Synthesis and Simulation Design Guide

About This Manual

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

The design examples in this manual were created with Verilog and VHSIC Hardware Description Language (VHDL); compiled with various synthesis tools; and targeted for XC4000, Spartan, Spartan-II, Spartan-XL, Virtex, Virtex-E, Virtex-II and XC5200 devices. Xilinx equally endorses both Verilog and VHDL. VHDL may be more difficult to learn than Verilog and usually requires more explanation.

This manual 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. Refer to your synthesis tool's reference manuals and design methodology notes for additional information.

Before using this manual, you should be familiar with the operations that are common to all Xilinx software tools. These operations are covered in the *Quick Start Guide*.

Note This Xilinx software release is certified as Year 2000 compliant.

Manual Contents

This manual contains the following chapters:

- Chapter 1, "Introduction," provides a general overview of designing Field Programmable Gate Arrays (FPGAs) 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, “Architecture Specific HDL Coding Styles for XC4000XLA, Spartan, and Spartan-XL,” includes coding techniques to help you improve synthesis results.
- Chapter 5, “Architecture Specific HDL Coding Styles for Spartan-II, Virtex, Virtex-E, and Virtex-II,” includes coding techniques to help you use the latest Xilinx devices.
- Chapter 6, “Simulating Your Design,” describes simulation methods for verifying the function and timing of your designs.

Additional Resources

For additional information, go to `http://support.xilinx.com`. The following table lists some of the resources you can access from this Web site. You can also directly access these resources using the provided URLs.

Resource	Description/URL
Tutorials	Tutorials covering Xilinx design flows, from design entry to verification and debugging `http://support.xilinx.com/support/techsup/tutorials/index.htm`
Answers Database	Current listing of solution records for the Xilinx software tools Search this database using the search function at `http://support.xilinx.com/support/searchtd.htm`
Application Notes	Descriptions of device-specific design techniques and approaches `http://support.xilinx.com/apps/appsweb.htm`
Data Book	Pages from *The Programmable Logic Data Book*, which contain device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging `http://support.xilinx.com/partinfo/databook.htm`

Resource	Description/URL
Xcell Journals	Quarterly journals for Xilinx programmable logic users http://support.xilinx.com/xcell/xcell.htm
Technical Tips	Latest news, design tips, and patch information for the Xilinx design environment http://support.xilinx.com/support/techsup/journals/index.htm

Conventions

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

Typographical

The following conventions are used for all documents.

- `Courier font` indicates messages, prompts, and program files that the system displays.

  ```
  speed grade: - 100
  ```

- **`Courier bold`** indicates literal commands that you enter in a syntactical statement. However, braces "{ }" in Courier bold are not literal and square brackets "[ ]" in Courier bold are literal only in the case of bus specifications, such as bus [7:0].

  ```
  rpt_del_net=
  ```

  `Courier bold` also indicates commands that you select from a menu.

  `File` → **`Open`**

- *Italic font* denotes the following items.

  - Variables in a syntax statement for which you must supply values

    `edif2ngd` *`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 "[ ]" indicate an optional entry or parameter. However, in bus specifications, such as bus [7:0], they are required.

  edif2ngd *[option_name] design_name*

- Braces "{ }" enclose a list of items from which you must choose one or more.

  lowpwr ={on|off}

- A vertical bar "|" separates items in a list of choices.

  lowpwr ={on|off}

- A vertical ellipsis indicates repetitive material that has been omitted.

  ```
  IOB #1: Name = QOUT'
  IOB #2: Name = CLKIN'
  .
  .
  .
  ```

- A horizontal ellipsis "…." indicates that an item can be repeated one or more times.

  `allow block` *block_name loc1 loc2locn;*

Online Document

The following conventions are used for online documents.

- Red-underlined text indicates an interbook link, which is a cross-reference to another book. Click the red-underlined text to open the specified cross-reference.

- Blue-underlined text indicates an intrabook link, which is a cross-reference within a book. Click the blue-underlined text to open the specified cross-reference.

Contents

Chapter 5 Architecture Specific HDL Coding Styles for Spartan-II, Virtex, Virtex-E, and Virtex-II

Chapter 6 Simulating Your Design

Chapter 1

Introduction

This chapter provides a general overview of designing Field Programmable Gate Arrays (FPGAs) with HDLs and also includes installation requirements and instructions. It includes the following sections.

- "Architecture Support"
- "Overview of Hardware Description Languages"
- "Advantages of Using the Virtex-E FPGA Architecture"
- "Advantages of Using HDLs to Design FPGAs"
- "Designing FPGAs with HDLs"
- "Xilinx Internet Web Sites"

Architecture Support

The software supports the following architecture families in this release.

- Spartan™ / XL / II
- Virtex™ / E / II
- XC9500™ / XL / XV
- XC4000™ E / L / EX / XL / XLA
- XC3000™ A / L
- XC3100™ A / L
- XC5200™

Overview of Hardware Description Languages

Hardware Description Languages (HDLs) are used to describe the behavior and structure of system and circuit designs. This chapter includes a general overview of designing FPGAs with HDLs. HDL design examples are provided in subsequent chapters of this book, and design examples can be downloaded from the Xilinx web site. System requirements and installation instructions for designs available from the web are also provided in this chapter.

This chapter also includes a brief description of why using FPGAs is more advantageous than using ASICs for your design needs.

To learn more about designing FPGAs with HDLs, Xilinx recommends that you enroll in the appropriate training classes offered by Xilinx and by the vendors of synthesis software. An understanding of FPGA architecture allows you to create HDL code that effectively uses FPGA system features.

Before you start to create your FPGA designs, refer to the current version of the Quick Start Guide for Xilinx Alliance Series for a description of the design flow; installation information; and general information on the Xilinx tools.

For the latest information on Xilinx parts and software, visit the Xilinx web site at http://www.xilinx.com. On the Xilinx home page, click on Products. You can get answers to your technical questions from the Xilinx support web site at http://www.support.xilinx.com. On the support home page, click on Advanced Search to set up search criteria that match your technical questions. You can also download software service packs from www.support.xilinx.com. On the support home page, click on Software, and then Service Packs. Software documentation, tutorials, and design files, are available from the www.support.xilinx.com web site.

Advantages of Using the Virtex-E FPGA Architecture

Virtex-E devices are the most powerful and flexible devices offered from the Xilinx FPGA product lines. They deliver high performance and high capacity programmable logic solutions while reducing design time. One big advantage designers have with the Virtex-E device is that several other equipment manufacturers (OEMs) have

utilized the Virtex-E architecture specific algorithms in their synthesis tools. Synthesis tools such as Exemplar's *LeonardoSpectrum*, Synplicity's *Synplify* and Synopsys's *FPGA Express* and *FPGA Compiler II*. The intimate knowledge of the Virtex-E FPGA architecture each of these synthesis tools possess shortens the design cycle required to achieve performance goals.

Features of the Virtex-E FPGA.

- *Architectural Changes*—The heart of the new Virtex-E device is the configurable logic block (CLB) and its sub-unit known as a slice. Each CLB is made up logic cells (LC) which include a 4-input function generator, carry logic, and a storage element. There are four LCs per CLB organized in two similar slices. The "Virtex-E CLB" figure shows the structure of the CLB slices, and the "Virtex-E Slice" figure shows the slice in more detail.

  Output from the function generator in each LC drives both the CLB output and the D input of the flip-flop.

- *Increased Usage of the Function Generator(?)*—The FPGA synthesis tool writes a netlist in terms of the function generator, known as a look up table (LUT), the carry logic, and the storage elements. The LUT can be used as static RAM to supplement the memory available as Block RAM outside the CLB. The LUT can also provide a 16-bit shift register that is ideal for capturing high-speed or burst mode data. Using the 16-bit register you can store data in applications such as digital signal processing, multiplexers, or selected functions of up to nine inputs.

- *Increases in Silicon Efficiency*— Optimization of Place and Route have resulted in dramatic silicon efficiencies. The abundance of routing resources permits the Virtex-E family to accommodate even the largest and most complex designs, up to 3.2 million gates.

  V CCINT processing is 1.8v. VCCINT is the power supply voltage for the internal logic and memory for Virtex-E.

  The 0.18 mm design rules have resulted in smaller die, faster speed, and lower power consumption.

- *New I/O Standards Supported*—You can increase I/O performance up to 622 Mb/s by using source synchronous data transmission architectures. You can also increase the synchronous system performance up to 240 MHz by using the single-ended Select I/O

technology. New Differential I/O standards are supported, LVPECL, LVDS, and BLVDS, which use two pins per signal.

- *Faster Speed Rates*—Most signal pins for the new Virtex-E standards will achieve faster clock speeds. Virtex-E devices have up to 640 Kb of faster Block Select RAM. There are 8 DLLs with easier clock mirroring and 4x frequency multiplication than the original virtex. They can also achieve a higher performance to 311 MHz.

  I/O pins are 3v tolerant, and can be 5v tolerant with an external 100 watt resistor.

  Virtex-E FPGAs are SRAM-based, and are customized by loading configuration data into internal memory cells. Configuration data can be read from an external SPROM in master serial mode, or can be written into the FPGA in SelectMAP, slave serial, and JTAG modes. While performance is design dependent, many designs operate internally at speeds in excess of 133 MHz and can achieve over 311 MHz.

  For more compete details on the Virtex-E architecture, please refer to the “Virtex-E 1.8v Field Programmable Gate Arrays Datasheet” found on the Xilinx web site at www.xilinx.com.

Advantages of Using HDLs to Design FPGAs

Using HDLs to design high-density FPGAs is advantageous for the following reasons.

- *Top-Down Approach for Large Projects*—HDLs are used 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 the overall design plan is determined, 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 a design implemented with gates. This step

decreases design time by eliminating the need to define every gate. Synthesis to gates also reduces the number of errors that can occur during a manual translation of a hardware description to a schematic design. Additionally, you can apply the automation techniques used by the synthesis tool (such as machine encoding styles or automatic I/O insertion) during the optimization of your design to the original HDL code, resulting in greater efficiency.

- *Early Testing of Various Design Implementations*—HDLs allow you to test different implementations of your design early in the design flow. You can then use the synthesis tool to perform the logic synthesis and optimization into gates. Additionally, Xilinx FPGAs 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 FPGAs 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 FPGAs with HDLs

If you are more familiar with 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. You can 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; and the implementation tools. This section gives you some design hints to help you create FPGAs with HDLs.

Using Verilog

Verilog® is popular for synthesis designs because it is less verbose than traditional VHDL, and it is standardized as IEEE-STD-1364-95. It was not originally intended as an input to synthesis, and many Verilog constructs are not supported by synthesis software. The Verilog examples in this manual were tested and synthesized with current, commonly-used FPGA synthesis software. The coding strate-

gies presented in the remaining chapters of this manual can help you create HDL descriptions that can be synthesized.

Using 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 manual will work with most commonly used FPGA synthesis software. The coding strategies presented in the remaining chapters of this manual can help you create HDL descriptions that can be synthesized.

Comparing ASICs and FPGAs

Xilinx FPGAs are reprogrammable and when combined with an HDL design flow can greatly reduce the design and verification cycle seen with traditional ASICs.

Using Synthesis Tools

Most of the commonly-used FPGA synthesis tools have special optimization algorithms for Xilinx FPGAs. Constraints and compiling options perform differently depending on the target device. There are some commands and constraints in ASIC synthesis tools that do not apply to FPGAs and, if used, may adversely impact your results. You should understand how your synthesis tool processes designs before creating FPGA designs. Most FPGA synthesis vendors include information in their manuals specifically for Xilinx FPGAs.

Using FPGA System Features

You can improve device performance and area utilization by creating HDL code that uses FPGA system features, such as global reset, wide I/O decoders, and memory. FPGA system features are described in this manual.

Designing Hierarchy

Current HDL design methods are specifically written for ASIC designs. You can use some of these ASIC design methods when designing FPGAs; however, certain techniques may unnecessarily increase the number of gates or CLB levels. This Design Guide will train you in techniques for optional FPGA design methodlogies.

Design hierarchy is important in the implementation of an FPGA and also 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 current recommendations for 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 manual 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 and placement / routing tools. For more information, see “the chapter where this is explained”.

Xilinx Internet Web Sites

You can get product information and product support from the Xilinx internet web sites. Both sites are described in the following sections.

Xilinx World Wide Web Site

You can reach the Xilinx web site at http: / / www.xilinx.com. The following features can be accessed from the Xilinx web site.

- *Products* — You can find information about new Xilinx products that are being offered, as well as previously announced Xilinx products.
- *Service and Support* — You can jump to the xilinx technical support site by choosing Service and Support.

- *Xpresso Cafe* —You can purchase Xilinx software, hardware and software tool education classes through Xilinx and Xilinx distributors.

Technical Support Web Site

Answers to questions, tutorials, Application notes, software manuals and information on using Xilinx products can be found on the technical support web site. You can reach the support web site at http://www.xilinx.support.com. The following features can be accessed from the Xilinx support web site.

- *Troubleshoot* — You can do an advanced search on the answers database to troubleshoot questions or issues you have with your design.
- *Software* — You can download the latest software service packs, IP updates, and product information from the Xilinx support website.
- *Library* — You can view the Software manuals from this web site. The manuals are provided in both HTML, viewable through most HTML browsers, and PDF. The Databook, CORE Generator documentation and datasheets are also available.
- *Design* — You can find helpful application notes that illustrate specific design solutions and methodologies.
- *Services* — You can open a support case when you need to have information from a Xilinx technical support person. You can also find information about your hardware or software order.
- *Feedback* —We are always interested in how well we're serving our customers. You can let us know by filling out our customer service survey questionnaire.

You can contact Xilinx technical support and application support for additional information and assistance in the following ways.

Technical and Applications Support Hotlines

The telephone hotlines give you direct access to Xilinx Application Engineers worldwide. You can also e-mail or fax your technical questions to the same locations.

Table 1-1 Technical Support

Location	Telephone	Electronic Mail	Facsimile (Fax)
North America	1-800-255-7778	hotline@xilinx.com	1-408-879-4442
Japan	81-3-3297-9163	jhotline@xilinx.com	81-3-3297-0067
France	33-1-3463-0100	frhelp@xilinx.com	33-1-3463-0959
Germany	49- 89-93088-130	dlhelp@xilinx.com	49-89-904-4748
United Kingdom	44-1932-820821	ukhelp@xilinx.com	44-1932-828522
Corporate Switchboard	1-408-559-7778		

Note When e-mailing or faxing inquiries, provide your complete name, company name, and phone number. Also, provide a complete problem description including your design entry software and design stage.

Xilinx FTP Site

ftp://ftp.xilinx.com

The FTP site provides online access to automated tutorials, design examples, online documents, utilities, and published patches.

Vendor Support Sites

Vendor support for synthesis and verification products can be obtained at the following locations.

Table 1-2 Vendor Support Sites

Vendor Name - Product	Telephone	Electronic Mail	Web Site
Synopsys - XSI	1-800-245-8005	support_center@synopsys.com	www.synopsys.com
Cadence - Concept-HDL	1-877-237-4911	support@cadence.com	sourcelink.cadence.com

Table 1-2 Vendor Support Sites

Vendor Name - Product	Telephone	Electronic Mail	Web Site
Mentor Graphics	1-800-547-4303	support_net@xilinx.com	support-netweb.mentorg.com
Viewlogic	1-800-223-8439	WVOffice: pc-support@view-logic.com PowerView: pv-support@view-logic.com	www.view-logic.com
Synopsys FPGA Express.	1-800-445-1888	support_center@synopsys.com	www.synopsys.com
Synplicity	1-408-548-6000	support@synplicity.com	www.synplicity.com
ModelSim	N / A	support@model.com	www.model.com
Exemplar	1-408-487-7410	support@exem-plar.com	www.exem-plar.com

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. If necessary, refer to the current version of the *Quick Start Guide for the Xilinx Alliance Series* to familiarize yourself with the Xilinx and interface tools. This chapter includes the following sections.

- "Design Flow"
- "Entering your Design and Selecting Hierarchy"
- "Functional Simulation of your Design"
- "Synthesizing and Optimizing your Design"
- "Setting Constraints"
- "Evaluating Design Size and Performance"
- "Evaluating your Design for Coding Style and System Features"
- "Placing and Routing Your Design"
- "Timing Simulation of Your Design"
- "Downloading to the Device and In-system Debugging"
- "Creating a PROM File for Stand-Alone Operation"

Design Flow

An overview of the design flow steps is shown in the following figure.

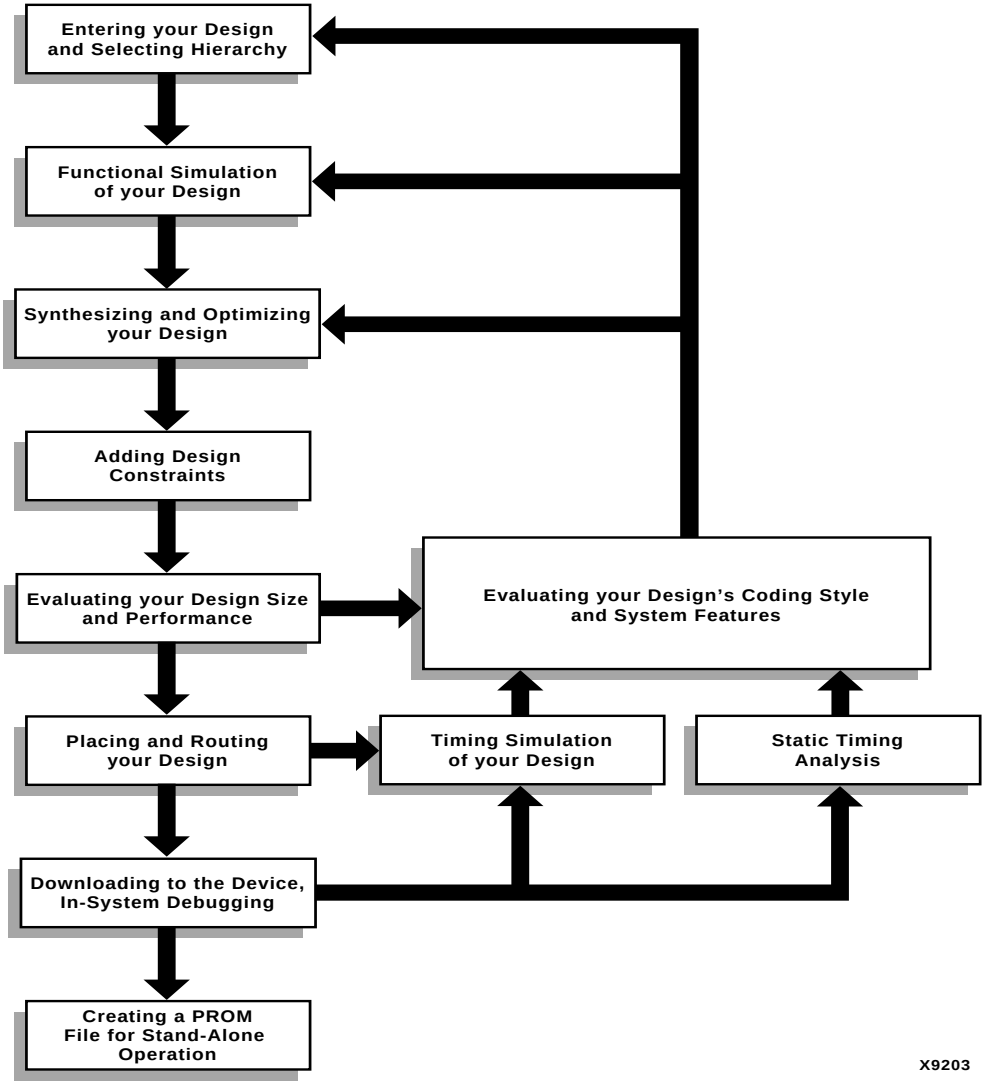

Figure 2-1 Design Flow Overview

Entering your Design and Selecting Hierarchy

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

Design Entry Recommendations

The following recommendations can help you create effective designs.

Using RTL Code

By using register transfer level (RTL) code and avoiding (when possible) instantiating specific components, you can create designs with the following characteristics.

Note In some cases instantiating optimized LogiBLOX, CORE Generator or LogiCORE modules is beneficial with RTL.

- Readable code
- Faster and simpler simulation
- Portable code for migration to different device families
- Reuseable code for future designs

Carefully Select Design Hierarchy

Selecting the correct design hierarchy is advantageous for the following reasons.

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

Functional Simulation of your Design

Use functional or RTL simulation to verify the syntax and functionality of your design. Use the following recommendations when simulating your design.

- Typically with larger hierarchical HDL designs, you should perform separate simulations on each module before testing your entire design. This makes it easier to debug your code.
- Once each module functions as expected, create a test bench to verify that your entire design functions as planned. You can use the test bench again for the final timing simulation to confirm that your design functions as expected under worst-case delay conditions.

Synthesizing and Optimizing your Design

This section includes recommendations for compiling your designs to improve your results and decrease the run time.

Note Refer to your synthesis tool documentation for more information on compilation options and suggestions.

Creating an Initialization File

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

Creating a Compile Run Script

FPGA Express, LeonardoSpectrum, and Synplify all support TCL scripting. Using 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. The following are some sample scripts that can be run from the command line or from the GUI.

FPGA Express

FPGA Scripting Tool (FST) implements a TCL-based command line interface for FPGA Express. FST can be accessed from a command line by typing the following.

- For FPGA Compiler II

  fc2_shell *synth_file.tcl*

- For FPGA Express

  fe_shell *-f synth.tcl*

The script will execute and put you back at the UNIX or DOS prompt.

FPGA Express FST Example

The following FST commands can be run in FPGA Express.

- To create the project, enter the following.

  create_project *-dir . d_register*

- To open the project, enter the following.

  open_project *d_register*

- To add the files to the project, enter the following.

  add_file *-format VHDL ../src/d_register.vhd*

- To analyze the design files enter the following.

  analyze_file *-progress*

- To create a chip for a device enter the following.

  create_chip *-progress -target Virtex -device v50PQ240 -speed -5 -name d_register d_register*

- To set the top level as the current design, enter the following.

  current_chip *d_register*

- To optimize the design, enter the following.

  set opt_chip [format "%s-Optimized" d_register]

  optimize_chip *-progress -name $opt_chip*

- To write out the messages enter the following.

  list_message

- To write out the netlist, enter the following.

  export_chip -*progress -dir .*

- **close_project**
- **quit**

LeonardoSpectrum

The following TCL script can be run from LeonardoSpectrum by doing one of the following.

1. Select the **File** → **Run Script** menu item from the LeonardoSpectrum graphical user interface.
2. Type in the Level 3 GUI 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. This will put you in a TCL prompt. Then at the TCL prompt type **source** *script_file.tcl*

LeonardoSpectrum TCL Examples

The following TCL commands can be entered in LeonardoSpectrum.

- To set the part type, enter the following.

  set part *v50ecs144*

- To read the HDL files, enter the following.

  read *macro1.vhd macro2.vhd top_level.vhd*

- To set assign buffers, enter the following.

  PAD *IBUF_LVDS data(7:0)*

- To optimize while preserving hierarchy, enter the following.

  optimize *-ta xcve -hier preserve*

- To write out the EDIF file, enter the following.

  auto_write *./M1/ff_example.edf*

Synplify

The following TCL script can be run from Synplify by doing one of the following:

1. Using the **File** → **Run TCL Script** menu item from the GUI
2. Typing **synplify** *-batch script_file.tcl* at a UNIX / DOS command prompt.

Synplify TCL Example

The following TCL commands can be entered in Synplify.

- To start a new project, enter the following.

  project *-new*

- To set device options, enter the following.

  set_option *-technology Virtex-E*

  set_option *-part XCV50E*

  set_option *-package CS144*

  set_option *-speed_grade -8*

- To add file options, enter the following.

  add_file *-constriant "watch.sdc"*

  add_file *-vhdl -lib work "macro1.vhd"*

  add_file *-vhdl -lib work "macro2.vhd"*

  add_file *-vhdl -lib work "top_levle.vhd"*

- To set compilation / mapping options, enter the following.

  set_option *-default_enum_encoding onehot*

  set_option *-symbolic_fsm_compiler true*

  set_option *-resource_sharing true*

- To set simulation options, enter the following.

  set_option *-write_verilog false*

  set_option *-write_vhdl false*

- To set automatic place and route (vendor) options, enter the following.

set_option *-write_apr_constraint true*

set_option *-part XCV50E*

set_option *-package CS144*

set_option *-speed_grade -8*

- To set result format / file options, enter the following.

  project *-result_format "edif"*

  project *-result_file "top_level.edf"*

  project *-run*

  project *-save "watch.prj"*

- **exit**

Compiling Your Design

Use the recommendations in this section to successfully compile your design.

Modifying your Design

You may need to modify your code to successfully compile 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.

Compiling 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. Refer to your synthesis tool documentation for details.

Saving Compiled Design as XNF or EDIF

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

Setting Constraints

You can define timing specifications for your design in the User Constraints File (UCF). You can use the Xilinx Constraints Editor which provides a graphical user interface allowing for easy constraints specification. You can also enter constraints directly into the UCF file. Both methods are described below. Most synthesis tools support an easy to use Constraints Editor interface for entering constraints in your design.

Using 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. Furthermore, you can group signals together to simplify timing specifications. Some synthesis tools translate certain synthesis constraints to Xilinx implementation constraints. The translated constraints are placed in a special TIMESPEC component. For more information on timing specifications in the UCF file, refer to the *Quick Start Guide for Xilinx Alliance Series*, the *Libraries Guide*, and the Answers Database on the Xilinx Support Web site, http://support.xilinx.com.

Using the Xilinx Constraints Editor

The Xilinx Constraints Editor is a GUI based tool that can be accessed from the Design Manager GUI (**`Utilities -> Constraints Editor`**), or from the command line (**`constraints_editor`**). The Constraints Editor allows the user to easily enter design constraints in a spreadsheet form and writes out the constraints in the UCF file. This eliminates the need to know the UCF file syntax. The other benefit is the Constraints Editor reads the design and lists all the nets and elements in the design. This is very helpful in the HDL flow when the synthesis tool creates the names.

Some constraints are not available through the Constraints Editor. The unavailable constraints will need to be entered directly in the UCF file using a text editor. The new UCF file needs to be re-run

through the Translate step in the Flow Engine or NGDBuild using the command line method. For more information on using the Xilinx Constraints Editor, please refer to the Constraints Editor Guide onthe Xilinx Support Web site, http://support.xilinx.com.

Using Synthesis Tools' Constraints Editor

The FPGA Express, LeonardoSpectrum, and Synplify synthesis tools all have constraint editors to apply constraints to your HDL design. Refer to your synthesis tool's documentation for information on how to use the constraints editor specific to your synthesis environment. 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 will be written out to an NCF file, and all other constraints will be written to the output EDIF or XNF file. Please refer to the documentation for your synthesis tool to obtain more information on Constraint Editors.

Evaluating Design Size and Performance

Your design should meet the following requirements.

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

After your design is compiled, you can determine preliminary device utilization and performance with your synthesis tool's reporting options. 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 incorporate any future changes or additions, and that your design will perform as specified.

Using your Synthesis Tool to Estimate Device Utilization and Performance

Use your synthesis tool's area and timing reporting options 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. Refer to your synthesis tool documentation for correct command syntax.

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, you should compare reports from the Xilinx place and route tool after implementation. Also, any instantiated components, such as LogiBLOX or CORE Generator modules, EDIF files, XNF 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 in your design, you must include the logic area used by these components when estimating design size. Also, sections of your design may get trimmed during the mapping process, and may result in a smaller design.

Using the Timing Report Command

Use your synthesis tool's timing report command to obtain a report with estimated data path delays. Refer to your synthesis vendor's documentation for command syntax.

Note See the "Report Files" appendix for sample report files from various synthesis vendors.

The timing report is based on the logic level delays from the cell libraries and estimated wire-load models for your design. This report is an estimate of how close you are to your timing goals; however, it is not the actual timing for your design. An accurate report of your design's timing is only available after your design is placed and routed. This timing report does not include information on any instantiated components, such as LogiBLOX or CORE Generator

modules, EDIF files, XNF files, or other components that are not recognized by your synthesis tool during compilation.

Determining Actual Device Utilization and Pre-routed Performance

To determine if your design fits the specified device, you must map it with the Xilinx Map program. The generated report file *design_name*.mrp contains the implemented device utilization information. The report file can be located through the **Reports** section of the Design Manager. You can run the Map program from the Design Manager or from the command line.

Using the Design Manager to Map Your Design

Use the following steps to map your design using the Design Manager.

Note For more information on using the Design Manager, see the *Design Manager/Flow Engine Reference/User Guide*.

1. To start the Design Manager, enter the following command.

   xilinx

2. To create a new project, select the XNF or EDIF file generated by your synthesis tool as your input file from the **File** → **New Project** menu command.

3. To start design implementation, click the Implement toolbar button or select **Design** → **Implement**.

   The Implement dialog box appears.

4. If necessary, select a part in the dialog box.

5. Select the Options button in the Implement dialog box.

   The Options dialog box appears.

6. Select the Produce Logic Level Timing Report option.

   This option creates a timing report prior to place and route, but after map, as described in the following five steps.

7. Select the Edit Template button next to the Implementation drop-down list.

The Implementation Template dialog box appears.

8. Select the Timing tab.
9. Select the Produce Logic Level Timing Report radio button.
10. Select the type of report you want to create.

    The default is Report Paths in Timing Constraints.

11. Use the Implementation Template dialog box tabs (Optimize & Map, Place & Route, or Interface) to select any other options applicable to your design. Select **OK** to exit the Implementation Template dialog box.

    Note Xilinx recommends using the default Map options for your designs. Also, do not use the guided map option with your synthesized designs.

12. Select Run in the Implement dialog box to begin implementing your design.
13. When the Flow Engine is displayed, stop the processing of your design after mapping by selecting **Setup** → **Stop After** or by selecting the Set Target toolbar button.

    The Stop After dialog box appears.

14. Select Map and select **OK**.
15. After the Flow Engine is finished mapping your design, select **Utilities** → **Report Browser** to view the map report. Double-click the report icon that you want to view. The map report includes a Design Summary section that contains the device utilization information.
16. 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.
17. At this point, you may want to start the Timing Analyzer from the Design Manager to create a more specific report of design paths.
18. 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 estimated routing delays.

