmartModels are not extracted by default. The Xilinx ISE installer sets the environment variable `LMC_HOME`, which points to the location to which the smartmodels are extracted. In order to extract the SmartModels, run **`compxlib`** with the appropriate switches. For more information, see "Compiling Xilinx Simulation Libraries (COMPXLIB)" in Chapter 6 of this guide.

Method Two

Use this method only if "Method One" did not work correctly.

- "Method Two on Linux"
- "Method Two on Windows"
- "Method Two on Solaris"

Method Two on Linux

To install SmartModels on Linux:

1. Run the `sl_admin.csh` program from the `$XILINX/smartmodel/lin/image` directory using the following commands:

   ```
   a. $ cd $XILINX/smartmodel/lin/image
   b. $ sl_admin.csh
   ```

2. Select **SmartModels To Install**.
   a. In the **Set Library Directory** dialog box, change the default directory from `image/x86_linux to installed`.
   b. Click **OK**.
   c. If the directory does not exist, the program asks if you want to create it. Click **OK**.
   d. In the **Install From...** dialog box, click **Open** to use the default directory.

   e. In the **Select Models to Install**, click **Add All** to select all models.
   f. Click **Continue**.
   g. In the **Select Platforms for Installation** dialog box:
      - For Platforms, select **Linux on x86**.
      - For EDAV Packages, select **Other**.
   h. Click **Install**.
   i. When **Install complete** appears, and the status line changes to **Ready**, the SmartModels have been installed
3. Continue to perform other operations such as accessing documentation and running checks on your newly installed library (optional).
4. Select **File > Exit**.

To properly use the newly compiled models, set the LMC_HOME variable to the image directory. For example:

```
setenv LMC_HOME $XILINX/smartmodel/lin/installed_lin
```

Method Two on Windows

To install SmartModels on Windows:

1. Run **sl_admin.exe** from the `%XILINX%\smartmodel\nt\image\`*`pcnt`* directory.
2. Select **SmartModels To Install**.
   a. In the **Set Library Directory** dialog box, change the default directory from `image\pcnt` to `installed`.
   b. Click **OK**.
   c. If the directory does not exist, the program asks if you want to create it. Click **OK**.
   d. Click **Install** on the left side of the sl_admin window. This allows you choose the models to install.
   e. In the **Install From...** dialog box, click **Browse**.
   f. Select the `%XILINX%\smartmodel\nt\image` directory. Click **OK** to select that directory.
   g. In the **Select Models to Install** dialog box, click **Add All**.
   h. Click **OK.**
   i. In the **Choose Platform** window:
      - For **Platforms**, select **Wintel**.
      - For **EDAV Packages**, select **Other**.
   j. Click **OK**.
   k. When **Install complete** appears, the SmartModels have been installed.
3. Continue to perform other operations such as accessing documentation and running checks on your newly installed library (optional).
4. Select **File > Exit**.

To properly use the newly compiled models, set the LMC_HOME variable to the image directory. For example:

```
Set LMC_HOME=%XILINX%\smartmodel\nt\installed_nt
```

Method Two on Solaris

To install SmartModels on Solaris:

1. Run **sl_admin.csh** from the `$XILINX/smartmodel/sol/image` directory using the following commands:

   ```
   a. $ cd $XILINX/smartmodel/sol/image
   b. $ sl_admin.csh
   ```

2. Select **SmartModels To Install.**
   a. In the **Set Library Directory dialog box ,** change the default directory from `image/sol` to `installed`.
   b. Click **OK**.
   c. If the directory does not exist, the program asks if you want to create it. Click **OK**.
   d. In the **Install From...** dialog box, click **Open** to use the default directory.
   e. In the **Select Models to Install** dialog box, click **Add All** to select all models.
   f. Click **Continue**.
   g. In the Select Platforms for Installation dialog box:
      - For Platforms, select **Sun-4**.
      - For EDAV Packages, select **Other**.
   h. Click **Install**.
   i. When **Install complete** appears, and the status line changes to **Ready**, the SmartModels have been installed.
   j. Continue to perform other operations such as accessing documentation and running checks on your newly installed library (optional).
   k. Select **File > Exit**.

To properly use the newly compiled models, set the LMC_HOME variable to the image directory. For example:

```
setenv LMC_HOME $XILINX/smartmodel/sol/installed_sol
```

Setting Up and Running Simulation

This section describes how to set up and run simulation on the supported simulators.

- "MTI ModelSim SE and ModelSim PE"
- "Cadence NC-Verilog"
- "Cadence NC-VHDL"
- "Synopsys VCS-MX"
- "Synopsys VCS-MXi"

MTI ModelSim SE and ModelSim PE

This section describes how to set up and run simulation for MTI ModelSim SE and ModelSim PE (6.0 or later) on the following platforms:

- "MTI ModelSim SE and ModelSim PE on Linux"
- "MTI ModelSim SE and ModelSim PE on Windows"
- "MTI ModelSim SE and ModelSim PE on Solaris"

MTI ModelSim SE and ModelSim PE on Linux

This section describes how to set up and run simulation for MTI ModelSim SE and ModelSim PE (6.0 or later) on Linux.

Simulator Setup

ModelSim SE and PE support the SWIFT interface required for use with SmartModel. Some modifications must be made to the default ModelSim setup to use the SWIFT interface. The SWIFT interface is not enabled by default on ModelSim PE (5.7 or later). Contact MTI to enable this option.

Make the following changes in the `modelsim.ini` file located in the `%MODEL_TECH%` directory.

1. After the lines:

   ```
   ; Simulator resolution
   ; Set to fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or
   100.
   ```

   Edit the statement that follows from **`Resolution = ns`** to **`Resolution = ps`**

2. After the lines:

   ```
   ; Specify whether paths in simulator commands should be described
   ; in VHDL or Verilog format. For VHDL, PathSeparator = /
   ; for Verilog, PathSeparator = .
   ```

   Comment out the following statement by adding a ";" at the start of the line.

   ```
   PathSeparator = /
   ```

3. After the line:

   ```
   ; List of dynamically loaded objects for Verilog PLI applications
   ```

   Add the following statement:

   ```
   Veriuser = $MODEL_TECH/libswiftpli.sl
   ```

4. After the line:

   ```
   ; Logic Modeling's SmartModel SWIFT software (Linux)
   ```

   add the following statements:

   ```
   libsm = $MODEL_TECH/libsm.sl
   libswift = $LMC_HOME/lib/x86_linux.lib/libswift.so
   ```

Make these changes in the order in which the commands appear in the `modelsim.ini` file. The simulation may not work if you do not follow the recommended order.

Running Simulation

Once the simulator is set up, run the command line library compiling utility (CompXLib) to compile the SmartModel wrapper files into the UNISIM and SIMPRIM libraries. To see the exact commands for your system, type **`compxlib -help`** at the command line.

MTI ModelSim SE and ModelSim PE on Windows

This section describes how to set up and run simulation for MTI ModelSim SE and ModelSim PE (6.0 or later) on Windows.

Simulator Setup

ModelSim SE and PE support the SWIFT interface required for use with SmartModel. Some modifications must be made to the default ModelSim setup to use the SWIFT interface. The SWIFT interface is not enabled by default on ModelSim PE (5.7 or later). Contact MTI to enable this option.

Make the following changes to the `modelsim.ini` file located in the `%MODEL_TECH%` directory.

1. After the lines:

   ```
   ; Simulator resolution
   ; Set to fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or
   100.
   ```

   Edit the statement that follows from **`Resolution = ns`** to **`Resolution = ps`**

2. After the lines:

   ```
   ; Specify whether paths in simulator commands should be described
   ; in VHDL or Verilog format. For VHDL, PathSeparator = /
   ; for Verilog, PathSeparator = .
   ```

   Comment out the following statement by adding a ";" at the start of the line.

   ```
   PathSeparator = /
   ```

3. After the line:

   ```
   ; List of dynamically loaded objects for Verilog PLI applications
   ```

   Add the following statement:

   ```
   Veriuser = %MODEL_TECH%/libswiftpli.dll
   ```

4. After the line:

   ```
   ; Logic Modeling's SmartModel SWIFT software (Windows NT)
   ```

   add the following statements:

   ```
   libsm = %MODEL_TECH%/libsm.dll
   libswift = %LMC_HOME%/lib/pcnt.lib/libswift.dll
   ```

Make these changes in the order in which the commands appear in the modelsim.ini file. The simulation may not work if you do not follow the recommended order.

Running Simulation

Once the simulator is set up, run the command line library compiling utility (CompXLib) to compile the SmartModel wrapper files into the UNISIM and SIMPRIM libraries. To see the exact commands for your system, type **`compxlib -help`** at the command line.

MTI ModelSim SE and ModelSim PE on Solaris

This section describes how to set up and run simulation for MTI ModelSim SE and ModelSim PE (6.0 or later) on Solaris.

Simulator Setup

ModelSim SE and PE support the SWIFT interface required for use with SmartModel. Some modifications must be made to the default ModelSim setup to use the SWIFT interface. The SWIFT interface is not enabled by default on ModelSim PE (5.7 or later). Contact MTI to enable this option.

Make the following changes in the `modelsim.ini` file located in the `$MODEL_TECH` directory.

1. After the lines:

   ```
   ; Simulator resolution
   ; Set to fs, ps, ns, us, ms, or sec with optional prefix of 1, 10, or
   100.
   ```

   Edit the statement that follows from **`Resolution = ns`** to **`Resolution = ps`**

2. After the lines:

   ```
   ; Specify whether paths in simulator commands should be described
   ; in VHDL or Verilog format. For VHDL, PathSeparator = /
   ; for Verilog, PathSeparator = .
   ```

   Comment out the following statement by adding a ";" at the start of the line.

   ```
   PathSeparator = /
   ```

3. After the line

   ```
   ; List of dynamically loaded objects for Verilog PLI applications
   ```

   add the following statement:

   ```
   Veriuser = $MODEL_TECH/libswiftpli.sl
   ```

4. After the line

   ```
   ;  Logic Modeling's SmartModel SWIFT software (Sun4 Solaris 2.x)
   ```

   add the following statements:

   ```
   libsm = $MODEL_TECH/libsm.sl
   libswift = $LMC_HOME/lib/sun4Solaris.lib/libswift.so
   ```

Make these changes in the order in which the commands appear in the modelsim.ini file. The simulation may not work if you do not follow the recommended order.

Running Simulation

Once the simulator is set up, run the command line library compiling utility (CompXLib) to compile the SmartModel wrapper files into the UNISIM and SIMPRIM libraries. To see the exact commands for your system, type **`compxlib -help`** at the command line.

Cadence NC-Verilog

This section describes how to set up and run simulation for Cadence NC-Verilog on the following platforms:

- "Cadence NC-Verilog on Linux"
- "Cadence NC-Verilog on Windows"
- "Cadence NC-Verilog on Solaris"

Cadence NC-Verilog on Linux

This section describes how to set up and run simulation for Cadence NC-Verilog on Linux.

Running Simulation

Several files in the `$XILINX/smartmodel/lin/simulation/ncverilog` directory can help you set up and run a simulation utilizing the SWIFT interface. You can run the simulation after you have updated each of these files. Following is a description of each file.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
setenv XILINX <Xilinx path>
setenv CDS_INST_DIR <Cadence path>
setenv LM_LICENSE_FILE <license.dat>:$LM_LICENSE_FILE

setenv LMC_HOME $XILINX/smartmodel/lin/installed_lin
setenv LMC_CONFIG $LMC_HOME/data/x86_linux.lmc

setenv LD_LIBRARY_PATH
$CDS_INST_DIR/tools/lib:$LMC_HOME/sim/pli/src:$LMC_HOME/lib/x86_linux.
lib:$LD_LIBRARY_PATH
setenv LMC_CDS_VCONFIG $CDS_INST_DIR/tools/verilog/bin/vconfig

setenv PATH ${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin ${PATH}
setenv PATH ${XILINX}/bin/lin ${PATH}
```

Change the parameters in the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample NC-Verilog compilation simulation script. It shows the files that must be compiled and loaded for simulation. To modify this file to simulate a design, include the appropriate design and test bench files. For example:

```
ncverilog  \
<design>.v <testbench>.v  \
${XILINX}/verilog/src/glbl.v  \
-y ${XILINX}/verilog/src/unisims +libext+.v \
-y ${XILINX}/verilog/src/simprims +libext+.v \
-y ${XILINX}/smartmodel/lin/wrappers/ncverilog +libext+.v \
+loadpli1=swiftpli:swift_boot +incdir+$LMC_HOME/sim/pli/src \
+access+r+w
```

Change the parameters in the angle brackets (< >) to match your design and test bench files.

Cadence NC-Verilog on Windows

This section describes how to set up and run simulation for Cadence NC-Verilog on Windows.

Running Simulation

Several files in the `%XILINX%\smartmodel\nt\simulation\ncverilog` directory can help you set up and run a simulation utilizing the SWIFT interface. You can run the simulation after you have updated each of these files. Following is a description of each file.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
set XILINX =  <Xilinx path>
set CDS_INST_DIR = <Cadence path>
set LM_LICENSE_FILE = <license.dat>;%LM_LICENSE_FILE%

set LMC_HOME = %XILINX%\smartmodel\nt\installed_nt
set LMC_CONFIG = %LMC_HOME%\data\pcnt.lmc

set PATH =
%LMC_HOME%\bin;%CDS_INST_DIR%\tools\lib;%LMC_HOME%\sim\pli\src;%LMC_HO
ME%\lib\pcnt.lib;%PATH%
```

Change the parameters in the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample NC-Verilog compilation simulation script. It shows the files that must be compiled and loaded for simulation. To modify this file to simulate a design, include the appropriate design and test bench files. For example:

```
ncverilog \
  <design.v> <testbench.v> \
  %XILINX%\verilog\src\glbl.v\
  -y %XILINX%\verilog\src\unisims +libext+.v\
  -y %XILINX%\verilog\src\simprims +libext+.v\
  -y %XILINX%\smartmodel\nt\wrappers\ncverilog +libext+.v \
  +access+rw\
  +loadpli1=swiftpli:swift_boot +incdir+%LMC_HOME%\sim\pli\src
```

Change the parameters in the angle brackets (< >) to match your design and test bench files.

These environment variables and settings can also be set globally. Change global variables and settings in the System Environment Variables, not in the User Environment Variables.

Cadence NC-Verilog on Solaris

This section describes how to set up and run simulation for Cadence NC-Verilog on Solaris.

Running Simulation

Several files in the `$XILINX/smartmodel/sol/simulation/ncverilog` directory can help you set up and run a simulation utilizing the SWIFT interface. You can run the simulation after you have updated each of these files. Following is a description of each file.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
setenv XILINX <Xilinx path>
setenv CDS_INST_DIR <Cadence path>
setenv LM_LICENSE_FILE <license.dat>:$LM_LICENSE_FILE

setenv LMC_HOME $XILINX/smartmodel/sol/installed_sol
setenv LMC_CONFIG $LMC_HOME/data/solaris.lmc

setenv LD_LIBRARY_PATH
$LMC_HOME/sim/pli/src:$LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
setenv LMC_CDS_VCONFIG $CDS_INST_DIR/tools/verilog/bin/vconfig

setenv PATH ${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin ${PATH}
setenv PATH ${XILINX}/bin/sol ${PATH}
```

Change the parameters in the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample NC-Verilog compilation simulation script. It shows the files that must be compiled and loaded for simulation. To modify this file to simulate a design, include the appropriate design and test bench files. For example:

```
ncverilog  \
<design>.v <testbench>.v  \
${XILINX}/verilog/src/glbl.v  \
-y ${XILINX}/verilog/src/unisims +libext+.v \
-y ${XILINX}/verilog/src/simprims +libext+.v \
-y ${XILINX}/smartmodel/sol/wrappers/ncverilog +libext+.v \
+loadpli1=swiftpli:swift_boot +incdir+$LMC_HOME/sim/pli/src \
+access+r+w
```

Change the parameters in the angle brackets (< >) to match your design and test bench files.

Cadence NC-VHDL

This section describes how to set up and run simulation for Cadence NC-VHDL on the following platforms:

- "Cadence NC-VHDL on Linux"
- "Cadence NC-VHDL on Windows"
- "Cadence NC-VHDL on Solaris"

Cadence NC-VHDL on Linux

This section describes how to set up and run simulation for Cadence NC-VHDL on Linux.

Running Simulation

Several files in the `$XILINX/smartmodel/lin/simulation/ncvhdl` directory can help you set up and run a simulation utilizing the SWIFT interface. You can run the simulation after you have updated each of these files. Following is a description of each file.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
setenv XILINX <Xilinx_path>
setenv CDS_INST_DIR <Cadence_path>
setenv LM_LICENSE_FILE <license.dat>:$LM_LICENSE_FILE

setenv LMC_HOME $XILINX/smartmodel/lin/installed_lin
setenv LMC_CONFIG $LMC_HOME/data/x86_linux.lmc

setenv LD_LIBRARY_PATH
$CDS_INST_DIR/tools/lib:$LMC_HOME/sim/pli/src:$LMC_HOME/lib/x86_linux.
lib:$LD_LIBRARY_PATH
setenv LMC_TIMEUNIT -12

setenv PATH ${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin ${PATH}
setenv PATH ${XILINX}/bin/lin ${PATH}
```

Change the parameters within the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample NC-VHDL compilation simulation script. It shows the files that must be compiled and loaded for simulation. This file can be modified to simulate a design by including the design and test bench files appropriately. For example:

```
ncvhdl -v93 <testbench>.vhd <design>.vhd
ncelab -work worklib -cdslib cds.lib -access +wc
worklib.<testbench>:<view>
ncsim +access+rw -gui -cdslib cds.lib worklib.<testbench>:<view>
```

Change the parameters within the angle brackets (< >) to match your system configuration.

Cadence NC-VHDL on Windows

This section describes how to set up and run simulation for Cadence NC-VHDL on Windows.

Running Simulation

Several files in the `%XILINX%\smartmodel\nt\simulation\ncvhdl` directory can help you set up and run a simulation utilizing the SWIFT interface. You can run the simulation after you have updated each of these files. Following is a description of each file.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
set XILINX =  <Xilinx path>
set CDS_INST_DIR = <Cadence path>
set LM_LICENSE_FILE = <license.dat>;%LM_LICENSE_FILE%

set LMC_HOME = %XILINX%\smartmodel\lin\installed_nt
set LMC_CONFIG = %LMC_HOME%\data\pcnt.lmc
set PATH =
%LMC_HOME%\bin;%CDS_INST_DIR%\tools\lib;%LMC_HOME%\sim\pli\src;%LMC_HO
ME\lib\pcnt.lib;%PATH%
```

Change the parameters within the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample NC-VHDL compilation simulation script. It shows the files that must be compiled and loaded for simulation. To modify this file to simulate a design, include the appropriate design and test bench files. For example:

```
ncvhdl -v93  <testbench>.vhd <design>.vhd
ncelab -work worklib -cdslib cds.lib -access +wc
worklib.<testbench>:<view>
ncsim +access+rw -gui -cdslib cds.lib worklib.<testbench>:<view>
```

Change the parameters within the angle brackets (< >) to match your system configuration.

Cadence NC-VHDL on Solaris

This section describes how to set up and run simulation for Cadence NC-VHDL on Solaris.

Running Simulation

Several files in the `$XILINX/smartmodel/sol/simulation/ncvhdl` directory can help you set up and run a simulation utilizing the SWIFT interface. The following is a description of each file.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
setenv XILINX <Xilinx_path>
setenv CDS_INST_DIR <Cadence_path>
setenv LM_LICENSE_FILE <license.dat>:$LM_LICENSE_FILE

setenv LMC_HOME $XILINX/smartmodel/sol/installed_sol
setenv LMC_CONFIG $LMC_HOME/data/solaris.lmc

setenv LD_LIBRARY_PATH
$CDS_INST_DIR/tools/lib:$LMC_HOME/sim/pli/src:$LMC_HOME/lib/sun4Solari
s.lib:$LD_LIBRARY_PATH
setenv LMC_TIMEUNIT -12

setenv PATH ${LMC_HOME}/bin ${CDS_INST_DIR}/tools/bin ${PATH}
setenv PATH ${XILINX}/bin/sol ${PATH}
```

Change the parameters within the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample NC-VHDL compilation simulation script. It shows the files that must be compiled and loaded for simulation. This file can be modified to simulate a design by including the design and test bench files appropriately. For example:

```
ncvhdl -v93 <testbench>.vhd <design>.vhd
ncelab -work worklib -cdslib cds.lib -access +wc
worklib.<testbench>:<view>
ncsim +access+rw -gui -cdslib cds.lib worklib.<testbench>:<view>
```

Change the parameters within the angle brackets (< >) to match your system configuration.