Using the Command Line to Map Your Design

1. Translate your design as follows.

   `ngdbuild -p` *target_device design_name*`.xnf`

   or

   `ngdbuild -p` *target_device design_name*`.edf`

2. Map your design as follows.

   `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*`.ngd`

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

   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.

```
Xilinx Mapping Report File for Design 'area_con14'
Copyright (c) 1995-2000 Xilinx, Inc.  All rights reserved.
```

```
Design Information
------------------
Command Line   : map -u area_con14.ngd
Target Device  : xv50
Target Package : bg256
Target Speed   : -6
Mapper Version : virtex -- HEAD
Mapped Date    : Tue Mar  7 13:30:28 2000

Design Summary
--------------
  Number of errors: 0
  Number of warnings: 56
  Number of Slices: 0 out of  768   0%
  Number of Slices containing
    unrelated logic: 0 out of        0    0%
  Number of 4 input LUTs: 0 out of 1,536  0%
  Number of Block RAMs: 2 out of 8      25%
Total equivalent gate count for design:  32,768

Table of Contents
-----------------
Section 1 - Errors
Section 2 - Warnings
Section 3 - Design Attributes
Section 4 - Removed Logic Summary
Section 5 - Removed Logic
Section 6 - Added Logic
Section 7 - Expanded Logic
Section 8 - Signal Cross-Reference
Section 9 - Symbol Cross-Reference
Section 10 - IOB Properties
Section 11 - RPMs
Section 12 - Guide Report
Section 13 - Area Group Summary

Section 1 - Errors
------------------

Section 2 - Warnings
```

```
--------------------
WARNING:DesignRules:368 - Netcheck: Sourceless.
Net clk has
 no source.
WARNING:DesignRules:368 - Netcheck: Sourceless.
Net en has no source.
WARNING:DesignRules:368 - Netcheck: Sourceless.
Net rst has no source.
WARNING:DesignRules:368 - Netcheck: Sourceless.
 Net we has no source.
WARNING:DesignRules:368 - Netcheck: Sourceless.
 Net addr11 has no source.
.
.
.
WARNING:DesignRules:367 - Netcheck: Loadless.
 Net m2_m2_doa0 has no load.
WARNING:DesignRules:367 - Netcheck: Loadless.
 Net m2_m2_doa1 has no load.
WARNING:DesignRules:367 - Netcheck: Loadless.
 Net m2_m2_dob0 has no load.
WARNING:DesignRules:367 - Netcheck: Loadless.
 Net m2_m2_dob1 has no load.

Section 3 - Design Attributes
------------------------------

Section 4 - Removed Logic Summary
---------------------------------

Section 5 - Removed Logic
-------------------------

Section 6 - Added Logic
-----------------------

Section 7 - Expanded Logic
--------------------------
To enable this section, set the detailed map report option
 and rerun map.
```

```
Section 8 - Signal Cross-Reference
----------------------------------
To enable this section, set the detailed map report option
 and rerun map.

Section 9 - Symbol Cross-Reference
----------------------------------
To enable this section, set the detailed map report option
 and rerun map.

Section 10 - IOB Properties
---------------------------

Section 11 - RPMs
-----------------

Section 12 - Guide Report
-------------------------
Guide not run on this design.

Section 13 - Area Group Summary
-------------------------------
  AREA_GROUP AG_ONE
    RANGE: RAMB4_R1C1:RAMB4_R3C1
  No COMPRESSION specified for AREA_GROUP AG_ONE
    Number of BlockRAMs:            2 out of      3 66%
```

The following is a sample Logic Level Timing Report.

```
--------------------------------------------------------------------
Xilinx TRACE, Version M1.4.12
Copyright (c) 1995-1997 Xilinx, Inc.  All rights reserved.


Design file:              map.ncd
Physical constraint file: demo_board.pcf
Device, speed: xc4003e,-2 (x1_0.86  PRELIMINARY)
Report level:             summary report
--------------------------------------------------------------------

====================================================================
Timing constraint: NET "FAST_CLOCK" PERIOD =  15.200 nS
```

```
HIGH 50.000 % ;
 1 item analyzed, 0 timing errors detected.
 Minimum period is   5.585ns.

--------------------------------------------------------------------------------

================================================================================
Timing constraint: NET "control_logic/SLOW_CLOCK"PERIOD =
  121.600nS HIGH 50.000 % ;
 677 items analyzed, 0 timing errors detected.
 Minimum period is  17.295ns.
--------------------------------------------------------------------------------

All constraints were met.
Timing summary:
---------------
Timing errors: 0  Score: 0

Constraints cover 811 paths, 0 nets, and
232 connections (73.2% coverage)
Design statistics:
  Minimum period:  17.295ns (Maximum frequency:  57.820MHz)

Analysis completed Tue Jan 27 12:07:59 1998

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

Evaluating your Design for Coding Style and System Features

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

Tips for Improving Design Performance

This section includes ways of improving design performance by modifying your code and by incorporating FPGA system features. Most of these techniques are described in more detail in this manual.

Modifying Your Code

You can improve design performance with the following design modifications.

- Reduce levels of logic to improve timing
- Redefine hierarchical boundaries to help the compiler optimize design logic
- Pipeline
- Logic replication
- Use of CORE Generator modules
- Resource sharing
- Restructure logic

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.

Note Each device family has a unique set of system features. Review the current version of the *The Programmable Logic Data Book* for the system features available for the device you are targeting.

- Use global set/reset and global tristate nets to reduce routing congestion and improve design performance
- Use clock enables
- Place the highest fanout signals on the global buffers
- Modify large multiplexers to use tristate buffers
- Use one-hot encoding for large or complex state machines
- Use I/O registers when applicable
- Use I/O decoders when applicable
- Use I/O multiplexers when applicable

Using Xilinx-specific Features of Your Synthesis Tool

Most synthesis tools have special options for the Xilinx-specific features listed in the previous section. Refer to your synthesis tool

white papers, application notes, documentation and online help for detailed information on using Xilinx-specific features.

Placing and Routing Your Design

Note For more information on placing and routing your design, refer to the *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.
- During the day, you may want the tools to quickly process your design while you are waiting for the results. However, you may be less concerned with a quick run time, and more concerned about the quality of results when you run your designs for an extended period of time (during the night or weekend).
- 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 options you select for the placement and routing of your design directly influence the 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.

Note If you are using the command line, the appropriate command line option is provided in the following procedure.

Use the following steps to decrease implementation time in the Design Manager.

1. Select **`Design`** → **`Implement`**

The Implement dialog box appears.

2. Select the Options button in the Implement dialog box.

   The Options dialog box appears.

3. Select the Edit Template button next to the Implementation drop-down list in the Program Options Templates field. The Implementation Template dialog box appears.

4. Select the Place & Route tab.

5. Set options in this dialog box as follows.

   - Place & Route Effort Level

     Generally, you can reduce placement times by selecting a less CPU-intensive algorithm for placement. You can set the placement level from 1 (fastest run time) to 5 (best results) with the default equal to 2. Use the –l switch at the command line to perform the same function.

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

   - Router Options

     You can limit router iterations to reduce routing times. However, this may prevent your design from meeting timing requirements, or your design may not completely route. From the command line, you can control router passes with the –i switch.

   - Use Timing Constraints During Place and Route

     You can improve run times by not specifying 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 Design Manager, deselect the Use Timing Constraints During Place and Route button. To disable timing constraints at the command line, use the –x switch with PAR.

6. Select **OK** to exit the Implementation Template dialog box.

7. Select any applicable options in the Options dialog box.

8. Select **OK**.

9. Select Run in the Implement dialog box to begin implementing your design.

Improving Implementation Results

Conversely, 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). The following options can be used to improve implementation results. Detailed information for these options can be found in Chapter 3 of the *"Design Manager/Flow Engine Guide"*.

Multi-Pass Place and Route Option

Use this option to place and route your design with several different cost tables (seeds) to find the best possible placement for your design. This optimal placement results in shorter routing delays and faster designs. This option works well when the router passes are limited (with the –i option). After an optimal cost table is selected, use the re-entrant routing feature to finish the routing of your design. You may select this option from the Design menu in the Design Manager, or specify this option at the command line with the –n switch.

Turns Engine Option (UNIX only)

This option is a Unix-only feature that works with the Multi-Pass Place and Route option to allow parallel processing of placement and routing on several Unix 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 the Design Manger, specify a node list when selecting the Multi-Pass Place and Route option. To use this feature at the command line, use the –m switch to specify a node list, and the –n switch to specify the number of place and route iterations.

Note For more information on the turns engine option, refer to the Xilinx *Development System Reference Guide.*

Re-entrant Routing Option

Use the re-entrant 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 the Design Manager

To initiate a re-entrant route from the Design Manager interface, follow these steps.

1. From the Design Manager, select the placed and routed design revision for the re-entrant option.
2. Select **Tools** → **Flow Engine** to start the Flow Engine from the Design Manager.
3. From the Flow Engine menu, select **Setup** → **Re-entrant Route**.
4. In the Advanced dialog box that is displayed, select the Allow Re-entrant Routing option.
5. Select the appropriate options in the Re-entrant Route Options field.
6. Select **OK**.
7. The Place and Route icon in the Flow Engine is replaced with the Re-entrant Route icon. If this step is completed, use the Step Back button until the Re-entrant Route icon no longer indicates completed.
8. Select Run to complete the re-entrant routing.

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.

For more information on PAR and Cost Tables, refer to Chapter 12 of the *"Development System Reference Guide"*.

From the Command Line

To initiate a re-entrant route from the command line, you can 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 re-entrant 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**

Cost-Based Clean-up Option

This option specifies clean-up passes after routing is completed to substitute more appropriate routing options available from the initial routing process. For example, if several local routing resources are used to transverse the chip and a longline is available, the longline is substituted in the clean-up pass. The default value of cost-based cleanup passes is 1. To change the default value, use the Template Manager in the Design Manager, or the –c switch at the command line.

Delay-Based Clean-up Option

This option specifies clean-up passes after routing is completed to substitute more appropriate routing options to reduce delays. The default number of passes for delay-based clean-up is 0. You can change the default in the Design Manager in the Implementation Options window, or at the command line with the –d switch.

Guide Option

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 information on guide in modular design flows, refer to XAPP 404 at http://www.xilinx.com/xapp/xapp404.pdf.

Timing Simulation of Your Design

Note Refer to the "Simulating Your Design" chapter for more information on design 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. You can 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.

Timing Analysis Using TRACE

Timing-driven PAR is based upon Xilinx's timing analysis software, an integrated static timing analysis tool (that is, it does not depend on input stimulus to the circuit). This means that placement and routing are executed according to timing constraints that you specify in the beginning of the design process. The timing analysis software interacts with PAR to ensure that the timing constraints you impose on the design are met.

You can only use the analysis software if you are not supplying any timing constraints (in a PCF file) to TRACE. The timing analysis tool option writes out a timing report containing the following.

- 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 and Timing Analysis, refer to Chapter 14 of the *"Development System Reference Guide"*.

Downloading to the Device and In-system Debugging

After you have verified the functionality and timing of your placed and routed design, you can create a design data file to download for in-system verification. The design data or bitstream (.bit) file is created from the placed and routed .ncd file. In the Design Manager, use the Configuration step in the Flow Engine to create this file. From the command line, run BitGen on your placed and routed .ncd file to create the .bit file as follows.

`bitgen [options]` *design*`.ncd`

Use the .bit file with the XChecker cable and the Hardware Debugger to download the data to your device. You can run the Hardware Debugger from the Design Manager, or from the command line as follows.

`hwdebugr` *design*`.bit`

The Hardware Debugger allows you to download the data to the FPGA using your computer's serial port. The Hardware Debugger can also synchronously or asynchronously probe external or internal nodes in the FPGA. Waveforms can be created from this data and correlated to the simulation data for true in-system verification of your design.

Creating a PROM File for Stand-Alone Operation

After verifying that the FPGA works in the circuit, you can create a PROM file from the .bit file to program a PROM or other data storage device. You can then use this file to program the FPGA in-circuit during normal operation.

Use the Prom File Formatter to create the PROM file, or from the command line use PROMGen. You can run the Prom File Formatter from the Design Manager, or from the command line as follows.

`promfmtr` *design*`.bit`

Run PROMGen from the command line by typing the following.

`promgen [options]` *design*`.bit`

Note For more information on using these programs, refer to the Xilinx *Development System Reference Guide*.

Chapter 3

General HDL Coding Styles

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

- "Naming and Labeling Styles"
- "Specifying Constants"
- "Choosing Data Type (VHDL only)"
- "Coding for Synthesis"
- "Implementing Latches and Registers,"
- "Resource Sharing,"
- "Reducing Gates,"
- "Using Preset Pin or Clear Pin,"

HDLs contain many complex constructs that are difficult to understand at first. Also, the methods and examples included in HDL manuals do not always apply to the design of FPGAs. If you currently use HDLs to design ASICs, 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, and refer to 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 coding 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 and Labeling Styles

Because HDL designs are often created by design teams, Xilinx recommends that you 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 fellow team members. Also, inefficient coding styles can adversely impact synthesis and simulation, which can result in slow circuits. Additionally, 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 section of the manual provides a list of suggested 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.

Note Most synthesis tools convert illegal characters to legal ones.

- User-defined names can contain A–Z, a–z, $, _, –, <, and >. A “/” is also valid, however, it is not recommended because it is used as a hierarchy separator
- Names must contain at least one non-numeric character
- Names cannot be more than 256 characters long

  The following FPGA resource names are reserved and should not be used to name nets or components.

- Components (Comps), Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), basic elements (bels), clock buffers (BUFGs), tristate buffers (BUFTs), oscillators (OSC), CCLK, DP, GND, VCC, and RST
- CLB names such as AA, AB, and R1C2
- Primitive names such as TD0, BSCAN, M0, M1, M2, or STARTUP

- Do not use pin names such as P1 and A4 for component names
- Do not use pad names such as PAD1 for component names

  Refer to the language reference manual for Verilog or VHDL for language-specific naming restrictions. Xilinx does not recommend using escape sequences for illegal characters. Also, if you plan on importing schematics into your design, use the most restrictive character set.

Matching File Names to Entity and Module Names

The VHDL or Verilog source code file name should match 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. Xilinx also recommends that if your design contains more than one entity or module, each should be contained in a separate file with the appropriate file name. It is also 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, Types, and Packages

You can use long (256 characters maximum) identifier names with underscores and embedded punctuation in your code. Use meaningful names for signals and variables, such as CONTROL_REGISTER. Use meaningful names when defining VHDL types and packages as shown in the following examples.

```
type LOCATION_TYPE is ...;
package STRING_IO_PKG is
```

Labeling Flow Control Constructs

You can use optional labels on flow control constructs to make the code structure more obvious, as shown in the following VHDL and Verilog examples. However, you should note that these labels are not translated to gate or register names in your implemented design. Flow control constructs can slow down simulations in some Verilog simulators.

- VHDL Example

```
-- D_REGISTER.VHD
-- May 1997

-- Changing Latch into a D-Register

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

- Verilog Example

```
/* Changing Latch into a D-Register
* D_REGISTER.V
* May 1997
*/

module d_register (CLK, DATA, Q);
```

```
input CLK;
input DATA;
output Q;

reg Q;

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

endmodule
```

Using Named and Positional Association

Use positional association in function and procedure calls, and in port lists only when you assign all items in the list. Use named association when you assign only some of the items in the list. Also, Xilinx suggests that you use named association to prevent incorrect connections for the ports of instantiated components. Do not combine positional and named association in the same statement as illustrated in the following examples.

- VHDL

  Incorrect

  ```
  CLK_1: BUFGS port map (I=>CLOCK_IN,CLOCK_OUT);
  ```

  Correct

  ```
  CLK_1: BUFGS port map(I=>CLOCK_IN,O=>CLOCK_OUT);
  ```

- Verilog

  Incorrect

  ```
  BUFGS CLK_1 (.I(CLOCK_IN), CLOCK_OUT);
  ```

```
Correct
BUFGS CLK_1 (.I(CLOCK_IN), .O(CLOCK_OUT));
```

Passing Attributes

An attribute is attached to HDL objects in your design. You can pass attributes to HDL objects in two ways; you can predefine data that describes an object, or 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 will illustrate passing attributes in HDL code only. For information on passing attribute via the command file, please refer to your synthesis tool manual.

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

Note: For FPGA Express, attribute passing is available beginning with version 3.0 and the attributes can only be applied to instantiated components or ports (but not inferred logic and nets).

VHDL Attribute Examples

The following are examples of VHDL attributes.

- Attribute use on a 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 "";
```

```
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 INIT_00: string;
attribute INIT_00 of INST_RAMB4_S4: label is
"1F1E1D1C1B1A19181716151413121110 0F0E0D0C0B09087
   06050403020100";
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
       );
```

- 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
```

- Attribute use in FPGA Express syntax:

  / / synopsys **attribute** <*name*> <*value*>

  Example:

```
BUFG CLOCKB (.I(oscout), .O(clkint)); //synopsys
attribute LOC "BR"
```

  or

```
RAMB4_S4 U1 (.WE(w), .EN(en), .RST(r), .CLK(ck)
.ADDR(ad), .DI(di), .DO(do)); /* synopsys
   attribute INIT_00
"AAAAAAAAAAAAAAAABBBBBBBBBBBBBBBB" INIT_09
"99999988888888887777777766666666" */
```

- Atrribute use in Leonardo Spectrum syntax:

  / / exemplar **attribute** <*object_name*> <*attribute_name*> <*attribute_value*>

  Examples:

```
RAMB4_S8 U0 (.WE(WE), .EN(1'b1), .RST(1'b0),
.CLK(CLK),.ADDR(ADDR), .DI(DIN), .DO(DOUT));
//exemplar attribute U0 INIT_00
1F1E1D1C1B1A19181716151413121110 0F0E0D0C0B0A0908
   070605040
3020100
```

- Attribute use in Synplfy syntax:

  / / **synthesis** <*directive*>

  / / **synthesis** <*attribute_name*>=<*value*>

or

/* **synthesis** <*directive*> */

/* **synthesis** <*attribute_name*>=<*value*> */

Examples :

```
FDCE u2(.D (q1),.CE(ce),.C (clk),.CLR (rst),
.Q (qo)) /* synthesis rloc="r1c0.s0" */;
```

or

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

Understanding Synthesis Tools Naming Convention

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 will discuss how different synthesis tools generate names from your VHDL / Verilog codes. This will help you corollate nets and component names appearing in the EDIF netlist. It will also help corollate nets and names during your after-synthesis design view of the VHDL / Verilog source.

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

- FPGA Express Naming Styles:

  Register instance: <output_signal>_reg

  Output of register: <output_signal>_reg

  Output of clock buffer: <signal>_BUFGed

  Output of tristate: <signal>_tri

  Port names: preserved

  Hierarchy notation: '_', e.g., <hier_1>_<hier_2>

Other inferred component and net names are machine generated.

- Leonardo Spectrum Naming Styles:

  Register instance: reg_<output signal>

  Output of register: preserved, except if the output is also external port of the design. In this case, it will be <signal>_dup0

  Clock buffer / ibuf: <driver_signal>_ibuf

  Output of clock buffer / ibuf: <driver_signal>_int

  Tristate instance: tri_<output_signal>

  Driver and output of tristate: 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 tristate instance: <output_signal>_obuft or

  <output_signal>_iobuf

  Internal tristate instance: un<n>_<signal_name>_tb, when <n> is any number or <signal_name>_tb

  Output of tristate driving an output / inout : name of port

  Output of internal tristate: <signal_name>_tb_<number>

  RAM instance and its output

- *Dual Port RAM:*

  ram instance: <memory_name>_<n>.I_<n>

  ram output : DPO-><memory_name>_<n>.rout_bus, SPO->

  <memory_name>_<n>.wout_bus

- *Single Port RAM:*

  ram instance: <memory_name>.I_<n>

ram output: <memory_name>

Single Port Block RAM:

ram_instance: <memory_name>.I_<n>

ram output: <memory_name>

- *Dual Port Block RAM:*

  ram_instance: <memory_name>.I_<n>

  ram output: <memory_name>[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. The use of constants help make a design more readable and portable.

Using Constants to Specify OPCODE Functions (VHDL)

Do not use variables for constants in your code. Define constant numeric values in your code as constants and use them by name. This coding convention allows you to easily determine if several occurrences of the same literal value have the same meaning. In some simulators, using constants allows greater optimization. In the following code example, the OPCODE values are declared as constants, and the constant names refer to their function. This method produces readable code that may be easier to modify.

```
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;
```

Using Parameters to Specify OPCODE Functions (Verilog)

You can specify a constant value in Verilog using the parameter special data type, as shown in the following examples. The first example includes a definition of OPCODE constants as shown in the previous VHDL example. The second example shows how to use a parameter statement to define module bus widths.

- Example 1

```
//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 @ (OPCODE or A or B)
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
```

- Example 2

```
//Using a parameter for Bus Size
parameter BUS_SIZE = 8;

output ['BUS_SIZE-1:0] OUT;
```

```
input [‘BUS_SIZE-1:0] X,Y;
```

Choosing Data Type (VHDL only)

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

- *Applies as a wide range of state values*—It has nine different values that represent most of the states found in digital circuits.
- *Automatically initializes to an unknown value*—Automatic initialization is important for HDL designs because it forces you to initialize your design to a known state, which is similar to what is required in a schematic design. Do not override this feature by initializing signals and variables to a known value when they are declared because the result may be a gate-level circuit that cannot be initialized to a known value.
- *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 will 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 testbench, you cannot reuse your functional testbench 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 package for all entity port declarations. This package makes it easier to integrate the synthesized netlist back into the design hierarchy without requiring conversion functions for the ports. A VHDL example using the Std_logic package for port declarations is shown below.

```
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;
```

Since the downto convention for vectors is supported in a back-annotated netlist, the RTL and synthesized netlists should use the same convention if you are using the same test bench. This is necessary because of the loss of directionality when your design is synthesized to an EDIF or XNF netlist.

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 < =UNSIGNED(A) + UNSIGNED(B) +
          UNSIGNED(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
-- May 1997
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;
```

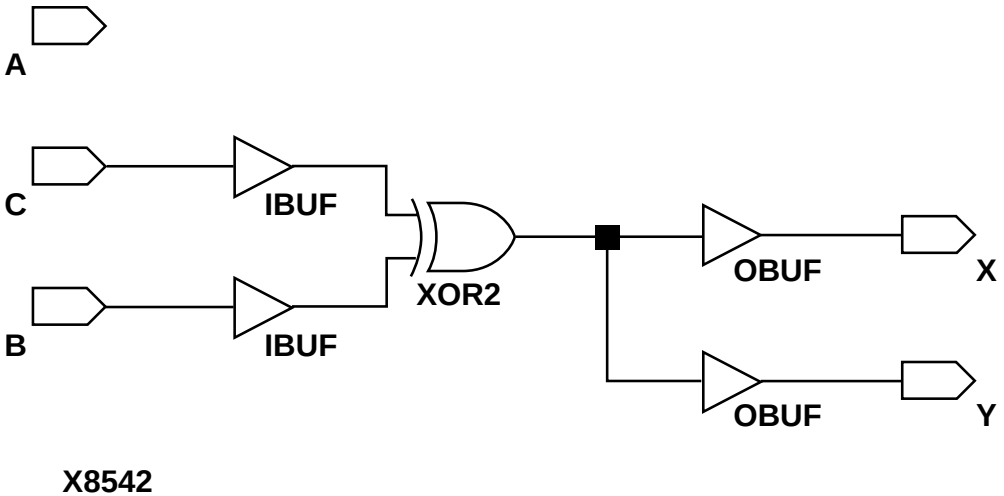

Figure 3-1 Gate implementation of XOR_SIG

Using Variables (VHDL)

```
-- XOR_VAR.VHD
-- May 1997

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;
```

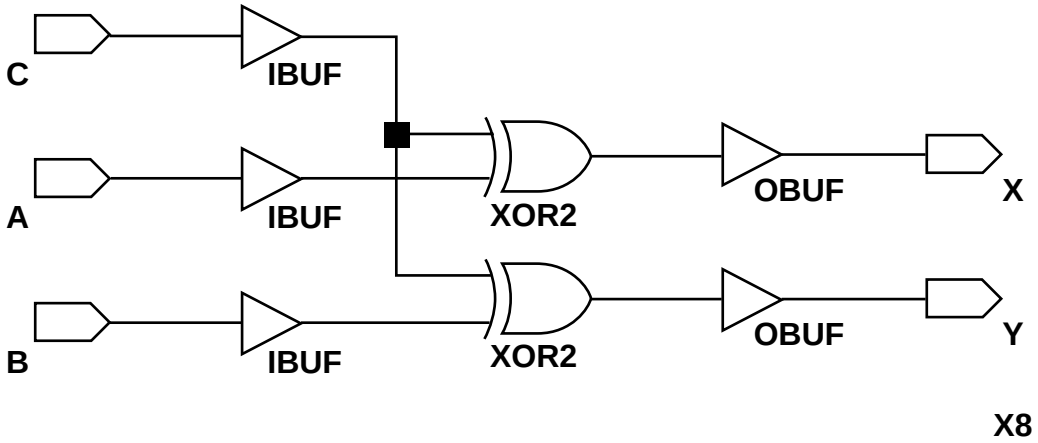

Figure 3-2 Gate Implementation of XOR_VAR

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 ensure that designs simulate the same way before and after synthesis. Follow the guidelines presented below to create code that simulates the same way before and after synthesis.

Omit the Wait for XX ns Statement

Do not use the Wait for XX ns 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 statement, the functionality of the simulated design does not 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;

Omit the ...After XX ns or Delay Statement

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.

Omit Initial Values

Do not assign signals and variables initial values because initial values are ignored by most synthesis tools. The functionality of the simulated design may not match the functionality of the synthesized design.

For example, do not use initialization statements like the following VHDL and Verilog statements.

- VHDL

  ```
  variable SUM:INTEGER:=0;
  ```

- Verilog

  ```
  wire SUM=1'b0;
  ```

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.

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 current synthesis tools can determine if the if-elsif conditions are mutually exclusive, and will not create extra logic to build the priority tree. The following are points to consider when writing if statements.

- Make sure that all outputs are defined in all branches of an if statement. If not it can creates latches or long equations on the CE signal. A good way to prevent this is to have default values for all outputs before the iff statements.
- 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.

The following examples use an If construct in a 4–to–1 multiplexer design. The “If_Ex Implementation” figure shows the implementation of these designs.

4–to–1 Multiplexer Design with If Construct

- VHDL Example

```
-- IF_EX.VHD
-- May 1997
```

```
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 FPGAs     //
// November 1997                            //
//////////////////////////////////////////////


   module if_ex (A, B, C, D, SEL, MUX_OUT);


       input        A, B, C, D;
       input  [1:0] SEL;
       output       MUX_OUT;


       reg          MUX_OUT;


       always @ (A or B or C or D or SEL)
       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
```

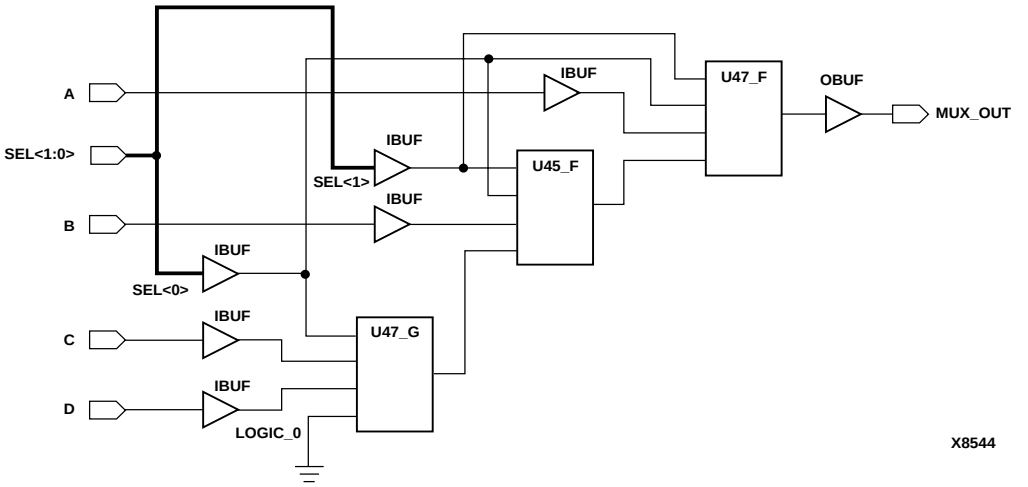

Figure 3-3 If_Ex Implementation

The following VHDL and Verilog examples use a Case construct for the same multiplexer. The “Case_Ex Implementation” figure shows the implementation of these designs. In these examples, the Case implementation requires only one XC4000 CLB while the If construct requires two CLBs 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

4–to–1 Multiplexer Design with Case Construct

- VHDL Example

```
-- CASE_EX.VHD
-- May 1997


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 FPGAs //
// November 1997 //
////////////////////////////////////////

   module case_ex (A, B, C, D, SEL, MUX_OUT);

   input        A, B, C, D;
   input  [1:0] SEL;
   output       MUX_OUT;

   reg          MUX_OUT;

      always @ (A or B or C or D or SEL)
      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
```

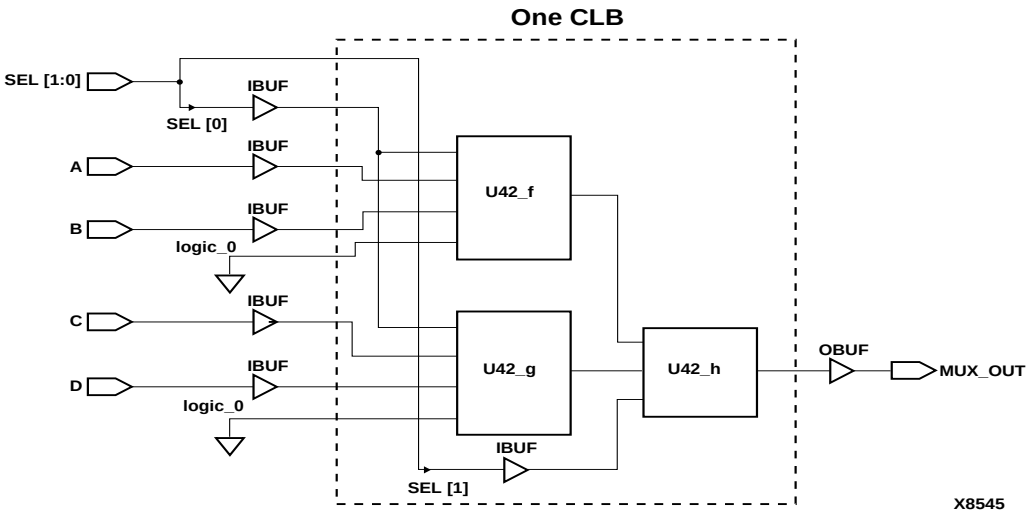

Figure 3-4 Case_Ex Implementation

Implementing Latches and Registers

Synthesizers infer latches from incomplete conditional expressions, such as an If statement without an Else clause. This can be problematic for FPGA designs because not all FPGA devices have latches available in the CLBs. In addition, you may think that a register is created, and the synthesis tool actually created a latch. The XC4000XLA, Spartan-XL / Spartan-II and Virtex / Virtex-E FPGAs do have registers that can be configured to act as latches. For these devices, synthesizers infer a dedicated latch from incomplete conditional expressions. Spartan device does not have latches in its CLBs. For these device, latches described in RTL code are implemented with gates in the CLB function generators. For Spartan device, if the latch is directly connected to an input port, it is implemented in an IOB as a dedicated input latch. For example, the D latch described in the following VHDL and Verilog designs is implemented with one function generator as shown in the “D Latch Implemented with Gates” figure.

D Latch Inference

- VHDL Example

```
-- D_LATCH.VHD
-- May 1997

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 LATCH

end BEHAV;
```

- Verilog Example

```
/* Transparent High Latch
 * D_LATCH.V
 * May 1997
*/
```

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

input GATE;
input DATA;
output Q;

reg Q;


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

   end  // End Latch

endmodule
```

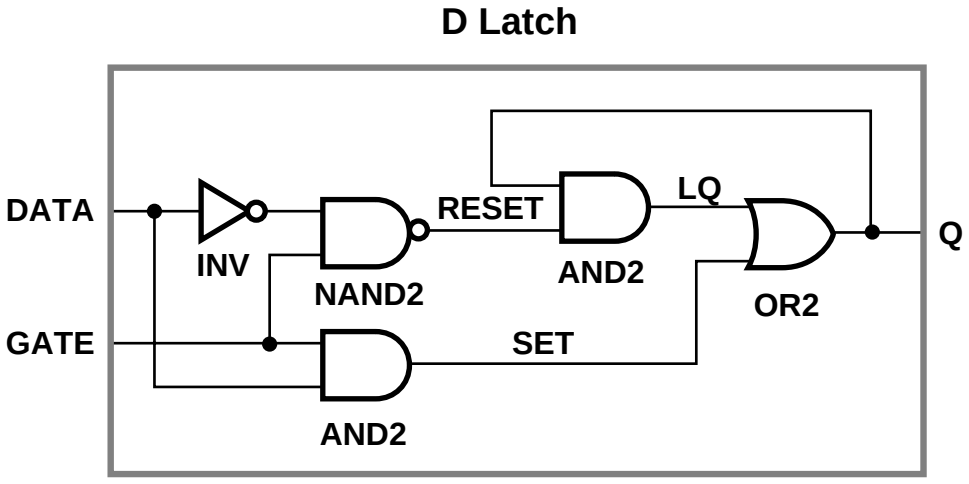

Figure 3-5 D Latch Implemented with Gates

In this example, a combinatorial loop results in a hold-time requirement on DATA with respect to GATE. Since most synthesis tools do not process hold-time requirements because of the uncertainty of routing delays, Xilinx does not recommend implementing latches with combinatorial feedback loops. A recommended method for implementing latches is described in this section.