Synopsys VCS-MX

This section describes how to set up and run simulation for Synopsys VCS-MX (Verilog Only) on the following platforms:

- "Synopsys VCS-MX on Linux"
- "Synopsys VCS-MX on Solaris"

Synopsys VCS-MX on Linux

This section describes how to set up and run simulation for Synopsys VCS-MX (Verilog Only) on Linux.

Running Simulation

Several files in the `$XILINX/smartmodel/lin/simulation/vcsmxverilog` directory can help you set up and run a simulation utilizing the SWIFT interface. You can run the simulation after you have updated each of these files. Following is a description of each file.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
setenv XILINX <Xilinx path>
setenv VCS_HOME <VCS path>
setenv LM_LICENSE_FILE <license.dat>:${LM_LICENSE_FILE}
setenv VCS_SWIFT_NOTES 1

setenv LMC_HOME $XILINX/smartmodel/lin/installed_lin
setenv LMC_CONFIG $LMC_HOME/data/x86_linux.lmc

setenv VCS_CC gcc

setenv LD_LIBRARY_PATH
$LMC_HOME/sim/pli/src:$LMC_HOME/lib/x86_linux.lib:$LD_LIBRARY_PATH
setenv PATH ${LMC_HOME}/bin ${VCS_HOME}/bin ${PATH}
setenv PATH ${XILINX}/bin/lin ${PATH}
```

Change the parameters in the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample VCS compilation simulation script. It shows the files that must be compiled and loaded for simulation. To modify this file to simulate a design, include the appropriate design and test bench files. For example:

```
vcs  -lmc-swift \
<design>.v <testbench>.v  \
${XILINX}/verilog/src/glbl.v  \
-y ${XILINX}/verilog/src/unisims +libext+.v \
-y ${XILINX}/verilog/src/simprims +libext+.v \
-y ${XILINX}/smartmodel/lin/wrappers/vcsmxverilog +libext+.v \
sim -l vcs.log
```

Change the parameters in the angle brackets (< >) to match your design and test bench files.

Synopsys VCS-MX on Solaris

This section describes how to set up and run simulation for Synopsys VCS-MX (Verilog Only) on Solaris.

Running Simulation

Several files in the `$XILINX/smartmodel/sol/simulation/vcsmxverilog` directory can help you set up and run a simulation utilizing the SWIFT interface. You can run the simulation after you have updated each of these files. Following is a description of each file.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
setenv XILINX <Xilinx path>
setenv VCS_HOME <VCS path>
setenv LM_LICENSE_FILE <license.dat>:${LM_LICENSE_FILE}
setenv VCS_SWIFT_NOTES 1

setenv LMC_HOME $XILINX/smartmodel/sol/installed_sol
setenv LMC_CONFIG ${LMC_HOME}/data/solaris.lmc

setenv VCS_CC gcc

setenv LD_LIBRARY_PATH
$LMC_HOME/sim/pli/src:$LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
setenv PATH ${LMC_HOME}/bin ${VCS_HOME}/bin ${PATH}
setenv PATH ${XILINX}/bin/sol ${PATH}
```

Change the parameters in the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample VCS compilation simulation script. It shows the files that must be compiled and loaded for simulation. To modify this file to simulate a design, include the appropriate design and test bench files. For example:

```
vcs  -lmc-swift \
<design>.v <testbench>.v  \
${XILINX}/verilog/src/glbl.v  \
-y ${XILINX}/verilog/src/unisims +libext+.v \
-y ${XILINX}/verilog/src/simprims +libext+.v \
-y ${XILINX}/smartmodel/sol/wrappers/vcsmxverilog +libext+.v \
sim -l vcs.log
```

Change the parameters in the angle brackets (< >) to match your design and test bench files.

Synopsys VCS-MXi

This section describes how to set up and run simulation for Synopsys VCS-MXi (Verilog Only) on the following platforms:

- "Synopsys VCS-MXi on Linux"
- "Synopsys VCS-MXi on Solaris"

Synopsys VCS-MXi on Linux

This section describes how to set up and run simulation for Synopsys VCS-MXi (Verilog Only) on Linux.

Running Simulation

In the `$XILINX/smartmodel/lin/simulation/`vcsmxiverilog directory there are several files to help set up and run a simulation utilizing the SWIFT interface.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
setenv XILINX <Xilinx_path>
setenv VCS_HOME <VCS-MXI_path>
setenv LM_LICENSE_FILE <license.dat>:${LM_LICENSE_FILE}

setenv LMC_HOME $XILINX/smartmodel/lin/installed_lin
setenv LMC_CONFIG $LMC_HOME/data/x86_linux.lmc

setenv VCS_CC gcc

setenv PATH ${LMC_HOME}/bin ${VCSI_HOME}/bin ${PATH}
setenv PATH ${XILINX}/bin/lin ${PATH}
```

Change the parameters within the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample VCSi compilation simulation script. It shows the files that must be compiled and loaded for simulation. To modify this file to simulate a design, include the appropriate design and test bench files. For example:

```
rm simv\
vcsi -lmc-swift \
<design>.v <testbench>.v \
${XILINX}/verilog/src/glbl.v \
${XILINX}/smartmodel/lin/wrappers/vcsmxiverilog/GT_SWIFT.v \
${XILINX}/smartmodel/lin/wrappers/vcsmxiverilog/GT_SWIFT_BIT.v \
${XILINX}/smartmodel/lin/wrappers/vcsmxiverilog/GT10_SWIFT.v \
${XILINX}/smartmodel/lin/wrappers/vcsmxiverilog/GT10_SWIFT_BIT.v \
${XILINX}/smartmodel/lin/wrappers/vcsmxiverilog/PPC405_SWIFT.v \
${XILINX}/smartmodel/lin/wrappers/vcsmxiverilog/PPC405_SWIFT_BIT.v \
-y ${XILINX}/verilog/src/unisims +libext+.v \
-y ${XILINX}/verilog/src/simprims +libext+.v \
sim -l vcs.log
```

Change the parameters within the angle brackets (< >) to match your system configuration.

Synopsys VCS-MXi on Solaris

This section describes how to set up and run simulation for Synopsys VCS-MXi (Verilog Only) on Solaris.

Running Simulation

Several files in the `$XILINX/smartmodel/sol/simulation/vcsmxiverilog` directory can help you set up and run a simulation utilizing the SWIFT interface. You can run the simulation after you have updated each of these files. Following is a description of each file.

Setup File

The setup file describes the variables that must be set for correct simulation. For example:

```
setenv XILINX <Xilinx path>
setenv VCS_HOME <VCS-MXi path>
setenv LM_LICENSE_FILE <license.dat>:${LM_LICENSE_FILE}
setenv VCS_SWIFT_NOTES 1

setenv LMC_HOME $XILINX/smartmodel/sol/installed_sol
setenv LMC_CONFIG ${LMC_HOME}/data/solaris.lmc

setenv VCS_CC gcc

setenv LD_LIBRARY_PATH
$LMC_HOME/sim/pli/src:$LMC_HOME/lib/sun4Solaris.lib:$LD_LIBRARY_PATH
setenv PATH ${LMC_HOME}/bin ${VCS_HOME}/bin ${PATH}
setenv PATH ${XILINX}/bin/sol ${PATH}
```

Change the parameters in the angle brackets (< >) to match your system configuration.

Simulate File

The simulate file is a sample VCSi compilation simulation script. It shows the files that must be compiled and loaded for simulation. To modify this file to simulate a design, include the appropriate design and test bench files. For example:

```
vcsi  -lmc-swift \
<design>.v <testbench>.v  \
${XILINX}/verilog/src/glbl.v  \
-y ${XILINX}/verilog/src/unisims +libext+.v \
-y ${XILINX}/verilog/src/simprims +libext+.v \
-y ${XILINX}/smartmodel/sol/wrappers/vcsmxverilog +libext+.v \
sim -l vcsi.log
```

Change the parameters in the angle brackets (< >) to match your design and test bench files.

Chapter 6

Simulating Your Design

This chapter describes the basic HDL simulation flow using Xilinx® and third party software. This chapter includes the following sections.

Introduction

Increasing design size and complexity, as well as improvements in design synthesis and simulation tools, have made HDL the preferred design language of most integrated circuit designers. The two leading HDL synthesis and simulation languages today are Verilog and VHDL. Both of these languages have been adopted as IEEE standards.

The Xilinx software is designed to be used with several HDL synthesis and simulation tools that provide a solution for programmable logic designs from beginning to end. The Xilinx software provides libraries, netlist readers, and netlist writers, along with powerful place and route software that integrates with your HDL design environment on PC, Linux, and UNIX workstation platforms.

Adhering to Industry Standards

Xilinx adheres to relevant industry standards.

Standards Supported by Xilinx Simulation Flow

The standards in the following table are supported by the Xilinx simulation flow.

Table 6-1: **Standards Supported by Xilinx Simulation Flow**

Description	Version
VHDL Language	IEEE-STD-1076-1993
VITAL Modeling Standard	IEEE-STD-1076.4-2000
Verilog Language	IEEE-STD-1364-2001
Standard Delay Format (SDF)	OVI 3.0
Std_logic Data Type	IEEE-STD-1164-93

Although the Xilinx HDL netlisters produce IEEE-STD-1076-93 VHDL code or IEEE-STD-1364-2001 Verilog code, that does not restrict the use of newer or older standards for the creation of test benches or other simulation files. If the simulator being used supports both older and newer standards, then generally, both standards can be used in these simulation files. Be sure to indicate to the simulator during code compilation which standard was used for the creation of the file.

Xilinx Supported Simulators

Xilinx currently tests and supports the following simulators for VHDL and Verilog simulation.

Table 6-2: **Xilinx Supported Simulators**

VHDL	Verilog
Xilinx ISE™ Simulator	Xilinx ISE Simulator
Model Technology ModelSim	Model Technology ModelSim
Cadence NC-SIM (NC-VHDL)	Cadence NC-SIM (NC-Verilog)
Synopsys VCS-MX	Synopsys VCS-MX

In general, you should run the most current version of the simulator available to you.

Xilinx develops its libraries and simulation netlists using IEEE standards, so you should be able to use most current VHDL and Verilog simulators. Check with your simulator vendor to confirm that the standards are supported by your simulator, and to verify the settings for your simulator.

Xilinx Libraries

The Xilinx VHDL libraries are tied to the IEEE-STD-1076.4-2000 VITAL standard for simulation acceleration. VITAL 2000 is in turn based on the IEEE-STD-1076-93 VHDL language. Because of this, the Xilinx libraries must be compiled as 1076-93.

VITAL libraries include some additional processing for timing checks and back-annotation styles. The UNISIM library turns these timing checks off for unit delay functional simulation. The SIMPRIM back-annotation library keeps these checks on by default to allow accurate timing simulations.

Simulation Points

Xilinx supports functional and timing simulation of HDL designs at five points in the HDL design flow:

- "Primary Simulation Points for HDL Designs"
- "Register Transfer Level (RTL)"
- "Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation"
- "Post-NGDBuild (Pre-Map) Gate-Level Simulation"
- "Post-Map Partial Timing (Block Delays)"
- "Timing Simulation Post-Place and Route"

Primary Simulation Points for HDL Designs

Figure 6-1 shows the points of the design flow.

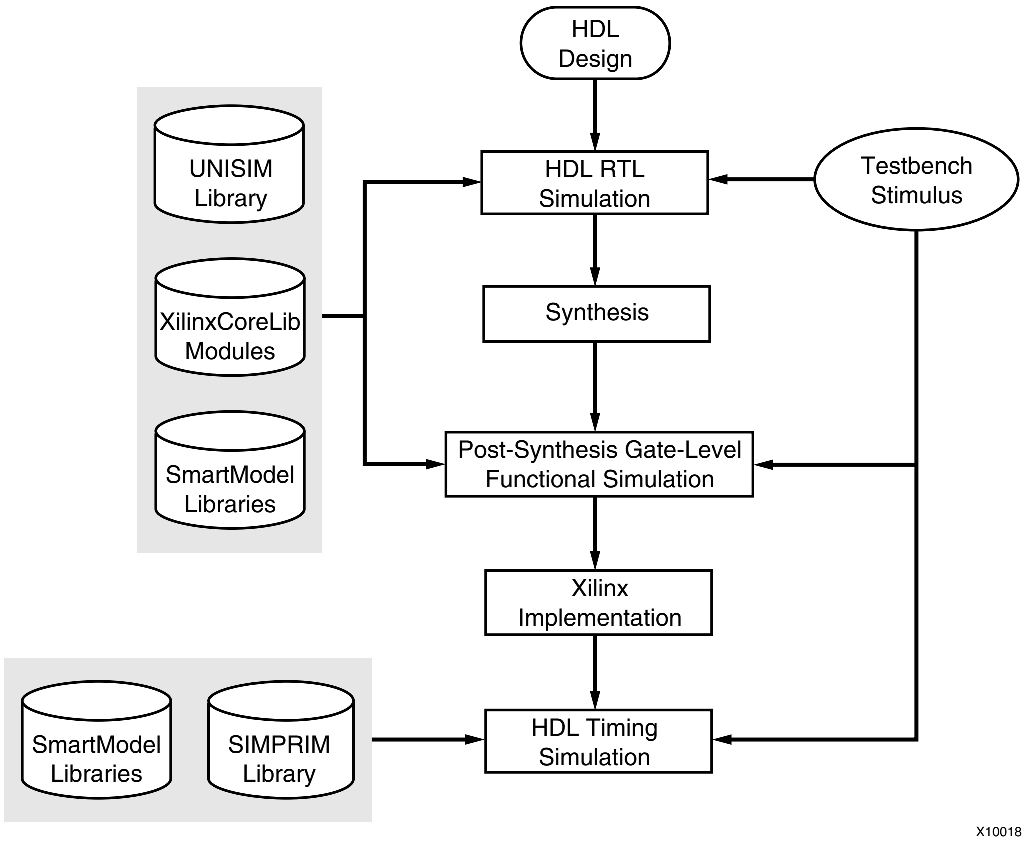

Figure 6-1: **Primary Simulation Points for HDL Designs**

The Post-NGDBuild and Post-Map simulations can be used when debugging synthesis or map optimization issues

Table 6-3: **Five Simulation Points in HDL Design Flow**

Simulation		**UNISIM**	**XilinxCoreLib Models**	**SmartModel**	**SIMPRIM**	**SDF**
1.	RTL	X	X	X		
2.	Post-Synthesis (optional)	X	X	X		
3.	Functional Post-NGDBuild (optional)			X	X	
4.	Functional Post-Map (optional)			X	X	X
5.	Post-Route Timing			X	X	X

These Xilinx simulation points are described in detail in the following sections. The libraries required to support the simulation flows are described in detail in "VHDL and Verilog Libraries and Models" in this chapter. The flows and libraries support functional equivalence of initialization behavior between functional and timing simulations.

Different simulation libraries are used to support simulation before and after running NGDBuild. Prior to NGDBuild, your design is expressed as a UNISIM netlist containing Unified Library components that represents the logical view of the design. After NGDBuild, your design is a netlist containing SIMPRIMs that represents the physical view of the design.

Although these library changes are fairly transparent, there are two important considerations to keep in mind:

- You must specify different simulation libraries for pre- and post-implementation simulation.
- There are different gate-level cells in pre- and post-implementation netlists.

For Verilog, within the simulation netlist there is the Verilog system task `$sdf_annotate`, which specifies the name of the SDF file to be read. If the simulator supports the `$sdf_annotate` system task, the Standard Delay Format (SDF) file is automatically read when the simulator compiles the Verilog simulation netlist. If the simulator does not support `$sdf_annotate`, in order to get timing values applied to the gate-level netlist, you must manually specify to the simulator to annotate the SDF file.

For VHDL, you must specify:

- the location of the SDF file
- which instance to annotate during the timing simulation

The method for doing this depends on the simulator being used. Typically, a command line or program switch is used to read the SDF file. For more information on annotating SDF files, see your simulation tool documentation.

Register Transfer Level (RTL)

Register Transfer Level (RTL) may include the following:

- RTL Code
- Instantiated UNISIM library components
- XilinxCoreLib and UNISIM gate-level models (CORE Generator™)
- SmartModels

The RTL-level (behavioral) simulation enables you to verify or simulate a description at the system or chip level. This first pass simulation is typically performed to verify code syntax, and to confirm that the code is functioning as intended. At this step, no timing information is provided, and simulation should be performed in unit-delay mode to avoid the possibility of a race condition.

RTL simulation is not architecture-specific unless the design contains instantiated UNISIM or CORE Generator components. To support these instantiations, Xilinx provides the UNISIM and XilinxCoreLib libraries. You can instantiate CORE Generator components if you do not want to rely on the module generation capabilities of the synthesis tool, or if the design requires larger memory structures.

Keep the code behavioral for the initial design creation. Do not instantiate specific components unless necessary. This allows for:

- more readable code
- faster and simpler simulation
- code portability (the ability to migrate to different device families)
- code reuse (the ability to use the same code in future designs)

However, you may find it necessary to instantiate components if the component is not inferable (for example, DCM, GT, and PPC405), or in order to control the mapping, placement, or structure of a function.

Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation

Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation may include one of the following (optional):

- Gate-level netlist containing UNISIM library components
- XilinxCoreLib and UNISIM gate-level models (CORE Generator)
- SmartModels

Most synthesis tools can write out a post-synthesis HDL netlist for a design. If the VHDL or Verilog netlists are written for UNISIM library components, you may use the netlists to simulate the design and evaluate the synthesis results. However, Xilinx does not support this method if the netlists are written in terms of the vendor's own simulation models.

The instantiated CORE Generator models are used for any post-synthesis simulation because these modules are processed as a "black box" during synthesis. It is important that you maintain the consistency of the initialization behavior with the behavioral model used for RTL, post-synthesis simulation, and the structural model used after implementation. In addition, the initialization behavior must work with the method used for synthesized logic and cores.

Post-NGDBuild (Pre-Map) Gate-Level Simulation

Post-NGDBuild (Pre-Map) Gate-Level Simulation (optional) may include the following:

- Gate-level netlist containing SIMPRIM library components
- SmartModels

The post-NGDBuild (pre-map) gate-level functional simulation is used when it is not possible to simulate the direct output of the synthesis tool. This occurs when the tool cannot write UNISIM-compatible VHDL or Verilog netlists. In this case, the NGD file produced from NGDBUILD is the input into the Xilinx simulation netlister, NetGen. NetGen creates a structural simulation netlist based on SIMPRIM models.

Like post-synthesis simulation, post-NGDBuild simulation allows you to verify that your design has been synthesized correctly, and you can begin to identify any differences due to the lower level of abstraction. Unlike the post-synthesis pre-NGDBuild simulation, there are GSR and GTS nets that must be initialized, just as for post-Map and post-PAR simulation. For more information on using the GSR and GTS signals for post-NGDBuild simulation, see "Understanding the Global Reset and 3-state for Simulation" in this chapter.

Post-Map Partial Timing (Block Delays)

Post-Map Partial Timing (Block Delays) may include the following (optional):

- Gate-level netlist containing SIMPRIM library components
- Standard Delay Format (SDF) files
- SmartModels

You may also perform simulation after mapping the design. Post-Map simulation occurs before placing and routing. This simulation includes the block delays for the design, but not the routing delays. This is generally a good metric to test whether the design is meeting the timing requirements before additional time is spent running the design through a complete place and route.

As with the post-NGDBuild simulation, NetGen is used to create the structural simulation Running the simulation netlister tool, NetGen, creates an SDF file. The delays for the design are stored in the SDF file which contains all block or logic delays. However, it does not contain any of the routing delays for the design since the design has not yet been placed and routed. As with all NetGen created netlists, GSR and GTS signals must be accounted for. For more information on using the GSR and GTS signals for post-NGDBuild simulation, see "Understanding the Global Reset and 3-state for Simulation" in this chapter.

Timing Simulation Post-Place and Route

Timing Simulation Post-Place and Route Full Timing (Block and Net Delays) may include the following:

- Gate-level netlist containing SIMPRIM library components
- Standard Delay Format (SDF) files
- SmartModels

After your design has completed the place and route process in the Xilinx Implementation Tools, a timing simulation netlist can be created. It is not until this stage of design implementation that you start to see how your design behaves in the actual circuit. The overall functionality of the design was defined in the beginning stages, but it is not until the design has been placed and routed that all of the timing information of the design can be accurately calculated.

The previous simulations that used NetGen created a structural netlist based on SIMPRIM models. However, this netlist comes from the placed and routed NCD file. This netlist has GSR and GTS nets that must be initialized. For more information on initializing the GSR and GRTS nets, see "Understanding the Global Reset and 3-state for Simulation" in this chapter.

When you run timing simulation, an SDF file is created as with the post-Map simulation. However, this SDF file contains all block and routing delays for the design.

Providing Stimulus

Before you perform simulation, create a test bench or test fixture to apply the stimulus to the design.

Test Benches

A test bench is HDL code written for the simulator that:

- instantiates the design netlists
- initializes the design
- applies stimuli to verify the functionality of the design

You can also set up the test bench to display the desired simulation output to a file, waveform or screen.

A test bench can be very simple in structure and sequentially apply stimulus to specific inputs. A test bench can also be very complex, and include:

- subroutine calls
- stimulus read in from external files
- conditional stimulus
- other more complex structures

The test bench has several advantages over interactive simulation methods:

- It allows repeatable simulation throughout the design process.
- It provides documentation of the test conditions.

Creating a Test Bench

Use any of the following to create a test bench and simulate a design:

- "Creating a Test Bench in ISE Tools"
- "Creating a Test Bench in Waveform Editor"
- "Creating a Test Bench in NetGen"

Creating a Test Bench in ISE Tools

The ISE tools create a template test bench containing the proper structure, library references, and design instantiation based on your design files from Project Navigator. This greatly eases test bench development at the beginning stages of the design.

Creating a Test Bench in Waveform Editor

You may use Waveform Editor to automatically create a test bench by drawing the intended stimulus and the expected outputs in a waveform viewer. For more information, see the ISE help and the ISE Simulator help.

Creating a Test Bench in NetGen

You can use NetGen to create a test bench file. The **`-tb`** switch for NetGen creates a test fixture or test bench template. The Verilog test fixture file has a `.tv` extension, and the VHDL test bench file has a `.tvhd` extension.

Test Bench Recommendations

Xilinx recommends the following when you create and run a test bench.

- Give the name *`testbench`* to the main module or entity name in the test bench file.
- Specify the instance name for the instantiated top-level of the design in the test bench as *UUT*.

  These names are consistent with the default names used by ISE for calling the test bench and annotating the SDF file when invoking the simulator.
- Initialize all inputs to the design within the test bench at simulation time zero in order to properly start simulation with known values.
- Apply stimulus data *after* 100 ns in order to account for the default Global Set/Reset pulse used in SIMPRIM-based simulation. However, the clock source should begin before the GSR is released. For more information on GSR, see "Understanding the Global Reset and 3-state for Simulation" in this chapter.

VHDL and Verilog Libraries and Models

The five simulation points require the following libraries:

- UNISIM
- CORE Generator (XilinxCoreLib)
- SmartModel
- SIMPRIM

Required Libraries

This section shows the libraries required for each of the five simulation points.

First Simulation Point

The first point, "Register Transfer Level (RTL)", is a behavioral description of your design at the register transfer level. RTL simulation is not architecture-specific unless your design contains instantiated UNISIM, or CORE Generator components.

To support these instantiations, Xilinx provides a functional UNISIM library, a CORE Generator Behavioral XilinxCoreLib library, and a SmartModelLibrary. You can also instantiate CORE Generator components if you do not want to rely on the module generation capabilities of your synthesis tool, or if your design requires larger memory structures.

Second Simulation Point

The second simulation point is "Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation". If the UNISIM library and CORE Generator components are used, then the UNISIM, the XilinxCoreLib and SmartModel Libraries must all be used.

The synthesis tool must write out the HDL netlist using UNISIM primitives. Otherwise, the synthesis vendor provides its own post-synthesis simulation library, which is not supported by Xilinx.

Third, Fourth, and Fifth Simulation Points

The third, fourth, and fifth points simulation points are:

- "Post-NGDBuild (Pre-Map) Gate-Level Simulation"
- "Post-Map Partial Timing (Block Delays)"
- "Timing Simulation Post-Place and Route"

These simulation points use the SIMPRIM and SmartModel Libraries.

Simulation Phase Library Information

The following table shows the library required for each of the five simulation points.

Table 6-4: **Simulation Phase Library Information**

Simulation Point	Compilation Order of Library Required
"Register Transfer Level (RTL)",	UNISIM XilinxCoreLib SmartModel
"Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation"	UNISIM XilinxCoreLib SmartModel
"Post-NGDBuild (Pre-Map) Gate-Level Simulation"	SIMPRIM SmartModel
"Post-Map Partial Timing (Block Delays)"	SIMPRIM SmartModel
"Timing Simulation Post-Place and Route"	SIMPRIM SmartModel

Locating Library Source Files

The following table shows:

- the location of the simulation library source files
- the order for a typical compilation

Table 6-5: **Simulation Library Source Files**

Library	**Location of Source Files**		**Compile Order**	
	Verilog	**VITAL VHDL**	**Verilog**	**VITAL VHDL**

UNISIM
Spartan-II, Spartan-IIE, Spartan-3, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, Xilinx IBM FPGA Core
	`$XILINX/verilog /src/unisims`
	`- Unix/Linux`
	
	`%XILINX%\verilo g \src\unisims - Windows`
		`$XILINX/vhdl/ src/unisims - Unix/Linux`
		
		`%XILINX%\vhdl\`
		`src\unisims - Windows`
			No special compilation order required for Verilog libraries	Required;
				typical compilation order:
				`unisim_VCOMP.vhd`
				`unisim_VPKG.vhd`
				`unisim_VITAL.vhd`

UNISIM
9500, CoolRunner, CoolRunner-II
	`$XILINX/verilog /src/uni9000- Unix/Linux`
	
	`%XILINX%\verilo g\src\uni9000- Windows`
		`$XILINX/vhdl/`
		`src/unisims -`
		`Unix/Linux`
		
		`%XILINX%\vhdl\`
		`src\unisims -`
		`Windows`
			No special compilation order required for Verilog libraries	Required;
				typical compilation order:
				`unisim_VCOMP.vhd`
				`unisim_VPKG.vhd`
				`unisim_VITAL.vhd`

XilinxCoreLib
FPGA Families only
	`$XILINX/verilog /src/XilinxCore -Lib-Unix/Linux`
	
	`%XILINX%\verilo g\src\XilinxCor eLib-Windows`
		`$XILINX/vhdl/ src/XilinxCore- Lib-Unix/Linux`
		
		`%XILINX%\vhdl\`
		`src\XilinxCoreL ib-Windows`
			No special compilation order required for Verilog libraries	Compilation order required;
				See the vhdl_analyze_order file located in `$XILINX/vhdl/src/ XilinxCoreLib/ -` Unix/Linux
				
				%XILINX%\vhdl\src\
				XilinxCoreLib\-Windows
				for the required compile order

Table 6-5: **Simulation Library Source Files**

Library	Location of Source Files		Compile Order	
	Verilog	VITAL VHDL	Verilog	VITAL VHDL

SmartModel
Virtex-II Pro
Virtex-II Pro X
Virtex-4
	`$XILINX/smartmodel/`*`<platform>/`*`wrappers/`*`<simulator>-Unix/Linux`*
	
	`%XILINX%\`*`smartmodel\<platform>\wrappers\<simulator>-Windows`*
		`$XILINX/smartmodel/<platform>/wrappers/<simulator>-Unix/Linux`
		
		`%XILINX%\smartmodel\<platform>\wrappers\<simulator>-Windows`
			No special compilation order required for Verilog libraries	Required.
				Typical compilation order for Functional Simulation:
				`unisim_VCOMP.vhd`
				`smartmodel_wrappers.vhd`
				`unisim_SMODEL.vhd`
				Typical compilation order for Timing Simulation:
				`simprim_Vcomponents.vhd`
				`(Simprim_Vcomponents_mti.vhd) - for MTI only`
				`smartmodel_wrappers.vhd`
				`simprim_SMODEL.vhd`
				`(simprim_SMODEL_mti.vhd) - for MTI only`

SIMPRIM
(All Xilinx Technologies)
	`$XILINX/verilog/src/simprims-Unix/Linux`
	
	`%XILINX%\verilog\src\simprims-Windows`
		`$XILINX/vhdl/src/simprims-Unix/Linux`
		
		`%XILINX%\vhdl\src\simprims-Windows`
			No special compilation order required for Verilog libraries	Required;
				typical compilation order:
				`simprim_Vcomponents.vhd`
				`(simprim_Vcomponents_mti.vhd)`[a]
				`(simprim_Vpackage_mti.vhd)`[b]
				`simprim_Vpackage.vhd`
				`simprim_VITAL.vhd`
				`(simprim_VITAL_mti.vhd)`[c]

a. `for MTI only`
b. `for MTI only`
c. `for MTI only`

Using the Libraries

This section discusses using the following simulation libraries:

- "Using the UNISIM Library"
- "Using the VHDL UNISIM Library"
- "Using the Verilog UNISIM Library"
- "Using the CORE Generator XilinxCoreLib Library"
- "Using the SIMPRIM Library"
- "Using the SmartModel Library"

Using the UNISIM Library

The UNISIM Library is used for functional simulation only. This library includes:

- all of the Xilinx Unified Library primitives that are inferred by most synthesis tools
- primitives that are commonly instantiated, such as DCMs, BUFGs, and GTs

You should generally infer most design functionality using behavioral RTL code unless:

- the desired component is not inferable by your synthesis tool, or
- you want to take manual control of mapping and placement of a function

The UNISIM library structure is different for VHDL and Verilog.

Using the VHDL UNISIM Library

The VHDL UNISIM library is split into four files containing:

- the component declarations (`unisim_VCOMP.vhd`)
- package files (`unisim_VPKG.vhd`)
- entity and architecture declarations (`unisim_VITAL.vhd`)
- SmartModel declarations (`unisim_SMODEL.vhd`)

All primitives for all Xilinx device families are specified in these files. The VHDL UNISIM Library source directories are located as shown in the following table.

Table 6-6: **VHDL UNISIM Library Source Files**

Platform	Location
`Unix/Linux`	`$XILINX/vhdl/src/unisims`
`Windows`	`.%XILINX%\vhdl\src\unisims`

Using the Verilog UNISIM Library

For Verilog, each library component is specified in a separate file. This allows automatic library expansion using the **`-y`** library specification switch. All Verilog module names and file names are all upper case (for example, module BUFG would be BUFG.v, and module IBUF would be IBUF.v).

Since Verilog is a case-sensitive language, make sure that all UNISIM primitive instantiations adhere to this upper-case naming convention. The library sources are split into two directories. See the following table.

Table 6-7: **Verilog UNISIM Library Source Files**

	FPGA device families	CPLD device families
`Unix/Linux`	`$XILINX/verilog/src/unisims`	`$XILINX/verilog/src/uni9000`
`Windows`	`%XILINX%\verilog\src\unisims`	`%XILINX%\verilog\src\uni9000`

Using the CORE Generator XilinxCoreLib Library

The Xilinx CORE Generator is a graphical intellectual property design tool for creating high-level modules like FIR Filters, FIFOs and CAMs, as well as other advanced IP. You can customize and pre-optimize modules to take advantage of the inherent architectural features of Xilinx FPGA devices, such as block multipliers, SRLs, fast carry logic, and on-chip, single-port or dual-port RAM. You can also select the appropriate HDL model type as output to integrate into your HDL design.

The CORE Generator HDL library models are used for RTL simulation. The models do not use library components for global signals.

For the location of the CORE Generator library source files, see the following table.

Table 6-8: **CORE Generator Library Source Files**

	VHDL	Verilog
`Unix/Linux`	`$XILINX/vhdl/src/XilinxCoreLib`	`$XILINX/verilog/src/XilinxCoreLib`
`Windows`	`%XILINX%\vhdl\src\XilinxCoreLib`	`.%XILINX%\verilog\src\XilinxCoreLib`

Using the SIMPRIM Library

The SIMPRIM library is used for post Ngdbuild (gate level functional), post-Map (partial timing), and post-place-and-route (full timing) simulations. This library is architecture independent.

For the location of the SIMPRIM Library source files,ee the following table.

Table 6-9: **SIMPRIM Library Source Files**

	VHDL	Verilog
`Unix/Linux`	`$XILINX/vhdl/src/simprims`	`$XILINX/verilog/src/simprims`
`Windows`	`%XILINX%\vhdl\src\simprims`	`%XILINX%\verilog\src\simprims`

Using the SmartModel Library

The SmartModel Libraries are used to model very complex functions of modern FPGA devices such as the PowerPC™ and the RocketIO™. SmartModels are encrypted source files that communicate with simulators via the SWIFT interface.

The SmartModel Libraries are located at:

```
$XILINX/smartmodel or %XILINX%\smartmodel
```

The SmartModel Libraries require additional installation steps to properly install on your system. Additional setup within the simulator may also be required. For more information on how to install and set up the SmartModel Libraries, see Chapter 5, "Using SmartModels" in this guide.

Compiling Xilinx Simulation Libraries (COMPXLIB)

Note: Do NOT use with ModelSim XE (Xilinx Edition) or ISE Simulator.

Before starting the functional simulation of your design, you must compile the Xilinx Simulation Libraries for the target simulator. For this purpose Xilinx provides a tool called COMPXLIB.

COMPXLIB is a tool for compiling the Xilinx HDL based simulation libraries using the tools provided by the simulator vendor. Libraries should generally be compiled or recompiled any time a new version of a simulator is installed, including a new service pack.

Compiling Simulation Libraries

You can compile the simulation libraries from Project Navigator, or from the command line, as described below.

Compiling Simulation Libraries from Project Navigator

To compile a simulation library from Project Navigator.

1. Create or open an existing project for Project Navigator.
2. In the Sources window, highlight the target device.
3. In the Processes window, under the Design Entry Utilities toolbox, right-click **Compile HDL Simulation Libraries.**
4. Select **Properties** to open the Process Properties dialog box.
5. Choose one or more of the "Project Navigator Options" from the Process Properties dialog box.
6. Click **OK**.

   Double-click **Compile HDL Simulation Libraries**.

   Project Navigator compiles the libraries using the properties you specified.

To see the compilation results after the libraries are compiled, double-click **View Compilation Log** to open the COMPXLIB.log file.

Project Navigator Options

The following options are available from the Process Properties dialog box. Project Navigator shows only the options that apply to your specific design flow. For example, for a Virtex-II project, it shows only the list of libraries required to simulate a Virtex-II design.

Target Simulator

Select the target simulator for which the libraries are to be compiled. Click anywhere in the Value field to display the list of supported simulators. You must choose a Target Simulator before starting the compile process.

Language

Language is selected by default according to the Project Properties.

Compiled Library Directory

Specify the directory where the compiled libraries will be saved.

To change the directory path:

1. Type a new path in the Value field, or
   a. Click anywhere in the Value field.
   b. Double-click the button to the right of the current value.
2. Choose an output directory from the **Browse for File** dialog box.

The default directory path is `$XILINX/language/simulator`, where:

- `language` is the selected language to compile
- `simulator` is the name of the selected simulator

Simulator Location

Specify the path to the simulator executables. By default, Project Navigator searches the path environment variable for the simulator path. If you have multiple simulators, or if the simulator path is not defined in the environment variable, set the Simulator Location property to specify the path to your simulator.

To change the simulator path:

1. Click the **Value** field.
2. Double-click the button to the right of the current value.
3. Choose a directory from the **Browse for File** dialog box.

Existing Compiled Library

Choose whether to overwrite the previously compiled library, or map to the previously compiled library. The compiled libraries are overwritten by default.

Compile UNISIM (Functional) Simulation Library

Choose whether to compile the UNISIM library. The UNISIM libraries are compiled by default.

Compile SIMPRIM (Timing) Simulation Library

Choose whether to compile the SIMPRIM library. The SIMPRIM libraries are compiled by default.

Compile XilinxCoreLib (Coregen) Simulation Library

Choose whether to compile the XilinxCoreLib library. The XilinxCoreLib libraries are compiled by default.

Note: Since XilinxCoreLib libraries depend on the UNISIM libraries, COMPXLIB automatically compiles UNISIM prior to compiling the XilinxCoreLib libraries.

Compile SmartModels (PPC, MGT) Simulation SmartModels

Choose whether to compile the SmartModels library. The SmartModels libraries are compiled by default.

Update modelsim.ini File for Xilinx SmartModel Use

Instructs COMPXLIB to make the necessary modifications to the `modelsim.ini` file to run SmartModel simulations.

Compiling Simulation Libraries from the Command Line

To compile libraries from the command line, type:

```
compxlib [options]
```

For options and syntax details, see "COMPXLIB Syntax" in this chapter.

To view COMPXLIB help, type:

```
compxlib -help
```

COMPXLIB Support

For the libraries, device families, and simulators that COMPXLIB supports, see the following sections.

Libraries

COMPXLIB Support supports the compilation of the following Xilinx HDL Simulation Libraries:

- UNISIM (Functional)
- Uni9000 (Verilog Functional CPLDs only)
- SIMPRIM (Timing)
- XilinxCoreLib (Functional)
- SmartModel Library (Functional & Timing)
- CoolRunner (Functional)
- Abel (Functional)

Device Families

COMPXLIB supports the compilation of libraries for all Xilinx Device Families.

Simulators

Caution! Do NOT use with ModelSim XE (Xilinx Edition) or ISE Simulator.

COMPXLIB supports the compilation of Xilinx HDL Simulation Libraries for the following simulators:

- ModelSim SE (all Xilinx supported platforms)
- ModelSim PE (all Xilinx supported platforms)
- NCSIM (all Xilinx supported platforms)
- VCS-MX (only on Solaris and Linux based platforms)
- VCS-MXi (only on Solaris and Linux based platforms)

The VHDL SIMPRIM library is VITAL2000 compliant. Make sure that your simulator is also VITAL2000 compliant to successfully compile the SIMPRIM library.

COMPXLIB Syntax

The following command compiles all Xilinx Verilog libraries for the Virtex device family on the ModelSim SE simulator:

```
compxlib -s mti_se -arch virtex -l verilog
```

The compiled results are saved in the default location:

```
$XILINX/verilog/mti_se
```

COMPXLIB Command Line Options

This section describes COMPXLIB command line options.

Target Simulator (**`-s`**)

Specify the simulator for which the libraries will be compiled.

If **`-s`** is not specified, COMPXLIB exits without compiling the libraries.

Valid values for **`-s`** are:

```
-s mti_se
-s mti_pe
-s ncsim
-s vcs_mx
-s vcs_mxi
```

Language (**`-l`**)

Specify the language from which the libraries will be compiled.

By default, COMPXLIB detects the language type from the **`-s`** (Target Simulator) option. If the simulator supports both Verilog and VHDL, COPMXLIB:

- sets the **`-l`** option to *all*
- compiles both Verilog and VHDL libraries

If the simulator does *not* support both Verilog and VHDL, COMXPLIB:

- detects the language type supported by the simulator
- sets the **`-l`** option value accordingly

If the **`-l`** option is specified, COMXPLIB compiles the libraries for the language specified with the **`-l`** option.

Valid values for **`-l`** are:

```
-l verilog
-l vhdl
-l all
```

Device Family (**`-arch`**)

Specify the device family.

If **`-arch`** is not specified, COMPXLIB exits with an error message without compiling the libraries.

Valid values for **`-arch`** are:

`-arch all`[a]	`-arch virtex`	`-arch virtexe`
`-arch virtex2`	`-arch virtex2p`	`-arch virtex4`
`-arch spartan2`	`-arch spartan3`	`-arch spartan2e`
`-arch spartan3e`	`-arch cpld`	`-arch fpga`
`-arch cr2s`	`-arch xpla3`	`-arch xbr`
`-arch xc9500`	`-arch xc9500xl`	`-arch xc9500xv`
`-arch acr2`	`-arch aspartan2e`	`-arch aspartan3`
`-arch qrvirtex`	`-arch qrvirtex2`	`-arch qvirtex`
`-arch qvirtex2`	`-arch qvirtexe`	

a. all device families

To compile selected libraries, use the following **`-arch`** syntax.