To eliminate this possible problem, use D registers instead of latches. For example, to convert the D latch to a D register, use an Else statement or modify the code to resemble the following example.

Converting a D Latch to a D Register

- VHDL Example

```
-- D_REGISTER.VHD
-- May 1997
```

```
-- Changing Latch into a D-Register

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

- Verilog Example

```
/* Changing Latch into a D-Register
 * D_REGISTER.V
 * May 1997                          */
```

```
module d_register (CLK, DATA, Q);

input CLK;
input DATA;
output Q;

reg Q;


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

endmodule
```

With some synthesis tools you can determine the number of latches that are implemented in your design. Check the manuals that came with your software 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. Use the recommended register coding styles in the synthesis tool documentation to complete this conversion.

In XC4000E devices, you can implement a D latch by instantiating a RAM 16x1 primitive, as illustrated in the following figure.

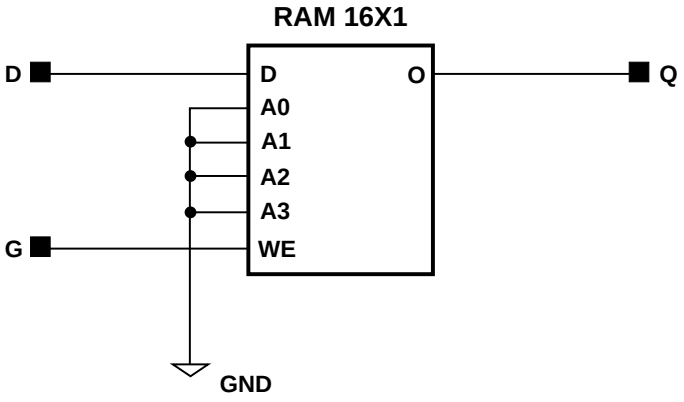

Figure 3-6 D Latch Implemented by Instantiating a RAM

In all other cases (such as latches with reset/set or enable), use a D flip-flop instead of a latch. This rule also applies to JK and SR flip-flops.

The following table provides a comparison of area and speed for a D latch implemented with gates, a 16x1 RAM primitive, and a D flip-flop.

Table 3-1 D Latch Implementation Comparison

Comparison	Spartan, CLB Latch Implemented with Gates	XC4000XLA, Spartan-XL, Spartan-II, Virtex, Virtex-E CLB Latch	All Spartan, XC4000 and Virtex Input Latch	XC4000XLA Instantiated RAM Latch	All Families D Flip Flop
Advantages	RTL HDL infers latch	RTL HDL infers latch, no hold times	RTL HDL infers latch, no hold times (if not specifying NODELAY, saves CLB resources)	No hold time or combinatorial loops.	No hold time or combinatorial loop. FPGAs are register abundant.
Disadvantages	Feedback loop results in hold time requirement, not suggested	Not available in Spartan	Input to latch must directly connect to port	Must be instantiated, uses logic resources	Requires change in code to convert latch to register
Area, see footnote "a"	1 Function Generator	1 CLB Register / Latch	1 IOB Register / Latch	1 Function Generator	1 CLB Register / Latch

a. Area is the number of function generators and registers required. XC4000XLA and Spartan / Spartan-XL CLBs have two function generators and two registers. For Spartan-II and Virtex / Virtex-E devices, replace CLB with slice.

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 XC4000 family, Spartan family, and Virtex family CLBs / slices. 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 the Figure 3-7.

- VHDL Example

```
-- RES_SHARING.VHD
-- May 1997

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
 * May 1997
*/

module res_sharing (A1, B1, C1, D1, COND_1, Z1);

input       COND_1;
input  [7:0] A1, B1, C1, D1;
output [7:0] Z1;

reg [7:0] Z1;
```

```
    always @(A1 or B1 or C1 or D1 or COND_1)
    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 the "Implementation without Resource Sharing" figure.

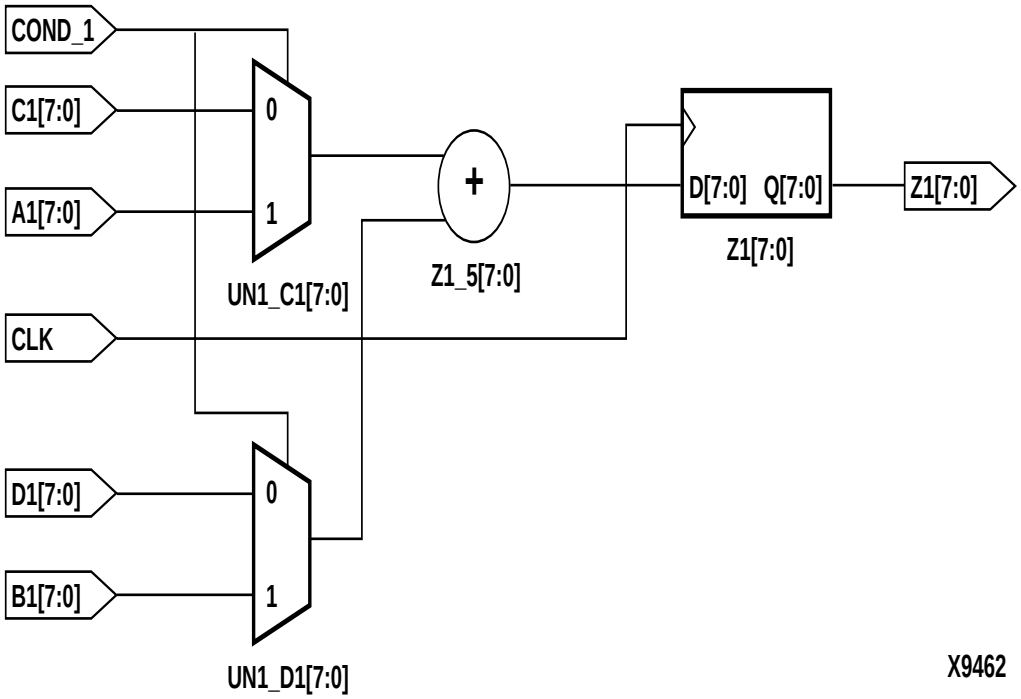

Figure 3-7 Implementation of Resource Sharing

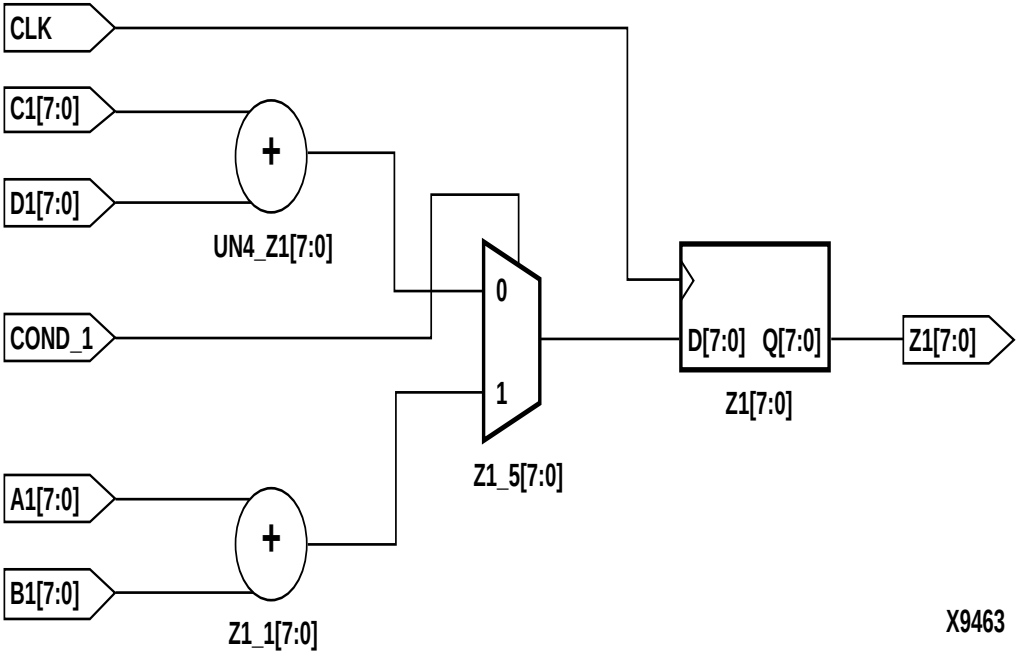

Figure 3-8 Implementation without Resource Sharing

Some synthesis tools generate modules from special Xilinx module generation algorithms. Generally, this module generation is used for operators such as adders, subtracters, incrementers, decrementers, and comparators. The following table provides a

comparison of the number of CLBs used and the delay for the VHDL and Verilog designs with and without resource sharing.

Table 3-2 Resource Sharing/No Resource Sharing Comparison for XC4005EPC84-2

Comparison	Resource Sharing with Xilinx Module Generation	No Resource Sharing with Xilinx Module Generation	Resource Sharing without Xilinx Module Generation	No Resource Sharing without Xilinx Module Generation
F/G Functions	24	24	19	28
H Function Generators	0	0	11	8
Fast Carry Logic CLBs	5	10	0	0
Longest Delay	27.878 ns	23.761 ns	47.010 ns	33.386 ns
Advantages/ Disadvantages	Potential for area reduction	Potential for decreased critical path delay	No carry logic increases path delays	No carry logic increases CLB count

Note Refer to the appropriate reference manual for more information on resource sharing.

Reducing Gates

Use the generated module components to reduce the number of gates in your designs. The module generation algorithms use Xilinx carry logic to reduce function generator logic and improve routing and speed performance. Further gate reduction can occur with synthesis tools that recognize the use of constants with the modules.

Using Preset Pin or Clear Pin

Xilinx FPGAs consist of CLBs that contain function generators and flip-flops. The XC4000 family and Spartan family flip-flops have a dedicated clock enable pin and either a clear (asynchronous reset) pin or a preset (asynchronous set) pin. All synchronous preset or clear functions can be implemented with combinatorial logic in the function generators.

You can configure XC4000XLA and Spartan / Spartan-XL CLB registers to have either a preset pin or a clear pin. Spartan-II and Virtex / Virtex-E registers can be configured to have either or both preset and clear pins.

For XC4000XLA and Spartan / Spartan-XL family you must modify any process that requires both pins to use only one pin or you must use three registers and a mux to implement the process. In this devices, if a register is described with an asynchronous set and reset, your synthesis tool may issue an error message similar to the following during the compilation of your design.

```
Warning: Target library contains no replacement for
   register ‘Q_reg’ (**FFGEN**) . (TRANS-4)
Warning: Cell ‘Q_reg’ (**FFGEN**) not translated.
   (TRANS-1)
```

During the implementation of the synthesized netlist, NGDBuild issues the following error message.

```
ERROR:basnu – logical block “Q_reg” of type
   “_FFGEN_” is unexpanded.
```

An XC4000 CLB and a Virtex slice is shown in the following figure.

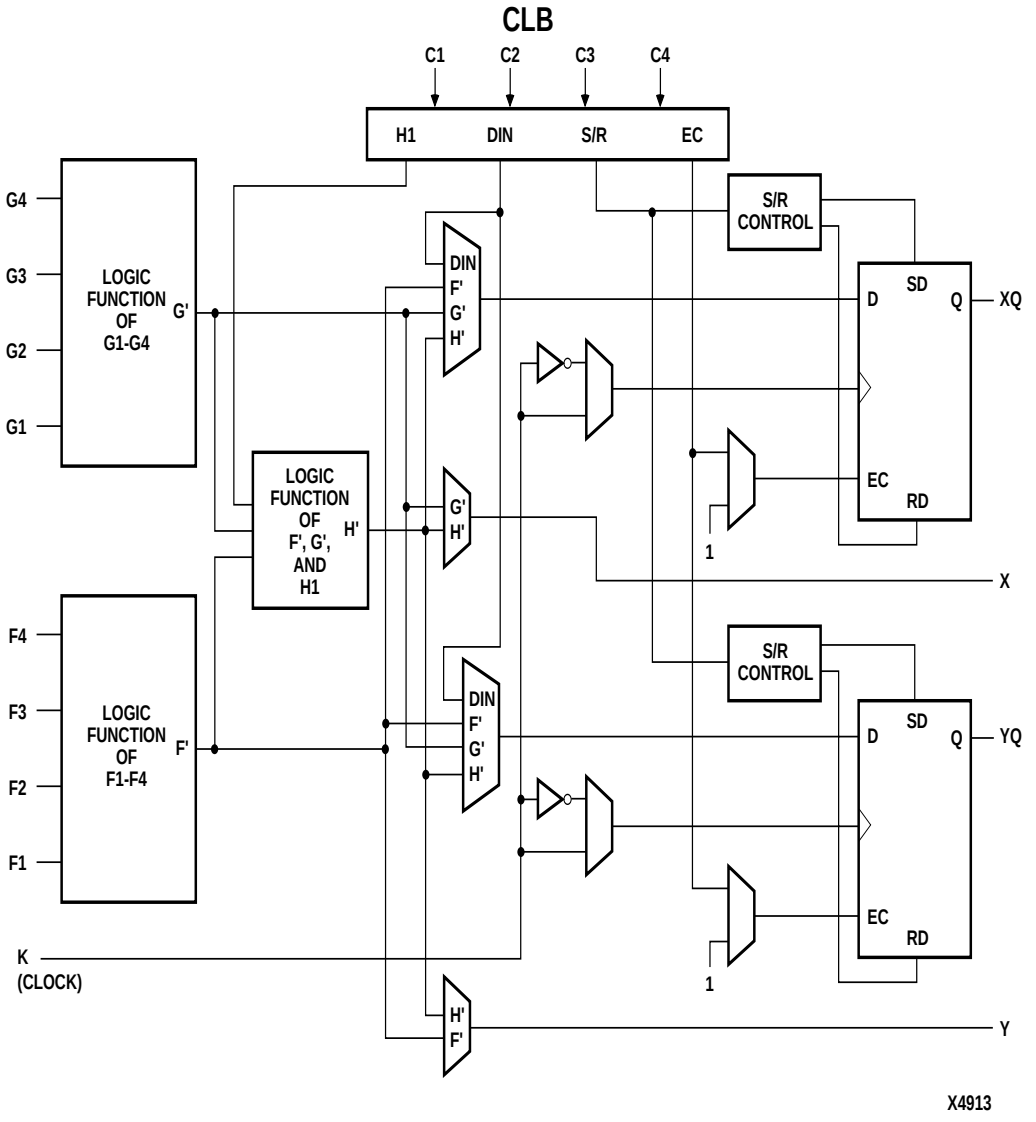

Figure 3-9 XC4000 Configurable Logic Block

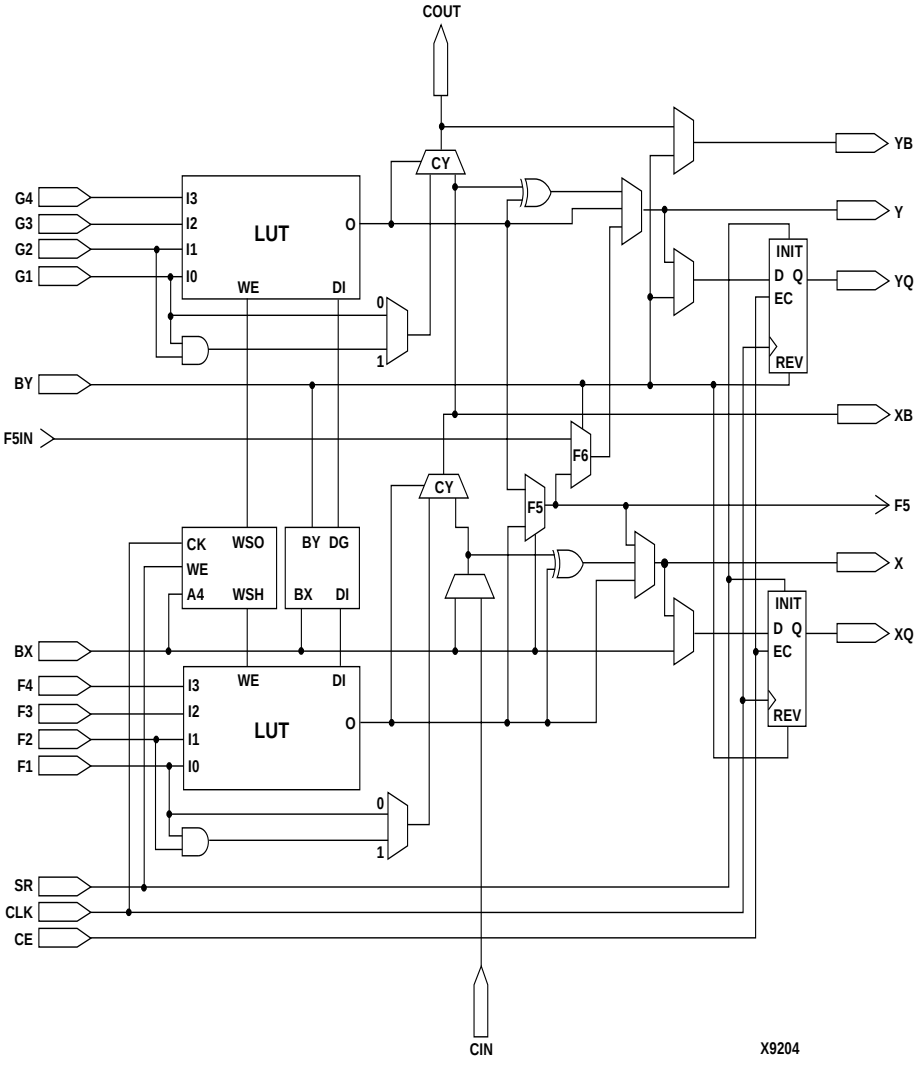

Figure 3-10 Detailed View of Virtex Slice

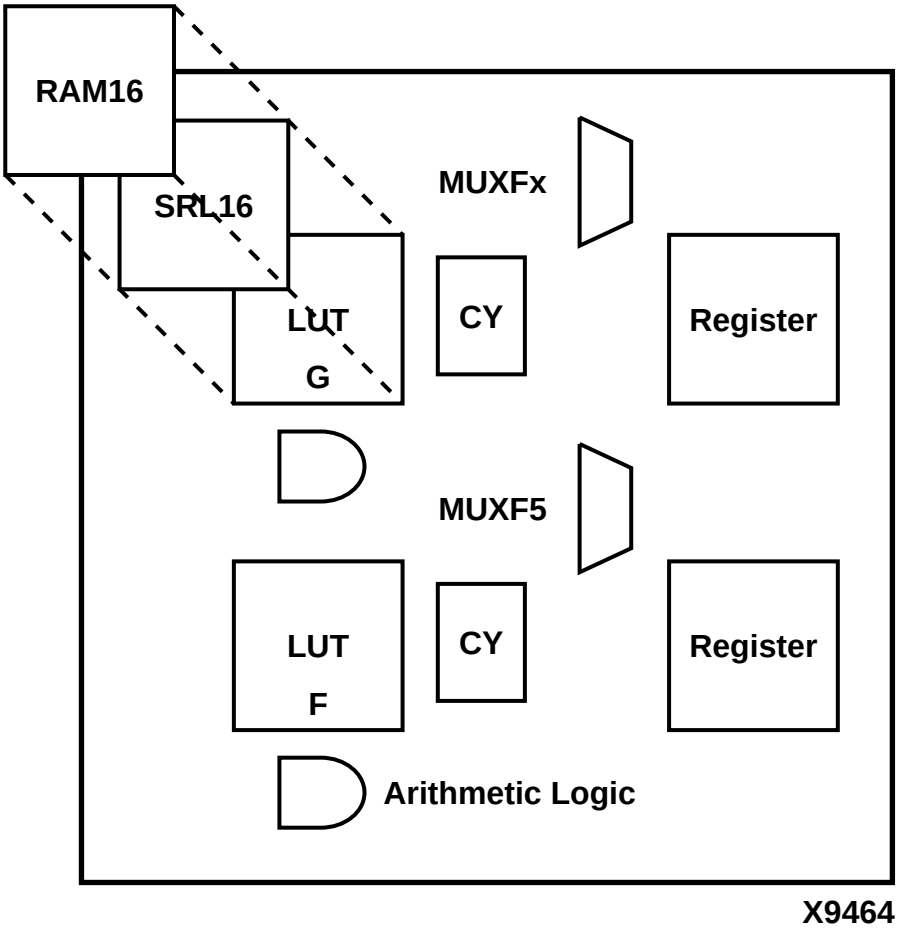

Figure 3-11 Virtex-II Slice

Register Inference

The following VHDL and Verilog designs show how to describe a register with a clock enable and either an asynchronous preset or a clear.

- VHDL Example

```
-- FF_EXAMPLE.VHD
```

```
-- May 1997
-- Example of Implementing Registers

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


entity ff_example is
 port ( RESET, CLOCK, ENABLE: in STD_LOGIC;
   D_IN: in STD_LOGIC_VECTOR (7 downto 0);
   A_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
   B_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
   C_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0);
   D_Q_OUT: out STD_LOGIC_VECTOR (7 downto 0));
end ff_example;

architecture BEHAV of ff_example is
begin

    -- D flip-flop
    FF: process (CLOCK)
    begin
        if (CLOCK'event and CLOCK='1') then
            A_Q_OUT <= D_IN;
        end if;
    end process; -- End FF
```

```
-- Flip-flop with asynchronous reset
FF_ASYNC_RESET: process (RESET, CLOCK)
begin
    if (RESET = '1') then
        B_Q_OUT <= “00000000”;
    elsif (CLOCK'event and CLOCK='1') then
        B_Q_OUT <= D_IN;
    end if;
end process; -- End FF_ASYNC_RESET

-- Flip-flop with asynchronous set
FF_ASYNC_SET: process (RESET, CLOCK)
begin
    if (RESET = '1') then
        C_Q_OUT <= “11111111”;
    elsif (CLOCK'event and CLOCK = '1') then
        C_Q_OUT <= D_IN;
    end if;
end process; -- End FF_ASYNC_SET

-- Flip-flop with asynchronous reset and
clock enable
FF_CLOCK_ENABLE: process (ENABLE, RESET,
CLOCK)
begin
    if (RESET = '1') then
       D_Q_OUT <= “00000000”;
    elsif (CLOCK'event and CLOCK='1') then
       if (ENABLE='1') then
```

```
                                D_Q_OUT <= D_IN;
                            end if;
                        end if;
                    end process; -- End FF_CLOCK_ENABLE


                end BEHAV;
```

- Verilog Example

```
/* Example of Implementing Registers
 * FF_EXAMPLE.V
 * May 1997
*/

module ff_example (RESET, CLOCK, ENABLE, D_IN,
              A_Q_OUT, B_Q_OUT, C_Q_OUT, D_Q_OUT);

input RESET, CLOCK, ENABLE;
input       [7:0] D_IN;
output      [7:0] A_Q_OUT;
output      [7:0] B_Q_OUT;
output      [7:0] C_Q_OUT;
output      [7:0] D_Q_OUT;

reg         [7:0] A_Q_OUT;
reg         [7:0] B_Q_OUT;
reg         [7:0] C_Q_OUT;
reg         [7:0] D_Q_OUT;

    // D flip-flop
    always @(posedge CLOCK)
    begin
        A_Q_OUT <= D_IN;
    end

    // Flip-flop with asynchronous reset
    always @(posedge RESET or posedge CLOCK)
    begin
        if (RESET)
            B_Q_OUT <= 8'b00000000;
```

```
            else
              B_Q_OUT <= D_IN;
    end

    // Flip-flop with asynchronous set
    always @(posedge RESET or posedge CLOCK)
    begin
        if (RESET)
            C_Q_OUT <= 8'b11111111;
            else
                C_Q_OUT <= D_IN;
    end

//Flip-flop with asynchronous reset & clock enable always
// @(posedge RESET or posedge CLOCK)
    begin
           if (RESET)
                 D_Q_OUT <= 8'b00000000;
            else if (ENABLE)
                 D_Q_OUT <= D_IN;
    end

endmodule
```

- VHDL Example

  The following VHDL and Verilog designs show how to describe a register with a clock enable and both clear and preset pins for Spartan-II and Virtex / Virtex-E devices.

```
library IEEE;

use IEEE.std_logic_1164.all;

entity setreset is

   port (CLK: in std_logic;

         DIN1, DIN2: in STD_LOGIC;

         SET, RESET: in STD_LOGIC;

         DOUT1, DOUT2: out STD_LOGIC);

end setreset ;

architecture RTL of setreset is
```

```
begin
set_then_reset: process (CLK, SET, RESET)
begin
     if (SET = '1') then
         DOUT1 <= '1';
     elsif (RESET = '1') then
         DOUT1 <= '0';
     elsif ( CLK'event and CLK ='1') then
         DOUT1  <= DIN1;
     end if;
end process;
reset_then_set: process (CLK, SET, RESET)
   begin
     if (RESET = '1') then
         DOUT2 <= '0';
     elsif (SET = '1') then
         DOUT2 <= '1';
     elsif ( CLK'event and CLK ='1') then
         DOUT2  <= DIN2;
     end if;
end process;
end RTL;
```

- Verilog Example

```
module setreset (CLK, DIN1, DIN2, SET, RESET,
DOUT1, DOUT2);
  input CLK;
  input DIN1, DIN2;
  input SET, RESET;
  output DOUT1, DOUT2;
```

```
  reg DOUT1, DOUT2;
always @ (posedge SET or posedge RESET or posedge CLK)
  begin: set_then_reset
    if (SET)
      DOUT1 <= 1'b1;
    else if (RESET)
      DOUT1 <= 1'b0;
    else
      DOUT1  <= DIN1;
  end
always @ (posedge RESET or posedge SET or posedge CLK)
  begin: reset_then_set
    if (RESET)
      DOUT2 <= 1'b0;
    else if (SET)
      DOUT2 <= 1'b1;
    else
      DOUT2  <= DIN2;
  end
endmodule
```

Using Clock Enable Pin Instead of Gated Clocks

Use the CLB clock enable pin instead of gated clocks in your designs. Gated clocks can introduce glitches, increased clock delay, clock skew, and other undesirable effects. The first two examples in this section (VHDL and Verilog) illustrate a design that uses a gated clock. The Figure 3-12 shows this design implemented with gates. Following these examples are VHDL and Verilog designs that show how you can modify the gated clock design to use the clock enable pin of the CLB. The Figure 3-13 shows this design implemented with gates.

- VHDL Example

```
-----------------------------------------
-- GATE_CLOCK.VHD Version 1.1 --
-- Illustrates clock buffer control --
-- Better implementation is to use --
-- clock enable rather than gated clock  --
-- May 1997 --
```

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

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

entity gate_clock is
    port (IN1,IN2,DATA,CLK,LOAD: in STD_LOGIC;
          OUT1: out STD_LOGIC);
end gate_clock;

architecture BEHAVIORAL of gate_clock is

signal GATECLK: STD_LOGIC;

begin

GATECLK <= (IN1 and IN2 and CLK);

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

```
end BEHAVIORAL;
```

- Verilog Example

```
////////////////////////////////////////
// GATE_CLOCK.V Version 1.1 //
// Gated Clock Example //
// Better implementation to use clock //
// enables than gating the clock //
// May 1997 //
////////////////////////////////////////

module gate_clock(IN1, IN2, DATA,
   CLK,LOAD,OUT1);
input       IN1 ;
input       IN2 ;
input       DATA ;
input       CLK ;
input       LOAD ;
output      OUT1 ;
reg         OUT1 ;

wire GATECLK ;

assign GATECLK = (IN1 & IN2 & CLK);

always @(posedge GATECLK)
begin
   if (LOAD == 1'b1)
      OUT1 = DATA;
```

```
end

endmodule
```

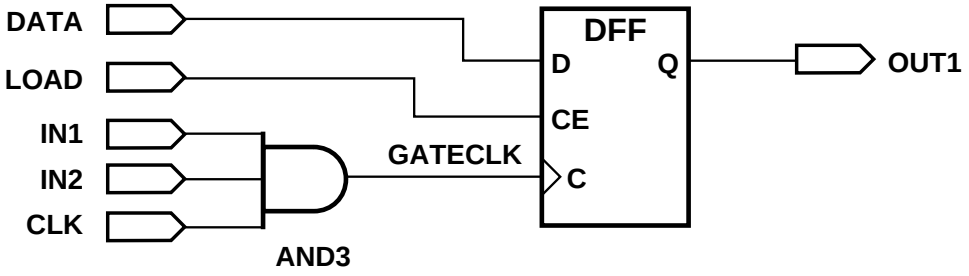

Figure 3-12 Implementation of Gated Clock

- VHDL Example

```
-- CLOCK_ENABLE.VHD
-- May 1997

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

entity clock_enable is
    port (IN1,IN2,DATA,CLOCK,LOAD: in STD_LOGIC;
          DOUT: out STD_LOGIC);
end clock_enable;

architecture BEHAV of clock_enable is
signal ENABLE: STD_LOGIC;
```

```
begin

    ENABLE <= IN1 and IN2 and LOAD;

    EN_PR: process (ENABLE,DATA,CLOCK)
    begin
        if (CLOCK'event and CLOCK='1') then
            if (ENABLE='1') then
                DOUT <= DATA;
            end if;
        end if;
    end process; -- End EN_PR

end BEHAV;
```

- Verilog Example

```
/* Clock enable example
 * CLOCK_ENABLE.V
 * May 1997
*/

module clock_enable (IN1, IN2, DATA, CLK, LOAD,
   DOUT);

input IN1, IN2, DATA;
input CLK, LOAD;
output DOUT;

wire ENABLE;
```

```
reg DOUT;

assign ENABLE = IN1 & IN2 & LOAD;

    always @(posedge CLK)
    begin
            if (ENABLE)
                 DOUT <= DATA;
    end

endmodule
```

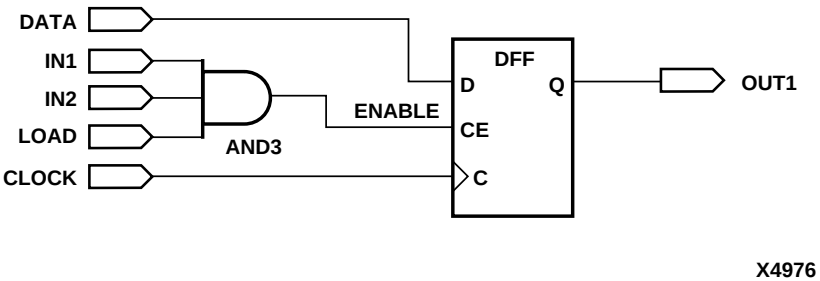

Figure 3-13 Implementation of Clock Enable

Chapter 4

Architecture Specific HDL Coding Styles for XC4000XLA, Spartan, and Spartan-XL

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

- "Introduction"
- "Instantiating Components"
- "Using Boundary Scan (JTAG 1149.1)"
- "Using Global Clock Buffers"
- "Using Dedicated Global Set/Reset Resource"
- "Implementing Inputs and Outputs"
- "Encoding State Machines"
- "Implementing Operators and Generate Modules"
- "Implementing Memory"
- "Implementing Multiplexers"
- "Using Pipelining"
- "Design Hierarchy"
- "Incremental Design (ECO)"

Introduction

Xilinx XC4000XLA, Spartan, and Spartan-XL FPGA devices are architecturally similar. They provide the benefits of custom CMOS VLSI and allow you to avoid the initial cost, time delay, and risk of conventional masked gate array devices. In addition to the logic in the CLBs

and IOBs, the XC4000XLA family, Spartan, and Spartan-XL FPGAs contain the following system-oriented features.

- Global low-skew clock or signal distribution network
- Wide edge decoders (XC4000XLA family only)
- 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
- 12-mA sink current per output and 24-mA sink per output pair.
- 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 are available for instantiation. These components are marked as *primitive* in the *"Libraries Guide"*. Components marked as *macro* in the *"Libraries Guide"* should not be instantiated in HDL code.

FPGA primitives can be instantiated in VHDL and Verilog.

- VHDL example (declaring component and port map)

```
library IEEE;
use IEEE.std_logic_1164.all;
-- Add the following two lines if using Synplify:
-- library xc4000;
-- use xc4000.components.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 declaration
-- if using 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;
```

Note To use this example in Synplify, you need to add the Xilinx primitive library and remove the component declarations as noted above.

The XC4000 library contains primitives of XC4000XLA and Spartan / Spartan-XL architecture. Replace 'xc4000' with the appropriate device family if you are targeting other Xilinx FPGA architecture

- Verilog Example.

```
module flops (d1, ce, clk, q1, rst);
input d1;
input ce;
input clk;
output q1;
input rst;

FDCE u1 (.D(d1),
                  .CE(ce),
```

```
                    .C (clk),
                    .CLR(rst),
                    .Q (q1));
endmodule
```

Note To use the above example in Synplify, add the following line.

```
'include "<path_to>/<architecture>.v"
```

The <architecture>.v files are located in $SYNPLICITY/lib/xilinx. Where $SYNPLICITY identifies your Synplicity install area.

To use the above example with XC4000XLA and Spartan/Spartan-XL, replace <architecture> with **`xc4000.v`**.

Instantiating CORE Generator Modules

CORE Generator allows you to generate complex ready-to-use functions such as FIFO, Filter, Divider, RAM, and ROM. Core Generator will generate EDIF netlist to describe the functionality and a component instantiation template for HDL instantiation. For more information on the use and functions created by the CORE Generator, see the *"CORE Generator Guide"*.

In VHDL, you can declare the component and port map as shown in *"Instantiating FPGA Primitives"* section above. Synthesis tools will assume a black box to components that do not have a VHDL functional description.

In Verilog, an empty module must be declared to get port directionality. In addition, Synplify requires a synthesis syn_black_box directive declared on a black box as shown in the example below. FPGA Express and LeonardoSpectrum will assume a black box to empty modules.

Example of Black Box Directive and Empty Module Declaration

```
module r256x16s (
      addr,
      di,
      clk,
      we,
      en,
      rst,
```

```
      do); //synthesis syn_black_box
input [7:0] addr;
input [15:0] di;
input clk;
input we;
input en;
input rst;
output [15:0] do;
endmodule
module top (addrp, dip, clkp, wep, enp, rstp, dop);
input [7:0] addrp;
input [15:0] dip;
input clkp;
input wep;
input enp;
input rstp;
output [15:0] dop;
r256x16s U0(
        .addr(addrp),
        .di(dip),
        .clk(clkp),
        .we(wep),
        .en(enp),
        .rst(rstp),
        .do(dop));
endmodule
```

Using Boundary Scan (JTAG 1149.1)

Note Refer to the Development System Reference Guide for a detailed description of the boundary scan capabilities.

XC4000XLA, Spartan, and Virtex FPGAs contain boundary scan facilities that are compatible with IEEE Standard 1149.1.

Xilinx devices support external (I/O and interconnect) testing and have limited support for internal self-test.

You can access the built-in boundary scan logic between power-up and the start of configuration. Optionally, the built-in logic is available after configuration if you specify boundary scan in your design. During configuration, a reduced boundary scan capability (sample/preload and bypass instructions) is available.

In a configured FPGA device, the boundary scan logic is enabled or disabled by a specific set of bits in the configuration bitstream. In the XC4000XLA family and Spartan/Spartan-XL families, you must instantiate the boundary scan symbol, BSCAN, and the boundary scan I/O pins, TDI, TMS, TCK, and TDO to access the boundary scan logic after configuration in HDL designs.

Instantiating the Boundary Scan Symbol in XC4000XLA and Spartan/Spartan-XL

To incorporate the boundary scan capability in a configured XC4000XLA, or Spartan/Spartan-XL FPGA using synthesis tools, you must manually instantiate boundary scan library primitives at the source code level. These primitives include TDI, TMS, TCK, TDO, and BSCAN. The following VHDL and Verilog examples show how to instantiate the boundary scan symbol, BSCAN, into your HDL code. Note that the boundary scan I/O pins are not declared as ports in the HDL code. The schematic for this design is shown in the "Bnd_scan Schematic" figure.

The following example will work as is in FPGA Express, LeonardoSpectrum, and Synplify provided you load or include the correct library. In other synthesis tools, you may be required to assign a **`Set Don't Touch`** or equivalent attribute to the net connected to the TDO pad before using the Insert Pads (or equivalent) and compile commands. Otherwise, the TDO pad is removed by the compiler. In addition, you do not need IBUFs or OBUFs for the TDI, TMS, TCK, and TDO pads. These special pads connect directly to the Xilinx boundary scan module.

Boundary Scan VHDL Example

The following is an example of the Boundary Scan in VHDL.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity bnd_scan is
        port (TDI_P, TMS_P, TCK_P : in STD_LOGIC;
        LOAD_P, CE_P, CLOCK_P, RESET_P: in
        STD_LOGIC;
        DATA_P: in STD_LOGIC_VECTOR(3 downto 0);
        TDO_P: out STD_LOGIC;
```

```
        COUT_P: out STD_LOGIC_VECTOR(3 downto 0));
end bnd_scan;
architecture XILINX of bnd_scan is
    component BSCAN
      port (TDI, TMS, TCK out STD_LOGIC;
            TDO: in STD_LOGIC);
    end component;
    component TDI
      port (I: out STD_LOGIC);
    end component;
    component TMS
      port (I: out STD_LOGIC);
    end component;
    component TCK
      port (I: out STD_LOGIC);
    end component;
    component TDO
      port (O: out STD_LOGIC);
    end component;
    component count4
      port (LOAD, CE, CLOCK, RST: in STD_LOGIC;
      DATA: in STD_LOGIC_VECTOR (3 downto 0);
      COUT: out STD_LOGIC_VECTOR (3 downto 0));
    end component;
  -- Defining signals to connect BSCAN to Pins --
  signal TCK_NET  : STD_LOGIC;
  signal TDI_NET  : STD_LOGIC;
  signal TMS_NET  : STD_LOGIC;
  signal TDO_NET  : STD_LOGIC;
begin
  U1: BSCAN port map (TDO = TDO_NET,
                      TDI = TDI_NET,
                      TMS = TMS_NET,
                      TCK = TCK_NET);
  U2: TDI port map (I =TDI_NET);
  U3: TCK port map (I =TCK_NET);
  U4: TMS port map (I =TMS_NET);
  U5: TDO port map (O =TDO_NET);
  U6: count4 port map (LOAD  = LOAD_P,
                   CE    = CE_P,
                   CLOCK = CLOCK_P,
                   RST   = RESET_P,
```

```
                    DATA  = DATA_P,
                    COUT  = COUT_P);
end XILINX;
```

Boundary Scan Verilog Example

The following is an example of the Boundary Scan in Verilog.

```
////////////////////////////////////////////////
// BND_SCAN.V //
// Example of instantiating the BSCAN symbol in //
// activating the Boundary Scan circuitry //
// Count4 is an instantiated .v file of a counter //
// September 1997
////////////////////////////////////////////////
module bnd_scan (LOAD_P, CLOCK_P, CE_P, RESET_P,
  DATA_P, COUT_P);
    input         LOAD_P, CLOCK_P, CE_P, RESET_P;
input  [3:0] DATA_P;
    output [3:0] COUT_P;
    wire         TDI_NET, TMS_NET, TCK_NE, TDO_NET;
    BSCAN U1 (.TDO(TDO_NET), .TDI(TDI_NET),
            .TMS(TMS_NET), .TCK(TCK_NET));
    TDI U2 (.I(TDI_NET));
    TCK U3 (.I(TCK_NET));
    TMS U4 (.I(TMS_NET));
    TDO U5 (.O(TDO_NET));
    count4 U6 (.LOAD(LOAD_P), .CLOCK(CLOCK_P),
              .RST(RESET_P),
  .DATA(DATA_P), .COUT(COUT_P));
endmodule
```

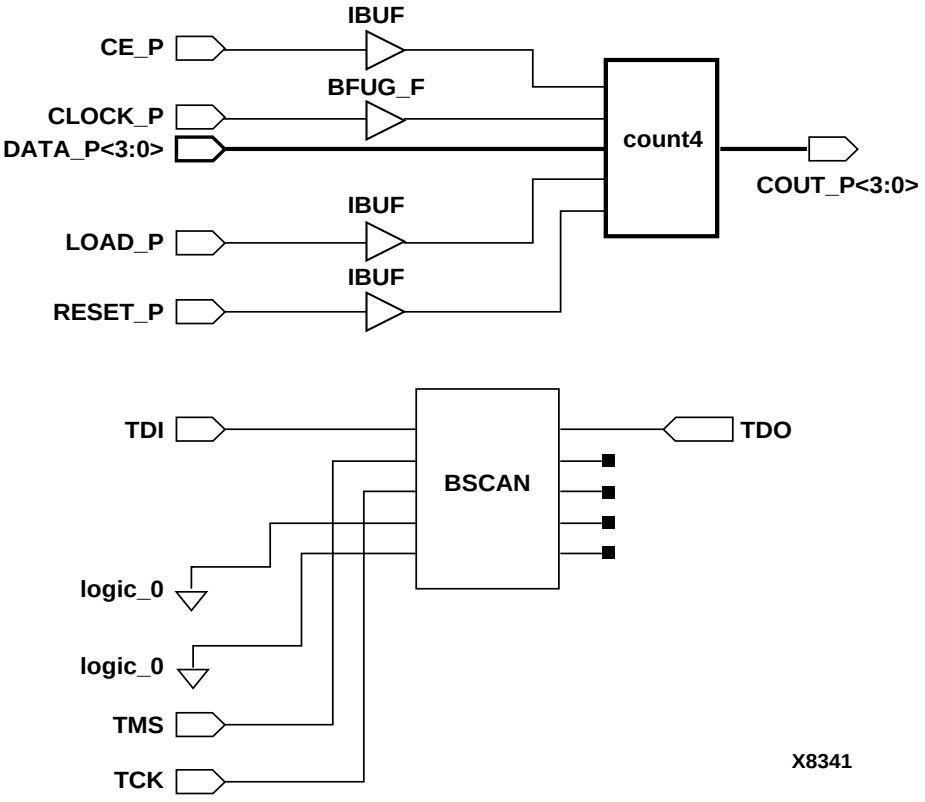

Figure 4-1 Bnd_scan Schematic

Using Global Clock Buffers

For designs with global clock signals, use global buffers (BUFGs) 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 generic clock buffer whenever an input signal drives a clock signal. It may also limit the amount of clock buffer insertion in the design. The Xilinx implementation software automatically selects the clock buffer that is appropriate for your specified design architecture. If you want to use a specific global buffer, you must instantiate it.

You can instantiate an architecture-specific buffer if you understand the architecture and want to specify how the resources should be used. Devices can contain Primary Global Buffers (BUFGP), and Secondary Global Buffers (BUFGS) that share the same routing resources. Devices can also contain Global Low Skew Buffers (BUFGLS). Table 4-1 summarizes global buffer resources in XC4000XLA, Spartan and Spartan-XL devices.

Table 4-1 Global Buffer Resources

Buffer Type	XC4000XLA	Spartan	Spartan-XL
BUFGP	N/A	4	N/A
BUFGS	N/A	4	N/A
BUFGLS	8	N/A	8
BUFGE	8	N/A	N/A
Total BUFGs	16	8	8

In XC4000XLA and Spartan-XL devices, BUFGLS are standard global buffers that should be used for most internal clocking or high fanout signals that must drive a large portion of the device. There are eight BUFGLS buffers available, two in each corner of the device. The Global Early Buffers (BUFGEs) in XC4000XLA are designed to provide faster clock access, but CLB access is limited to one quadrant of the device. I/O access is also limited. Similarly, there are eight BUFGEs, two in each corner of the device.

Because BUFGEs and BUFGLS share a single pad, a single IPAD can drive a BUFGE, BUFGLS, or both in parallel in a XC4000XLA device. The parallel configuration is especially useful for clocking the fast capture latches of the device. Since the BUFGE and BUFGLS share a common input, they cannot be driven by two different signals.

You can use the following criteria to help select the appropriate global buffer for a given design path.

- The simplest option is to use a BUFGLS.
- If you want a faster clock path, use a BUFG. Initially, the software will try to use a BUFGLS. If timing requirements are not met, a BUFGE is automatically used if possible.

- If a single quadrant of the chip is sufficient for the clocked logic, and timing requires a faster clock than the BUFGLS, use a BUFGE.

Note For more information on using the XC4000XLA device family global buffers, refer to the online version of *The Programmable Logic Data Book* or the Xilinx web site at http://www.xilinx.com.

For Spartan devices, you can use a BUFGS to buffer high-fanout, low-skew signals that are sourced from inside the FPGA. To access the secondary global clock buffer for an internal signal, instantiate the BUFGS cell. You can use a BUFGP to distribute signals applied to the FPGA from an external source. Internal signals can be globally distributed with a primary global buffer, however, the signals must be driven by an external pin.

Spartan devices have four primary (BUFGP) and four secondary (BUFGS) global clock buffers that share four global routing lines, as shown in the following figure.

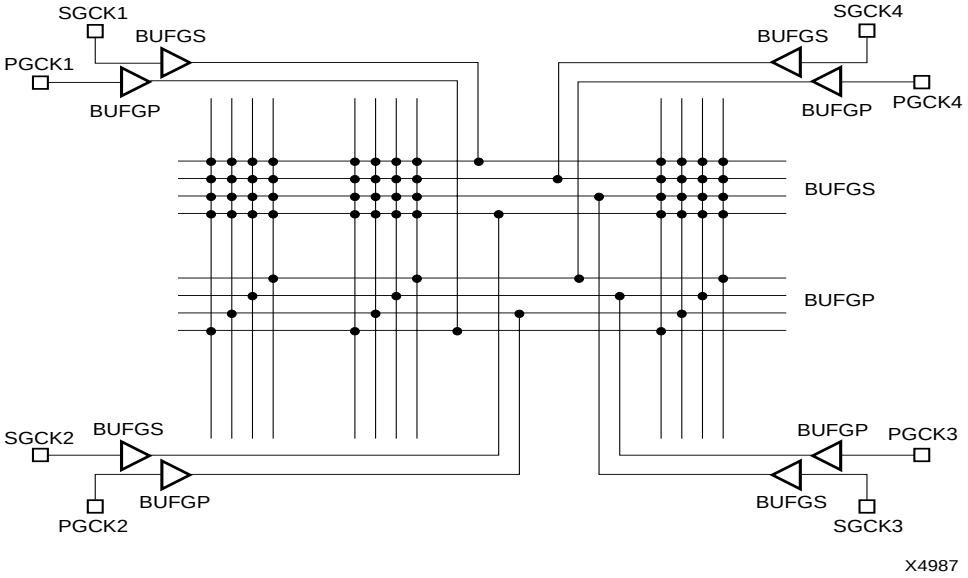

Figure 4-2 Global Buffer Routing Resources (Spartan)

These global routing resources are only available for the eight global buffers. The eight global nets run horizontally across the middle of the device and can be connected to one of the four vertical longlines

that distribute signals to the CLBs in a column. Because of this arrangement only four of the eight global signals are available to the CLBs in a column. These routing resources are “free” resources because they are outside of the normal routing channels. Use these resources whenever possible. You may want to use the secondary buffers first because they have more flexible routing capabilities.

In Spartan-XL, the four vertical longlines can be driven by any of the eight BUFGLS.

You should use the global buffer routing resources primarily for high-fanout clocks that require low skew, however, you can use them to drive certain CLB pins, as shown in the following figure. In addition, you can use these routing resources to drive high-fanout clock enables, clear lines, and the clock pins (K) of CLBs and IOBs.

In the following figure, the C pins drive the input to the H function generator, Direct Data-in, Preset, Clear, or Clock Enable pins. The F and G pins are the inputs to the F and G function generators, respectively.

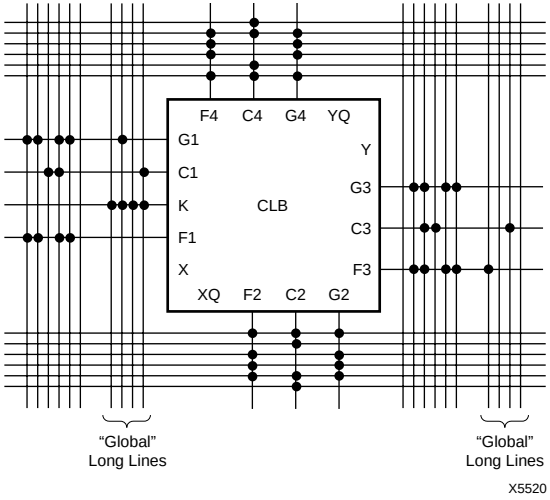

Figure 4-3 Global Longlines Resource CLB Connections

If your design does not contain four high-fanout clocks, use these routing resources for signals with the next highest fanout. To reduce routing congestion, use the global buffers to route high-fanout signals. These high-fanout signals include clock enables and reset

signals (*not* global reset signals). Use global buffer routing resources to reduce routing congestion; enable routing of an otherwise unroutable design; and ensure that routing resources are available for critical nets.

Xilinx recommends that you assign up to four secondary global clock buffers to the four signals in your design with the highest fanout (such as clock nets, clock enables, and reset signals). Clock signals that require low skew have priority over low-fanout non-clock signals. You can source the signals with an input buffer or a gate internal to the design. Generate internally sourced clock signals with a register to avoid unwanted glitches. The synthesis tool can insert global clock buffers or you can instantiate them in your HDL code.

Note Use Global Set/Reset resources when applicable. Refer to the "Using Dedicated Global Set/Reset Resource" section in this chapter for more information.

Inserting Clock Buffers

Many synthesis tools automatically insert a global buffer (BUFG) on all input ports that drive a register's clock pin or a gated clock signal. If you have more clock pins than the available BUFGs resources, most synthesis tools will allow you to control BUFG insertions manually.

FPGA Express will infer up to four clock buffers for pure clock nets. You can also instantiate clock buffers or assign them via the Express Constraints Editor.

LeonardoSpectrum will 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 the user will not have to instantiate any BUFG components. As long as "chip" options is used to optimize the IBUFs, they will be 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 will assign a BUFG to any input signal that directly drives a clock. The maximum number of global buffers is defined as 4. Auto-insertion of the BUFG for internal clocks occur 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, you may set an attribute in the .sdc file as follows.

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

Refer to your synthesis tool documentation for a detailed syntax information.

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.

In XC4000XLA, Spartan, and Spartan-XL designs verify that the Pad parameter is not specified for the buffer so that IBUF is not inserted.

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 the internal oscillator or non-dedicated I/O pin. The following VHDL and Verilog examples instantiate a BUFGS for an internal multiplexed clock circuit.

Note Synplify will insert a buffer on a CLOCK signal that has a fanout of more than 16.

- VHDL Example

```
-----------------------------------------------
-- CLOCK_MUX.VHD Version 1.1                 --
-- This is an example of an instantiation of --
```

```
-- global buffer (BUFGS) from an internally  --
-- driven signal, a multiplexed clock.       --
-- March 1998                                --
-----------------------------------------------
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;
component BUFGS
    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: BUFGS
   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.V Version 1.1          //
//This is an example of an instantiation of //
// global buffer (BUFGS) from an internally //
// driven signal, a multipled clock. //
// March 1998              //
//////////////////////////////////////////

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

    reg   CLOCK;
    wire   CLOCK_GBUF;
    reg    DOUT;
        always @ (SEL or FAST_CLOCK or SLOW_CLOCK)
        begin
        if (SEL == 1'b1)
            CLOCK <= FAST_CLOCK;
        else
            CLOCK <= SLOW_CLOCK;
          end

        BUFGS GBUF_FOR_MUX_CLOCK (.O(CLOCK_GBUF),
                                  .I(CLOCK));

    always @ (posedge CLOCK_GBUF)
        DOUT = DATA;
endmodule
```

Using Dedicated Global Set/Reset Resource

XC4000XLA, Spartan, and Spartan-XL devices have a dedicated Global Set/Reset (GSR) net that you can use to initialize all CLBs and IOBs. When the GSR is asserted, *every* flip-flop in the FPGA is simultaneously preset or cleared. You can access the GSR net from the GSR pin on the STARTUP block or the GSRIN pin of the STARTBUF (VHDL).

Since the GSR net has dedicated routing resources that connect to the Preset or Clear pin of the flip-flops, you do not need to use general purpose routing or global buffer resources to connect to these pins. If your design has a Preset or Clear signal that affects every flip-flop in your design, use the GSR net to increase design performance and reduce routing congestion.

For XC4000XLA, Spartan, and Spartan-XL devices, the Global Set/Reset (GSR) signal is, by default, set to active high (globally resets device when logic equals 1). You can change this to active low by inferring it properly in your code. Most new synthesis tools will automatically insert the STARTUP block and infer active low reset correctly.

Note See the "Simulating Your Design" chapter for more information on simulating the Global Set/Reset.

Startup State

The GSR pin on the STARTUP block or the GSRIN pin on the STARTBUF block drives the GSR net and connects to each flip-flop's Preset and Clear pin. When you connect a signal from a pad to the STARTUP block's GSR pin, the GSR net is activated. Because the GSR net is built into the silicon it does not appear in the pre-routed netlist file. When the GSR signal is asserted High (the default), all flip-flops and latches are set to the state they were in at the end of configuration. When you simulate the routed design, the gate simulator translation program correctly models the GSR function.

Note See the "Simulating Your Design" chapter for more information on STARTUP and STARTBUF.

Preset vs. Clear

The XC4000XLA, Spartan, and Spartan-XL family flip-flops are configured as either preset (asynchronous set) or clear (asynchronous reset) during startup. Automatic assertion of the GSR net presets or clears each flip-flop. You can assert the GSR pin at any time to produce this global effect. You can also preset or clear individual flip-flops with the flip-flop's dedicated Preset or Clear pin. When a Preset or Clear pin on a flip-flop is connected to an active signal, the state of that signal controls the startup state of the flip-flop. For example, if you connect an active signal to the Preset pin, the flip-flop starts up in the preset state. If you do not connect the Clear or Preset pin, the

default startup state is a clear state. To change the default to preset, assign an INIT=S attribute to the XC4000XLA / Spartan / Spartan-XL flip-flop.

I / O flip-flops and latches in XC4000XLA / Spartan / Spartan-XL do not have individual Preset or Clear pins. The default value of these flip-flops and latches is clear. To change the default value to preset, assign an INIT=1, or INIT=S attribute.

Below is an example of setting register INIT using ROCBUF. In the HDL code, the instantiated ROCBUF connects the set / reset signal. The Xilinx tools will automatically remove the ROCBUF during implementation leaving the set / reset signal active only during power-up.

- VHDL Example.

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

entity d_register is
 port (CLK : in std_logic;
   RESET : in std_logic;
   D0: in std_logic;
   D1: in std_logic;
   Q0 : out std_logic;
   Q1 : out std_logic);

end d_register;

architecture XILINX of d_register is
signal RESET_int : std_logic;
component ROCBUF is  port (I : in STD_LOGIC;
                           O : out STD_LOGIC);
end component;
```

```
begin

U1: ROCBUF port map (I => RESET, O =>
   RESET_int);

process (CLK, RESET_int)
begin
 if RESET_int = '1' then
     Q0 <= '0';
     Q1 <= '1';

 elsif  rising_edge(CLK) then
             Q0 <= D0;
             Q1 <= D1;
 end if;
end process;
end XILINX;
```

- Verilog Example

```
/*
 Note: In Synplify, set blackbox attribute for
   ROCBUF as follows:
module ROCBUF (I, O); //synthesis syn_black_box
input I;
output O;
endmodule
*/
module rocbuf_example (reset, clk, d0, d1, q0,
   q1)
```

```
 ;
input reset;
input clk ;
input d0;
input d1;
output q0 ;
output q1 ;
reg q0, q1;
wire reset_int;
ROCBUF u1 (.I(reset), .O(reset_int));
always @ (posedge clk or posedge reset_int)
   begin
if  (reset_int) begin
 q0 = 1'b0;
 q1 = 1'b1;
 end
else
 begin
 q0 = d0;
 q1 = d1;
 end
end
endmodule
module ROCBUF (I, O);
input I;
output O;
endmodule
```

Performance with the GSR Net

Many designs contain a net that initializes most of the flip-flops in the design. If this signal can initialize *all* the flip-flops, you can use the GSR net. You should always include a net that initializes your design to a known state.

To ensure that your HDL simulation results at the RTL level match the synthesis results, write your code so that every flip-flop and latch is preset or cleared when the GSR signal is asserted. The synthesis tool cannot infer the GSR net from HDL code. To utilize the GSR net, you must instantiate the STARTUP or STARTBUF block (VHDL), as shown below in the figure, "No_GSR Implementation with Gates". You can also set the option to infer a GSR in your synthesis tool.

Design Example without Dedicated GSR Resource

In the following VHDL and Verilog designs, the RESET signal initializes all the registers in the design; however, it does not use the dedicated global resources. The RESET signal is routed using regular routing resources. These designs include two 4-bit counters. One counter counts up and is reset to all zeros on assertion of RESET and the other counter counts down and is reset to all ones on assertion of RESET. The "No_GSR Implemented with Gates" figure shows the No_GSR design implemented with gates.

- No GSR VHDL Example

```
-- NO_GSR Example
-- The signal RESET initializes all registers
-- May 1997
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all

entity no_gsr is
port (CLOCK: in STD_LOGIC;
      RESET: in STD_LOGIC;
      UPCNT: out STD_LOGIC_VECTOR (3 downto 0);
      DNCNT: out STD_LOGIC_VECTOR (3 downto 0));
end no_gsr;

architecture SIMPLE of no_gsr is
```

```
signal UP_CNT: STD_LOGIC_VECTOR (3 downto 0);
signal DN_CNT: STD_LOGIC_VECTOR (3 downto 0);

begin
    UP_COUNTER: process (CLOCK, RESET)
    begin
        if (RESET = '1') then
            UP_CNT <= "0000";
        elsif (CLOCK'event and CLOCK = '1') then
          UP_CNT <= UP_CNT + 1;
            end if;
    end process;

    DN_COUNTER: process (CLOCK, RESET)
    begin
        if (RESET = '1') then
            DN_CNT <= "1111";
        elsif (CLOCK'event and CLOCK = '1') then
            DN_CNT <= DN_CNT - 1;
        end if;
    end process;

    UPCNT <= UP_CNT;
    DNCNT <= DN_CNT;

end SIMPLE;
```

- No GSR Verilog Example

```
/* NO_GSR Example
 * The signal RESET initializes all registers
 * December 1997 */

module no_gsr ( CLOCK, RESET, UPCNT, DNCNT);

input CLOCK, RESET;
output [3:0] UPCNT;
output [3:0] DNCNT;

reg [3:0] UPCNT;
reg [3:0] DNCNT;
```

```
always @ (posedge CLOCK or posedge RESET) begin
      if (RESET) begin
          UPCNT = 4'b0000;
          DNCNT = 4'b1111;
 end else begin

              UPCNT = UPCNT + 1'b1;
           DNCNT = DNCNT - 1'b1;
 end
end
endmodule
```

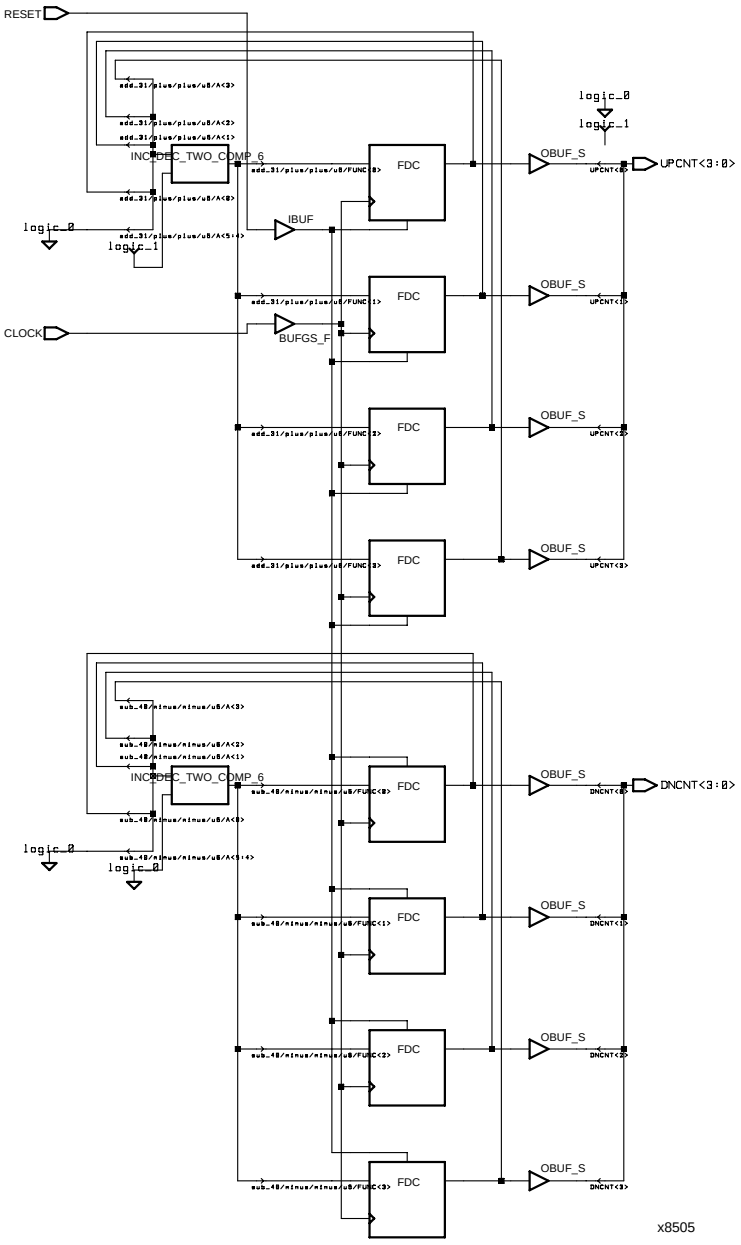

Figure 4-4 No_GSR Implemented with Gates

Design Example with Dedicated GSR Resource

To reduce routing congestion and improve the overall performance of the reset net in the No_GSR design, use the dedicated GSR net instead of the general purpose routing. Instantiate the STARTUP, STARTBUF, ROC, or TOC block in your design and use the GSR pin on the STARTUP block (or the GSRIN pin on the STARTBUF block) to access the global reset net. This is not necessary with many synthesis tools. If you fully define the behavior of the GSR net, the tool infers a STARTUP block. The modified designs (Use_GSR) are included at the end of this section. The Use_GSR design implemented with gates is shown in the Figure 4-5.