```
-arch device_family
```

Output Directory (**`-dir`**)

Specify the directory path where you want to compile the libraries. By default, COMXPLIB compiles the libraries as shown in in the following table.

Table 6-10: **Default COMXPLIB Output Directories**

Operating System	Default Output Directory
Unix/Linux	`$XILINX/`*`language/target_simulator`*
Windows	`%XILINX%\language\target_simulator`

Simulator Path (**`-p`**)

Specify the directory path where the simulator executables reside. By default, COMPXLIB automatically searches for the path from the `$PATH` or `%PATH%` environment variable. This option is required if the target simulator is not specified in the `$PATH` or `%PATH%` environment variable.

Overwrite Compiled Library (**`-w`**)

Overwrite the precompiled libraries. By default, COMXPLIB does not overwrite the precompiled libraries.

Create Configuration File (**`-cfg`**)

Create a configuration file with default settings. By default, COMPXLIB creates the `compxlib.cfg` file if it is not present in the current directory.

Use the configuration file to pass run time options to COMPXLIB while compiling the libraries. For more information on the configuration file, see "Specifying Run Time Options" in this chapter.

Print Precompiled Library Info (**-info**)

Print the precompiled information of the libraries. Specify a directory path with **-info** to print the information for that directory.

Specify Name of Library to Compile (**-lib**)

Specify the name of the library to compile.

Valid values for **-lib** are:

```
unisim
simprim
uni9000
xilinxcorelib
smartmodel
abel
coolrunner
```

For multiple libraries, separate the **-lib** options with spaces. For example:

```
.. -lib uisim -lib simprim ..
```

If **-lib** is not used, all the libraries are compiled by default.

Print COMXPLIB Help (**-help**)

View COMPXLIB help.

SmartModel Setup (MTI only)

The **-smartmodel_setup** option instructs COMPXLIB to modify the **modelsim.ini** file to make the amendments necessary to run SmartModel simulations.

COMPXLIB Command Line Examples

This section shows examples of the following:

- "Compiling Libraries as a System Administrator"
- "Compiling Libraries as a User"

Compiling Libraries as a System Administrator

System administrators compiling the libraries using COMPXLIB should compile the libraries in a default location that is accessible to all users.

The following example shows how to compile the libraries for ModelSim SE for all devices and all libraries and all languages setting up the ini file for smartmodels

```
compxlib -s mti_se -arch all -smartmodel_setup
```

This compiles the libraries needed for simulation using Model sim SE. For the location to which the libraries are compiled, see the following table.

Table 6-11: **ModelSim SE Libraries Locations**

	VHDL	Verilog
Unix/Linux	$XILINX/vhdl/mti_se	$XILINX/verilog/mti_se
Windows	%XILINX%\vhdl\mti_se	%XILINX%\verilog\mti_se

Compiling Libraries as a User

When you run COMPXLIB as a user, Xilinx recommends that you compile the libraries on a per project basis. If your project targets a single Xilinx device, compile the libraries for that specific device only.

The following example shows how to compile UNISIM and SIMPRIM libraries for NCSIM (VHDL) for a Virtex-4 design:

```
compxlib -s ncsim -arch virtex4 -lib unisim -lib simprim -lang vhdl -o ./
```

This compiles the libraries to the current working directory.

Mapping to Pre-Compiled Libraries as a User

If the system administrator has compiled all the libraries to the default location, each individual user can map to these libraries as needed. Xilinx recommends that each user map to the libraries on a per project basis to minimize the need for unnecessary library mappings in the project location.

The example below shows how to map to the pre-compiled UNISIM and XILINXCORELIB libraries for ModelSim PE for a Virtex-4 design:

```
    compxlib -s mti_pe -arch virtex4 -lib unisim -lib xilinxcorelib
```

When you map to a pre-compiled location, do not specify the **-w** switch. If there are no pre-compiled libraries in the default location, COMPXLIB starts to compile the libraries.

Additional Compxlib Examples

Following are additional examples of using Compxlib.

Table 6-12: **Additional Compxlib Examples**

Task	Command
Display the COMPXLIB help onscreen	`compxlib -h`
Obtain help for a specific option	`compxlib -h <option>`
Obtain help for all the available architectures	`compxlib -h arch`
Compile all of the Verilog libraries for a Virtex device (UNISIM, SIMPRIM and XilinxCoreLib) on the ModelSim SE simulator and overwrite the results in $XILINX/verilog/mti_se	`compxlib -s mti_se -arch virtex -l verilog -w`
Compile the Verilog UNISIM, Uni9000 and SIMPRIM libraries for the ModelSim PE simulator and save the results in the $MYAREA directory	`compxlib -s mti_pe -arch all -lib uni9000 -lib simprim-l verilog -dir $MYAREA`
Compile the VHDL and Verilog SmartModels for the Cadence NC-Sim simulator and save the results in /tmp directory	`compxlib -s ncsim -arch virtex2p -lib smartmodel -l all -dir /tmp`
Compile the Verilog Virtex-II XilinxCoreLib library for the Synopsys VCS simulator and save the results in the default directory, `$XILINX/verilog/vcs`	`compxlib -s vcs_mx -arch virtex2 -lib xilinxcorelib`
Compile the Verilog CoolRunner library for the Synopsys VCSi simulator and save the results in the `current` directory	`compxlib -s vcs mxi -arch coolrunner -lib -dir ./`

Table 6-12: **Additional Compxlib Examples**

Task	Command
Compile the Spartan-IIE and Spartan-3 libraries (VHDL UNISIMs, SIMPRIMs and XilinxCoreLib) for the Synopsys Scirocco simulator and save the results in the default directory (`$XILINX/vhdl/scirocco`), and use the simulator executables from the path specified with the **`-p`** option,	`compxlib -s vcs mx -arch spartan2e -l vhdl -arch spartan3 -p /products/eproduct.ver2_0/2.0/comnon/sunos5/bin`
Print the precompiled library information for the libraries compiled in `%XILINX%\xilinxlibs`	`compxlib -info %XILINX%\xilinxlibs`
Print the precompiled library information for the libraries compiled in `$XILINX` for the ModelSim SE simulator	`compxlib -info $XILINX/mti_se/`
Create `compxlib.cfg` with default options	`compxlib -cfg`

Specifying Run Time Options

Use the `compxlib.cfg` file to specify run time options for COMPXLIB. By default, COMPXLIB creates this file in the current directory. To automatically create this file with its default settings, use **`-cfg`**.

You can specify the following run time options in the configuration file.

`EXECUTE: ON|OFF`

By default, the value is ON.

If the value is ON, COMPXLIB compiles the libraries.

If the value is OFF, COMPXLIB generates only the list of compilation commands in the `compxlib.log` file, without executing them.

`EXTRACT_LIB_FROM_ARCH: ON|OFF`

This option supports Early Access devices. Do not change this option.

`LOCK_PRECOMPILED: ON |OFF`

By default, the value is OFF.

If the value is OFF, COMPXLIB compiles the dependent libraries automatically if they are not precompiled.

If the value is ON, COMPXLIB does not compile the precompiled libraries.

For example, if you want to compile the SmartModel Library, COMPXLIB looks for this variable value to see if the dependent libraries, UNISIM and SIMPRIM, are to be compiled.

`LOG_CMD_TEMPLATE: ON|OFF`

By default, the value is OFF.

If the value is OFF, COMPXLIB does not print the compilation command line in the `compxlib.log` file.

If the value is ON, COMXPLIB prints the compilation commands in the `compxlib.log` file.

`PRECOMPILED_INFO: ON|OFF`

By default, the value is ON.

If the value is ON, COMPXLIB prints the precompiled library information including the date the library was compiled.

If the value is OFF, COMXPLIB does not print the precompiled library information.

`BACKUP_SETUP_FILES: ON|OFF`

By default, the value is ON.

If the value is ON, COMPXLIB creates a backup of the all the simulator specific setup files (`modelsim.ini/cds/lib/hdl.var/`) that it wrote out in the previous run.

If the value is OFF, COMXPLIB does not create a backup of the setup files.

`FAST_COMPILE: ON|OFF`

By default, the value is ON.

If the value is ON, COMPXLIB uses advanced compilation techniques for faster library compilation for select libraries.

If the value is OFF, COMXPLIB does not use the advanced compilation methods and reverts to traditional methods for compilation.

`ABORT_ON_ERROR: ON|OFF`

By default, the value is OFF.

If the value is OFF, COMPXLIB does not error out if a compilation error occurs.

If the value is ON, COMXPLIB errors out if a compilation error occurs.

`ADD_COMPILATION_RESULTS_TO_LOG: ON|OFF`

By default, the value is ON.

If the value is ON, COMPXLIB writes to the log file with the name specified by **`-log`**.

If the value is OFF, COMXPLIB ignores **`-log`**.

`USE_OUTPUT_DIR_ENV: NONE|<NAME_OF_ENVIRONMENT_VARIABLE>`

By default, the value is NONE.

If the value is NONE, COMPXLIB does not look for an environment variable for the output directory. Instead, it uses the directory specified by **`-o`**.

If the value is <NAME_OF_ENV_VAR>, COMXPLIB looks on the system for an environment variable with the name listed in this option, and compiles the libraries to that folder. See the following example.

cfg file	`USE_OUTPUT_DIR_ENV:MY_LIBS`
system setting	`setenv MY_LIBS  /my_compiled_libs`
compiles the libraries to the folder	`/my_compiled_libs`

`INSTALL_SMARTMODEL: ON|OFF`

By default, the value is ON.

If the value is ON, COMPXLIB installs the smartmodels when **`-lib smartmodel`** is used.

If the value is OFF, COMXPLIB does not install the SmartModels even if the **`-lib smartmodel`** is used.

`INSTALL_SMARTMODEL_DIR:`

By default, the value is left blank.

If the value is blank, COMPXLIB writes to the location pointed to by the LMC_HOME environment variable.

If the LMC_HOME environment variable is not set, the SmartModels are installed to the directory specified here. This option is used only if the INSTALL_SMARTMODEL option is set to ON

`OPTION`

Simulator language command line options.

```
OPTION:Target_Simulator:Language:Command_Line_Options
```

By default, COMXPLIB picks the simulator compilation commands specified in the *`Command_Line_Options`*.

You can add or remove the options from *`Command_Line_Options`* depending on the compilation requirements.

Sample Configuration File (Windows Version)

```
#*****************************************************************
#   compxlib initialization file (compxlib.cfg)                  *
#                                                                *
#   Copyright (c) 1995-2005 Xilinx, Inc.  All rights reserved.   *
#                                                                *
#   Important :-                                                 *
#       All options/variables must start from first column       *
#                                                                *
#*****************************************************************

#
RELEASE_VERSION:I.23
#
# set current simulator name
SIMULATOR_NAME:mti_se
#
# set current language name
LANGUAGE_NAME:verilog
#
# set compilation execution mode
EXECUTE:on
#
# compile additional libraries in architecture specific directories
EXTRACT_LIB_FROM_ARCH:on
#
MAP_PRE_COMPILED_LIBS:off
#
```

```
# donot re-compile dependent libraries
LOCK_PRECOMPILED:off
#
# print compilation command template in log file
LOG_CMD_TEMPLATE:off
#
# print Pre-Compiled library info
PRECOMPILED_INFO:on
#
# create backup copy of setup files
BACKUP_SETUP_FILES:on
#
# use enhanced compilation techniques for faster library compilation
# (applicable to selected libraries only)
FAST_COMPILE:on
#
# abort compilation process if errors are detected in the library
ABORT_ON_ERROR:off
#
# save compilation results to log file with the name specified with -log option
ADD_COMPILATION_RESULTS_TO_LOG:on
#
# compile library in the directory specified by the environment variable if the
# -dir option is not specified
USE_OUTPUT_DIR_ENV:NONE
#
# turn on/off smartmodel installation process
INSTALL_SMARTMODEL:on
#
# smartmodel installation directory
INSTALL_SMARTMODEL_DIR:
#
#/////////////////////////////////////////////////////////////////////////
# MTI-SE setup file name
SET:mti_se:MODELSIM=modelsim.ini
#
# MTI-SE options for VHDL Libraries
# Syntax:-
# OPTION:<simulator_name>:<language>:<library>:<options>
# <library> :- u (unisim) s (simprim) c (xilinxcorelib)
#              m (smartmodel) a (abel) r (coolrunner)
#
OPTION:mti_se:vhdl:u:-source -93
OPTION:mti_se:vhdl:s:-source -93
OPTION:mti_se:vhdl:c:-source -93 -explicit
OPTION:mti_se:vhdl:m:-source -93
OPTION:mti_se:vhdl:a:-source -93
OPTION:mti_se:vhdl:r:-source -93
#
# MTI-SE options for VERILOG Libraries
# Syntax:-
# OPTION:<simulator_name>:<language>:<library>:<options>
# <library> :- u (unisim) s (simprim) c (xilinxcorelib)
#              m (smartmodel) a (abel) r (coolrunner)
#
OPTION:mti_se:verilog:u:-source -93
OPTION:mti_se:verilog:s:-source -93
OPTION:mti_se:verilog:n:-source -93
OPTION:mti_se:verilog:c:-source -93
```

```
OPTION:mti_se:verilog:m:-source -93
OPTION:mti_se:verilog:a:-source -93
OPTION:mti_se:verilog:r:-source -93
#
#/////////////////////////////////////////////////////////////////////
# MTI-PE setup file name
SET:mti_pe:MODELSIM=modelsim.ini
#
# MTI-PE options for VHDL Libraries
# Syntax:-
# OPTION:<simulator_name>:<language>:<library>:<options>
# <library> :- u (unisim) s (simprim) c (xilinxcorelib)
#              m (smartmodel) a (abel) r (coolrunner)
#
OPTION:mti_pe:vhdl:u:-source -93
OPTION:mti_pe:vhdl:s:-source -93
OPTION:mti_pe:vhdl:c:-source -93 -explicit
OPTION:mti_pe:vhdl:m:-source -93
OPTION:mti_pe:vhdl:a:-source -93
OPTION:mti_pe:vhdl:r:-source -93
#
# MTI-PE options for VERILOG Libraries
# Syntax:-
# OPTION:<simulator_name>:<language>:<library>:<options>
# <library> :- u (unisim) s (simprim) c (xilinxcorelib)
#              m (smartmodel) a (abel) r (coolrunner)
#
OPTION:mti_pe:verilog:u:-source -93
OPTION:mti_pe:verilog:s:-source -93
OPTION:mti_pe:verilog:n:-source -93
OPTION:mti_pe:verilog:c:-source -93
OPTION:mti_pe:verilog:m:-source -93
OPTION:mti_pe:verilog:a:-source -93
OPTION:mti_pe:verilog:r:-source -93
#
#/////////////////////////////////////////////////////////////////////
# NCSIM setup file names
SET:ncsim:CDS=cds.lib
SET:ncsim:HDL=hdl.var
#
# NCSIM options for VHDL Libraries
# Syntax:-
# OPTION:<simulator_name>:<language>:<library>:<options>
# <library> :- u (unisim) s (simprim) c (xilinxcorelib)
#              m (smartmodel) a (abel) r (coolrunner)
#
OPTION:ncsim:vhdl:u:-MESSAGES -v93 -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:vhdl:s:-MESSAGES -v93 -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:vhdl:c:-MESSAGES -v93 -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:vhdl:m:-MESSAGES -v93 -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:vhdl:a:-MESSAGES -v93 -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:vhdl:r:-MESSAGES -v93 -NOLOG -CDSLIB $CDS -HDLVAR $HDL
#
# NCSIM options for VERILOG Libraries
# Syntax:-
# OPTION:<simulator_name>:<language>:<library>:<options>
# <library> :- u (unisim) s (simprim) c (xilinxcorelib)
#              m (smartmodel) a (abel) r (coolrunner)
#
```

```
OPTION:ncsim:verilog:u:-MESSAGES -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:verilog:s:-MESSAGES -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:verilog:n:-MESSAGES -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:verilog:c:-MESSAGES -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:verilog:m:-MESSAGES -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:verilog:a:-MESSAGES -NOLOG -CDSLIB $CDS -HDLVAR $HDL
OPTION:ncsim:verilog:r:-MESSAGES -NOLOG -CDSLIB $CDS -HDLVAR $HDL
#/////////////////////////////////////////////////////////////////////////
# End
```

Running NetGen

Xilinx provides a program that can create a verification netlist file from your design files. You can create a timing simulation netlist as follows:

- "Running NetGen from Project Navigator"
- "Running NetGen from XFLOW"
- "Running NetGen from the Command Line or a Script File"

Running NetGen from Project Navigator

To create a simulation netlist from Project Navigator:

1. Highlight the top level design in the **Sources** window.
2. Make sure that the flow is set to **Synthesis/Implementation**.
3. In the **Processes** window, click the "+" sign next to **Implement Design**.
4. Click the "+" sign next to **Place & Route**.
5. To change default options:
   a. Right-click the **Generate Post Place & Route Simulation Model** process.
   b. Select **Properties**.
   c. Select **Standard** or **Advanced** from **Property display level** in the **Simulation Model Properties** dialog box.
   d. Choose options. For more information about options, see the Project Navigator help.
6. Double-click **Generate Post Place & Route Simulation Model**.

Project Navigator creates the back-annotated simulation netlist.

Running NetGen from XFLOW

To display the available options for XFLOW, and for a complete list of the XFLOW option files, type **`xflow`** at the prompt without any arguments. For complete descriptions of the options and the option files, see the *Xilinx Development System Reference Guide*.

Creating a Simulation Netlist from XFLOW

To create a Simulation netlist from XFLOW:

1. Open a command terminal.
2. Change directory to the project directory.
3. Type the following at the command prompt:

a. To create a functional simulation (Post NGD) netlist from an input design EDIF file:

```
> xflow -fsim option_file.opt design_name.edif
```

b. To create a timing simulation (post PAR) netlist from an input EDIF design file:

```
> xflow -implement option_file -tsim option_file design_name.edf
```

c. To create a timing simulation (Post PAR) netlist from an NCD file:

```
> xflow -tsim option_file.opt design_name.ncd
```

XFLOW runs the appropriate programs with the options specified in the option file.

Changing Options

To change the options:

1. Run XFLOW with the **–norun** option.

   XFLOW copies the option files to the project directory.

2. Edit the option files to modify the run parameters for the flow.

For more information, see the Xilinx *Development System Reference Guide*.

Running NetGen from the Command Line or a Script File

To create a simulation netlist from the command line or a script file, follow the instructions below.

Post-NGD simulation

To run a post-NGD simulation for VHDL, type:

```
netgen -sim -ofmt vhdl [options] design.ngd
```

To run a post-NGD simulation for Verilog, type:

```
netgen -sim -ofmt verilog [options] design.ngd
```

Post-Map Simulation

To run a post-Map simulation, perform the following command line operations:

```
ngdbuild options design
map options design.ngd
```

- Verilog

  ```
  netgen -sim -ofmt verilog [options] design.ncd
  ```

- VHDL

  ```
  netgen -sim -ofmt vhdl [options] design.ncd
  ```

Post-PAR simulation

To run a post-PAR simulation, perform the following command line operations:

```
ngdbuild options design
map options design.ngd
par options design.ncd -w design_par.ncd
```

- Verilog

  `netgen -sim -ofmt verilog [`*`options`*`]` *`design`*`_par.ncd`

- VHDL

  `netgen -sim -ofmt vhdl [`*`options`*`]` *`design`*`_par.ncd`

Disabling X Propagation

When a timing violation occurs during a timing simulation, the default behavior of a latch, register, RAM, or other synchronous element outputs an X to the simulator.

This occurs because the actual output value is not known. The output of the register could:

- retain its previous value
- update to the new value
- go metastable, in which a definite value is not settled upon until some time after the clocking of the synchronous element

Since this value cannot be determined, and accurate simulation results cannot be guaranteed, the element outputs an X to represent an unknown value. The X output remains until the next clock cycle in which the next clocked value updates the output if another violation does not occur.

X generation can significantly affect simulation. For example, an X generated by one register can be propagated to others on subsequent clock cycles. This may cause large portions of the design being tested to become unknown. This can be corrected as follows:

- On a synchronous path, analyze the path and fix any timing problems associated with this or other paths to ensure a properly operating circuit.
- On an asynchronous path, if you cannot otherwise avoid timing violations, disable the X propagation on synchronous elements during timing violations.

  When X propagation is disabled, the previous value is retained at the output of the register. In the actual silicon, the register may have changed to the 'new' value. Disabling X propagation may yield simulation results that do not match the silicon behavior.

  Caution! Exercise care when using this option. Use it only if you cannot otherwise avoid timing violations.

Using the ASYNC_REG Constraint

The ASYNC_REG constraint:

- identifies asynchronous registers in the design
- disables X propagation for those registers

ASYNC_REG can be attached to a register in the front end design by:

- an attribute in the HDL code, or
- a constraint in the UCF

The registers to which ASYNC_REG is attached retain the previous value during timing simulation, and do not output an X to simulation.

Caution! A timing violation error may still occur. Use care, as the new value may have been clocked in as well.

The ASYNC_REG constraint is applicable to CLB and IOB registers and latches only. If you cannot avoid clocking in asynchronous data, Xilinx recommends that you do so on IOB or CLB registers only. Clocking in asynchronous signals to RAM, SRL, or other synchronous elements has less deterministic results, and therefore should be avoided. Xilinx highly recommends that you first properly synchronize any asynchronous signal in a register, latch, or FIFO before writing to a RAM, SRL, or any other synchronous element.

For more information on ASYNC_REG, see the Xilinx *Constraints Guide*.

SIM_COLLISION_CHECK

Xilinx block RAM memory is a true dual-port RAM where both ports can access any memory location at any time. Be sure that the same address space is not accessed for reading and writing at the same time. This will cause a block RAM address collision. These are valid collisions, since the data that is read on the read port is not valid. In the hardware, the value that is read might be the old data, the new data, or a combination of the old data and the new data. In simulation, this is modeled by outputting X since the value read is unknown. For more information on block RAM collisions, see the architecture specific user guides.

In certain applications, this situation cannot be avoided or designed around. In these cases, the block RAM can be configured not to look for these violations. This is controlled by the generic (VHDL) or parameter (Verilog) SIM_COLLISION_CHECK in all the Xilinx block RAM primitives.

Use With Care

Xilinx strongly recommends that you disable X propagation ONLY on paths that are truly asynchronous where it is impossible to meet synchronous timing requirements. This capability is present for simulation in the event that timing violations cannot be avoided, such as when a register must input asynchronous data. Use extreme caution when disabling X propagation; simulation results may no longer accurately reflect what is happening in the silicon.

SIM_COLLISION_CHECK Strings

Use the following strings with SIM_COLLISION_CHECK to control what happens in the event of a collision.

Table 6-13: **SIM_COLLISION_CHECK Strings**

String	Write Collision Messages	Write Xs on the Output
ALL	Yes	Yes
WARNING_ONLY	Yes	No
GENERATE_X_ONLY	No	Yes
None	No	No

SIM_COLLISION_CHECK can be applied at an instance level. This enables you to change the setting for each block RAM instance.

MIN/TYP/MAX Simulation

The Standard Delay Format (SDF) file allows you to specify three sets of delay values for simulation:

- "Maximum (MAX)"
- "Typical (TYP)"
- "Minimum (MIN)"

These values are usually abbreviated as MIN:TYP:MAX.

Xilinx uses these values to allow the simulation of the target architecture under various operating conditions. By allowing for the simulation across various operating conditions, you can perform more accurate setup and hold timing verification.

Definitions

Following are the definitions of the three values:

- "Maximum (MAX)"
- "Typical (TYP)"
- "Minimum (MIN)"

Maximum (MAX)

The Maximum (MAX) field is used to represent the delays under the worst case operating conditions of the device. The worst case operating conditions are defined as the maximum operating temperature, the minimum voltage, and the worst case process variations. Under worst case conditions, the data paths of the device have the maximum delay possible, while the clock path delays are the minimum possible relative to the data path delays. This situation is ideal for setup time verification of the device.

Typical (TYP)

The Typical (TYP) field is used to represent the typical operating conditions of the device. In this situation, the clock and data path delays are both the maximum possible. This is different from the MAX field, in which the clock paths are the minimum possible relative to the maximum data paths.

Minimum (MIN)

The Minimum (MIN) field is used to represent the device under the best case operating conditions. The base case operating conditions are defined as the minimum operating temperature, the maximum voltage, and the best case process variations. Under best case conditions, the data paths of the device have the minimum delay possible, while the clock path delays are the maximum possible relative to the data path delays. This situation is ideal for hold time verification of the device.

Obtaining Accurate Results

In order to obtain the most accurate setup and hold timing simulations, you should perform two simulations using the proper SDF values.

To perform a setup simulation, specify values in the Maximum (MAX) field with the following command line modifier:

```
-SDFMAX
```

To perform the most accurate hold simulation, specify values in the Minimum (MIN) field with the following command line modifier:

```
-SDFMIN
```

For a simulation that matches the results of previous software releases, specify values in the Typical (TYP) field with the following command line modifier:

```
-SDFTYP
```

For more information on how to pass the SDF switches to the simulator, see your simulator tool documentation.

Using NetGen

NetGen can optionally produce absolute minimum delay values for simulation by applying the **`-s min`** switch. The resulting SDF file produced from NetGen has the absolute process minimums populated in all three SDF fields: MIN, TYP, and MAX.

Absolute process MIN values are the absolute fastest delays that a path can run in the target architecture given the best operating conditions within the specifications of the architecture:

- lowest temperature
- highest voltage
- best possible silicon

Generally, these process minimum delay values are only useful for checking board-level, chip-to-chip timing for high-speed data paths in best/worst case conditions.

By default, the worst case delay values are derived from the worst temperature, voltage, and silicon process for a particular target architecture. If better temperature and voltage characteristics can be ensured during the operation of the circuit, you can use prorated worst case values in the simulation to gain better performance results. The default would apply worst case timing values over the specified TEMPERATURE and VOLTAGE within the operating conditions recommended for the device.

Using the VOLTAGE and TEMPERATURE Constraints

Prorating is a linear scaling operation. It applies to existing speed file delays, and is applied globally to all delays. The prorating constraints, the “VOLTAGE Constraint” and the “TEMPERATURE Constraint”, provide a method for determining timing delay characteristics based on known environmental parameters.

VOLTAGE Constraint

The VOLTAGE constraint provides a means of prorating delay characteristics based on the specified voltage applied to the device. The UCF syntax is:

```
VOLTAGE=value[V]
```

where:

```
value
```

is an integer or real number specifying the voltage, and

```
units
```

is an optional parameter specifying the unit of measure.

TEMPERATURE Constraint

The TEMPERATURE constraint provides a means of prorating device delay characteristics based on the specified junction temperature. The UCF syntax is:

```
TEMPERATURE=value[C|F|K]
```

where:

```
value
```

is an integer or a real number specifying the temperature, and

```
C, K, and F
```

are the temperature units:

- F = degrees Fahrenheit
- K =degrees Kelvin
- C =degrees Celsius (default)

The resulting values in the SDF fields when using prorated TEMPERATURE and VOLTAGE values are the prorated worst case values.

To determine the specific range of valid operating temperatures and voltages for the target architecture, see the product data sheet. If the temperature or voltage specified in the constraint does not fall within the supported range, the constraint is ignored and an architecture specific default value is used instead. Not all architectures support prorated timing values. For simulation, the VOLTAGE and TEMPERATURE constraints are processed from the UCF file into the PCF file. The PCF file must then be referenced when running NetGen in order to pass the operating conditions to the delay annotator.

To generate a simulation netlist using prorating for VHDL, type:

```
netgen -sim -ofmt vhdl [options] -pcf design.pcf design.ncd
```

To generate a simulation netlist using prorating for Verilog, type:

```
netgen -sim -ofmt verilog [options] -pcf design.pcf design.ncd
```

Combining both minimum values overrides prorating, and results in issuing only absolute process MIN values for the simulation SDF file. Prorating may be available only for select FPGA families. It is not intended for military and industrial ranges. It is applicable only within the commercial operating ranges.

Table 6-14: **NetGen Options**

NetGen Option	MIN:TYP:MAX Field in SDF File Produced by NetGen –sim
default	MAX:MAX:MAX
–s min	Process MIN: Process MIN: Process MIN
Prorated voltage/temperature in UCF/PCF	Prorated MAX: Prorated MAX: Prorated MAX

Understanding the Global Reset and 3-state for Simulation

Xilinx FPGA devices have dedicated routing and circuitry that connects to every register in the device. The dedicated global GSR (Global Set-Reset) net is asserted and is released during configuration immediately after the device is configured. All the flip-flops and latches receive this reset and are either set or reset, depending on how the registers are defined.

Although you can access the GSR net after configuration, Xilinx does not recommend using the GSR circuitry in place of a manual reset. This is because the FPGA devices offer high-speed backbone routing for high fanout signals like a system reset. This backbone route is faster than the dedicated GSR circuitry, and is easier to analyze than the dedicated global routing that transports the GSR signal.

In back-end simulations, a GSR signal is automatically pulsed for the first 100 ns to simulate the reset that occurs after configuration. A GSR pulse can optionally be supplied in front end functional simulations, but is not necessary if the design has a local reset that resets all registers. When you create a test bench, remember that the GSR pulse occurs automatically in the back-end simulation. This holds all registers in reset for the first 100 ns of the simulation. For more information about controlling the GSR pulse or inserting a GSR pulse in the front end simulation, see "Simulating VHDL" and "Simulating Verilog" in this chapter.

In addition to the dedicated global GSR, all output buffers are set to a high impedance state during configuration mode with the dedicated GTS (global output 3-state enable) net. All general-purpose outputs are affected whether they are regular, 3-state, or bi-directional outputs during normal operation. This ensures that the outputs do not erroneously drive other devices as the FPGA is being configured.

In simulation, the GTS signal is usually not driven. The circuitry for driving GTS is available in the back-end simulation and can be optionally added for the front end simulation, but the GTS pulse width is set to 0 by default. For more information about controlling the GTS pulse or inserting the circuitry to pulse GTS in the front end simulation, see "Simulating VHDL" and "Simulating Verilog" in this chapter.

The following figure shows how the global GTS and GSR signals are used in the FPGA.

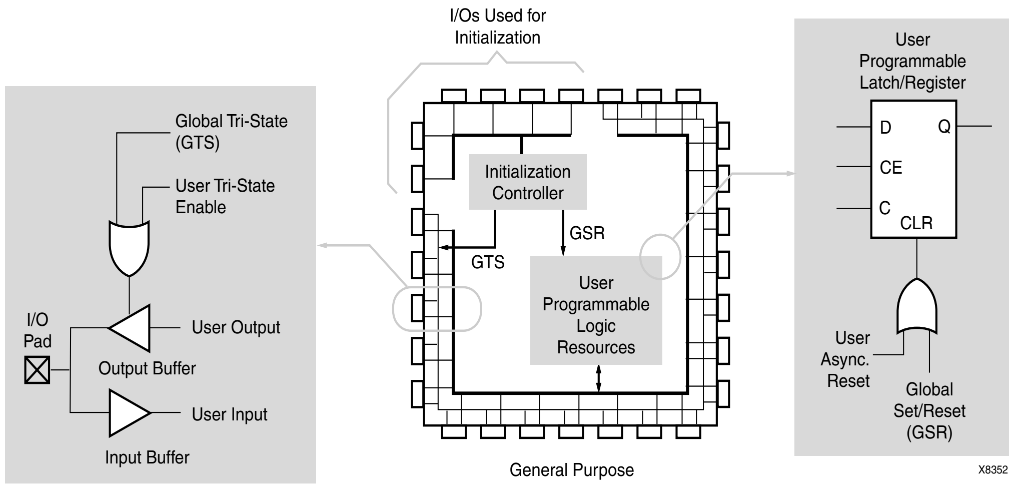

Figure 6-2: **Built-in FPGA Initialization Circuitry**

Simulating VHDL

This section discusses simulation in VHDL.

Emulating the Global GSR Pulse in VHDL in Functional Simulation

Many HDL designs targeted for Xilinx FPGA devices have a user reset that initializes all registers in the design during the functional simulation. For these designs, it is not necessary to emulate the GSR pulse in the functional simulation. If the design contains registers that are not connected to a user reset, the GSR pulse can be emulated to ensure that the functional simulation matches the timing simulation. There are two methods that can be used to emulate the GSR pulse:

- Use the ROC cell to generate a one-time GSR pulse at the beginning of the simulation. See "Using VHDL Reset-On-Configuration (ROC) Cell" in this chapter.
- Use the ROCBUF cell to control the emulated GSR signal in the test bench. See"Using VHDL ROCBUF Cell" in this chapter.

Using VHDL Reset-On-Configuration (ROC) Cell

The ROC cell, which is modeled in the UNISIM library, can be used to emulate the GSR pulse at the beginning of a functional simulation. This is the same component that is automatically inserted into the back-end netlist. It generates a one time pulse at the beginning of the simulation that lasts for a default value of 100ns.

During implementation, the signal connected to the output of the ROC component will automatically be mapped to the Global GSR network and will not be routed on local routing.

Code Example

The following example shows how to use the ROC cell.

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

```
library UNISIM;
use UNISIM.all;
entity EX_ROC is
  port (
        CLOCK, ENABLE : in std_logic;
        CUP, CDOWN : out std_logic_vector (3 downto 0)
        );
end EX_ROC;
architecture A of EX_ROC is
  signal GSR : std_logic;
  signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
  component ROC
    port (O : out std_logic);
  end component;
begin
  U1 : ROC port map (O => GSR);

  UP_COUNTER : process (CLOCK, ENABLE, GSR)
    begin
      if (GSR = '1') then
          COUNT_UP <= "0000";
      elsif (CLOCK'event AND CLOCK = '1') then
        if (ENABLE = '1') then
            COUNT_UP <= COUNT_UP + "0001";
        end if;
      end if;
  end process UP_COUNTER;
  DOWN_COUNTER : process (CLOCK, ENABLE, GSR, COUNT_DOWN)
  begin
    if (GSR = '1' OR COUNT_DOWN = "0101") then
        COUNT_DOWN <= "1111";
    elsif (CLOCK'event AND CLOCK = '1') then
        if (ENABLE = '1') then
            COUNT_DOWN <= COUNT_DOWN - "0001";
        end if;
    end if;
  end process DOWN_COUNTER;
  CUP <= COUNT_UP;
  CDOWN <= COUNT_DOWN;
end A;
```

Progression of the ROC Model

The following figure shows the progression of the ROC model and its interpretation in the four main design phases.

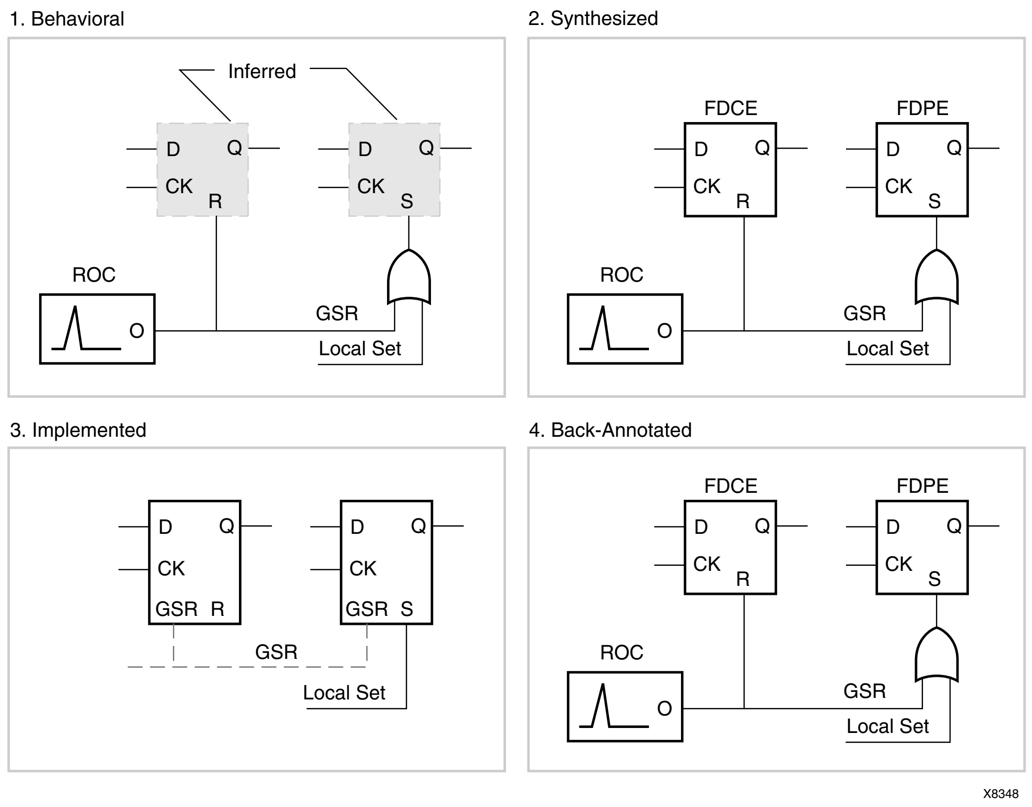

Figure 6-3: **ROC Simulation and Implementation**

Behavioral Phase

In the behavioral phase, the behavioral or RTL description registers are inferred from the coding style, and the ROC cell is instantiated. This ensures that GSR behavior at the RTL level matches the behavior of the post-synthesis and implementation netlists.

Synthesized Phase

In the synthesized phase, inferred registers are mapped to a technology and the ROC instantiation is carried from the RTL to the implementation tools. As a result, consistent global set/reset behavior is maintained between the RTL and synthesized structural descriptions during simulation.

Implemented Phase

During the implemented phase, the ROC is removed from the logical description and is placed and routed as a pre-existing circuit on the chip. All set/resets for the registers are automatically assumed to be driven by the global set/reset net so data is not lost.

Back-Annotated Phase

In the back-annotated phase, the Xilinx VHDL netlist program:

- assumes all buffers are driven by the GSR net

- uses the X_ROC simulation model for the ROC
- rewires it to the GSR nets in the back-annotated netlist

For a non-hierarchical netlist, the GSR net is a fully wired net and the X_ROC cell drives it. For a hierarchical netlist, the GSR net is not wired across hierarchical modules. The GSR net in each hierarchical module is driven by an X_ROC cell. The ROC pulse width of the X_ROC component can be controlled using the -rpw switch for NetGen.

Using VHDL ROCBUF Cell

A second method of emulating GSR in the functional simulation is to use the ROCBUF. This component creates a buffer for the global set/reset signal, and provides an input port on the buffer to drive the global set reset line. This port must be declared in the entity list and driven in RTL simulation.

This method is applicable when system-level issues make your design's initialization synchronous to an off-chip event. In this case, you provide a pulse that initializes your design at the start of simulation time, and you possibly provide further pulses as simulation time progresses (perhaps to simulate cycling power to the device).

During the place and route process, this port is removed; it is not implemented on the chip. ROCBUF does not by default reappear in the post-routed netlist unless the **`-gp`** switch is used during NetGen netlisting. The nets driven by a ROCBUF must be an active High set/reset.

Code Example

The following example illustrates how to use the ROCBUF in your designs.

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

library UNISIM;
use UNISIM.all;

entity EX_ROCBUF is
  port (
        CLOCK, ENABLE, SRP : in std_logic;
        CUP, CDOWN : out std_logic_vector (3 downto 0)
        );
end EX_ROCBUF;

architecture A of EX_ROCBUF is
  signal GSR : std_logic;
  signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
  component ROCBUF
    port (
        I : in std_logic;
        O : out std_logic
        );
  end component;
begin
  U1 : ROCBUF port map (I => SRP, O => GSR);
  UP_COUNTER : process (CLOCK, ENABLE, GSR)
  begin
    if (GSR = '1') then
        COUNT_UP <= "0000";
    elsif (CLOCK'event AND CLOCK = '1') then
```

```
        if (ENABLE = '1') then
            COUNT_UP <= COUNT_UP + "0001";
        end if;
    end if;
  end process UP_COUNTER;
  DOWN_COUNTER : process (CLOCK, ENABLE, GSR, COUNT_DOWN)
  begin
    if (GSR = '1' OR COUNT_DOWN = "0101") then
        COUNT_DOWN <= "1111";
    elsif (CLOCK'event AND CLOCK = '1') then
        if (ENABLE = '1') then
            COUNT_DOWN <= COUNT_DOWN - "0001";
        end if;
    end if;
  end process DOWN_COUNTER;
  CUP <= COUNT_UP;
  CDOWN <= COUNT_DOWN;
end A;
```

Progression of the ROC Model

The following figure shows the progression of the ROCBUF model and its interpretation in the four main design phases.