On assertion of the GSR net, flip-flops return to a clear (or Low) state by default. You can override this default by describing an asynchronous preset in your code, or by adding the INIT=1 or INIT=R.

The Use_GSR design explicitly state that the down-counter resets to all ones, therefore, asserting the reset net causes this counter to set to a default of all ones. If the design does not explicitly state the reset value, you can attach the INIT = 1 or INIT=R attribute to the flip-flops. This attribute allows you to override the default clear (or Low) state. However, because attributes are assigned outside the HDL code, the code no longer accurately represents the behavior of the design.

Refer to your synthesis tool documentation for more information on assigning attributes.

The STARTUP or STARTBUF block must not be optimized during the synthesis process. Add the appropriate attribute to prevent optimization before compiling your design.

- Use GSR VHDL Example (XC4000XLA / Spartan / Spartan-XL family)

```
-- USE_GSR.VHD Example
-- The signal RESET is connected to the
-- GSRIN pin of the STARTBUF block
-- May 1997

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

```
use UNISIM.all;

entity use_gsr is
port ( CLOCK: in STD_LOGIC;
       RESET: in STD_LOGIC;
       UPCNT: out STD_LOGIC_VECTOR (3 downto 0);
       DNCNT: out STD_LOGIC_VECTOR (3 downto 0));
end use_gsr;
architecture XILINX of use_gsr is
component STARTBUF
    port (GSRIN: in STD_LOGIC);
          GSROUT: out STD_LOGIC);
end component;
signal RESET_INT: STD_LOGIC;
signal UP_CNT: STD_LOGIC_VECTOR (3 downto 0);
signal DN_CNT: STD_LOGIC_VECTOR (3 downto 0);
begin
    U1: STARTBUF port map(GSRIN=>RESET,
                          GSROUT=>RESET_INT);
    UP_COUNTER: process(CLOCK, RESET_INT)
    begin
        if (RESET_INT = '1') then
            UP_CNT <= "0000";
        elsif (CLOCK'event and CLOCK = '1') then
            UP_CNT <= UP_CNT - 1;
        end if;
    end process;
    DN_COUNTER: (CLOCK, RESET_INT)
    begin
        if (RESET_INT = '1') then
            DN_CNT <= "1111";
        elsif (CLOCK'event and CLOCK = '1') then
            DN_CNT <= DN_CNT - 1;
        end if;
    end process;
    UPCNT <= UP_CNT;
    DNCNT <= DN_CNT;
end XILINX;
```

- Use GSR Verilog Example.

```
////////////////////////////////////////////
// USE_GSR.V Version 1.0                  //
```

```
// The signal RESET initializes all registers//
// Using the global reset resources (STARTUP)//
// December 1997                             //
////////////////////////////////////////////////
module use_gsr ( CLOCK, RESET, UPCNT, DNCNT);
input CLOCK, RESET;
output [3:0] UPCNT;
output [3:0] DNCNT;

reg [3:0] UPCNT;
reg [3:0] DNCNT;

STARTUP U1 (.GSR(RESET));

always @ (posedge CLOCK or posedge RESET) begin
    if (RESET) begin
        UPCNT = 4'b0000;
        DNCNT = 4'b1111;
    end else begin
        UPCNT = UPCNT + 1'b1;
        DNCNT = DNCNT - 1'b1;
    end
end
endmodule
```

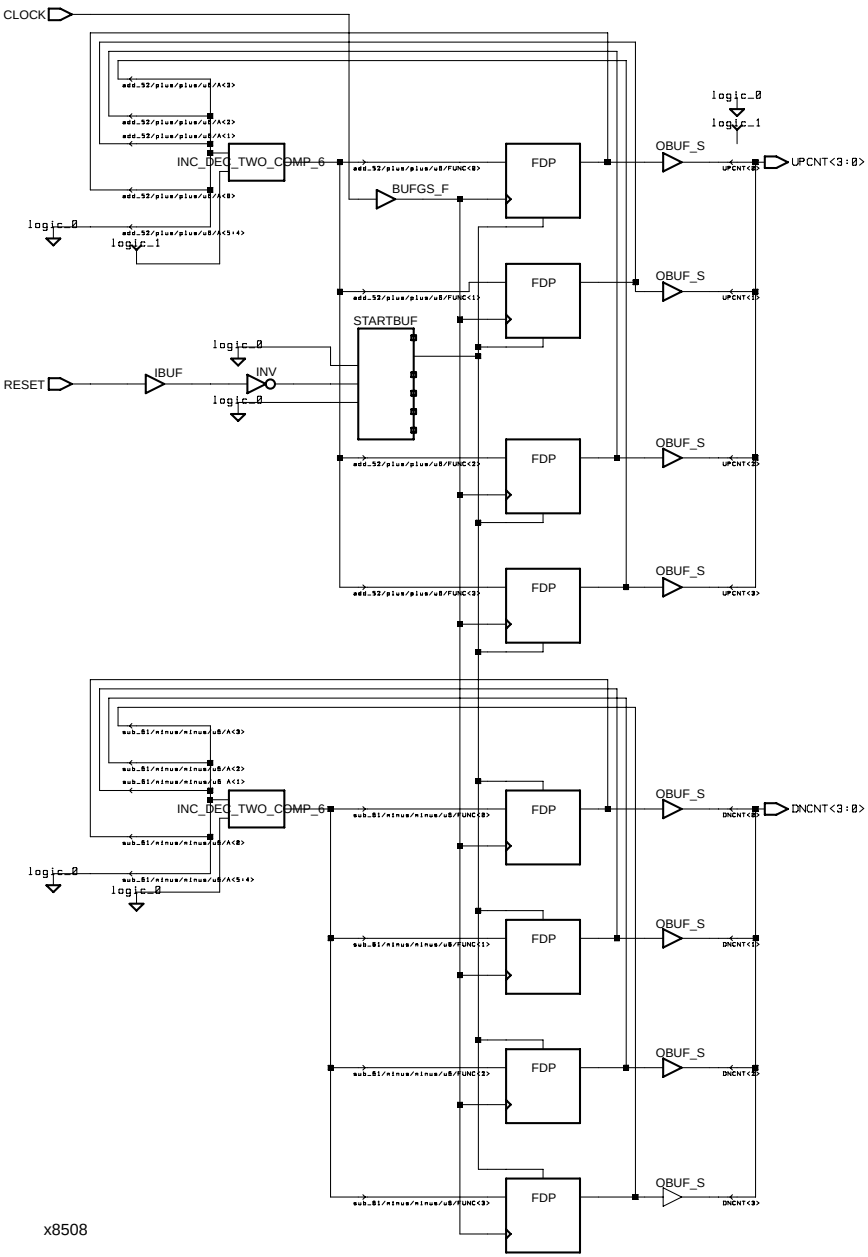

Figure 4-5 Active_Low_GSR Implemented with Gates

Design Example with Active Low GSR Signal

The Active_Low_GSR design is identical to the Use_GSR design except an INV is instantiated and connected between the RESET port and the STARTUP block. Also, a Set Don't Touch (or equivalent) attribute is added to the synthesis tool script for both the INV and STARTUP, or STARTBUF (VHDL) symbols. By instantiating the inverter, the global set/reset signal is now active low (logic level 0 resets all FPGA flip-flops). The inverter is absorbed into the STARTUP block in the device and no CLB resources are used to invert the signal. This is not necessary with many synthesis tools. If all registers and latches are described in the RTL code as reset or set, then a GSR is inferred. Some tools also give you the option to select any signal as the GSR net. This allows you to correct problems if the RTL code does not completely describe the GSR behavior. However, the RTL code will not match the place and route behavior because not all registers are described as set or reset with the GSR signal. Some tools provide a report of the inferred registers that are missing the GSR behavior, and allow you to change the RTL behavior. The following examples show VHDL and Verilog Active_Low_GSR designs.

- Active Low GSR VHDL Example

```
----------------------------------------------------
-- ACTIVE_LOW_GSR.VHD Version 1.0                  --
-- The signal RESET is inverted before being       --
-- connected to the GSRIN pin of the STARTBUF      --
-- The inverter will be absorbed by the STARTBUF   --
-- September 1997                                  --
----------------------------------------------------

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

entity active_low_gsr is
    port ( CLOCK: in STD_LOGIC;
           RESET: in STD_LOGIC;
           UPCNT: out STD_LOGIC_VECTOR (3 downto 0);
           DNCNT: out STD_LOGIC_VECTOR (3 downto 0));
end active_low_gsr;
```

```
architecture XILINX of active_low_gsr is

    component INV
      port (I: in  STD_LOGIC;
            O: out STD_LOGIC);
    end component;

    component STARTBUF
      port (GSRIN:  in  STD_LOGIC;
            GSROUT: out STD_LOGIC);
    end component;

    signal RESET_NOT:     STD_LOGIC;
    signal RESET_NOT_INT: STD_LOGIC;
    signal UP_CNT:        STD_LOGIC_VECTOR (3 downto 0);
    signal DN_CNT:        STD_LOGIC_VECTOR (3 downto 0);

  begin

  U1: INV port map(I => RESET, O => RESET_NOT);

  U2: STARTBUF port map(GSRIN=>RESET_NOT,
                        GSROUT=>RESET_NOT_INT);

  UP_COUNTER: process(CLOCK, RESET_NOT_INT)
  begin
    if (RESET_NOT_INT = '1') then
       UP_CNT <= "0000";
    elsif (CLOCK'event and CLOCK = '1') then
       UP_CNT <= UP_CNT + 1;
    end if;
 end process;

 DN_COUNTER: process(CLOCK, RESET_NOT_INT)
 begin
    if (RESET_NOT_INT = '1') then
        DN_CNT <= "1111";
   elsif (CLOCK'event and CLOCK = '1') then
        DN_CNT <= DN_CNT - 1;
  end if;
```

```
  end process;

  UPCNT <= UP_CNT;
  DNCNT <= DN_CNT;

end XILINX;
```

- Active Low GSR Verilog Example

```
////////////////////////////////////////////
// ACTIVE_LOW_GSR.V Version 1.0//
// The signal RESET is inverted before being //
// connected to the GSR pin of the STARTUP bloc/
// The inverter will be absorbed by STARTUP in//
// M1                                        //
// September 1997  //
////////////////////////////////////////////

module active_low_gsr ( CLOCK, RESET, UPCNT,
   DNCNT);
 input        CLOCK, RESET
 output [3:0] UPCNT;
 output [3:0] DNCNT;

    wire       RESET_NOT;
 reg  [3:0] UPCNT;
 reg  [3:0] DNCNT;

 INV U1 (.O(RESET_NOT), .I(RESET));

 STARTUP U2 (.GSR(RESET_NOT));

always @ (posedge CLOCK or posedge RESET_NOT)
begin
  if (RESET_NOT)
begin
   UPCNT = 4'b0000;
      DNCNT = 4'b1111;
 end
 else
        begin
          UPCNT = UPCNT + 1'b1;
```

```
            DNCNT = DNCNT - 1'b1;
          end
        end
endmodule
```

Implementing Inputs and Outputs

FPGAs have limited logic resources in the user-configurable inputs/output blocks(IOB). You can move logic that is normally implemented with CLBs to IOBs. By moving logic from CLBs to IOBs, additional logic can be implemented in the available CLBs. Using IOBs also improves design performance by increasing the number of available routing resources.

The XC4000XLA, Spartan, and Spartan-XL devices have different IOB functions. The following sections provide a general description of the IOB function in these devices. A description of how to manually implement additional I/O features is also provided.

XC4000XLA and Spartan/Spartan-XL IOBs

You can configure XC4000XLA and Spartan/Spartan-XL IOBs as input, output, or bidirectional signals. You can also specify pull-up or pull-down resistors, independent of the pin usage.

These various buffer and I/O structures can be inferred from commands executed in a script or in your synthesis tool. You can add attributes to these commands to further control pull-up, pull-down, and clock buffer insertion, as well as slew-rate control. Some tools operate on I/Os by selecting a chip level (inserts I/O) or module level (no I/O) synthesis.

Inputs

The buffered input signal that drives the data input of a storage element can be configured as either a flip-flop or a latch. Additionally, the buffered signal can be used in conjunction with the input flip-flop or latch, or without the register.

If an IOB or register is instantiated in your HDL code, you may not be able to use the Set Port Is Pad (or equivalent) command on that port. Doing so may automatically infer a buffer on that port and create an invalid double-buffer structure. This varies with the tool you are

using. Check with your synthesis vendor to see if partial instantiation interferes with automatic I/O insertion or the use of IOB registers.

Registers that connect to an input or output pad and require a Direct Clear or Preset pin are not implemented by the synthesis tool in the IOB. The VHDL emulation of GSR on these registers prevents them from being pulled into the IOB. The VHDL emulation of GSR through direct clear or preset pins is described in the "Simulating Your Design" chapter. If GSR behavior is not completely described, automatic inferencing of GSR does not occur. In this case, instantiate STARTBUF in VHDL, and fully describe the GSR behavior except for registers that you want in the IOB. In VHDL, these registers do not initialize pre-route, but do indicate X's until the first data is registered. However, they do initialize properly during back-annotation. Verilog models initialize properly and do not interfere with the automatic use of IOB registers instead of CLB registers.

Outputs

The output signal can be registered, 3-stated or a direct output. The register is a positive-edge triggered flip-flop and the clock polarity can be inverted inside the IOB. (Xilinx software automatically optimizes any inverters into the IOB.) The XC4000XLA and Spartan/Spartan-XL output buffers can sink 12 mA. Two adjacent outputs can be inter-connected externally to sink up to 24mA.

Note Most FPGA synthesis tools can optimize flip-flops attached to output pads into the IOB. However, some of these tools cannot optimize flip-flops into an IOB configured as a bidirectional pad. Refer to your synthesis tool documentation for more information.

XC4000XLA Output Multiplexer/2-Input Function Generator

A function added to XC4000XLA and Spartan-XL families is a two input multiplexer connected to the IOB output allowing the output clock to select either the output data or the IOB clock enable as the output pad. This allows you to share output pins between two signals, effectively doubling the number of device outputs without requiring a larger device or package. Additionally, this multiplexer can be configured as a two-input function generator allowing you to implement any 2-input logic function in the IOB thus freeing up addi-

tional logic resources in the device and allowing for very fast pin-to-pin data paths.

To use the output multiplexer (OMUX), you must instantiate it in your code. See the following VHDL and Verilog examples. Instantiation of the other types of two-input output primitives (such as OAND2, OOR2, and OXOR2) are similar to these examples.

Note Since the OMUX uses the IOB output clock and clock enable routing structures, the output flip-flop (OFD) can not be used within the same IOB. The input flip-flop (IFD) can be used if the clock enable is not used.

- Output Multiplexer VHDL Example

```
----------------------------------------
-- OMUX_EXAMPLE.VHD                   --
-- Example of OMUX instantiation      --
-- For an XC4000EX/XL/XV device       --
-- HDL Synthesis Design Guide for FPGAs --
-- August 1997                        --
----------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

entity  omux_example is
 port (DATA_IN: in STD_LOGIC_VECTOR (1 downto 0);
SEL: in STD_LOGIC
          DATA_OUT: out STD_LOGIC);
end omux_example;
architecture XILINX of omux_example is
component OMUX2
 port (D0, D1, S0 : in  STD_LOGIC;
       O :          out STD_LOGIC);
end component;
begin

DUEL_OUT: OMUX2 port map (O=>DATA_OUT,
 D0=>DATA_IN(0), D1=>DATA_IN(1), S0=>SEL);
end XILINX;
```

- Output Multiplexer Verilog Example

```
////////////////////////////////////////
// OMUX_EXAMPLE.V                     //
```

```
// Example of instantiating an OMUX2    //
// in an XC4000EX/XL IOB                //
// HDL Synthesis Design Guide for FPGAs //
// August 1997                          //
//////////////////////////////////////////
module omux_example (DATA_IN, SEL, DATA_OUT) ;
input  [1:0] DATA_IN ;
input        SEL ;
output       DATA_OUT ;
OMUX2 DUEL_OUT (.O(DATA_OUT), .D0(DATA_IN[0]),
 .D1(DATA_IN[1]), .S0(SEL));
endmodule
```

Bi-directional I/O

You can create bi-directional I/O with one or a combination of the following methods.

- Behaviorally describe the I/O path
- Structurally instantiate appropriate IOB primitives

Xilinx FPGA IOBs consist of a direct input path into the FPGA through an input buffer (IBUF) and an output path to the FPGA pad through a high impedance buffer (OBUFT). The input path can be registered or latched; the output path can be registered. If you instantiate or behaviorally describe the I/O, you must describe this bi-directional path in two steps. First, describe an input path from the declared INOUT port to a logic function or register. Second, describe an output path from an internal signal or function in your code to a high impedance output with a high impedance control signal that can be mapped to an OBUFT.

You should always describe the I/O path at the top level of your code. If the I/O path is described in a lower level module, your synthesis tool may incorrectly create the I/O structure.

Inferring Bi-directional I/O

This section includes VHDL and Verilog examples that show how to infer a bi-directional I/O. In these examples, the input path is latched by a CLB latch that is gated by the active high READ_WRITE signal.

The output consists of two latched outputs with an AND and OR, and connected to a described high impedance buffer. The active low READ_WRITE signal enables the high impedance gate.

- Inferring a Bi-directional Pin VHDL Example

```
------------------------------------------------------
--  BIDIR_INFER.VHD                                 --
--  Example of inferring a Bi-directional pin --
--  August 1997                                     --
------------------------------------------------------

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
entity bidir_infer is

port (DATA :       inout STD_LOGIC_VECTOR(1 downto 0);
    READ_WRITE : in    STD_LOGIC);
end bidir_infer;
architecture XILINX of bidir_infer is

   signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);

  begin

 process(READ_WRITE, DATA)
 begin
    if (READ_WRITE = '1') then
        LATCH_OUT <= DATA;
    end if;
 end process;
  process(READ_WRITE, LATCH_OUT)
    begin
    if (READ_WRITE = '0') then
        DATA(0) <= LATCH_OUT(0) and LATCH_OUT(1);
        DATA(1) <= LATCH_OUT(0) or LATCH_OUT(1);
    else
        DATA(0) <= 'Z';
        DATA(1) <= 'Z';
    end if;
 end process;
end XILINX;
```

- Inferring a Bi-directional Pin Verilog Example

```
////////////////////////////////////////////////////////////////
// BIDIR_INFER.V Version 1.1  //
// This is an example of an inference of a bi-directional //
// signal.                                                //
// Note: Logic description of port should always be on //
// top-level                                              //
// code when using Synopsys Compiler and verilog.//
// March 1998                                             //
////////////////////////////////////////////////////////////////
module bidir_infer (DATA, READ_WRITE);
    input       READ_WRITE ;
inout [1:0] DATA ;

reg   [1:0] LATCH_OUT ;

always @ (READ_WRITE or DATA)
 begin
 if (READ_WRITE == 1'b1)
     LATCH_OUT <= DATA;
 end

assign DATA[0] = READ_WRITE ?  1'bZ : (LATCH_OUT[0] & LATCH_OUT[1]);
assign DATA[1] = READ_WRITE ? 1'bZ : (LATCH_OUT[0] | LATCH_OUT[1]);
endmodule
```

Instantiating Bi-directional I/O

Instantiating the bi-directional I/O gives you more control over the implementation of the circuit; however, as a result, your code is more architecture-specific and usually more verbose. The VHDL and Verilog examples in this section are identical to the examples in the "Inferring B-directional I/O" section; however, since there is more control over the implementation, an input latch is specified rather than the CLB latch inferred in the previous examples. The following examples are a more efficient implementation of the same circuit.

When instantiating I/O primitives, do not specify the Set Port Is Pad (or equivalent) command on the instantiated ports to prevent the I/O buffers from being inferred by your synthesis tool. This precaution also prevents the creation of an illegal structure.

- Instantiation of a Bi-directional Pin VHDL Example

```
----------------------------------
--  BIDIR_INSTANTIATE.VHD        --
--  Example of an instantiation  --
--  of a Bi-directional pin      --
--  August 1997                  --
----------------------------------
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity bidir_instantiate is

 port (DATA :       inout STD_LOGIC_VECTOR(1 downto 0);
       READ_WRITE : in    STD_LOGIC);
end bidir_instantiate;
architecture XILINX of bidir_instantiate is

    signal LATCH_OUT : STD_LOGIC_VECTOR(1 downto 0);
    signal DATA_OUT :  STD_LOGIC_VECTOR(1 downto 0);
    signal GATE :      STD_LOGIC;

  component ILD_1
      port (D, G : in  STD_LOGIC;
          Q    : out STD_LOGIC);
  end component;

  component OBUFT_S
      port (I, T : in  STD_LOGIC;
          O :    out STD_LOGIC);
  end component;
  begin

  DATA_OUT(0) <= LATCH_OUT(0) and LATCH_OUT(1);
  DATA_OUT(1) <= LATCH_OUT(0) or LATCH_OUT(1);

  GATE <= not READ_WRITE;

  INPUT_PATH_0 : ILD_1
    port map (D => DATA(0), G => GATE,
            Q => LATCH_OUT(0));

  INPUT_PATH_1 : ILD_1
```

```
    port map (D => DATA(1), G => GATE,
            Q => LATCH_OUT(1));

  OUPUT_PATH_0 : OBUFT_S
    port map (I => DATA_OUT(0), T => READ_WRITE,
            O => DATA(0));

  OUPUT_PATH_1 : OBUFT_S
    port map (I => DATA_OUT(1), T => READ_WRITE,
            O => DATA(1));
end XILINX;
```

- Instantiation of a Bi-directional Pin Verilog Example

```
     ////////////////////////////////////////
    // BIDIR_INSTANTIATE.V                  //
   // This is an example of an instantiation //
  // of a bi-directional port.             //
 // August 1997                           //
////////////////////////////////////////
module bidir_instantiate (DATA, READ_WRITE);

 input       READ_WRITE ;
 inout [1:0] DATA ;

reg   [1:0] LATCH_OUT ;
wire  [1:0] DATA_OUT ;
 wire       GATE ;

assign GATE = ~READ_WRITE;

assign DATA_OUT[0] = LATCH_OUT[0] & LATCH_OUT[1];
assign DATA_OUT[1] = LATCH_OUT[0] | LATCH_OUT[1];
// I/O primitive instantiation

ILD_1 INPUT_PATH_0 (.Q(LATCH_OUT[0]), .D(DATA[0]), .G(GATE));

ILD_1 INPUT_PATH_1 (.Q(LATCH_OUT[1]), .D(DATA[1]), .G(GATE));
OBUFT_S      OUPUT_PATH_0      (.O(DATA[0]),      .I(DATA_OUT[0]),
.T(READ_WRITE));

OBUFT_S      OUPUT_PATH_1      (.O(DATA[1]),      .I(DATA_OUT[1]),
.T(READ_WRITE));
```

```
endmodule
```

Delay and Slew Rate

You can avoid external hold-time requirements by using the IOB input flip-flops and latches with a delay block between the external pin and the D input. You can remove this default delay by instantiating a flip-flop or latch with a NODELAY attribute. The NODELAY attribute decreases the setup-time requirement and introduces a small hold time.

FPGA Express currently does not support the NODELAY attribute. You can set the NODELAY attribute through the user constraints file (.ucf).

In LeonardoSpectrum, set the NODELAY attribute in a TCL script or from the command line. See the following example.

set_attribute -port *data_in* **-name NODELAY -value TRUE**

or

set_attribute -instance *my_ifd* **-name NODELAY -value TRUE**

or

set_attribute -net *net_name* **-name NODELAY -value TRUE.**

In Synplify, use the xc_nodelay attribute in the constraint editor or in your HDL code. Slew rate on the IOB outputs can be controlled by declaring either FAST or SLOW attributes.

In FPGA Express, slew rate can be specified after elaborating the design (before optimization) in the Constraints Editor graphical user interface.

In LeonardoSpectrum, set the slew rate in a TCL script or from the command line. The command line attribute would look like the following.

set_attribute -port <*portname*> **-name FAST**

In Synplify, use the xc_fast attribute in the constraints editor or in your HDL code.

Pull-ups and Pull-downs

XC4000XLA, Spartan, and Spartan-XL devices have programmable pull-up and pull-down resistors available in the I/O. The resisters are available regardless of whether it is configured as an input, output, or bi-directional I/O. By default, all unused IOBs are configured as an input with a pull-up resistor. The value of the pull-ups and pull-downs vary depending on operating conditions and device process variances but should be approximately 50 K Ohms to 100 K Ohms. If a more precise value is required, use an external resistor. Refer to your synthesis tool documentation for information on how to specify internal pull-up or pull-down I/O resistors.

Specifying Pad Locations

Although Xilinx recommends allowing the software to select pin locations to ensure the best possible pin placement in terms of design timing and routing resources, sometimes you must define the pad locations prior to placement and routing. You can assign pad locations either from the Xilinx Implementation Tool's script prior to writing out the netlist file, or from a User Constraints File (UCF). Use one or the other method, but not both.

For XC4000XLA designs it is best to use a higher placement effort in the software when allowing your synthesis tool to pick the I/Os.

Refer to your synthesis tool documentation for the correct syntax for configuring your I/O with the LOC property. Also, refer to *The Programmable Logic Data Book* or the Xilinx Web site (http://support.xilinx.com) for the pad locations for your device and package.

Moving Registers into the IOB

IOBs contain an input register or latch and an output register. IOB inputs can be register or latch inputs as well as direct inputs to the device array. Registers without a direct reset or set function can be moved into IOBs. Moving registers or latches into IOBs may reduce the number of CLBs used and decreases the routing congestion. In addition, moving input registers and latches into the IOB reduces the external setup time, as shown in the following figure.

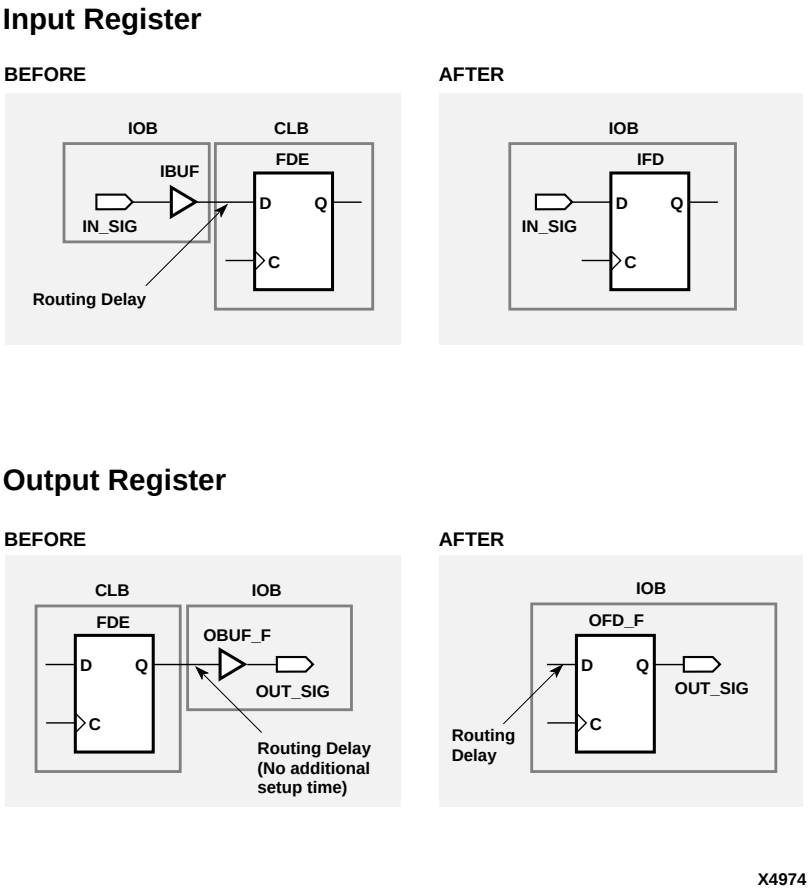

Figure 4-6 Moving Registers into the IOB

Although moving output registers into the IOB may increase the internal setup time, it may reduce the clock-to-output delay, as shown in this figure. Most FPGA synthesis tools automatically move registers into IOBs if the Preset, Clear, and Clock Enable pins are *not* used.

Use –pr Option with Map

Use the –pr (pack registers) option when running MAP. The –pr {i | o | b} (input | output | both) option specifies to the MAP program to move registers into IOBs under the following circumstances.

1. The input of the register must be connected to an input port, or the Q pin must be connected to an output port. For the XC4000XLA this applies to non-I/O latches, as well as flip-flops.
2. The flip-flop does not use an asynchronous set or reset signal.
3. In XC4000, Spartan, and XC3000 devices, a flop/latch is not added to an IOB if it has a BLKNM or LOC conflict with the IOB.
4. In XC4000XLA or Spartan/Spartan-XL devices, a flop/latch is not added to an IOB if its control signals (clock or clock enable) are not compatible with those already defined in the IOB. This occurs when a flip-flop (latch) is already in the IOB with different clock or clock enable signals, or when the XC4000XLA or Spartan-XL output MUX is used in the same IOB. In Virtex, clock enable can be different for each flip-flop, but clock and set/reset must be the same for all registers and latches in one IOB.

Using Unbonded IOBs (XC4000XLA and Spartan/ Spartan-XL Only)

In some package/device pairs, not all pads are bonded to a package pin. You can use these unbonded IOBs and the flip-flops inside them in your design by instantiating them in the HDL code. The VHDL and Verilog examples in this section show how to instantiate unbonded IOB flip-flops in a 4-bit shift register in XC4000XLA or Spartan/Spartan-XL devices.

Note The synthesis tool compilers cannot infer unbonded primitives. Refer to your synthesis tool documentation for a list of library primitives that can be used for instantiations.

4-bit Shift Register Using Unbonded I/O VHDL Example

```
------------------------------------------------
-- UNBONDED_IO.VHD Version 1.0                 --
```

```
-- XC4000 LCA has unbonded IOBs which have     --
-- storage elements that can be used to build  --
-- shift registers.                            --
-- Below is a 4-bit Shift Register using       --
-- Unbonded IOB Flip Flops                     --
-- Xilinx HDL Synthesis Design Guide for FPGAs --
-- May 1997                                    --
-------------------------------------------------
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity unbonded_io is
    port (A, B: in STD_LOGIC;
        CLK: in STD_LOGIC;
        Q_OUT: out STD_LOGIC);
end unbonded_io;
architecture XILINX of unbonded_io is

    component IFD_U  -- Unbonded Input FF with INIT=Reset
        port (Q: out std_logic;
            D, C: in  std_logic);
    end component;

    component IFDI_U -- Unbonded Input FF with INIT=Set
        port (Q: out std_logic;
            D, C: in  std_logic);
    end component;

    component OFD_U -- Unbonded Output FF with INIT=Reset
        port (Q: out std_logic;
            D, C: in  std_logic);
    end component;

    component OFDI_U -- Unbonded Output FF with INIT=Set
        port (Q: out std_logic;
            D, C: in  std_logic);
    end component;
--- Internal Signal Declarations -----
    signal U_Q : STD_LOGIC_VECTOR (3 downto 0);
    signal U_D : STD_LOGIC;
begin
U_D <= A and B;
```

```
Q_OUT <= U_Q(0);
U3: OFD_U  port map (Q => U_Q(3),
                     D => U_D,
                     C => CLK);

U2: IFDI_U port map (Q => U_Q(2),
                     D => U_Q(3),
                     C => CLK);

U1: OFDI_U port map (Q => U_Q(1),
                     D => U_Q(2),
                     C => CLK);

U0: IFD_U  port map (Q => U_Q(0),
                     D => U_Q(1),
                     C => CLK);
end XILINX;
```

4-bit Shift Register Using Unbonded I/O Verilog Example

```
////////////////////////////////////////////////
// UNBONDED.V                                 //
// XC4000 family has unbonded IOBs which have  //
// storage elements that can be used to build  //
// functions like shift registers.              //
// Below is a 4-bit Shift Register using Unbonded //
// IOB Flip Flops                             //
// HDL Synthesis Design Guide for FPGAs       //
// May 1997                                   //
////////////////////////////////////////////////

module unbonded_io (A, B, CLK, Q_OUT);

input A, B, CLK;
output Q_OUT;

wire[3:0] U_Q;
wire      U_D;
```

```
assign U_D = A & B;
assign Q_OUT = U_Q[0];

    OFD_U U3 (.Q(U_Q[3]), .D(U_D), .C(CLK));
    IFDI_U U2 (.Q(U_Q[2]), .D(U_Q[3]), .C(CLK));

    OFDI_U U1 (.Q(U_Q[1]), .D(U_Q[2]), .C(CLK));

    IFD_U  U0 (.Q(U_Q[0]), .D(U_Q[1]), .C(CLK));
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 are acceptable for PAL and gate array architectures. However, because FPGAs have many flip-flops and narrow 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. 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). Refer to

your synthesis tool documentation for information on attribute-driven state machine synthesis.

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.

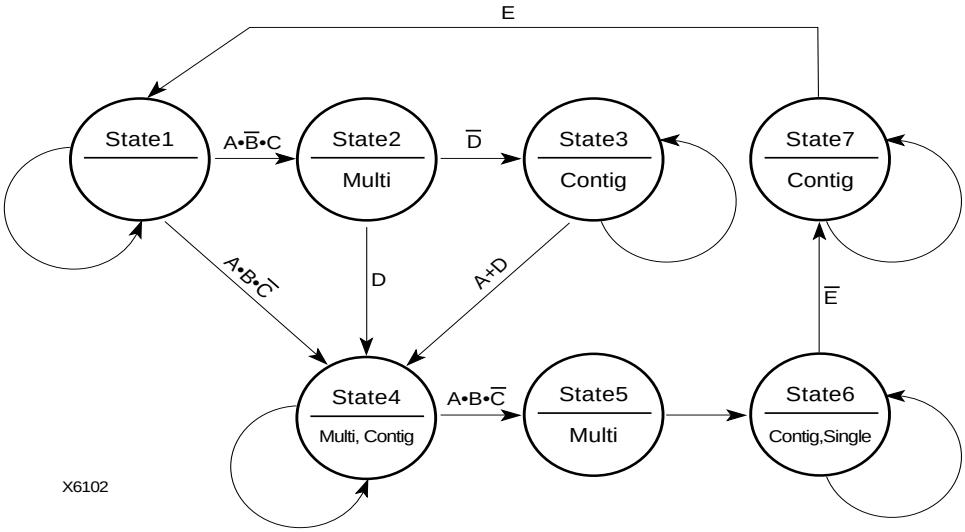

Figure 4-7 State Machine Bubble Diagram

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   --
-- May 1997                                    --
```

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

```
////////////////////////////////////////////////
// BINARY.V Version 1.0                       //
// Example of a binary encoded state machine  //
// May 1997                                   //
////////////////////////////////////////////////
module binary (CLOCK, RESET, A, B, C, D, E, SINGLE, MULTI, CONTIG);

input    CLOCK, RESET;
input    A, B, C, D, E;
output   SINGLE, MULTI, CONTIG;

reg      SINGLE, MULTI, CONTIG;
// Declare the symbolic names for states
parameter [2:0]
    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 or posedge RESET)
    begin
        if (RESET == 1'b1)
            CS = S1;
        else
            CS = NS;
    end
    always @ (CS or A or B or C or D or D or E)
    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.

Note Refer to the previous VHDL and Verilog Binary Encoded State Machine examples for the complete Case statement portion of the code.

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//
// May 1997                                     //
////////////////////////////////////////////////

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

input  CLOCK, RESET;
input  A, B, C, D, E;
output SINGLE, MULTI, CONTIG;
```

```
reg    SINGLE, MULTI, CONTIG;

// Declare the symbolic names for states
parameter [2:0]
    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 or posedge RESET)
begin
    if (RESET == 1'b1)
        CS = S1;
    else
        CS = NS;
end

always @ (CS or A or B or C or D or D or E)
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. If you are using FPGA Express, use enumerated type, and avoid using the "when others" construct in the VHDL case statement. This construct can result in a very large state machine.

Note Refer to the previous VHDL and Verilog Binary Encoded State Machine examples for the complete Case statement portion of the code.

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 FPGAs  //
// May 1997                                    //
////////////////////////////////////////////////

module one_hot (CLOCK, RESET, A, B, C, D, E,
                SINGLE, MULTI, CONTIG);
```

```
input   CLOCK, RESET;
input   A, B, C, D, E;
output  SINGLE, MULTI, CONTIG;

reg SINGLE, MULTI, CONTIG;

// Declare the symbolic names for states
parameter [6:0]
    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
reg [2:0] CS;
reg [2:0] NS;

// state_vector CS

 always @ (posedge CLOCK or posedge RESET)
 begin
      if (RESET == 1'b1)
          CS = S1;
      else
          CS = NS;
end

always @ (CS or A or B or C or D or D or E)
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
        .
        .
        .
```

Accelerate FPGA Macros with One-Hot Approach

Most synthesis tools provide a setting for finite state machine (FSM) encoding. This setting will prompt synthesis tools 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, grey, sequential, etc.

In FPGA Express, FSM encoding is set to “One-Hot” by default. To change this setting, select Synthesis-> Options -> Project Tab. Available options are: One-Hot, Binary, and Zero One-Hot.

In LeonardoSpectrum, FSM encoding is set to “Auto” by default which differs depending on Bit Width of your state machine. To change this setting to a specific value, select Input tab. Available options are: Binary, One-Hot, Random, Gray, and Auto.

In Synplify, Symbolic FSM Complier option can be accessed from the main GUI. When set, the default value is One-Hot. However, you may override the default on a register by register bases with syn_encoding directive / attribute. Available options are: One-Hot, Gray, Sequential, and Safe.

Summary of Encoding Styles

In the three 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 three 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.

Note Refer to your synthesis tool documentation for instructions on how to extract the state machine and change the encoding style.

Comparing Synthesis Results for Encoding Styles

The following table summarizes the synthesis results from the different methods used to encode the state machine in the three previous VHDL and Verilog state machine examples. The results are for an XC4013XLABG256-7 device.

Note The Timing Analyzer was used to obtain the timing results in this table.

Table 4-2 State Machine Encoding Styles Comparison (XC4013XLABG256-7)

Comparison	One-Hot	Binary	Enum (One-hot)
Occupied CLBs	6	7	6
CLB Flip-flops	7	3	7
PadToSetup	6.6 ns (3a)	7.2 ns (4)	3.6 ns (3)
ClockToPad	10.4 ns (3)	10.8 ns (3)	9.9 ns (3)
ClockToSetup	7.8 ns (4)	8.5 ns (4)	4.8 ns (3)

a. The number in parentheses represents the CLB block level delay.

The binary-encoded state machine has the longest ClockToSetup delay. Generally, the FSM extraction tool provides the best results because the compiler reduces any redundant states and optimizes the state machine after the extraction.

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.

- VHDL Example

```
SYNC_PROC: process (CLOCK, RESET)
begin
    if (RESET='1') then
       CS <= s1;
```

- Verilog Example

```
always @ (posedge CLOCK or posedge RESET)
begin
   if (RESET == 1'b 1)
       CS = S1;
```

  Alternatively, you can assign an INIT=S attribute to the initial state register to specify the initial state. Refer to your synthesis tool documentation for information on assigning this attribute.

  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 Generate Modules

Xilinx FPGAs feature carry logic elements that can be used for optimal implementation of operators and generate modules. Synthesis tools infer the carry logic automatically when a specific coding style or operator is used.

Adder and Subtracter

Synthesis tools will infer carry logic in XC4000XLA, Spartan, and Spartan-XL devices when an adder and Subtracter is described (+ or - operator).

Multiplier

Synthesis tools will utilize the carry logic when a multiplier is described.

LeonardoSpectrum Pipelined Multiplier Example

- VHDL Example

```
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;
```

- Verilog Example.

```
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
```

Counters

When describing a counter in HDL, the arithmetic operator '+' will infer the carry chain.

```
count <= count + 1; -- This will infer carry logic
```

This implementation will provide a very effective solution especially for all purpose counters.

Below is an example of a loadable binary counter:

- VHDL Example

```
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, rst)
 begin
   if rst = '1' then
     count <= (others => '0');
   elsif rising_edge(clk) then
       if ld = '1' then
           count <= d;
```

```
          elsif ce = '1' then
              count <= count + '1';
          end if;
      end if;
    end process;
    q <= count;
   end behave;
```

- Verilog Example

```
module counter(d, ld, ce, clk, rst, q);
  input [7:0]  d;
  input        ld, ce, clk, rst;
  output [7:0] q;
  reg [7:0]    count;
  always @(posedge clk or posedge rst)
    begin
        if (rst)
          count <= 0;
        else if (ld)
          count <= d;
        else if (ce)
          count <= count + 1;
    end
  assign q = count;
endmodule
```

For application that require faster counters, LFSR can implement high performance and area efficient counters. LFSR will require very minimum logic (only a XOR or XNOR feedback).

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

- VHDL Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity johnson is
    generic (size : integer := 3);
    port (clk   : in std_logic;
        reset : in std_logic;
        qout  : out std_logic_vector(size-1 downto 0));
end johnson;
architecture RTL of johnson is
    signal q : std_logic_vector(size-1 downto 0);
begin  -- RTL
    process(clk, reset)
    begin
      if reset = '1' then
          q <= (others => '0');
      elsif clk'event and clk='1' then
          for i in 1 to size - 1 loop
              q(i) <= q(i-1);
          end loop;  -- i
          q(0) <= not q(size-1);
      end if;
    end process;
    qout <= q;
end RTL;
```

- Verilog Example

```
module johnson (clk, reset, q);
parameter size = 4;
  input     clk, reset;
  output [size-1:0] q;
  reg [size-1:0]    q;
  integer           i;
  always @(posedge clk or posedge reset)
    if (reset)
      q <= 0;
    else
```

```
      begin
        for (i=1;i<size;i=i+1)
            q[i] <= q[i-1];
            q[0] <= ~q[size-1];
      end
endmodule // johnson
```

Comparator

Magnitude comparator '>' or '<' will infer carry chain logic and result in fast implementations in Xilinx devices. Equality comparator '==' will be implemented using LUTs

- VHDL Example

```
-- Unsigned 8-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.
module compar(A, B, cmp);
input [7:0] A;
input [7:0] B;
output cmp;
```

```
assign cmp = A >= B ? 1'b1 : 1'b0;
endmodule
```

Implementing Memory

XC4000XLA, Spartan, and Spartan-XL FPGAs provide distributed on-chip RAM and ROM. CLB function generators can be configured as ROM (ROM16X1, ROM32X1); level-sensitive RAM (RAM16X1, RAM 32X1); edge-triggered, single-port (RAM16X1S, RAM32X1S); or dual-port (RAM16x1D) RAM. Level sensitive RAMs are not available for the Spartan and Spartan-XL families.

Note For more information on XC4000 family RAM, refer to the Xilinx Web site (http://support.xilinx.com) or the current release of *The Programmable Logic Data Book*.

Implementing Distributed SelectRAM+

Distributed SelectRAM+ can either be instantiated or inferred.Thefollowing sections describe and give examples of both instantiating and inferring distributed SelectRAM+.

The following RAM Primitives are available for instantiation.

- Static level-sensitive RAM (RAM16x1, RAM32x1). These primitives are not available in Spartan and Spartan-XL.
- Static synchronous single-port RAM (RAM16x1S, RAM32x1S)
- Static synchronous dual-port RAM (RAM16x1D, RAM32x1D)

For more information on distributed SelectRAM+, please see the *Libraries Guide.*

Instantiating Distributed SelectRAM+ in VHDL

Below are VHDL coding examples for RAM instantiation in FPGA Express/ FPGA Compiler II, LeonardoSpectrum, and Synplify.

- FPGA Express/FPGA Compiler II example

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.
library IEEE;
use IEEE.std_logic_1164.all;
--use IEEE.std_logic_unsigned.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

component RAM16x1S is
  port (O : out std_logic;
  D : in std_logic;
        A3, A2, A1, A0 : in std_logic;
        WE, WCLK : in std_logic);
end component;
attribute INIT: string;
attribute INIT of U0: label is "FFFF";
attribute INIT of U1: label is "ABCD";
attribute INIT of U2: label is "BCDE";
attribute INIT of U3: label is "CDEF";

begin
U0 : RAM16x1S
      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
      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
      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
      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;
```

- LeonardoSpectrum example

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.


   library IEEE;

use IEEE.std_logic_1164.all;
entity ram_16x1s is

   generic (init_val : string := "0000" );

   port (O : out std_logic;

        D : in std_logic;
        A3, A2, A1, A0: in std_logic;
        WE, CLK : in std_logic);
end ram_16x1s;


   architecture xilinx of ram_16x1s is


   attribute INIT: string;

attribute INIT of u1 : label is init_val;
component RAM16X1S is port (O : out std_logic;
         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);
end component;

begin
U1 : RAM16X1S port map (O => O, WE => WE,  WCLK =>
    CLK, D => D, A0 => A0, A1 => A1, A2 => A2, A3 =>
   A3);
end xilinx;
library IEEE;
use IEEE.std_logic_1164.all;
```

```
--use IEEE.std_logic_unsigned.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
component ram_16x1s
    generic (init_val: string := "0000");
    port (O : out std_logic;
    D : in std_logic;
          A3, A2, A1, A0 : in std_logic;
          WE, CLK : in std_logic);
end component;
begin
U0 : ram_16x1s generic map ("FFFF")
    port map (O => o(0), WE => we, CLK => clk,
    D => d(0), A0 => a(0), A1 => a(1),
    A2 => a(2),A3 => a(3));
U1 : ram_16x1s generic map ("ABCD")
    port map (O => o(1), WE => we, CLK => clk,
    D => d(1), A0 => a(0), A1 => a(1),
    A2 => a(2), A3 => a(3));
U2 : ram_16x1s generic map ("BCDE")
    port map (O => o(2), WE => we, CLK => clk,
    D => d(2), A0 => a(0), A1 => a(1),
    A2 => a(2), A3 => a(3));
U3 : ram_16x1s generic map ("CDEF")
    port map (O => o(3), WE => we, CLK => clk,
    D => d(3), A0 => a(0), A1 => a(1),
    A2 => a(2), A3 => a(3));
end xilinx;
```

- Synplify example

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.
library IEEE;
use IEEE.std_logic_1164.all;
library xc4000;
use xc4000.components.all;
```

```
library synplify;
use synplify.attributes.all;

entity ram_16x1s is
  generic (init_val : string := "0000" );
  port (O : out std_logic;
        D : in std_logic;
        A3, A2, A1, A0: in std_logic;
        WE, CLK : in std_logic);
end ram_16x1s;

architecture xilinx of ram_16x1s is

attribute xc_props: string;
attribute xc_props of u1 : label is "INIT=" &
   init_val;

begin

U1 : RAM16X1S port map (O => O, WE => WE,  WCLK =>
  CLK, D => D, A0 => A0, A1 => A1, A2 => A2, A3 =>
   A3);

end xilinx;

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

component ram_16x1s
```

```
    generic (init_val: string := "0000");
    port (O : out std_logic;
    D : in std_logic;
          A3, A2, A1, A0 : in std_logic;
          WE, CLK : in std_logic);
end component;

begin
U0 : ram_16x1s generic map ("FFFF")
    port map (O => o(0), WE => we, CLK => clk,
    D =>d(0), A0 => a(0), A1 => a(1),
    A2 => a(2), A3 => a(3));
U1 : ram_16x1s generic map ("ABCD")
    port map (O => o(1), WE => we, CLK => clk,
    D => d(1), A0 => a(0), A1 => a(1),
    A2 => a(2), A3 => a(3));
U2 : ram_16x1s generic map ("BCDE")
    port map (O => o(2), WE => we, CLK => clk,
    D => d(2), A0 => a(0), A1 => a(1),
    A2 => a(2), A3 => a(3));
U3 : ram_16x1s generic map ("CDEF")
    port map (O => o(3), WE => we, CLK => clk,
    D => d(3), A0 => a(0), A1 => a(1),
    A2 => a(2), A3 => a(3));

   end xilinx;
```

Instantiating Distributed SelectRAM+ in Verilog

Below are Verilog coding examples for RAM instantiation in FPGA Express/FPGA Compiler II, LeonardoSpectrum, and Synplify.

- FPGA Express/FPGA Compiler II example.

```
// 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.

// synopsys translate_off
defparam RAM0.INIT="0101", RAM1.INIT="AAAA",
             RAM2.INIT="FFFF", RAM3.INIT="5555";
// synopsys translate_on
assign DATA_BUS = !WE ? DATA_OUT : 4'hz;
// Instantaition of 4 16X1 Synchronous RAMs
// Use the xc_props attribute to pass the INIT
   property
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)) ;
      /* synopsys attribute 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));
      /* synopsys attribute 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));
      /* synopsys attribute 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));
      /* synopsys attribute 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
```

- LeonardoSpectrum example

```
// 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;
// Instantaition of 4 16X1 Synchronous RAMs
// Use the xc_props attribute to pass the INIT
   property
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 example

```
// This example shows how to create a
// 16x4 RAM using Xilinx RAM16X1S component.
`include "xc4000.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;
// Only for Simulation -- the defparam will not
// synthesize
// Use the defparam for RTL simulation.
// There is no defparam needed for Post P&R
//simulation.
// synthesis translate_off
defparam RAM0.INIT="0101", RAM1.INIT="AAAA",
             RAM2.INIT="FFFF", RAM3.INIT="5555";
// synthesis translate_on
assign DATA_BUS = !WE ? DATA_OUT : 4'hz;
// Instantaition of 4 16X1 Synchronous RAMs
// Use the xc_props attribute to pass the INIT
   property
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))
      /* synthesis xc_props="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))
      /* synthesis xc_props="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))
      /* synthesis xc_props="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))
      /* synthesis xc_props="INIT=0101" */;

   endmodule
```

Inferring Distributed SelectRAM+ in VHDL

The following examples provide VHDL coding styles for LeonardoSpectrum and Synplify. FPGA Express / FPGA Compiler II currently do not infer RAM.

- VHDL example of 32x8 (32 words by 8 bits per word) synchronous dual port RAM.

```
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 of 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 examples provide Verilog coding hints for Synplify and LeonardoSpectrum. FPGA Express / FPGA Compiler II currently do not infer RAM.

- Verilog example of 32x8 (32 words by 8 bits per word) synchronous dual port RAM.

```
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 [(1 << addr_width) 1:0];
always @(posedge clk)
  if (we)
    mem[waddr] = d;
always @(mem or raddr)
  o = mem[raddr];
endmodule
```

- Verilog example of 32x8 (32 words by 8 bits per word) synchronous, single-port RAM.

```
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 [(1 << 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

The following examples are RTL VHDL and Verilog ROM coding examples.

RTL Description of a ROM VHDL Example

Note LeonardoSpectrum does not infer ROM.

Use the following coding example for FPGA Express and Synplify.

```
--
--  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 ROM Verilog Example

Note LeonardoSpectrum does not infer ROM.

Use the following coding example for FPGA Express and Synplify.

```
/*
 * 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
```

When using an RTL description of a ROM, the synthesis tool creates ROMs from random logic gates that are implemented using function generators.

Another method for implementing ROMs is instantiating 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.

Note Refer to your synthesis tool documentation for the correct syntax.

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. See the VHDL and Verilog RAM examples in the following section for examples of this property using a RAM primitive.

Implementing Distributed SelectRAM+

Distributed SelectRAM+ can either be instantiated or inferred.Thefollowing sections describe and give examples of both instantiating and inferring distributed SelectRAM+.

The following RAM Primitives are available for instantiation.

- Static level-sensitive RAM (RAM16x1, RAM32x1). These primitives are not available in Spartan and Spartan-XL.
- Static synchronous single-port RAM (RAM16x1S, RAM32x1S)
- Static synchronous dual-port RAM (RAM16x1D, RAM32x1D)

For more information on distributed SelectRAM+, please see the *Libraries Guide.*

Implementing FIFO

Xilinx provides several Application Notes describing the use of FIFO when implementing FPGAs. Please refer to the following Xilinx Application Notes for more information:

- XAPP053: *"Implementing FIFOs in XC4000 Series RAM"* (7/96) (http://www.xilinx.com/xapp/xapp053.pdf)
- XAPP051: *"Synchronous and Asynchronous FIFO Designs"* (9/96) (http://www.xilinx.com/xapp/xapp051.pdf)

Additionally, synchronous FIFO cores are available for XC4000XLA and Spartan/Spartan-XL with the Xilinx CORE Generator.

Using CORE Generator to Implement Memory

If you must instantiate memory, use CORE Generator to create a memory module larger than 32X1 (16X1 for Dual Port). Implementing memory with CORE Generator is similar to implementing any module with CORE Generator except for defining the Memory initialization file. Reference the memory module datasheets that come with every CORE Generator module for specific information on the initialization file.

Implementing Multiplexers

A 4-to-1 multiplexer is efficiently implemented in a single XC4000 or Spartan family CLB. The six input signals (four inputs, two select lines) use the F, G, and H function generators. Multiplexers that are larger than 4-to-1 exceed the capacity of one CLB. For example, a 16-to-1 multiplexer requires five CLBs and has two logic levels. These additional CLBs increase area and delay. Xilinx recommends that you use internal high impedance 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 high impedance 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 the "5-to-1 MUX Implemented with Gates" figure.

Some synthesis tools include commands that allow you to switch between multiplexers with gates or with 3-states. Check with your synthesis vendor for more information.

The VHDL and Verilog designs provided at the end of this section show a 5-to-1 multiplexer built with high impedance buffers. The high impedance buffer version of this multiplexer has one-hot encoded selector inputs and requires five select inputs (SEL<4:0>). The schematic representation of these designs is shown in the "5-to-1 MUX Implemented with Gates" figure.

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
-- May 1997

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
* May 1997 */


module mux_gate (A,B,C,D,E,SEL,SIG);

input A,B,C,D,E;
input [2:0] SEL;
output SIG;
reg SIG;

    always @ (A or B or C or D or SEL)
    case (SEL)
        3'b000:
          SIG=A;
        3'b001:
          SIG=B;
        3'b010:
          SIG=C;
        3'b011:
          SIG=D;
        3'b100:
          SIG=E;
default: SIG=A;
    endcase
endmodule
```

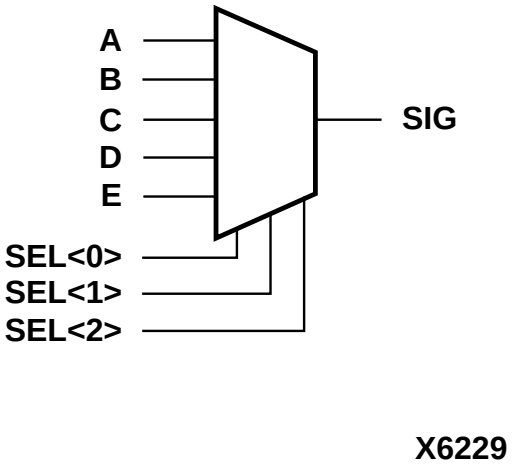

Figure 4-8 5-to-1 MUX Implemented with Gates

Mux Implemented with BUFTs VHDL Example

The following example shows a MUX implemented with BUFTs.

```
-- MUX_TBUF.VHD
-- 5-to-1 Mux Implemented in 3-State Buffers
-- May 1997

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

entity mux_tbuf is
port (SEL: in STD_LOGIC_VECTOR (4 downto 0);
A,B,C,D,E: in STD_LOGIC;
```

```
      SIG: out STD_LOGIC);
end mux_tbuf;

architecture RTL of mux_tbuf is
begin

    SIG <= A when (SEL(0)='0') else 'Z';
    SIG <= B when (SEL(1)='0') else 'Z';
    SIG <= C when (SEL(2)='0') else 'Z';
    SIG <= D when (SEL(3)='0') else 'Z';
    SIG <= E when (SEL(4)='0') else 'Z';
end RTL;
```

Mux Implemented with BUFTs Verilog Example

The following example shows a MUX implemented with BUFTs.

```
/* MUX_TBUF.V
 * May 1997 */

module mux_tbuf (A,B,C,D,E,SEL,SIG);

input A,B,C,D,E;
input [4:0] SEL;
output SIG;
reg SIG;

    always @ (SEL or A)
    begin
      if (SEL[0]==1'b0)
          SIG=A;
      else
          SIG=1'bz;
    end

always @ (SEL or B)
begin
      if (SEL[1]==1'b0)
          SIG=B;
      else
          SIG=1'bz;
    end
```

```
always @ (SEL or C)
begin
    if (SEL[2]==1'b0)
        SIG=C;
    else
        SIG=1'bz;
    end

always @ (SEL or D)
begin
    if (SEL[3]==1'b0)
        SIG=D;
    else
        SIG=1'bz;
    end

always @ (SEL or E)
begin
    if (SEL[4]==1'b0)
        SIG=E;
    else
        SIG=1'bz;
    end
endmodule
```

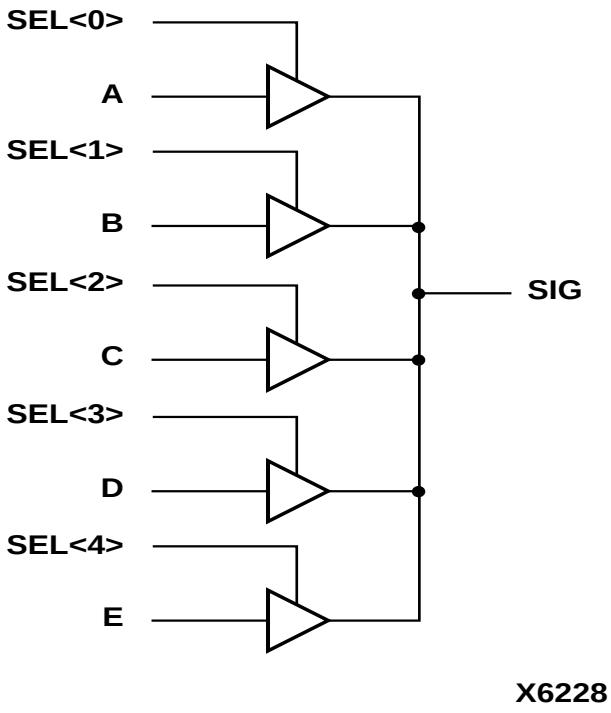

Figure 4-9 5-to-1 MUX Implemented with BUFTs

A comparison of timing and area for a 5-to-1 multiplexer built with gates and high impedance buffers in an XC4013XLABG256-07 device is provided in the following table. When the multiplexer is implemented with high impedance buffers, no CLBs are used and the delay is smaller.

Table 4-3 Timing/Area for 5-to-1 MUX (XC4013XLABG256-07)

Timing/Area	Using BUFTs	Using Gates
Longest Path	SEL(4) to SIG	D to SIG

Table 4-3 Timing/Area for 5-to-1 MUX (XC4013XLABG256-07)

Timing/Area	Using BUFTs	Using Gates
Timing	10.09 ns (3 level of logic)	9.65 ns (4 levels of logic)
Area	1 CLBs, 5 BUFTs	2 CLBs

Using Pipelining

You can use pipelining to dramatically improve device performance. 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 because 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 translate these constraints to Xilinx implementation constraints. Check your synthesis tool documentation for information on multi-cycle paths. 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; and the setup time of the destination register.

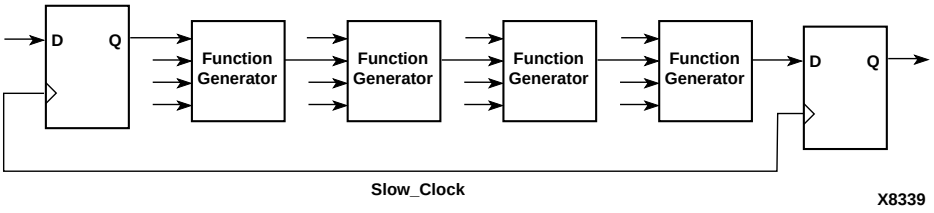

Figure 4-10 Before Pipelining

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; and the setup time of the destination register. In this example, the system clock runs much faster than in the previous example.