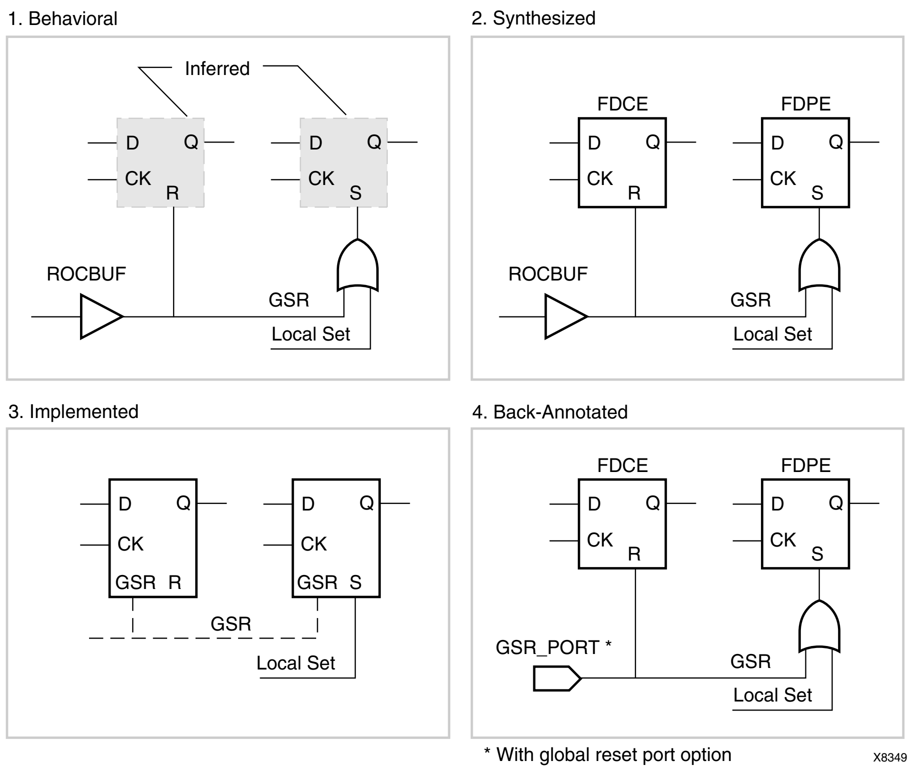

Figure 6-4: **ROCBUF Simulation and Implementation**

Behavioral Phase

In the behavioral phase, the behavioral or RTL description registers are inferred from the coding style, and the ROCBUF cell is instantiated. Use the ROCBUF cell instead of the ROC cell when you want test bench control of GSR simulation.

Synthesized Phase

In the synthesized phase, inferred registers are mapped to a technology and the ROCBUF instantiation is carried from the RTL to the implementation tools. As a result, consistent global set/reset behavior is maintained between the RTL and synthesized structural descriptions during simulation.

Implemented Phase

In the implemented phases, the ROCBUF is removed from the logical description of the design and the global resources are used for the set/reset function.

Back-Annotated Phase

In the back-annotated phase, use the NetGen option **`-gp`** to replace the port that was previously occupied by the ROCBUF in the RTL description of the design. A non-hierarchical netlist will have this port drive the fully wired GSR signal. For hierarchical designs, the GSR net is not wired across hierarchical modules. The VHDL global signal, X_GSR_GLOBAL_SIGNAL, is used to connect the top level port created by the **`-gp`** switch to the GSR signals of the hierarchical modules. Each hierarchical module contains an X_ROCBUF cell, which is driven by X_GSR_GLOBAL_SIGNAL. The output of the X_ROCBUF is connected to the GSR signal of the hierarchical module. Toggling the input to the top level port affects the entire design. The X_GSR_GLOBAL_SIGNAL global signal is defined in a package within the VHDL netlist. The package name is *design_name*_ROCTOC.

If desired, each of the hierarchical modules can be written out as a separate netlist using the NetGen –mhf switch to create multiple hierarchical files. This helps in analyzing lower level modules individually. Every netlist contains the *design_name*_ROCTOC package definition with X_GSR_GLOBAL_SIGNAL signal. When lower modules are simulated, though X_ROCBUF is connected to X_GSR_GLOBAL_SIGNAL, there will be no event on it. Hence the X_ROCBUF creates a pulse similar to X_ROC and the GSR of the module is toggled. The ROC pulse width of the X_ROCBUF component can be controlled using the –rpw switch for NetGen.

If all the modules, including the top level design module, are compiled and analyzed together, the top level port created via the -gp switch would drive the X_GSR_GLOBAL_SIGNAL and all X_ROCBUF cells. In this situation, the X_ROCBUF component does not create a pulse, but rather behaves like a buffer passing on the value of the X_GSR_GLOBAL_SIGNAL. The ROC pulse width of the X_ROCBUF component can be controlled using the -rpw switch for NetGen.

Simulating Special Components in VHDL

The following section provides a description and examples of using special components, such as the Block SelectRAM™ for Virtex.

Simulating CORE Generator Components in VHDL

For CORE Generator model simulation flows, see the *CORE Generator Guide*.

Differential I/O (LVDS, LVPECL)

When you target a Virtex-E or Spartan-IIE device, the inputs of the differential pair are currently modeled with only the positive side. In contrast, the outputs have both pairs, positive and negative. For more information, see Xilinx Answer Record 8187, “*Virtex-E, Spartan-IIE LVDS/LVPECL - How do I use LVDS/LVPECL I/O standards?*”

This is not an issue for Virtex-II, Virtex-II Pro, Virtex-II Pro X, or Spartan-3, since the differential buffers for Virtex-II and later architectures now accept both the positive and negative inputs. For newer devices, instantiate either an IBUFDS or IBUFGDS and connect and simulate normally. Instantiation templates for these components can be found in the ISE Project Navigator HDL Templates or the appropriate Xilinx *HDL Libraries Guide.*

Code Example

The following is an example of an instantiated differential I/O in a Virtex-E or Spartan-IIE design.

```
entity lvds_ex is
port (
        data: in std_logic;
        data_op: out std_logic;
        data_on: out std_logic
        );
end entity lvds_ex;
architecture lvds_arch of lvds_ex is
  signal data_n_int : std_logic;
  component OBUF_LVDS port (
        I : in std_logic;
        O : out std_logic
        );
  end component;
  component IBUF_LVDS port (
        I : in std_logic;
        O : out std_logic
        );
  end component;
begin
--Input side
  I0: IBUF_LVDS port map (I => data), O =>data_int);
--Output side
  OP0: OBUF_LVDS port map (I => data_int, O => data_op);
  data_n_int = not(data_int);
  ON0: OBUF_LVDS port map (I => data_n_int, O => data_on);
end arch_lvds_ex;
```

Simulating Verilog

This section discusses simulation in Verilog.

Defining Global Signals in Verilog

The global set/reset and global 3-state signals are defined in the `$XILINX/verilog/src/glbl.v` module. The VHDL UNISIM library contains the ROC, ROCBUF, TOC, TOCBUF, and STARTBUF cells to assist in VITAL VHDL simulation of the global set/reset and 3-state signals. However, Verilog allows a global signal to be modeled as a wire in a global module, and thus, does not contain these cells.

Using the glbl.v Module

The `glbl.v` module connects the global signals to the design, which is why it is necessary to compile this module with the other design files and load it along with the *`design`*`.v` file and the *`testfixture`*`.v` file for simulation.

Defining GSR/GTS in a Test Bench

In most cases, GSR and GTS need not be defined in the test bench. The `glbl.v` file declares the global GSR and GTS signals and automatically pulses GSR for 100 ns. This is all that is necessary for back-end simulations and is generally all that is necessary for functional simulations. If GSR or GTS needs to be emulated in the functional simulation, then it is necessary to add GSR and GTS to the test bench and connect them to the user defined global signals. This is discussed in more detail in the following section.

Note: The terms "test bench" and "test fixture" are used synonymously throughout this guide.

Emulating the Global GSR in a Verilog Functional Simulation

Many HDL designs that target Xilinx FPGA devices have a user reset that initializes all registers in the design during the functional simulation. For these designs, it is not necessary to emulate the GSR pulse that occurs after the device is configured. If the design contains registers that are not connected to a user reset, the GSR pulse can be emulated to ensure that the functional simulation matches the timing simulation.

In the design code, declare a GSR as a Verilog wire. The GSR is not specified in the port list for the module. Describe the GSR to reset or set every inferred register or latch in your design. GSR need not be connected to any instantiated registers or latches, as the UNISIM models for these components connect to GSR directly. This is shown in the following example.

Code Example

```
module my_counter (CLK, D, Q, COUT);
input CLK, D;
output Q;
output [3:0] COUT;

wire GSR;
reg [3:0] COUT;

always @(posedge GSR or posedge CLK)
  begin
    if (GSR == 1’b1)
        COUT = 4’h0;
    else
        COUT = COUT + 1’b1;
  end
// GSR is modeled as a wire within a global module.
// So, CLR does not need to be connected to GSR and
// the flop will still be reset with GSR.
FDCE U0 (.Q (Q), .D (D), .C (CLK), .CE (1'b1), .CLR (1’b0));
endmodule
```

Since GSR is declared as a floating wire and is not in the port list, the synthesis tool optimizes the GSR signal out of the design. GSR is replaced later by the implementation software for all post-implementation simulation netlists.

Simulating Special Components in Verilog

The following section provides descriptions and examples of simulating special components for Virtex.

Defparam Support Considerations

Some synthesis tools do not support the use of defparams to attach attributes. If your synthesis tool does not support defparams, use the special meta comment to make the code visible only to the simulator. Place the meta comments immediately before and after the defparam declarations and mappings as follows:

```
// synthesis translate_off
   defparam U0.INIT = 2'b01;
// synthesis translate_off
```

The attributes can then be passed to the implementation tools by defining them in the UCF file. Alternatively, the synthesis tool may support a mechanism to pass these attributes directly from the Verilog file without using the generics. For more information on attribute passing, see your synthesis tool documentation.

Differential I/O (LVDS, LVPECL)

For Virtex-E and Spartan-IIE families, the inputs of the differential pair are currently modeled with only the positive side, whereas the outputs have both pairs, positive and negative. For more information, see Xilinx Answer Record 8187, "*Virtex-E, Spartan-IIE LVDS/LVPECL - How do I use LVDS/LVPECL I/O standards?*"

This is not an issue for Virtex-II, Virtex-II Pro, Virtex-II Pro X, or Spartan-3, since the differential buffers for Virtex-II and later architectures now accept both the positive and negative inputs. For newer devices, instantiate either an IBUFDS or IBUFGDS and connect and simulate normally. Instantiation templates for these components can be found in the ISE Project Navigator HDL Templates or the appropriate Xilinx *Libraries Guide*.

The following is an example of an instantiated differential I/O in a Virtex-E or Spartan-IIE design.

```
module lvds_ex (data, data_op, data_on);
  input data;
  output data_op, data_on;

// Input side
  IBUF_LVDS I0 (.I (data), .O (data_int));

// Output side
  OBUF_LVDS OP0 (.I (data_int), .O (data_op));
  wire data_n_int = ~data_int;
  OBUF_LVDS ON0 (.I (data_n_int), .O (data_on));

endmodule
```

Simulation CORE Generator Components

The simulation flow for CORE Generator models is described in the *CORE Generator Guide*.

Design Hierarchy and Simulation

Most FPGA designs are partitioned into levels of hierarchy. This section discusses design hierarchy and simulation.

Advantages of Hierarchy

Hierarchy:

- Makes the design easier to read
- Makes the design easier to re-use
- Allows partitioning for a multi-engineer team
- Improves verification

Improving Design Utilization and Performance

To improve design utilization and performance, the synthesis tool or the Xilinx implementation tools often flatten or modify the design hierarchy. After this flattening and restructuring of the design hierarchy in synthesis and implementation, it may become impossible to reconstruct the hierarchy. As a result, much of the advantage of using the original design hierarchy in RTL verification is lost in back-end verification. In order to improve visibility of the design for back-end simulation, the Xilinx design flow allows for retention of the original design hierarchy with this described methodology.

To allow preservation of the design hierarchy through the implementation process with little or no degradation in performance or increase in design resources, stricter design rules should be followed and design hierarchy should be carefully selected so that optimization is not necessary across the design hierarchy.

Good Design Practices

Some good design practices to follow are:

- Register all outputs exiting a preserved entity or module.
- Do not allow critical timing paths to span multiple entities or modules.
- Keep related or possibly shared logic in the same entity or module.
- Place all logic that is to be placed or merged into the I/O (such as IOB registers, three state buffers, and instantiated I/O buffers) in the top-level module or entity for the design. This includes double-data rate registers used in the I/O.
- Manually duplicate high-fanout registers at hierarchy boundaries if improved timing is necessary.

Maintaining the Hierarchy

To maintain the entire hierarchy (or specified parts of the hierarchy) during synthesis, the synthesis tool must first be instructed to preserve hierarchy for all levels (or for each selected level of hierarchy). This may be done with:

- a global switch
- a compiler directive in the source files
- a synthesis command.

For more information on how to retain hierarchy, see your synthesis tool documentation.

After taking the necessary steps to preserve hierarchy, and properly synthesizing the design, the synthesis tool creates a hierarchical implementation file (EDIF or NGC) that retains the hierarchy.

Using the KEEP_HIERARCHY Constraint

Before implementing the design with the Xilinx software, place a KEEP_HIERARCHY constraint on each instance in the design in which the hierarchy is to be preserved. This tells the Xilinx software which parts of the design should not be flattened or modified to maintain proper hierarchy boundaries.

This constraint may be passed in the source code as an attribute, as an instance constraint in the NCF or UCF file, or may be automatically generated by the synthesis tool. For more information on this process, see your synthesis tool documentation. For more information on the KEEP_HIERARCHY constraint, see the Xilinx *Constraints Guide*.

After the design is mapped, placed, and routed, run NetGen using the following parameters to properly back-annotate the hierarchy of the design.

```
netgen -sim -ofmt {vhdl|verilog}design_name.ncd netlist_name
```

This is the NetGen default when you use ISE or XFLOW to generate the simulation files. It is only necessary to know this if you plan to execute NetGen outside of ISE or XFLOW, or if you have modified the default options in ISE or XFLOW. When you run NetGen in the preceding manner, all hierarchy that was specified to KEEP_HIERARCHY is reconstructed in the resulting VHDL or Verilog netlist.

NetGen can write out a separate netlist file and SDF file for each level of preserved hierarchy. This capability allows for full timing simulation of individual portions of the design, which in turn allows for:

- greater test bench re-use
- team-based verification methods
- the potential for reduced overall verification times

Use the **–mhf** switch to produce individual files for each KEEP_HIERARCHY instance in the design. You can also use the **–mhf** switch together with the **–dir** switch to place all associated files in a separate directory.

```
netgen -sim -ofmt {vhdl|verilog} -mhf -dir
        directory_name design_name.ncd
```

When you run NetGen with the **–mhf** switch, NetGen produces a text file called `design_mhf_info.txt`. This file lists all produced module/entity names, their associated instance names, SDF files, and sub modules. This file is useful for determining proper simulation compile order, SDF annotation options, and other information when you use one or more of these files for simulation.

Example File

The following is an example of an `mhf_info.txt` file for a VHDL produced netlist:

```
// Xilinx design hierarchy information file produced by netgen (I.23)
// The information in this file is useful for
//    - Design hierarchy relationship between modules
//    - Bottom up compilation order (VHDL simulation)
//    - SDF file annotation (VHDL simulation)
//
//  Design Name : stopwatch
//
//  Module      : The name of the hierarchical design module.
//  Instance    : The instance name used in the parent module.
//  Design File : The name of the file that contains the module.
//  SDF File    : The SDF file associated with the module.
//  SubModule   : The sub module(s) contained within a given module.
//     Module, Instance : The sub module and instance names.

  Module      : hex2led_1
  Instance    : msbled
  Design File : hex2led_1_sim.vhd
  SDF File    : hex2led_1_sim.sdf
  SubModule   : NONE

  Module      : hex2led
  Instance    : lsbled
  Design File : hex2led_sim.vhd
  SDF File    : hex2led_sim.sdf
  SubModule   : NONE

  Module      : smallcntr_1
  Instance    : lsbcount
  Design File : smallcntr_1_sim.vhd
  SDF File    : smallcntr_1_sim.sdf
  SubModule   : NONE

  Module      : smallcntr
  Instance    : msbcount
  Design File : smallcntr_sim.vhd
  SDF File    : smallcntr_sim.sdf
  SubModule   : NONE

  Module      : cnt60
  Instance    : sixty
  Design File : cnt60_sim.vhd
  SDF File    : cnt60_sim.sdf
  SubModule   : smallcntr, smallcntr_1
       Module : smallcntr, Instance : msbcount
       Module : smallcntr_1, Instance : lsbcount

  Module      : decode
  Instance    : decoder
  Design File : decode_sim.vhd
  SDF File    : decode_sim.sdf
  SubModule   : NONE

  Module      : dcm1
  Instance    : Inst_dcm1
```

```
Design File : dcm1_sim.vhd
SDF File    : dcm1_sim.sdf
SubModule   : NONE

Module      : statmach
Instance    : MACHINE
Design File : statmach_sim.vhd
SDF File    : statmach_sim.sdf
SubModule   : NONE

Module      : stopwatch
Design File : stopwatch_timesim.vhd
SDF File    : stopwatch_timesim.sdf
SubModule   : statmach, dcm1, decode, cnt60, hex2led, hex2led_1
     Module : statmach, Instance : MACHINE
     Module : dcm1, Instance : Inst_dcm1
     Module : decode, Instance : decoder
     Module : cnt60, Instance : sixty
     Module : hex2led, Instance : lsbled
     Module : hex2led_1, Instance : msbled
```

Note: Hierarchy created by generate statements may not match the original simulation due to naming differences between the simulator and synthesis engines for generated instances.