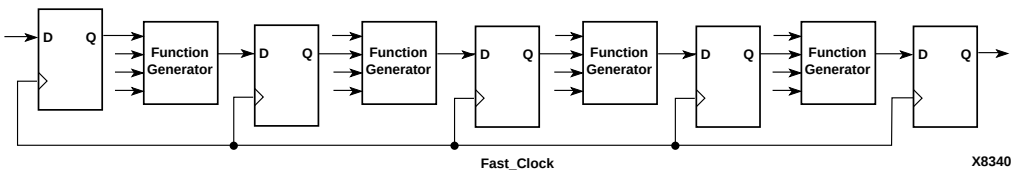

Figure 4-11 After Pipelining

Design Hierarchy

HDL designs can either be synthesized as a flat module or as many small modules. Each methodology has its advantages and disadvantages, but as higher density FPGAs are created, the advantages of hierarchical designs outweigh any disadvantages.

Advantages to building hierarchical designs are as follows.

- Easier and faster verification / simulation
- Allows several engineers to work on one design at the same time

- Speeds up design compilation
- Reduces design time by allowing design module re-use for this and future designs.
- Allows you to produce designs that are easier to understand
- Allows you to efficiently manage the design flow

Disadvantages to building hierarchical designs are as follows.

- Design mapping into the FPGA may not be as optimal across hierarchical boundaries; this can cause lesser device utilization and decreased design performance
- Design file revision control becomes more difficult
- Designs become more verbose

Most of the disadvantages listed above can be overcome with careful design consideration when choosing 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 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 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 bound-

aries, 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 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; the time required to complete each optimization run; if the design is worked on by a design team; and 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. Check with your synthesis vendor for guidelines.

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.

Note See your synthesis tool documentation for more information on optimizing logic across hierarchical boundaries and compiling hierarchical designs.

Incremental Design (ECO)

For information on Incremental Design (ECO), please refer to the following Application Notes:

- XAPP165: “Using Xilinx and Exemplar for Incremental Designing (ECO)”, application note, v1.0 (8/9/99) (http://www.xilinx.com/xapp xapp165.pdf).
- XAPP164: “Using Xilinx and Synplify for Incremental Designing(ECO)”, application note, v1.0 (8/6/99) (http://www.xilinx.com/xapp xapp164.pdf).

Chapter 5

Architecture Specific HDL Coding Styles for Spartan-II, Virtex, Virtex-E, and Virtex-II

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

- "Introduction"
- "Instantiating Components"
- "Using Boundary Scan (JTAG 1149.1)"
- "Using Global Clock Buffers"
- "Using Advanced Clock Management,"
- "Using Dedicated Global Set/Reset Resource"
- "Implementing Inputs and Outputs"
- "Encoding State Machines"
- "Implementing Operators and Generate Modules"
- "Implementing Memory"
- "Implementing Shift Register (Virtex/E/II and Spartan-II)"
- "Implementing Multiplexers"
- "Using Pipelining"
- "Design Hierarchy"
- "Modular Design and Incremental Design (ECO)"

Introduction

This chapter highlights the features and synthesis techniques in designing with Xilinx Virtex/E/II and Spartan-II FPGAs. Virtex/E

and Spartan-II devices share many architectural similarities. Virtex-II provides 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. Unless otherwise stated, the features and examples in this chapter apply to all Virtex/E/II and Spartan-II devices.

- 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 are available for instantiation. These components are marked as *primitive* in the *"Libraries Guide"*. Components marked as *macro* in the *"Libraries Guide"* should not be instantiated in HDL code.

FPGA primitives can be instantiated in VHDL and Verilog.

- VHDL example (declaring component and port map)

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

```
   -- Add the following two lines if using
      Synplify:

   -- library virtex;

-- use virtex.components.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 declaration
-- if using 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;
```

Note To use this example in Synplify, you need to add the Xilinx primitive library and remove the component declarations as noted above.

The Virtex library contains primitives of Virtex and Spartan-II architectures. Replace 'virtex' with the appropriate device family if you are targeting other Xilinx FPGA architecture

If you are designing with a Virtex-E device, use the **`virtexe`** library. If you are designing with a Virtex-II device, use the **`virtex2`** library.

- Verilog Example.

```
module flops (d1, ce, clk, q1, rst);
input d1;
input ce;
input clk;
output q1;
input rst;

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

Note To use the above example in Synplify, add the following line.

```
'include "<path_to>/<architecture>.v"
```

The <architecture>.v files are located in $SYNPLICITY/lib/xilinx. Where $SYNPLICITY identifies your Synplicity install area.

To use the above example with Virtex, Virtex-II and Spartan-II devices, replace <architecture> with **`virtex`**. To use a Virtex-E device, replace with **`virtexe`**. To use a Virtex-II device, replace <architecture> with **`virtex2`**.

Instantiating CORE Generator Modules

The CORE Generator allows you to generate complex ready-to-use functions such as FIFO, Filter, Divider, RAM, and ROM. Core Generator will generate EDIF netlist to describe the functionality and a component instantiation template for HDL instantiation. For more information on the use and functions created by the CORE Generator, see the *"CORE Generator Guide"*.

In VHDL, you can declare the component and port map as shown in the "*Instantiating FPGA Primitives*" section above. Synthesis tools will assume a black box for components that do not have a VHDL functional description.

In Verilog, an empty module must be declared to get port directionality. In addition, Synplify requires a syn_black_box directive declared on a black box as shown in the example below. FPGA

Express and LeonardoSpectrum will assume a black box for empty modules.

Example of Black Box Directive and Empty Module Declaration.

```
module r256x16s (
      addr,
      di,
      clk,
      we,
      en,
      rst,
      do); //synthesis syn_black_box
input [7:0] addr;
input [15:0] di;
input clk;
input we;
input en;
input rst;
output [15:0] do;
endmodule

module top (addrp, dip, clkp, wep, enp, rstp, dop);
input [7:0] addrp;
input [15:0] dip;
input clkp;
input wep;
input enp;
input rstp;
output [15:0] dop;
r256x16s U0(
        .addr(addrp),
        .di(dip),
        .clk(clkp),
        .we(wep),
        .en(enp),
        .rst(rstp),
        .do(dop));
endmodule
```

Using Boundary Scan (JTAG 1149.1)

Virtex/E/II and Spartan-II FPGAs 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 Virtex/E/II and Spartan-II device, basic boundary scan operations are always available. BSCAN_VIRTEX, BSCAN_VIRTEX2 and BSCAN_SPARTAN2 are instantiated only if users want to create internal boundary scan chains in a Virtex/Virtex-E /Virtex-II or Spartan-II device.

For specific information on boundary scan for an architecture, refer to the *"Libraries Guide"* and *"The Programmable Logic Data Book"*. For information on configuration and readback of Virtex/Virtex-E/Spartan-II FPGAs refer to XAPP 139 at http://www.xilinx.com/xapp/xapp139.pdf.

Instantiating the Boundary Scan Symbol in Virtex, Virtex-E, Virtex-II and Spartan-II

The BSCAN_VIRTEX, BSCAN_VIRTEX2 and BSCAN_SPARTAN2 symbol are used to create internal boundary scan chains in a Virtex/E/II and Spartan-II devices, respectively. The 4-pin JTAG interface (TDI, TDO, TCK, and TMS) are dedicated pins in these devices. To use normal JTAG for boundary scan purposes, just hook up the JTAG pins to the port and go. The pins on the BSCAN_VIRTEX, BSCAN_VIRTEX2 and BSCAN_SPARTAN2 symbols do not need to be instantiated and connected unless those special functions are needed to drive an internal scan chain.

The following example shows how to instantiate BSCAN_VIRTEX. Figure 4-2 shows how BSCAN_VIRTEX is connected in this design.

Replace BSCAN_VIRTEX with BSCAN_SPARTAN2 for a Spartan-II, and BSCAN_VIRTEX2 for a Virtex-II device. Refer to the "*Libraries Guide*" to verify component pinout information.

Note BSCAN_VIRTEX2 provides an additional output pin named "CAPTURE". This pin is will output HIGH when the boundary scan is in Capture-DR state.

Boundary Scan VHDL Example

The following is an example of how to instantiate a BSCAN_VIRTEX.

```
-- VIRTEX/VIRTEX-E Boundary Scan code

library IEEE;
use IEEE.std_logic_1164.all;

entity flops is port(
di: in std_logic;
ce : in std_logic;
clk: in std_logic;
qo: out std_logic
);
end flops;

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;
component BSCAN_VIRTEX port ( TDO1 : in std_logic;
                       TDO2: in std_logic;
                       UPDATE : out std_logic;
                       SHIFT: out std_logic;
                       RESET: out std_logic;
                       TDI: out std_logic;
                       SEL1: out std_logic;
                       DRCK1: out std_logic;
                       SEL2: out std_logic;
                       DRCK2: out std_logic);
end component;

signal q1,rst,tdo1,update, shift, reset,
tdi,sel1,drck1 : std_logic;
begin
U4: BSCAN_VIRTEX port map(TDO1 => tdo1,
      TDO2 => '0' ,
                       UPDATE => update,
                       SHIFT => shift,
```

```
                        RESET => rst,
                        TDI => tdi,
                        SEL1 => sel1,
                        DRCK1 => drck1,
                        SEL2 => open,
                        DRCK2 => open);
      U0 : FDCE port map(D => di,
                      CE=> update,
                      C => clk,
                      CLR => shift,
                      Q => tdo1);
      U1: FDCE port map(D => tdi,
                      CE=> sel1,
                      C => drck1,
                      CLR => rst,
                      Q => q1);
      U2: FDCE port map(D => q1,
                      CE=> ce,
                      C => clk,
                      CLR => rst,
                      Q => qo);
end inst;
```

Boundary Scan Verilog Example

The following is an example of how to instantiate a BSCAN_VIRTEX.

```
//VIRTEX/VIRTEX-E Boundary Scan code
//Undriven input pins (TDO1 in this example) may be
   connected to GND
module vbscan (di,clk, qo, rst);
input di;
input clk;
output qo;
input rst;
wire
q0,update,shift,reset,tdi,sel1,drck1,sel2,drck2,tdo
   1, tdo2;
BSCAN_VIRTEX bscanvirtex( //.TDO1(),
                    .TDO2(tdo2),
                    .UPDATE(update),
                    .SHIFT(shift),
                    .RESET(reset),
```

```
                        .TDI(tdi),
                        .SEL1(),
                        .DRCK1(),
                        .SEL2(sel2),
                        .DRCK2(drck2)) ;
FDCE u0(.D(di),
                        .CE(update),
                        .C (clk),
                        .CLR (reset),
                        .Q (tdo2));
FDCE u1 (.D (shift),
                        .CE(tdi),
                        .C (clk),
                        .CLR(sel2),
                          .Q (q1));
FDCE u2(.D (q1),
                          .CE(drck2),
                          .C (clk),
                          .CLR (rst),
                          .Q (qo)) ;
endmodule
```

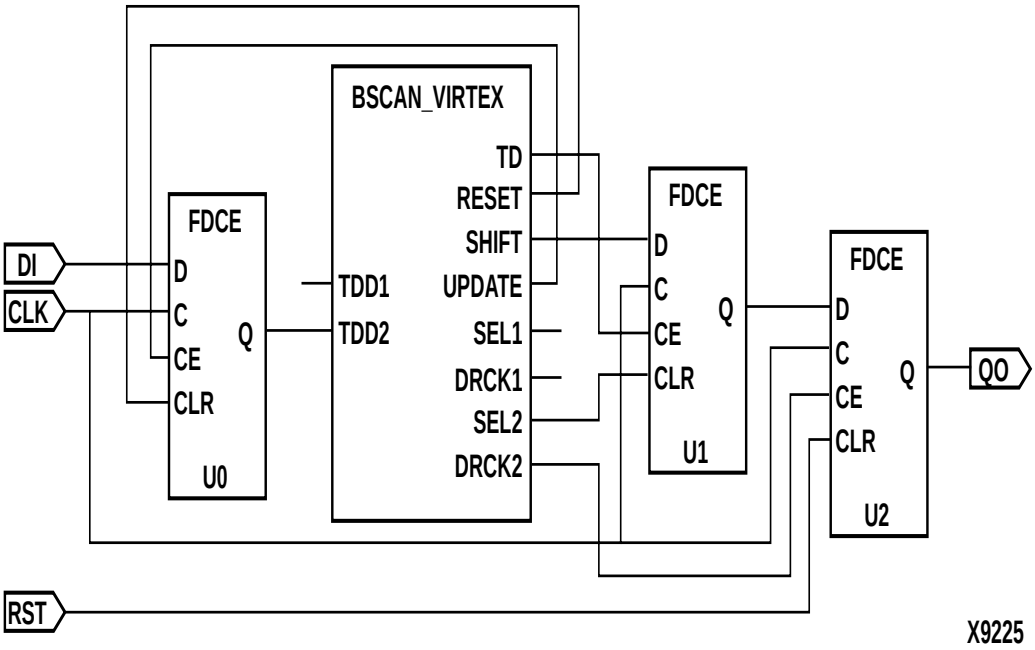

Figure 5-1 BSCAN_VIRTEX Schematic

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. The Xilinx implementation software automatically selects the clock buffer that is appropriate for your specified design architecture.

Some synthesis tools also limit global buffer insertions to match the number of buffers available on the device. Refer to your synthesis tool documentation for detailed information.

You can instantiate the clock buffers if your design requires a special architecture-specific buffer or if you want to specify how the clock buffer resources should be allocated.

Table 5-1 summarizes global buffer (BUFG) resources in Virtex, Virtex-E, Virtex-II and Spartan-II devices.

Table 5-1 Global Buffer Resources

Buffer Type	Virtex	Virtex-E	Virtex-II	Spartan-II
BUFG	4	4	N / A	4
BUFGMUX	N / A	N / A	16	N / A

Virtex / E / II, and Spartan-II devices include two tiers of global routing resources referred to as primary global and secondary local clock routing resources.

Note In Virtex-II, BUFG is available for instantiation, but will be implemented with BUFGMUX.

- The primary global routing resources are dedicated global nets with dedicated input pins that are designed to distribute high-fanout clock signals with minimal skew. Each global clock net can drive all CLB, IOB, and Block SelectRAM+ clock pins. The primary global nets may only be driven by the global buffers (BUFG), one for each global net. There are four primary global nets in Virtex / E and Spartan-II. There are sixteen in Virtex-II.
- The secondary local clock routing resources consist of backbone lines or longlines. These secondary resources are more flexible than the primary resources since they are not restricted to routing clock signal only. These backbone lines are accessed differently between Virtex / E / Spartan-II and Virtex-II devices as follows:
  - In Virtex / E and Spartan-II devices, there are 12 longlines across the top of the chip and 12 across bottom. From these lines, up to 12 unique signals per column can be distributed via the 12 longlines in the column. To use this, you must specify the USELOWSKEWLINES constraint in the UCF file. For more information on the USELOWSKEWLINES constraint syntax, refer to the *"Libraries Guide"*.
  - In Virtex-II, longlines resources are more abundant. There are many ways in which the secondary clocks or high fanout signals can be routed using a pattern of resources that result in low skew. The Xilinx Implementation tools will automatically use these resources based on various constraints in your design. Additionally, the USELOWSKEWLINES constraint can be applied to access this routing resource.

Inserting Clock Buffers

Many synthesis tools 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 will also allow you to control BUFG insertions manually if you have more clock pins than the available BUFGs resources.

FPGA Express will infer up to four clock buffers for pure clock nets. You can also instantiate clock buffers or assign them via the Express Constraints Editor.

Note Synthesis tools currently insert BUFGs for all Virtex/E/II and Spartan-II designs. If a BUFGMUX is needed in a Virtex-II design, it must be instantiated.

LeonardoSpectrum will 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 the user will not have to instantiate any BUFG components. As long as "chip" options is used to optimize the IBUFs, they will be 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 will assign a BUFG to any input signal that directly drives a clock. The maximum number of global buffers is defined as 4. Auto-insertion of the BUFG for internal clocks occur 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, you may set an attribute in the .sdc file as follows.

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

Refer to your synthesis tool documentation for a detailed syntax information.

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.

In Virtex/E/II and Spartan-II designs, 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.

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. The following VHDL and Verilog examples instantiate a BUFG for an internal multiplexed clock circuit.

Note Synplify will infer 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.       --
-- March 1998                                --
-----------------------------------------------
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;
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 multipled clock.         //
 // March 1998                                //
////////////////////////////////////////////

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

    reg   CLOCK;
    wire   CLOCK_GBUF;
    reg    DOUT;
     always @ (SEL or FAST_CLOCK or SLOW_CLOCK)
     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
```

Using Advanced Clock Management

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 detailed information on using CLKDLLs, refer to the *"Libraries Guide"* and application notes, XAPP 132 and XAPP 174 at http://www.xilinx.com/apps/xapp.htm.

In Virtex-II devices, the Digital Clock Manager (DCM) is available for advanced clock management. The DCM contains four main features listed below. For more information on the functionality of these features, refer to the "*Libraries Guide*" and the "*Virtex-II Handbook*."

- *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 frequency related to the input clock.
- *Digital Spread Spectrum (DSS)* — The DSS broadens the frequency spectrum of the output clock by speeding up and slowing down the clock within a few percent of the target frequency.

Table 5-2 CLKDLL and DCM Resources

	Virtex/ Spartan-II	Virtex-E	Virtex-II
CLKDLL	4	8	N / A
DCM	N / A	N / A	4 - 12

Using CLKDLL (Virtex/E, Spartan II)

There are four CLKDLLs in each Virtex / Spartan-II device and eight in each Virtex-E device. There are also four global clock input buffer (IBUFG) in the Virtex / E and Spartan-II devices to bring external clock in to the CLKDLL. The VHDL / Verilog example below shows a possible connection and usage of CLKDLL in your design. Cascading three CLKDLLs in Virtex / Spartan-II device is not allowed due to excessive jitter.

Synthesis tools currently do not infer CLKDLLs. The following examples shows how to instantiate CLKDLLs in your VHDL and Verilog code.

- VHDL Example.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity CLOCK_TEST is
  port(
   ACLK               : in  std_logic;
-- off chip feedback, connected to OUTBCLK on the board.
    BCLK               : in  std_logic;
--OUT CLOCK
    OUTBCLK            : out 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
  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;
  -- Glock signals
  signal ACLK_ibufg     : std_logic;
  signal BCLK_ibufg     : std_logic;
  signal ACLK_2x        : std_logic;
  signal ACLK_2x_design : std_logic;
  signal ACLK_lock      : std_logic;
begin
  ACLK_ibufg_inst : IBUFG
    port map (
      I => ACLK,
      O => ACLK_ibufg
      );
  BCLK_ibufg_inst : IBUFG
    port map (
      I => BCLK,
      O => BCLK_ibufg
```

```
      );
  ACLK_bufg : BUFG
    port map (
      I => ACLK_2x,
      O => ACLK_2x_design
      );
  ACLK_dll : CLKDLL
    port map (
CLKIN      => ACLK_ibufg,
      CLKFB      => ACLK_2x_design,
      RST        => '0',
      CLK2X      => ACLK_2x,
      CLK0       => OPEN,
      CLK90      => OPEN,
      CLK180     => OPEN,
      CLK270     => OPEN,
      CLKDV      => OPEN,
     LOCKED     => ACLK_lock
      );
BCLK_dll_out : CLKDLL
     port map (
       CLKIN     => ACLK_ibufg,
       CLKFB     => BCLK_ibufg,
       RST       => '0',
       CLK2X     => OUTBCLK,
       CLK0      => OPEN,
       CLK90     => OPEN,
       CLK180    => OPEN,
       CLK270    => OPEN,
       CLKDV     => OPEN,
      LOCKED    => BCLK_lock
       );
process (ACLK_2x_design, RESET)
begin
 if RESET = '1' then
  QOUT <= "00";
 elsif ACLK_2x_design'event and ACLK_2x_design = '1' then
  if ACLK_lock = '1' then
   QOUT <= DIN;
  end if;
 end if;
end process;
```

```
END RTL;
```

Note To use this example in Synplify, include the following library.

```
library virtex;
use virtex.components.all;
```

- Verilog Example.

```
// Verilog example
// In this example ACLK's frequency is doubled,
   used inside and outside the chip.
// BCLK and OUTBCLK are connected in the board
   outside the chip.
module clock_test(ACLK, DIN, QOUT, BCLK,
   OUTBCLK, BCLK_LOCK, RESET);
 input   ACLK, BCLK;
 input RESET;
 input [1:0] DIN;
 output [1:0] QOUT;
output OUTBCLK, BCLK_LOCK;
reg [1:0] QOUT;
IBUFG CLK_ibufg_A
      (.I (ACLK),
      .O(ACLK_ibufg)
      );
BUFG ACLK_bufg
      (.I (ACLK_2x),
       .O (ACLK_2x_design)
     );
IBUFG CLK_ibufg_B
      (.I (BCLK),     // connected to OUTBCLK
   outside
```

```
      .O(BCLK_ibufg)
      );
CLKDLL ACLK_dll_2x   // 2x clock
      (.CLKIN(ACLK_ibufg),
      .CLKFB(ACLK_2x_design),
      .RST(1'b0),
      .CLK2X(ACLK_2x),
      .CLK0(),
      .CLK90(),
     .CLK180(),
      .CLK270(),
      .CLKDV(),
      .LOCKED(ACLK_lock)
      );
CLKDLL BCLK_dll_OUT  // off-chip synchronization
      (.CLKIN(ACLK_ibufg),
.CLKFB(BCLK_ibufg), // BCLK and OUTBCLK is
   connected outside the chip.
.RST(1'b0),
.CLK2X(OUTBCLK),  //connected to BCLK outside
.CLK0(),
.CLK90(),
.CLK180(),
.CLK270(),
.CLKDV(),
.LOCKED(BCLK_LOCK)
      );
always @(posedge ACLK_2x_design or posedge
   RESET)
```

```
begin
if (RESET)
 QOUT[1:0] <= 2'b00;
else if (ACLK_lock)
 QOUT[1:0] <= DIN[1:0];
end
endmodule
```

Note To use this example in Synplify, add the following line:

```
'include "<path_to>/virtex.v"
```

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. Refer to the "DLLs in Virtex-E Devices" figure below. The basic operations of the DLLs in the Virtex-E devices remains the same as in the Virtex and Spartan-II devices, but the connections may have changed for some configurations.

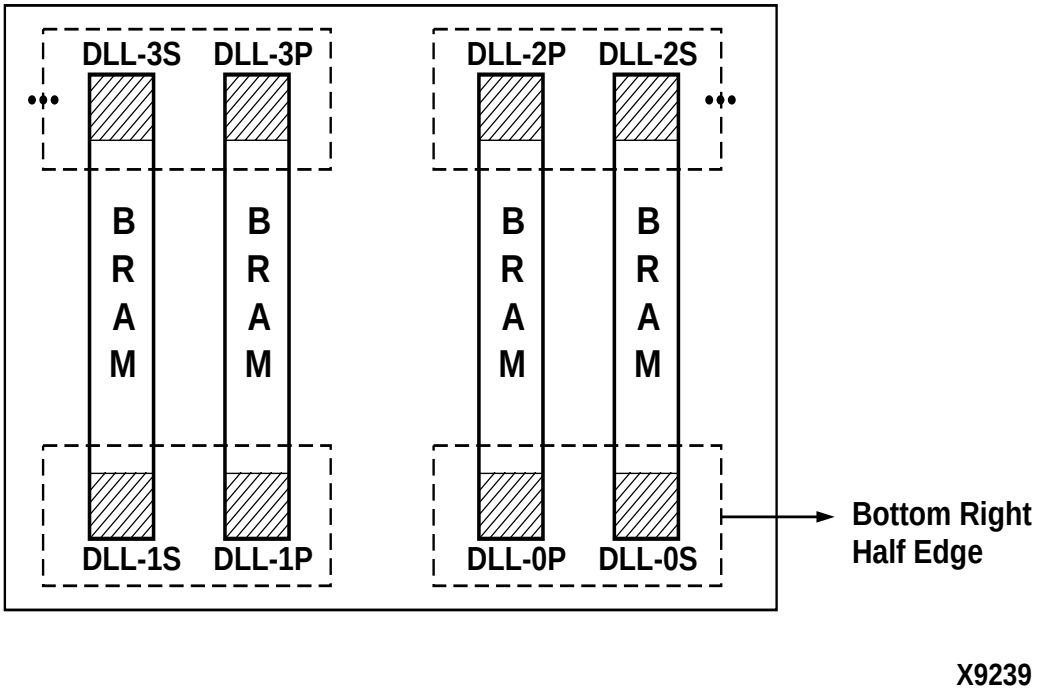

Figure 5-2 DLLs in Virtex-E Devices

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. Refer to the "DLL Generation of 4x Clock in Virtex-E Devices" figure below.

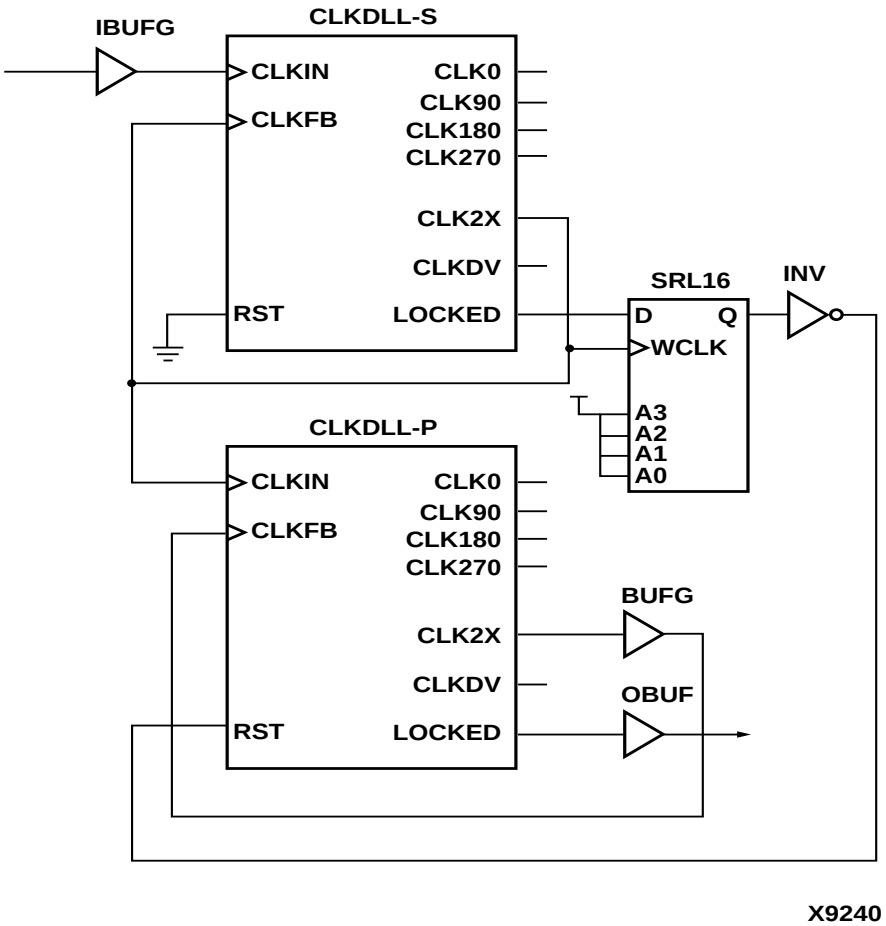

Figure 5-3 DLL Generation of 4x Clock in Virtex-E Devices

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

- VHDL Example.

```
library IEEE;
use IEEE.std_logic_1164.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
  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;
  -- Glock 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,
      CLKDV      => 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;
```

Note Synplify users need to add the following line,

```
library virtex;
use virtex.components.all;
```

- Verilog Example.

```
module clock_test(ACLK, DIN, QOUT, BCLK_LOCK,
   RESET);
 input   ACLK;
 input RESET;
 input [1:0] DIN;
 output [1:0] QOUT;
 output BCLK_LOCK;
reg [1:0] QOUT;
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 or posedge RESET)
begin
if (RESET)
 QOUT[1:0] <= 2'b00;
else if (BCLK_LOCK)
 QOUT[1:0] <= DIN[1:0];
end
endmodule
```

Note Synplify users should add the appropriate library. Please see the "Instantiating FPGA Primitives" section in this chapter.

Using BUFGDLL

BUFGDLL macro is the simplest way to provide zero propagation delay for a high-fanout on-chip clock from the external input. This macro uses the IBUFG, CLKDLL and BUFG primitive to implement the most basic DLL application. Refer to the "BUFGDLL Schematic" figure below.

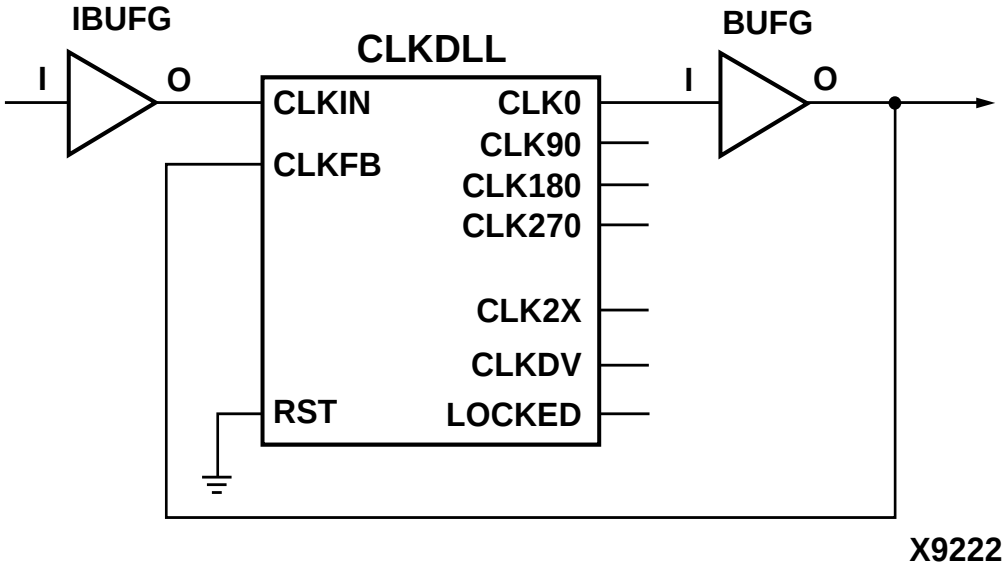

Figure 5-4 BUFGDLL Schematic

In LeonardoSpectrum, set the following attribute in the command line or TCL script.

set_attribute -port <*CLOCK_PORT*> **-name PAD -value BUFGDLL**

LeonardoSpectrum support implementation of BUFGDLL with CLKDLLHF component. To use this implementation, set the following attribute.

set_attribute -port <*CLOCK_PORT*> **-name PAD -value BUFGDLLHF**

In Synplify, set the following attribute in SDC file.

define_attribute <*port_name*> **xc_clockbuftype {BUFGDLL}**

This attribute can be applied to the clock port in HDL code as well.

CLKDLL Attributes

To specify how the signal on CLKDIV pin is frequency divided with respect to the CLK0 pin, the CLKDV_DIVIDE property can be set.

The values allowed for this property are 1.5, 2, 2.5, 3, 4, 5, 8, or 16. The default is 2.

In HDL code, CLKDV_DIVIDE property is set as an attribute to the CLKDLL instance.

The following are VHDL and Verilog coding examples of CLKDLL attributes.

- VHDL Example.