RTL Simulation Using Xilinx Libraries

Since Xilinx simulation libraries are VHDL-93 and Verilog-2001 compliant, they can be simulated using any simulator that supports these language standards. However, certain delay and modelling information is built into the libraries, which is required to correctly simulate the Xilinx hardware devices.

Xilinx recommends not changing data signals at clock edges, even for functional simulation. The simulators add a unit delay between the signals that change at the same simulator time. If the data changes at the same time as a clock, it is possible that the data input will be scheduled by the simulator to occur after the clock edge. Thus, the data will not go through until the next clock edge, although it is possible that the intent was to have the data get clocked in before the first clock edge. To avoid any such unintended simulation results, Xilinx recommends not switching data signals and clock signals simultaneously.

Simulating Certain Xilinx Components

This section discusses Xilinx Block RAM (RAMB4/RAMB16) models. The UNISIM dual-port block RAM models have a built-in collision checking function that monitors reads and writes to each port, and reports violations if data is improperly handled for the RAM. While this is reported similarly to a timing violation, in reality, it is warning of a potential functionality issue for the design. If you receive any collision warnings from the UNISIM dual-port block RAM models during functional simulation, you should investigate the warning to ensure it will not have any negative impact on the end design functionality. Generally, this is best handled either by trying to re-code to avoid such errors or by ensuring that the data written or received from the RAM will be discarded as the actual value cannot be determined and therefore should not be used.

CLKDLL, DCM and DCM_ADV

This section discusses the following:

- "CLKDLL/DCM Clocks Do Not Appear De-Skewed"
- "TRACE/Simulation Model Differences"
- "Non-LVTTL Input Drivers"
- "Viewer Considerations"
- "Attributes for Simulation and Implementation"
- "Simulating the DCM in Digital Frequency Synthesis Mode Only"
- "JTAG / BSCAN (Boundary Scan) Simulation"

CLKDLL/DCM Clocks Do Not Appear De-Skewed

The CLKDLL and DCM components remove the clock delay from the clock entering into the chip. As a result, the incoming clock and the clocks feeding the registers in the device have a minimal skew within the range specified in the databook for any given device. However, in timing simulation, the clocks may not appear to be de-skewed within the range specified. This is due to the way the delays in the SDF file are handled by some simulators.

The SDF file annotates the CLOCK PORT delay on the X_FF components. However, some simulators may show the clock signal in the waveform viewer before taking this delay into account. If it appears that the simulator is not properly de-skewing the clock, see your synthesis tool documentation to find out if it is not displaying the input port delays in the waveform viewer at the input nodes. If this is the case, then when the CLOCK PORT delay on the X_FF is added to the internal clock signal, it should line up within the device specifications in the waveform viewer with the input port clock. The simulation is still functioning properly, the waveform viewer is just not displaying the signal at the expected node. To verify that the CLKDLL/DCM is functioning correctly, delays from the SDF file may need to be accounted for manually to calculate the actual skew between the input and internal clocks.

TRACE/Simulation Model Differences

To fully understand the simulation model, you must first understand that there are differences in the way the DLL/DCM is built in silicon, the way TRACE reports their timing, and the way the DLL/DCM is modeled for simulation. The DLL/DCM simulation model attempts to replicate the functionality of the DLL/DCM in the Xilinx silicon, but it does not always do it exactly how it is implemented in the silicon. In the silicon, the DLL/DCM uses a tapped delay line to delay the clock signal. This accounts for input delay paths and global buffer delay paths to the feedback in order to accomplish the proper clock phase adjustment. TRACE or Timing Analyzer reports the phase adjustment as a simple delay (usually negative) so that you can adjust the clock timing for static timing analysis.

As for simulation, the DLL/DCM simulation model itself attempts to align the input clock to the clock coming back into the feedback input. Instead of putting the delay in the DLL or DCM itself, the delays are handled by combining some of them into the feedback path as clock delay on the clock buffer (component) and clock net (port delay). The remainder is combined with the port delay of the CLKFB pin. While this is different from the way TRACE or Timing Analyzer reports it, and the way it is implemented in the silicon, the end result is the same functionality and timing. TRACE and simulation both use a simple delay model rather than an adjustable delay tap line similar to silicon.

The primary job of the DLL/DCM is to remove the clock delay from the internal clocking circuit as shown in the following figure.

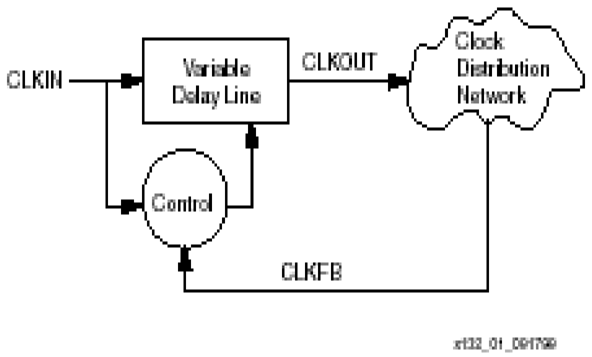

Figure 6-5: **Delay Locked Loop Block Diagram**

Do not confuse this with the function of de-skewing the clock. Clock skew is generally associated with delay variances in the clock tree, which is a different matter. By removing the clock delay, the input clock to the device pin should be properly phase aligned with the clock signal as it arrives at each register it is sourcing. This means that observing signals at the DLL/DCM pins generally does not give the proper view point to observe the removal of the clock delay. The place to see if the DCM is doing its job is to compare the input clock (at the input port to the design) with the clock pins of one of the sourcing registers. If these are aligned (or shifted to the desired amount) then the DLL/DCM has accomplished its job.

Non-LVTTL Input Drivers

When using non-LVTTL input buffer drivers to drive the clock, the DCM does not make adjustments as to the type of input buffer chosen, but instead has a single delay value to provide the best amount of clock delay across all I/O standards. If you are using the same input standard for the data, the delay values should track, and generally not cause a problem. Even if you are not using the same input standard, the amount of delay variance generally does not cause hold time failures because the delay variance is small compared to the amount of input delay. The delay variance is calculated in both static timing analysis and simulation so you should see proper setup time values during static timing analysis, as well as during simulation.

Viewer Considerations

Depending on which simulator you use, the waveform viewer might not depict the delay timing the way you expect to see it. Some simulators, including the current version of MTI ModelSim, combine interconnect delays (either interconnect or port delays) with the input pins of the component delays when you view the waveform on the waveform viewer. In terms of the simulation, the results are correct, but in terms of what you see in the waveform viewer, this may not always be what you expect to see.

Since interconnect delays are combined, when you look at a pin using the MTI ModelSim viewer, you do not see the transition as it happens on the pin. In terms of functionality, the simulation acts properly, and this is not very relevant, but when attempting to calculate clock delay, the interconnect delays before the clock pin must be taken into account if the simulator you are using combines these interconnect delays with component delays. For more information, see Xilinx Answer Record 11067, *"ModelSim Simulations: Input and Output clocks of the DCM and CLKDLL models do not appear to be de-skewed (VHDL, Verilog)."*

Attributes for Simulation and Implementation

Make sure that the same attributes are passed for simulation and implementation. During implementation of the design, DLL/DCM attributes may be passed either by the synthesis tool (via a synthesis attribute, generic or defparam declaration), or within the UCF file. For RTL simulation of the UNISIM models, the simulation attributes must be passed via a generic (VHDL), or inline parameters (Verilog).

If you do not use the default setting for the DLL/DCM, make sure that the attributes for RTL simulation are the same as those used for implementation. If not, there may be differences between RTL simulation and the actual device implementation.

To make sure that the attributes passed to implementation are the same as those used for simulation, use the generic mapping method (VHDL) or inline parameter passing (Verilog), provided your synthesis tool supports these methods for passing functional attributes.

Simulating the DCM in Digital Frequency Synthesis Mode Only

To simulate the DCM in Digital Frequency Synthesis Mode only, set the CLK_FEEDBACK attribute to NONE and leave the CLKFB unconnected. The CLKFX and CLKFX180 are generated based on CLKFX_MULTIPLY and CLKFX_DIVIDE attributes. These outputs do not have phase correction with respect to CLKIN.

JTAG / BSCAN (Boundary Scan) Simulation

Simulation of the BSCAN component is now supported for the Virtex-4 architecture. The simulation supports the interaction of the JTAG ports and some of the JTAG operation commands. Full support of the JTAG interface, including interface to the scan chain, is planned for a future release, but is not currently supported. In order to simulate this interface:

1. Instantiate the BSCAN_VIRTEX4 component and connect it to the design.
2. Instantiate the JTAG_SIM_VIRTEX4 component into the test bench (not the design).

This becomes:

- the interface to the external JTAG signals (such as TDI, TDO, and TCK).
- the communication channel to the BSCAN component

The communication between the components takes place in the VPKG VHDL package file or the glbl Verilog global module. Accordingly, no implicit connections are necessary between the JTAG_SIM_VIRTEX4 component and the design or the BSCAN_VIRTEX4 symbol.

Stimulus can be driven and viewed from the JTAG_SIM_VIRTEX4 component within the test bench to understand the operation of the JTAG/BSCAN function. Instantiation templates for both of these components are available in both the ISE HDL Templates in Project Navigator and the *Virtex-4 Libraries Guide.*

Timing Simulation

In back annotated (timing) simulation, the introduction of delays can cause the behavior to be different from what is expected. Most problems are caused due to timing violations in the design, and are reported by the simulator. However, there are a few other situations that can occur.

Glitches in Your Design

When a glitch (small pulse) occurs in an FPGA circuit or any integrated circuit, the glitch may be passed along by the transistors and interconnect (transport) in the circuit, or it may be swallowed and not passed (internal) to the next resource in the FPGA. This depends on the width of the glitch and the type of resource the glitch passes through. To produce more accurate simulation of how signals are propagated within the silicon, Xilinx models this behavior in the timing simulation netlist.

For VHDL simulation, there are two library components (X_BUF_PP and X_TRI_PP) in which proper values are annotated for pulse rejection in the simulation netlist. The result of these constructs in the simulation netlists is a more true-to-life simulation model, and therefore a more accurate simulation.

For Verilog simulation, this information is passed by the PATHPULSE construct in the SDF file. This construct is used to specify the size of pulses to be rejected or swallowed on components in the netlist.

Debugging Timing Problems

In back-annotated (timing) simulation, the simulator takes into account timing information that resides in the standard delay format (SDF) file. This may lead to eventual timing violations issued by the simulator if the circuit is operated too fast or has asynchronous components in the design. This section explains some of the more common timing violations, and gives advice on how to debug and correct them.

Identifying Timing Violations

After you run timing simulation, check the messages generated by your simulator. If you have timing violations, they are indicated by warning or error messages.

The following example is a typical setup violation message from MTI ModelSim for a Verilog design. Message formats vary from simulator to simulator, but all contain the same basic information. For more information, see your simulator tool documentation.

```
# ** Error:/path/to/xilinx/verilog/src/simprims/X_RAMD16.v(96):
$setup(negedge WE:29138 ps, posedge CLK:29151 ps, 373 ps);
# Time:29151 ps Iteration:0 Instance: /test_bench/u1/\U1/X_RAMD16\
```

Line One

```
# ** Error:/path/to/xilinx/verilog/src/simprims/X_RAMD16.v(96):
```

Line One points to the line in the simulation model that is in error. In this example, the failing line is line 96 of the Verilog file, X_RAMD16.

Line Two

```
$setup(negedge WE:29138 ps, posedge CLK:29151 ps, 373 ps);
```

Line Two gives information about the two signals that caused the error:

- The type of violation (for example, $setup, $hold, or $recovery). This example is a $setup violation.
- The name of each signal involved in the violation, followed by the simulation time at which that signal last changed values. In this example, the failing signals are the negative-going edge of the signal WE, which last changed at 29138 picoseconds, and the positive-going edge of the signal CLK, which last changed at 29151 picoseconds.
- The allotted amount of time for the setup. In this example, the signal on WE should be stable for 373 pico seconds before the clock transitions. Since WE changed only 13 pico seconds before the clock, this violation was reported.

Line Three

```
# Time:29151 ps Iteration:0 Instance: /test_bench/u1/\U1/X_RAMD16\
```

Line Three gives the simulation time at which the error was reported, and the instance in the structural design (time_sim) in which the violation occurred.

Verilog System Timing Tasks

Verilog system tasks and functions are used to perform simulation related operations such as monitoring and displaying simulation time and associated signal values at a specific time during simulation. All system tasks and functions begin with a dollar sign, for example **$setup**. For information about specific system tasks, see the *Verilog Language Reference Manual* available from IEEE.

Timing check tasks may be invoked in specific blocks to verify the timing performance of a design by making sure critical events occur within given time limits. Timing checks perform the following steps:

- Determine the elapsed time between two events.
- Compare the elapsed time to a specified limit.
- If the elapsed time does not fall within the specified time window, report timing violation.

The following system tasks may be used for performing timing checks:

$hold	$setup
$nochange	$setuphold
$period	$skew
$recovery	$width

VITAL Timing Checks

VITAL (VHDL Initiative Towards ASIC Libraries) is an addition to the VHDL specification that deals with adding timing information to VHDL models. One of the key aspects of VITAL is the specification of the package vital_timing. This package includes standard procedures for performing timing checks.

The package vital_timing defines the following timing check procedures:

- VitalSetupHoldCheck
- VitalRecoveryRemovalCheck
- VitalInPhaseSkewCheck
- VitalOutPhaseSkewCheck
- VitalPeriodPulseCheck.

VitalSetupHoldCheck is overloaded for use with test signals of type Std_Ulogic or Std_Logic_Vector. Each defines a CheckEnabled parameter that supports the modeling of conditional timing checks. For more information about specific VITAL timing checks, see the *VITAL Language Reference Manual* available from IEEE.

Timing Problem Root Causes

Timing violations, such as **`$setuphold`**, occur any time data changes at a register input (either data or clock enable) within the setup or hold time window for that particular register. There are a few typical causes for timing violations. The most common are:

- "Design Not Constrained"
- "Path Not or Improperly Constrained"
- "Design Does Not Meet Timespec"
- "Simulation Clock Does Not Meet Timespec"
- "Unaccounted Clock Skew"
- "Asynchronous Inputs, Asynchronous Clock Domains, Crossing Out-of-Phase"

Design Not Constrained

Timing constraints are essential to help you meet your design goals and obtain the best implementation of your circuit. Global timing constraints cover most constrainable paths in a design. These global constraints cover clock definitions, input and output timing requirements, and combinatorial path requirements. Specify global constraints such as PERIOD, OFFSET_IN_BEFORE, and OFFSET_OUT_AFTER to match your simulation stimulus with the timespecs of the devices used in the design.

PERIOD

The PERIOD constraint can be quickly applied to a design. It also leads in the support of OFFSET, which you can use to specify your I/O timing. This works well for a single-clock design, or multi-clock design that is not multi-cycle.

FROM-TO

This constraint works well with more complicated timing paths. Designs that are multi-cycle or have paths that cross clock domains are better handled this way. For I/O, however, you must add/subtract the delay of the global buffer. Note that using an OFFSET before for input and an OFFSET after for output is supported without the need to specify a period, so you can use the advantages of both.

Additional Resources

For more information on constraining your design, see the following.

- The Xilinx *Constraints Guide* lists all of the Xilinx constraints along with explanations and guides to their usage. For more information on constraining timing to achieve optimum results, see the Xilinx *Constraints Guide*, "Timing Constraint Strategies."
- Timing & Constraints on the Xilinx Support website provides a wealth of material, including What's New, Latest Answers, Documentation, and FAQ.
- The Timing Improvement Wizard provides suggestions for improving failing paths, and can help you find answers to your specific timing questions. The Timing Improvement Wizard is available from the Problem Solvers section of the Xilinx Support website.

Path Not or Improperly Constrained

Unconstrained or improperly constrained data and clock paths are the most common sources of setup and hold violations. Because data and clock paths can cross domain boundaries, global constraints are not always adequate to ensure that all paths are constrained. For example, a global constraint, such as PERIOD, does not constrain paths that originate at an input pin, and data delays along these paths could cause setup violations.

Use Timing Analyzer to determine the length of an individual data or clock path. For input paths to the design, if the length of a data path minus the length of the corresponding clock path, plus any data delay, is greater than the clock period, a setup violation occurs.

```
clock period < data path - clock path + data delay value setup value for
register
```

For more information on constraining paths driven by input pins, see the Xilinx *Constraints Guide*, "Timing Constraint Strategies." For other constraints resources, see "Design Not Constrained" in this chapter.

Design Does Not Meet Timespec

Xilinx software enables you to specify precise timing requirements for your Xilinx FPGA designs. Specify the timing requirements for any nets or paths in your design. The primary way to specify timing requirements is to assign timing constraints. You can assign timing constraints in:

- the Xilinx Constraints Editor
- your synthesis tool
- the User Constraint File (UCF)

For more information on entering timing specifications, see the Xilinx *Development System Reference Guide*. For more information about the constraints you can use with your schematic entry software, see the Xilinx *Constraints Guide*.

Once you define timing specifications, use TRACE (Timing Report, Circuit Evaluator, and TSI Report) or Timing Analyzer to analyze the results of your timing specifications. Review the timing report carefully to make sure that all paths are constrained, and that the constraints are specified properly. Check for any error messages in the report.

If your design still does not meet timespec after you apply timing constraints, see your tool documentation for additional options to improve timing performance.

If these additional options do not sufficiently improve timing performance, you may need to edit your source code to reconfigure parts of your design. Re-structuring code can reduce the levels of logic without necessarily changing end functionality, thereby reducing timing delays.

Simulation Clock Does Not Meet Timespec

If the frequency of the clock specified during simulation is greater than the frequency of the clock specified in the timing constraints, this over-clocking can cause timing violations. For example, if the simulation clock has a frequency of 5 ns, and a PERIOD constraint is set at 10 ns, a timing violation can occur. This situation can also be complicated by the presence of DLL or DCM in the clock path.

This problem is usually caused either by an error in the test bench or either by an error in the constraint specification. Make sure that the constraints match the conditions in the test bench, and correct any inconsistencies. If you modify the constraints, re-run the design through place and route to make sure that all constraints are met.

Unaccounted Clock Skew

Clock skew is the difference between the amount of time the clock signal takes to reach the destination register, and the amount of time the clock signal takes to reach the source register. The data must reach the destination register within a single clock period plus or minus the amount of clock skew. Clock skew is generally not a problem when you use global buffers; however, clock skew can be a concern if you use the local routing network for your clock signals.

To determine if clock skew is the problem, run a setup test in TRACE and read the report. For directions on how to run a setup check, see the Xilinx *Development System Reference Guide*, "TRACE." For information on using Timing Analyzer to determine clock skew, see the Timing Analyzer help.

Clock skew is modeled in the simulation, but not in TRACE, unless you invoke TRACE using the **`-skew`** switch. Simulation results may not equal TRACE results if the skew is significant (as when a non-BUFG clock is used). To account for skew in TRACE, use the following command:

```
trce -skew
```

or set the following environment variable:

```
setenv XILINX_DOSKEWCHECK yes
```

If your design has clock skew, consider redesigning your path so that all registers are tied to the same global buffer. If that is not possible, consider using the USELOWSKEWLINES constraint to minimize skew. For more information on USELOWSKEWLINES, see the Xilinx *Constraints Guide*.

Do not use the XILINX_DOSKEWCHECK environment variable with PAR. If you have clocks on local routing, the PAR timing score may oscillate. This is because the timing score is a function of both a clock delay and the data delay, and attempts to make the data path faster may make the clock path slower, or vice versa. It should only be used within PAR on designs with paths that make use of global clock resources.

Asynchronous Inputs, Asynchronous Clock Domains, Crossing Out-of-Phase

Timing violations can be caused by data paths that:

- are not controlled by the simulation clock
- are not clock controlled at all
- cross asynchronous clock boundaries
- have asynchronous inputs
- cross data paths out of phase

Asynchronous Clocks

If the design has two or more clock domains, any path that crosses data from one domain to another can cause timing problems. Although data paths that cross from one clock domain to another are not always asynchronous, it is always best to be cautious.

Always treat the following as asynchronous:

- Two clocks with unrelated frequencies
- Any clocking signal coming from off-chip
- Any time a register’s clock is gated (unless extreme caution is used)

To see if the path in question crosses asynchronous clock boundaries, check the source code and the Timing Analyzer report. If your design does not allow enough time for the path to be properly clocked into the other domain, you may have to redesign your clocking scheme. Consider using an asynchronous FIFO as a better way to pass data from one clock domain to another.

Asynchronous Inputs

Data paths that are not controlled by a clocked element are asynchronous inputs. Because they are not clock controlled, they can easily violate setup and hold time specifications.

Check the source code to see if the path in question is synchronous to the input register. If synchronization is not possible, you can use the ASYNC_REG constraint to work around the problem. For more information, see “Using the ASYNC_REG Constraint” in this chapter.

Out of Phase Data Paths

Data paths can be clock controlled at the same frequency, but nevertheless can have setup or hold violations because the clocks are out of phase. Even if the clock frequencies are a derivative of each other, improper phase alignment could cause setup violations.

To see if the path in question crosses another path with an out of phase clock, check the source code and the Timing Analyzer report.

Debugging Tips

When you are faced with a timing violation, ask the following questions:

- Was the clock path analyzed by TRACE or Timing Analyzer?
- Did TRACE or Timing Analyzer report that the data path can run at speeds being clocked in simulation?
- Is clock skew being accounted for in this path delay?

- Does subtracting the clock path delay from the data path delay still allow clocking speeds?
- Will slowing down the clock speeds eliminate the $setup/$hold time violations?
- Does this data path cross clock boundaries (from one clock domain to another)? Are the clocks synchronous to each other? Is there appreciable clock skew or phase difference between these clocks?
- If this path is an input path to the device, does changing the time at which the input stimulus is applied eliminate the $setup/$hold time violations?

Based on the answers to these questions, you may need to make changes to your design or test bench to accommodate the simulation conditions.

Special Considerations for Setup and Hold Violations

This section discusses special considerations for setup and hold violations.

Zero Hold Time Considerations

While Xilinx data sheets report that there are zero hold times on the internal registers and I/O registers with the default delay and using a global clock buffer, it is still possible to receive a `$hold` violation from the simulator. This `$hold` violation is really a `$setup` violation on the register. However, in order to get an accurate representation of the CLB delays, part of the setup time must be modeled as a hold time.

Negative Hold Times

In older versions of Xilinx simulation models, negative hold times were truncated and specified as zero hold times. While this does not cause inaccuracies for simulation, it does reveal a more pessimistic model in terms of timing than is possible in the actual FPGA device. Therefore, this made it more difficult to meet stringent timing requirements.

Negative hold times are now specified in the timing models to provide a wider, yet more accurate representation, of the timing window. This is accomplished by combining the setup and hold parameters for the synchronous models into a single setuphold parameter in which the timing for the setup/hold window can be expressed. This should not change the timing simulation methodology in any way; however, when using Cadence NC-Verilog, there are no longer separate violation messages for setup and hold. They are now combined into a single setuphold violation.

RAM Considerations

Timing Violations

Xilinx devices contain two types of memories, block RAM and distributed RAM. Both block RAM and distributed RAM are synchronous elements when you write data to them, so the same precautions must be taken as with all synchronous elements to avoid timing violations. The data input, address lines, and enables, all must be stable before the clock signal arrives to guarantee proper data storage.

Collision Checking

Block RAMs also perform synchronous read operations. During a read cycle, the addresses and enables must therefore be stable before the clock signal arrives, or a timing violation may occur.

When you use distribute RAM or block RAM in dual-port mode, take special care to avoid memory collisions. A memory collision occurs when (a) one port is being written to, and (b) an attempt is made to either read or write to the other port at the same address at the same time (or within a very short period of time thereafter). The model warns you if a collision occurs.

If the RAM is being read on one port as it is being written to on the other, the model outputs an X value signifying an unknown output. If the two ports are writing data to the same address at the same time, the model can write unknown data into memory. Take special care to avoid this situation, as unknown results may occur. For the hardware documentation on collision checking, see "Design Considerations: Using Block SelectRAM Memory," in the *Virtex-II Platform FPGA User Guide*.

You can use the generic (VHDL) or parameter (Verilog) SIM_COLLISION_CHECK to disable these checks in the model. See "SIM_COLLISION_CHECK" in this chapter.

Hierarchy Considerations

It is possible for the top-level signals to switch correctly, keeping the setup and hold times accounted for, and at the same time, have an error reported at the lowest level primitive in the design. This can happen because as the signals travel down through the hierarchy to this low-level primitive, the delays they go through can reduce the differences between them to the point where they begin to violate the setup time.

To correct this problem:

1. Browse the design hierarchy, and add the signals of the instance reporting the error to the top-level waveform. Make sure that the setup time is actually being violated at the lower level.
2. Step back through the structural design until a link between an RTL (pre-synthesis) design path and this instance reporting the error can be determined.
3. Constrain the RTL path using timing constraints so that the timing violation no longer occurs. Usually, most implemented designs have a small percentage of unconstrained paths after timing constraints have been applied, and these are the ones where $setup and $hold violations generally occur.

The debugging steps for **`$hold`** violations and **`$setup`** violations are identical.

$Width Violations

The **`$width`** Verilog system task monitors the width of signal pulses. When the pulse width of a specific signal is less than that required for the device being used, the simulator issues a $width violation. Generally, **`$width`** violations are specified only for clock signals and asynchronous set or reset signals.

For more information on device switching characteristics, see the product data sheet. Find the minimum pulse width requirements, and make sure that the device stimulus conforms to these specifications.

$Recovery Violations

The **`$recovery`** Verilog system task specifies a time constraint between an asynchronous control signal and a clock signal (for example, between clearbar and the clock for a flip-flop). A **`$recovery`** violation occurs when a change to the signal occurs within the specified time constraint.

The **`$recovery`** Verilog system task checks for one of two dual-port block RAM conflicts:

- If both ports write to the same memory cell simultaneously, violating the clock-to-setup requirement, the data stored is invalid.
- If one port attempts to read from the same memory cell to which the other is simultaneously writing (also violating the clock setup requirement), the write will be successful, but the data read will be invalid.

Recovery tasks are also used to detect if an asynchronous set/reset signal is released just before a clock event occurs. If this happens, the result is similar to a setup violation, in that it is undetermined whether the new data should be clocked in or not.

Simulation Flows

When simulating, you may compile the Verilog source files in any order since Verilog is compile order independent. However, VHDL components must be compiled bottom-up due to order dependency. Xilinx recommends that you specify the test fixture file before the HDL netlist of your design, as in the following examples.

Xilinx recommends giving the name *testbench* to the main module in the test fixture file. This name is consistent with the name used by default in the ISE Project Navigator. If this name is used, no changes are necessary to the option in ISE in order to perform simulation from that environment.

ModelSim SE/PE/XE VHDL

The following is information regarding ModelSim SE/PE/XE VHDL.

Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using ModelSim SE/PE/XE VHDL. For instructions on compiling the Xilinx VHDL libraries, see "Compiling Xilinx Simulation Libraries (COMPXLIB)" in this chapter.

Depending on the makeup of the design (Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the command line.

1. Create working directory.
```
vlib work
```
2. Compile design files and workbench.
```
vcom lower_level_files.vhd top_level.vhd testbench.vhd
    (testbench_cfg.vhd)
```
3. Simulate design.
```
vsim testbench_cfg
```

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line:

1. Create working directory.
```
vlib work
```
2. Compile design files and workbench.
```
vcom design.vhd testbench.vhd [testbench_cfg.vhd]
```
3. Simulate design.
```
vsim -sdfmax instance_name=design.sdf testbench_cfg
```

VCS-MX VHDL

Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using VCS-MX VHDL. For instructions on compiling the Xilinx VHDL librarie, see "Compiling Xilinx Simulation Libraries (COMPXLIB)" in this chapter.

Depending on the makeup of the design (Xilinx instantiated components, or CORE Generator components), for RTL simulation, specify the following at the command line.

1. Create working directory.

   ```
   mkdir work
   ```

2. Compile design files and workbench.

   ```
   vhdlan work_macro1.vhd top_level.vhd testbench.vhd
      testbench_cfg.vhd
   ```

   ```
   scs testbench_cfg
   ```

3. Simulate design.

   ```
   scsim
   ```

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line:

1. Create working directory.

   ```
   mkdir work
   ```

2. Compile design files and workbench.

   ```
   vhdlan work_design.vhd testbench.vhd
   ```

   ```
   scs testbench
   ```

3. Simulate design.

   ```
   scsim -sdf testbench:design.sdf
   ```

NC-SIM VHDL

The following is information regarding NC-SIM VHDL.

Using Shared Precompiled Libraries

Simulation libraries have to be compiled to *compiled_lib_dir* before using NC-SIM VHDLL. For instructions on compiling the Xilinx VHDL librarie, see "Compiling Xilinx Simulation Libraries (COMPXLIB)" in this chapter. It is assumed that the proper mapping and setup files are present before simulation. If you are unsure whether the simulation is properly set up, see your simulation tool documentation.

Depending on the makeup of the design (Xilinx instantiated components, or CORE Generator components), for RTL simulation, specify the following at the command line.

1. Create a working directory.

   ```
   mkdir test
   ```

2. Compile design files and workbench.

   ```
   ncvhdl -work test testwork_macro1.vhd top_level.vhd testbench.vhd
        testbench_cfg.vhd
   ```

3. Elaborate the design at the proper scope.

```
ncelab testbench_cfg:A
```

4. Invoke the simulation.

```
ncsim testbench_cfg:A
```

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line:

1. Compile the SDF annotation file:

```
ncsdfc design.sdf
```

2. Create an SDF command file, `sdf.cmd`, with the following data in it:

```
COMPILED_SDF_FILE = design.sdf.X
SCOPE = uut,
MTM_CONTROL = 'MAXIMUM';
```

3. Create a working directory.

```
mkdir test
```

4. Compile design files and workbench.

```
ncvhdl -work test work_design.vhd testbench.vhd
```

5. Elaborate the design at the proper scope.

```
ncelab -sdf_cmd_file.cmd testbench_cfg:A
```

6. Invoke the simulation.

```
ncsim testbench_cfg:A
```

NC-SIM Verilog

There are two methods to run simulation with NC-SIM Verilog.

- "Using Library Source Files With Compile Time Options"
- "Using Shared Precompiled Libraries"

Using Library Source Files With Compile Time Options

Depending on the makeup of the design (Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the command line:

```
ncverilog +libext+.v -y $XILINX/verilog/src/unisims \
    testfixture.v design.v $XILINX/verilog/src/glbl.v
```

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line.

```
ncverilog -y $XILINX/verilog/src/simprims \
    +libext+.v testfixture.v time_sim.v $XILINX/verilog/src/glbl.v
```

Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using NC-Verilog. For instructions on compiling the Xilinx Verilog libraries, see "Compiling Xilinx Simulation Libraries (COMPXLIB)" in this chapter.

Depending on the makeup of the design (Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, edit the `hdl.var` and `cds.lib` files to specify the library mapping.

```
# cds.lib
```

```
DEFINE simprims_ver compiled_lib_dir/simprims_ver
DEFINE xilinxcorelib_ver compiled_lib_dir/xilinxcorelib_ver
DEFINE worklib worklib

# hdl.var
DEFINE VIEW_MAP ($VIEW_MAP, .v => v)
DEFINE LIB_MAP ($LIB_MAP, compiled_lib_dir/unisims_ver => unisims_ver)
DEFINE LIB_MAP ($LIB_MAP, compiled_lib_dir/simprims_ver => simprims_ver)
DEFINE LIB_MAP ($LIB_MAP, + => worklib)
// After setting up the libraries, now compile and simulate the design:

ncvlog -messages -update $XILINX/verilog/src/glbl.v testfixture.v design.v
ncelab -messages testfixture_name glbl
ncsim -messages testfixture_name
```

The –update option of Ncvlog enables incremental compilation.

For timing or post-NGDBuild simulation, use the SIMPRIM-based libraries. Specify the following at the command line:

```
ncvlog -messages -update $XILINX/verilog/src/glbl.v testfixture.v time_sim.v
ncelab -messages -autosdf testfixture_name glbl
ncsim -messages testfixture_name
```

VCS-MX Verilog

VCS and VCSi are identical except that VCS is more highly optimized, resulting in greater speed for RTL and mixed level designs. Pure gate level designs run with comparable speed. However, VCS and VCSi are guaranteed to provide exactly the same simulation results. VCSi is invoked using the **vcsi** command rather than the **vcs**. command.

There are two methods to run simulation with VCS-MX Verilog.

- "Using Library Source Files With Compile Time Options"
- "Using Shared Precompiled Libraries"

Using Library Source Files With Compile Time Options

Depending on the makeup of the design (Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the command line.

```
vcs -y $XILINX/verilog/src/unisims \
    +libext+.v $XILINX/verilog/src/glbl.v \
    -Mupdate -R testfixture.v design.v
```

For timing or post-NGDBuild, use the SIMPRIM-based libraries. Specify the following at the command line.

```
vcs +compsdf -y $XILINX/verilog/src/simprims \
    $XILINX/verilog/src/glbl.v \
    +libext+.v -Mupdate -R testfixture.v time_sim.v
```

The **-R** option automatically simulates the executable after compilation.

The **-Mupdate** option enables incremental compilation. Modules are recompiled for any of the following reasons:

1. Target of a hierarchical reference has changed.
2. A compile time constant, such as a parameter, has changed.

3. Ports of a module instantiated in the module have changed.
4. Module inlining. For example, a group of module definitions merging, internally in VCS into a larger module definition which leads to faster simulation. These affected modules are again recompiled. This is done only once.

Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using VCS/VCSi.For instructions on compiling the Xilinx Verilog libraries, see "Compiling Xilinx Simulation Libraries (COMPXLIB)" in this chapter.

Depending on the makeup of the design (Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the command line.

```
vcs -Mupdate -Mlib=compiled_dir/unisims_ver -y \
    $XILINX/verilog/src/unisims -Mlib=compiled_dir/simprims_ver -y \
    $XILINX/verilog/src/simprims \
    -Mlib=compiled_dir/xilinxcorelib_ver +libext+.v \
    $XILINX/verilog/src/glbl.v -R testfixture.v design.v
```

For timing or post-NGDBuild simulation, the SIMPRIM-based libraries are used. Specify the following at the command line.

```
vcs +compsdf -y $XILINX/verilog/src/simprims \
    $XILINX/verilog/src/glbl.v +libext+.v-Mupdate -R \
    testfixture.v time_sim.v
```

The **-R** option automatically simulates the executable after compilation. Finally, the **-Mlib=*compiled_lib_dir*** option provides VCS with a central place to look for the descriptor information before it compiles a module and a central place to get the object files when it links together the executable.

The **-Mupdat**e option enables incremental compilation. Modules are recompiled for any of the following reasons:

1. Target of a hierarchical reference has changed.
2. A compile time constant such as a parameter has changed.
3. Ports of a module instantiated in the module have changed.
4. Module inlining. For example, a group of module definitions merging, internally in VCS, into a larger module definition which leads to faster simulation. These affected modules are again recompiled. This is done only once.

ModelSim Verilog

There are two methods to run simulation with ModelSim Verilog.

- "Using Library Source Files With Compile Time Options"
- "Using Shared Precompiled Libraries"

Using Library Source Files With Compile Time Options

Depending on the makeup of the design (Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the ModelSim prompt:

```
set XILINX $env(XILINX)
vlog -y $XILINX/verilog/src/unisims \
    +libext+.v $XILINX/verilog/src/glbl.v -incr testfixture.v design.v
    vsim testfixture glbl
```

For timing or post-NGDBuild simulation, the SIMPRIM-based libraries are used. Specify the following at the ModelSim prompt:

```
vlog -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \
    +libext+.v testfixture.v time_sim.v -incr
    vsim testfixture glbl +libext+.v testfixture.v
```

The **-incr** option enables incremental compilation.

Using Shared Precompiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using ModelSim Vlog. For instructions on compiling the Xilinx Verilog libraries, see "Compiling Xilinx Simulation Libraries (COMPXLIB)" in this chapter.

Depending on the makeup of the design (Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the ModelSim prompt:

```
set XILINX $env(XILINX)
vlog $XILINX/verilog/src/glbl.v testfixture.v time_sim.v -incr
vsim -L unisims_ver -L simprims_ver -L xilinxcorelib_ver \
    testfixture glbl
```

For timing or post-NGDBuild simulation, the SIMPRIM-based libraries are used. Specify the following at the ModelSim prompt:

```
vlog $XILINX/verilog/src/glbl.v testfixture.v time_sim.v -incr
vsim -L simprims_ver testfixture glbl
```

The **-incr** option enables incremental compilation. The **-L** *compiled_lib_dir* option provides VSIM with a library to search for design units instantiated from Verilog.

IBIS I/O Buffer Information Specification (IBIS)

The Xilinx IBIS models provide information on I/O characteristics. In particular, I/O Buffer Information Specification (IBIS) models provide information about I/O driver and receiver characteristics without disclosing proprietary knowledge of the IC design (as unencrypted SPICE models do). However, there are some limitations on the information that IBIS models can provide. These are limitations imposed by the IBIS specification itself.

IBIS models can be used for the following:

1. Model best-case and worst-case conditions (best-case = strong transistors, low temperature, high voltage; worst-case = weak transistors, high temperature, low voltage). Best-case conditions are represented by the "fast/strong" model, while worst-

case conditions are represented by the "slow/weak" model. Typical behavior is represented by the "typical" model.

2. Model varying drive strength and slew rate conditions for Xilinx I/Os that support such variation.

IBIS ***cannot*** be used for any of the following:

1. Provide internal timing information (propagation delays and skew).
2. Model power and ground structures.
3. Model pin-to-pin coupling.
4. Provide detailed package parasitic information. Package parasitics are provided in the form of lumped RLC data. This is typically not a significant limitation, as package parasitics have an almost negligible effect on signal transitions.

The implications of (2) and (3) above are that ground bounce, power supply droop, and simultaneous switching output (SSO) noise CANNOT be simulated with IBIS models. To ensure that these effects do not harm the functionality of your design, Xilinx provides device/package-dependent SSO guidelines based on extensive lab measurements. The locations of these guidelines are:

- Virtex-II
  Virtex-II Platform FPGA User Guide, "Design Considerations"
- Virtex-II Pro
  Virtex-II Pro Platform FPGA User Guide, "PCB Design Considerations"
- Virtex, Virtex-E
  Xilinx Application Note XAPP133, "Using the Virtex Select I/O Resource"
- Spartan-II, Spartan-IIE
  Xilinx Application Note XAPP179, "Using Select I/O Interfaces in Spartan-II FPGA Devices"
- IBIS models for Xilinx devices.
- For more information about the IBIS specification, see the IBIS Home Page.
- The Xilinx IBIS models are available for download.

Chapter 7

Equivalency Checking

Information on equivalency checking is no longer included in the *Synthesis and Simulation Design Guide*. See the following web page for the latest information on running formal verification with Xilinx devices:

http://www.xilinx.com/xlnx/xil_tt_product.jsp?BV_UseBVCookie=yes&sProduct=formal