```
library IEEE;
use IEEE.std_logic_1164.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)
    );
end CLOCK_TEST;
architecture RTL of CLOCK_TEST is
  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;
  -- Glock signals
signal ACLK_ibufg             : std_logic;
  signal div_2, div_2_design    : std_logic;
  signal ACLK0, ACLK0bufg       : std_logic;
attribute CLKDV_DIVIDE: string;
attribute CLKDV_DIVIDE of ACLK_dll : label is "2";
begin
  ACLK_ibufginst : IBUFG
    port map (
      I => ACLK,
      O => ACLK_ibufg
      );
  ACLK_bufg: BUFG
    port map (
      I => ACLK0, O => ACLK0bufg);
  DIV_bufg: BUFG
    port map (
      I => div_2, O => div_2_design);
  ACLK_dll : CLKDLL
    port map (
      CLKIN      => ACLK_ibufg,
      CLKFB      => ACLK0bufg,
      RST        => '0',
      CLK2X      => OPEN,
      CLK0       => ACLK0,
      CLK90      => OPEN,
      CLK180     => OPEN,
      CLK270     => OPEN,
      CLKDV      => div_2,
      LOCKED     => OPEN
      );
process (div_2_design, RESET)
begin
if RESET = '1' then
  QOUT <= "00";
 elsif div_2_design'event and div_2_design = '1'
   then
   QOUT <= DIN;
 end if;
end process;
END RTL;
```

- Verilog Example.

```
module clock_test(ACLK, DIN, QOUT, RESET);
 input   ACLK;
 input RESET;
 input [1:0] DIN;
 output [1:0] QOUT;
reg [1:0] QOUT;
IBUFG CLK_ibufg_A
      (.I (ACLK),
      .O(ACLK_ibufg)
      );
BUFG div_CLK_bufg
      (.I (div_2),
       .O (div_2_design)
      );
BUFG clk0_bufg ( .I(clk0), .O(clk_bufg));
CLKDLL ACLK_div_2   // div by 2
      (.CLKIN(ACLK_ibufg),
      .CLKFB(clk_bufg),
      .RST(1'b0),
      .CLK2X(),
      .CLK0(clk0),
      .CLK90(),
      .CLK180(),
      .CLK270(),
      .CLKDV(div_2),
      .LOCKED()
); //exemplar attribute ACLK_div_2 CLKDV_DIVIDE 2
always @(posedge div_2_design or posedge RESET)
begin
if (RESET)
 QOUT[1:0] <= 2'b00;
else
 QOUT[1:0] <= DIN[1:0];
end
endmodule
```

Using DCM In Virtex-II

Synthesis tools currently do not automatically infer DCM. Hence, the DCM has to be instantiated in your VHDL and Verilog designs.

Please refer to the Design Consideration section (Chapter 2) of the "*Virtex-II Handbook*" for information on the various features in the DCM. This book can be found on the Xilinx website at www.xilinx.com.

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

Note Please refer to Chapter 3, *"General HDL Coding Style"* for more information on passing attributes in the HDL code to different synthesis vendors.

```
VHDL Example
-- Using a DCM for Virtex-II (VHDL)
--
-- This code uses the phased clock output CLK0 of
-- the DCM
-- The Spread Spectrum option is enabled using the
-- attribute DSS_MODE set to SPREAD_8
--
-- The following code passes the attribute for the
-- 3 following synthesis tools: Synplify,
-- FPGA Compiler II and LeonardoSpectrum.
library IEEE;
use IEEE.std_logic_1164.all;
entity clock_block is
  port (
    CLK_PAD             : in  std_logic;
   SPREAD_SPECTRUM_YES : 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;
  attribute DSS_MODE : string;
  attribute DSS_MODE of U2: label is "SPREAD_8";
  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
                  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;
begin
  U1 : IBUFG port map ( I => CLK_PAD, O => CLK_int);
  U2 : DCM port map (
    CLKFB    => CLK,
    CLKIN    => CLK_int,
    DSSEN    => SPREAD_SPECTRUM_YES,
    PSCLK    => '0',
    PSEN     => '0',
    PSINCDEC => '0',
    RST      => RST_DLL,
    CLK0     => CLK_dcm,
    LOCKED   => LOCKED);
  U3 : BUFG port map (I => CLK_dcm, O => CLK);
  CLK_out <= CLK;
```

```
end architecture STRUCT;
```

- Verilog Example

```
// Using a DCM for Virtex-II (Verilog)
//
// This code uses the phased clock output CLK0 of
// the DCM
// The Spread Spectrum option is enabled using the
// attribute DSS_MODE set to SPREAD_8
//
// The following code passes the attribute for the
// 3 following synthesis tools: Synplify,
// FPGA Compiler II and LeonardoSpectrum.
module clock_top (clk_pad, spread_spectrum_yes,
  rst_dll,
    clk_out, locked);
  input    clk_pad, spread_spectrum_yes, rst_dll;
  output   clk_out, locked;
  wire     clk, clk_int, clk_dcm;
  IBUFG u1 (.I (clk_pad), .O (clk_int));
  DCM u2 (.CLKFB    (clk),
       .CLKIN    (clk_int),
       .DSSEN    (spread_spectrum_yes),
       .PSCLK    (1'b0),
       .PSEN     (1'b0),
       .PSINCDEC (1'b0),
       .RST      (rst_dll),
       .CLK0     (clk_dcm),
       .LOCKED   (locked)) /* synthesis
  DSS_MODE="SPREAD_8" */;
   // synopsys attribute DSS_MODE "SPREAD_8"
   // exemplar attribute u2 DSS_MODE SPREAD_8
    BUFG u3(.I (clk_dcm), .O (clk));
    assign clk_out = clk;
endmodule // clock_top
```

Using Dedicated Global Set/Reset Resource

Using GSR in Virtex/E/II and Spartan-II devices must be considered carefully. Synthesis tools automatically infer GSRs for these devices; however, STARTUP_VIRTEX, STARTUP_VIRTEX2 and STARTUP_SPARTAN2 can be instantiated in your code in order to

access the GSR resource. Xilinx's recommendation for Virtex / E / II and Spartan-II designs is to write the high fan out set / reset signal explicitly in the HDL code and not use the STARTUP_VIRTEX, STARTUP_VIRTEX2, or STARTUP_SPARTAN2 blocks. There are two advantages to this method.

1. This method gives you a faster speed. The set / reset signal will be routed onto the secondary longlines in the device, which are global lines with minimal skews and high speed. Therefore, the reset / set signal on the secondary lines has much faster speed than the speed of the GSR net of the STARTUP_VIRTEX block. Since Virtex is rich in routings, placing and routing this signal on the global lines can be easily done by our software.
2. The **`trce`** program will analyze the delays of the explicitly written set / reset signal. You can read the `.twr` file (report file of the **`trce`** program) and find out exactly how fast its speed is. The **`trce`** command does not analyze the delays on the GSR net of the STARTUP_VIRTEX, STARTUP_VIRTEX2, or STARTUP_SPARTAN2. Hence, using explicit set / reset signal will improve your design accountability.

For Virtex / E / II and Spartan-II devices, the Global Set / Reset (GSR) signal is, by default, set to active high (globally resets device when logic equals 1). You can change this to active low by inverting the GSR signal before connecting it to the GSR input of the STARTUP component.

Note See the "Simulating Your Design" chapter for more information on simulating the Global Set / Reset.

Startup State

The GSR pin on the STARTUP block or the GSRIN pin on the STARTBUF block drives the GSR net and connects to each flip-flop's Preset and Clear pin. When you connect a signal from a pad to the STARTUP block's GSR pin, the GSR net is activated. Because the GSR net is built into the silicon it does not appear in the pre-routed netlist file. When the GSR signal is asserted High (the default), all flip-flops and latches are set to the state they were in at the end of configuration. When you simulate the routed design, the gate simulator translation program correctly models the GSR function.

Note See the "Simulating Your Design" chapter for more information on STARTUP and STARTBUF.

The following VHDL and Verilog example shows a STARTUP_VIRTEX instantiation using both GSR and GTS pins for FPGA Express and LeonardoSpectrum.

- VHDL example.

```
-- This example uses both GTS and GSR pins.
-- Unconnected STARTUP pins are omitted from
-- component declaration.
library IEEE;
use IEEE.std_logic_1164.all;
entity setreset is
   port (CLK: in std_logic;
     DIN1 : in STD_LOGIC;
     DIN2: in STD_LOGIC;
     RESET: in STD_LOGIC;
     GTSInput: in STD_LOGIC;
     DOUT1: out STD_LOGIC;
     DOUT2: out STD_LOGIC;
     DOUT3: out STD_LOGIC);
end setreset ;
architecture RTL of setreset is
component STARTUP_VIRTEX
   port( GSR, GTS: in std_logic);
end component;
begin
startup_inst: STARTUP port map(GSR => RESET, GTS
   => GTSInput);
reset_process: process (CLK, RESET)
```

```
    begin
      if (RESET = '1') then
        DOUT1 <= '0';
      elsif ( CLK'event and CLK ='1') then
      DOUT1  <= DIN1;
      end if;
  end process;
  gtsprocess:process (GTSInput)
  begin
  if GTSInput = '0' then
  DOUT3 <= '0';
  DOUT2 <= DIN2;
  else
  DOUT2 <= 'Z';
  DOUT3 <= 'Z';
  end if;
  end process;
  end RTL;
```

- Verilog example.

```
// This example uses both GTS and GSR pins
// Unused STARTUP pins are omitted from module
// declaration.
module setreset(CLK,DIN1, DIN2,RESET, GTSInput,
    DOUT1,DOUT2,DOUT3);
    input CLK;
    input DIN1;
    input DIN2;
    input RESET;
```

```
    input GTSInput;
    output DOUT1;
    output DOUT2;
    output DOUT3;
    reg DOUT1;
STARTUP startup_inst(.GSR(RESET), .GTS(GTSInput));
always @(posedge CLK or posedge RESET)
begin
if (RESET)
    DOUT1 = 1'b0;
else
    DOUT1 = DIN1;
end
assign DOUT3 = (GTSInput == 1'b0)? 1'b0: 1'bZ;
assign DOUT2 = (GTSInput == 1'b0)? DIN2: 1'bZ;
endmodule
module STARTUP( GSR, GTS);
input GSR;
input GTS;
endmodule
```

The following VHDL / Verilog examples show a STARTUP_VIRTEX instantiation using both GSR and GTS pins in Synplify. In the examples, STARTUP_VIRTEX_GSR and STARTUP_VIRTEX_GTS are instantiated together to get the GSR and GTS pins connected. The resulting EDIF netlist will have only one STARTUP_VIRTEX block with GTS and GSR connections. The CLK pin of the STARTUP_VIRTEX will be unconnected. If all pins (GSR, GTS, and CLK) in the STARTUP block are needed, use STARTUP_VIRTEX to port map the pins.

- VHDL example

```
library IEEE,virtex,synplify;
```

```
use synplify.attributes.all;
use virtex.components.all;
use IEEE.std_logic_1164.all;
entity setreset is
   port (CLK: in std_logic;
         DIN1 : in STD_LOGIC;
         DIN2: in STD_LOGIC;
         RESET: in STD_LOGIC;
         GTSInput: in STD_LOGIC;
         DOUT1: out STD_LOGIC;
         DOUT2: out STD_LOGIC;
         DOUT3: out STD_LOGIC);
end setreset ;
architecture RTL of setreset is
begin
u0: STARTUP_VIRTEX_GSR port map(GSR => RESET);
u1: STARTUP_VIRTEX_GTS port map(GTS =>
   GTSInput);
reset_process: process (CLK, RESET)
   begin
     if (RESET = '1') then
       DOUT1 <= '0';
     elsif ( CLK'event and CLK ='1') then
       DOUT1  <= DIN1;
     end if;
end process;
gtsprocess:process (GTSInput)
begin
```

```
if GTSInput = '0' then
DOUT3 <= '0';
DOUT2 <= DIN2;
else
DOUT2 <= 'Z';
DOUT3 <= 'Z';
end if;
end process;
end RTL;
```

- Verilog example

```
`include "path/to/virtex.v"
 module setreset(CLK,DIN1, DIN2,RESET, GTSInput,
DOUT1,DOUT2,DOUT3);
     input CLK;
     input DIN1;
     input DIN2;
     input RESET;
     input GTSInput;
     output DOUT1;
     output DOUT2;
     output DOUT3;
     reg DOUT1;
STARTUP_VIRTEX_GSR startup_inst(.GSR(RESET));
STARTUP_VIRTEX_GTS startup_2(.GTS(GTSInput));
   always @(posedge CLK or posedge RESET)
   begin
   if (RESET)
     DOUT1 = 1'b0;
```

```
    else
      DOUT1 = DIN1;
end
assign DOUT3 = (GTSInput == 1'b0)? 1'b0: 1'bZ;
assign DOUT2 = (GTSInput == 1'b0)? DIN2: 1'bZ;
endmodule
```

Preset vs. Clear

The Virtex/E/II and Spartan-II family flip-flops are configured as either preset (asynchronous set) or clear (asynchronous reset) during startup. Automatic assertion of the GSR net presets or clears each flip-flop after the FPGA is configured. You can assert the GSR pin at any time to produce this global effect. You can also preset or clear individual flip-flops with the flip-flop's dedicated Preset or Clear pin. When a Preset or Clear pin on a flip-flop is connected to an active signal, the state of that signal controls the startup state of the flip-flop. For example, if you connect an active signal to the Preset pin, the flip-flop starts up in the preset state. If you do not connect the Clear or Preset pin, the default startup state is a clear state. To change the default to preset, assign an INIT=1 to the Virtex/E/II or Spartan-II flip-flop.

I/O flip-flops and latches in Virtex, Virtex-E, Virtex-II, and Spartan-II have SR pin which can be configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear. The SR pin can be driven by any user logic, but INIT will also work for these flip-flops.

Below are examples of setting register INIT using ROCBUF. In the HDL code, the instantiated ROCBUF connects the set/reset signal. The Xilinx tools will automatically remove the ROCBUF during implementation leaving the set/reset signal active only during power-up.

- VHDL Example.

```
library IEEE;
 use IEEE.std_logic_1164.all;
   entity d_register is
      port (CLK : in std_logic;
```

```
        RESET : in std_logic;
        D0: in std_logic;
        D1: in std_logic;
        Q0 : out std_logic;
        Q1 : out std_logic);
  end d_register;
  architecture XILINX of d_register is
signal RESET_int : std_logic;
component ROCBUF is  port (I : in STD_LOGIC;
                                    O : out STD_LOGIC);
end component;
begin
U1: ROCBUF port map (I => RESET, O => RESET_int);
process (CLK, RESET_int)
  begin
    if RESET_int = '1' then
          Q0 <= '0';
          Q1 <= '1';
    elsif  rising_edge(CLK) then
                  Q0 <= D0;
                  Q1 <= D1;
    end if;
end process;
```

```
  end XILINX;
```

- Verilog example

```
  /*
   Note: In Synplify, set blackbox attribute for
     ROCBUF as follows:
  module ROCBUF (I, O); //synthesis syn_black_box
  input I;
  output O;
  endmodule
  */
  module ROCBUF (I, O);
  input I;
  output O;
```

```
endmodule

module rocbuf_example (reset, clk, d0, d1, q0,
   q1);

input reset;

input clk ;

input d0;

input d1;

output q0 ;

output q1 ;

reg q0, q1;

wire reset_int;

ROCBUF u1 (.I(reset), .O(reset_int));

always @ (posedge clk or posedge reset_int)
   begin

if  (reset_int) begin

q0 = 1'b0;

q1 = 1'b1;

end

else

begin

q0 = d0;

q1 = d1;

end

end

endmodule
```

Implementing Inputs and Outputs

FPGAs have limited logic resources in the user-configurable inputs / output blocks (IOB). You can move logic that is normally implemented with CLBs to IOBs. By moving logic from CLBs to IOBs, addi-

tional logic can be implemented in the available CLBs. Using IOBs also improves design performance by increasing the number of available routing resources.

The Virtex/E/II, and Spartan-II IOBs feature SelectI/O inputs and outputs that support a wide variety of I/O signaling standards. In addition, each IOB provides three storage elements. The following sections discuss IOB features in more detail.

I/O Standards

The following table summarizes the I/O standards supported in Virtex/E/II and Spartan-II devices. A complete table is available in the *"Libraries Guide"*.

Table 5-3 I/O Standard in Virtex/E/II and Spartan-II Devices

I/O Standard	Virtex/ Spartan-II	Virtex-E	Virtex-II
LVTTL (default)	√	√	√
AGP	√	√	√
CTT	√	√	
GTL	√	√	√
GTLP	√	√	√
HSTL Class I	√	√	√
HSTL Class II			√
HSTL Class III	√	√	√
HSTL Class IV	√	√	√
LVCMOS2	√		
LVCMOS15			√
LVCMOS18		√	√
LVCMOS25, 33			√

Table 5-3 I/O Standard in Virtex/E/II and Spartan-II Devices

I/O Standard	Virtex/ Spartan-II	Virtex-E	Virtex-II
LVCZ_15, 18, 25, 33			√
LVCZ_DV2 _15, 18, 25, 33			√
LVDS		√	√
LVPECL		√	√
PCI33_5			
PCI33_3, PCI66_3	√	√	√
PCIX			√
SSTL2 Class I and Class II	√	√	√
SSTL3 Class I and Class II	√	√	√

For Virtex, Virtex-E, and Spartan-II devices, Xilinx provides a set of IBUF, IBUFG, IOBUF, and OBUF with its SelectI/O variants. For example, IBUF_GTL, IBUFG_PCI66_3, IOBUF_HSTL_IV, OBUF_LVCMOS2. Alternatively, an IOSTANDARD attribute can be set to a specific I/O standard and attached to an IBUF, IBUFG, IOBUF, and OBUF. The IOSTANDARD attribute can be set in the user constraint file (UCF) or in the netlist by the synthesis tool.

The Virtex-II library includes certain SelectI/O components for compatibility with other architectures. However, the recommended method for using SelectI/O components for Virtex-II is to attach an IOSTANDARD attribute to IBUF/IBUFG/IOBUF/OBUF. For example, attach IOSTANDARD=GTLP to an IBUF instead of using the IBUF_GTLP.

The default for the IOSTANDARD attribute is LVTTL. For all Virtex/E/II and Spartan-II devices, you must specify IBUF, IBUFG, IOBUF or OBUF on the IOSTANDARD attribute if LVTTL is not desired.

For more information on I/O standards and components, please refer to the *"Libraries Guide"*.

Inputs

Virtex/E/II, and Spartan-II inputs can be configured the I/O standards listed above.

In FPGA Express, these special IOB components exist in the synthesis library and can be instantiated in your HDL code or selected from the GUI. A complete list of components understood by FPGA Express can be found in the lib\virtex directory under the FPGA Express tree (%XILINX%\synth for Foundation users). FPGA Express will understand these components and will not attempt to place any I/O logic on these ports. Users will be alerted by this warning:

```
Warning: Existing pad cell '/ver1-Optimized/U1' is
connected to the port 'clk' - no pads cells inserted
at this port. (FPGA-PADMAP-1)
```

Note FPGA Express/FPGA Compiler II 3.4 and older does not recognize LVDS and LVPECL I/O standard. Use the IOSTANDARD attribute on I/O buffer components when instantiating these I/O standards.

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` *<IO_standard> <portname>*

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

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

In Synplify, users can set xc_padtype attribute in SCOPE (Synplify's constraint editor) or in HDL code as shown below:

- VHDL Example.

```
library ieee, synplify;
  use ieee.std_logic_1164.all;
  use synplify.attributes.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 xc_padtype of a : signal is
   "IBUF_SSTL3_I";
  attribute xc_padtype of bidir : signal is
   "IOBUF_HSTL_III";
  attribute xc_padtype of q : signal is "OBUF_S_8";
  end entity;
```

- Verilog Example

```
module test_padtype (a, b, clk, rst, en, bidir, q);
input [3:0] a /* synthesis xc_padtype = "IBUF_AGP"
   */;
input [3:0] b;
input clk, rst, en;
inout [3:0] bidir /* synthesis xc_padtype =
   "IOBUF_CTT" */;
output [3:0] q /* synthesis xc_padtype =
   "OBUF_F_12" */;
```

Note Refer to IBUF_selectIO in the *Libraries Guide* for a list of available IBUF_selectIO values.

Outputs

Virtex/E/II and Spartan-II outputs can also be configured to any of I/O standards listed in the I/O standards section. An OBUF that uses the LVTTL, LVCMOS15, LVCMOS18, LVCMOS25, or LVCMOS33 signaling standards has selectable drive and slew rates using the DRIVE and SLOW or FAST constraints. The defaults are DRIVE=12 mA and SLOW slew.

In addition, you can control the slew rate and drive power for LVTTL outputs using **OBUF**_*<slew>*_*<drive_power>*.

Refer to OBUF_selectIO in the *"Libraries Guide"* for a list of available OBUF_selectIO values. You can use the examples in the Inputs section to configure OBUF to an I/O standard.

Using IOB Register and Latch

Virtex/E, and Spartan-II IOBs 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 as a synchronous set, a synchronous reset, an asynchronous preset, or an asynchronous clear. FDCP (asynchronous reset and set) and FDRS (synchronous reset and set) register configurations are not available in IOBs.

Virtex-II 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 tristated enable register. A separate clock (ICLK1 and ICLK2) drive the input register. The set and reset signals (SR and REV) are shared by the three registers.

Virtex/E/II, and Spartan-II devices no longer have primitives that correspond to the synchronous elements in the IOB's. There are a few ways to infer usage of these FF's if the rules for pulling them into the IOB are followed. The following rules apply.

- All FF's that are to be pulled into the IOB must have a fanout of 1, this applies to input, output and tristated enable registers. For example, if there is a 32 bit bidirectional bus then the tristated enable signal must be replicated in the original design so that it will have a fanout of 1.
- In Virtex/E and Spartan-II devices, all FF's must share the same clock and reset signal, they can have independent clock enables.
- In Virtex-II devices, output and tristated enable registers must share the same clock. All FF's must share the same set and reset signals.

One way you can pull FF's into the IOB is to use the IOB=TRUE setting. Another way is to pull FF's into the IOB using the **`map -pr`** command which will be discussed in a later section. Some synthesis tools will apply IOB=TRUE attribute and allow you to merge a FF to

an IOB by setting an attribute. Refer to your synthesis tool documentation for the correct attribute and settings.

In FPGA Express, you can set the attribute for each port into which a flip flop should be merged. This will not work for tristated enable flip flops.

In LeonardoSpectrum you can select MAP IOB Registers from the Technology tab in the GUI or set the following attribute in your TCL script:

```
set virtex_map_iob_registers TRUE
```

In Synplify, attach the **syn_useioff** attribute to the module or architecture of 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 / 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 Output Enable IOB Register

The following VHDL and Verilog example illustrate how to infer output enable register. See the above section for an attribute to turn I/O register inference in synthesis tools.

Note If using FPGA Express to synthesis the VHDL example below, use map -pr option to force output enable register to IOB. Verilog example works fine.

- VHDL Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- the following two lines is used for
-- LeonardoSpectrum synthesis tool only.
library exemplar;
use exemplar.exemplar_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;
       tri_state_b :in std_logic;
  data_out : out std_logic_vector(7 downto 0));
end tri_state;
architecture behavioral of tri_state is
signal data_in : std_logic_vector(7 downto 0);
signal data_in_r :std_logic_vector(7 downto 0);
signal tri_state_cntrl: std_logic_vector(7 downto
   0);
signal temp_tri_state : std_logic_vector(7 downto
   0);
-- Attribute below is for LeonardoSpectrum
-- synthesis tool only
attribute preserve_signal : boolean;
attribute preserve_signal of temp_tri_state :
signal is TRUE;
begin
    G1:  for I in 0 to 7 generate
      temp_tri_state(I) <= tri_state_a AND
    tri_state_b;   -- create duplicate input signal
end generate;
    process (tri_state_cntrl, data_in_r) begin
      G2:  for J in 0 to 7 loop
      if (tri_state_cntrl(J) = '0') then  -- tri-
   state data_out
          data_out(J) <= data_in_r(J);
        else data_out(J) <= 'Z';
      end if;
    end loop;
  end process;
process(clk) begin
    if clk'event and clk='1' then
```

```
      data_in <= data_in_p;-- register for input
      data_in_r <= data_in;-- register for output
      for I in 0 to 7 loop
      tri_state_cntrl(I) <= temp_tri_state(I);--
register tri-state
      end loop;
    end if;
  end process;
end behavioral;
```

- Verilog Example

```
////////////////////////////////////////////////
// Inferring output enable register //

   // October 2000 //

////////////////////////////////////////////////
module tri_state (data_in_p, clk, tri_state_a,
tri_state_b, data_out);
   input[7:0] data_in_p;
   input clk;
   input tri_state_a;
   input tri_state_b;
   output[7:0] data_out;
   reg[7:0] data_out;
   reg[7:0] data_in;
   reg[7:0] data_in_r;
   reg[7:0] tri_state_cntrl;
   wire[7:0] temp_tri_state;
   assign temp_tri_state[0] = tri_state_a &
tri_state_b ; // create duplicate input signal
   assign temp_tri_state[1] = tri_state_a &
tri_state_b ;
   assign temp_tri_state[2] = tri_state_a &
tri_state_b ;
   assign temp_tri_state[3] = tri_state_a &
tri_state_b ;
   assign temp_tri_state[4] = tri_state_a &
tri_state_b ;
   assign temp_tri_state[5] = tri_state_a &
tri_state_b ;
   assign temp_tri_state[6] = tri_state_a &
tri_state_b ;
```

```
   assign temp_tri_state[7] = tri_state_a &
tri_state_b ;
// exemplar attribute temp_tri_state
preserve_signal TRUE
   always @(tri_state_cntrl or data_in_r)
   begin
      begin : xhdl_1
         integer J;
         for(J = 0; J <= 7; J = J + 1)
         begin : G2
            if (!(tri_state_cntrl[J]))
            begin
               data_out[J] <= data_in_r[J] ;
         end
            else// tri-state data_out
            begin
               data_out[J] <= 1'bz ;
            end
         end
      end
   end
   always @(posedge clk)
   begin
         data_in <= data_in_p ; // register for
input
         data_in_r <= data_in ; // register for
output
         tri_state_cntrl[0] <= temp_tri_state[0] ;
        tri_state_cntrl[1] <= temp_tri_state[1] ;
         tri_state_cntrl[2] <= temp_tri_state[2] ;
         tri_state_cntrl[3] <= temp_tri_state[3] ;
         tri_state_cntrl[4] <= temp_tri_state[4] ;
         tri_state_cntrl[5] <= temp_tri_state[5] ;
         tri_state_cntrl[6] <= temp_tri_state[6] ;
         tri_state_cntrl[7] <= temp_tri_state[7] ;
      end
endmodule
```

Using -pr Option with MAP

Use the –pr (pack registers) option when running MAP. The –pr {i | o | b} (input | output | both) option specifies to the MAP program to

move registers into IOBs when possible. For example: **`map -pr b <design_name.ngd>`**

Virtex-E IOBs

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

Additional I/O Standards

Virtex-E devices has 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 NCF file with complete pin loc information must be created to ensure that the I/O banking rules are not violated. If a UCF or NCF file is not used, PAR will issue errors.

The input buffer of these two I/O standards may be placed in 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 will be reserved and no other IOB will be allowed to be placed 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 will lead to DRC errors in the software.

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

- VHDL Example.

```
library IEEE;
```

```
use IEEE.std_logic_1164.all;
--Synplify users add appropriate virtex library.
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
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.

```
module LVDSIOinst (CLK, DATA, Tin,
   IODATA_p, IODATA_n, Q_p, Q_n) ;
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 tagetting 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
```

FPGA Express does not recognize LVDS and LVDS I/O buffers as a valid primitive. Special techniques must be used when synthesizing these primitives in your design. Otherwise, FPGA Express will insert I/O buffers, causing errors when the design run through the Xilinx implementation tools.

The following examples use the IOSTANDARD attribute 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 attribute.

- VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.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

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;
attribute IOSTANDARD : string;
attribute LOC : string;
attribute IOSTANDARD of u1 : label is "LVDS";
attribute IOSTANDARD of u2 : label is "LVDS";
attribute IOSTANDARD of u3 : label is "LVDS";
-----------------------------------------------
-- 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 port map (d,d_lvds);
u2: OBUF port map (q_lvds,q);
u3: OBUF 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.

```
module flip_flop (d, clk, q, q_n);
/*********************************/
// 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.
/*********************************/
input d;//synopsys attribute LOC "A5"
input clk;
output q;//synopsys attribute LOC "D8"
output q_n;//synopsys attribute LOC "C8"
wire d,clk,d_lvds,q;
reg q_lvds;
IBUF u1 (.I(d), .O(d_lvds));
//synopsys attribute IOSTANDARD "LVDS"
OBUF u2 (.I(q_lvds), .O(q));
//synopsys attribute IOSTANDARD "LVDS"
OBUF u3 (.I(q_lvds_n), .O(q_n));
//synopsys attribute IOSTANDARD "LVDS"
always @(posedge clk) q_lvds=d_lvds;
assign q_lvds_n=~q_lvds;
```

```
endmodule
```

Reference Xilinx Answer Database in http://support.xilinx.com for more information.

In LeonardoSpectrum and Synplify, you can instantiate the selectI/O components or use the attribute discussed in *"Inputs"* section, but make sure that the output and its inversion are declared and configured properly.

Virtex-II IOB

Virtex-II offers more Select I/O configuration than Virtex/E and Spartan-II as shown in Table 5-3. IOSTANDARD and synthesis tools' specific attribute can be use to configure the Select I/O.

Additionally, Virtex-II provides 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 you can instantiate the 'DCI' suffixed I/Os from the library such as HSTL_IV_DCI.

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

Differential Signaling in Virtex-II

Differential signaling in Virtex-II can be configured using IBUFDS, OBUFDS, OBUFTDS. The IBUFDS is two-inputs one-output buffer. The OBUFDS is one-input two-outputs buffer. Reference the "*Libraries Guide*" for the component diagram and description.

LVDS_25, LVDS_33, LVDSEXT_33, and LVPECL_33 are valid IOSTANDARD value to attach to differential signaling buffers. If no IOSTANDARD is attached, the default is LVDS_33.

The following is the VHDL and Verilog example instantiating 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 FPGAs   --
-- October  2000                          --
-------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
--use exemplar.exemplar_1164.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
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 FPGAs  --
// October 2000                          --
//----------------------------------------
module LVDS_33_IO (CLK_p, CLK_n, DATA_p, DATA_n,
DATAIN2_p, DATAIN2_n, Tin_p, Tin_n, ODATA_p,
ODATA_n, Q_p, Q_n) ;
input    CLK_p, CLK_n, DATA_p, DATA_n,
   DATAIN2_p, DATAIN2_n, Tin_p, Tin_n;
output    ODATA_p, ODATA_n;
output   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 are acceptable for PAL and gate array architectures. However, because FPGAs have many

flip-flops and narrow 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. 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). Refer to your synthesis tool documentation for information on attribute-driven state machine synthesis.

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.

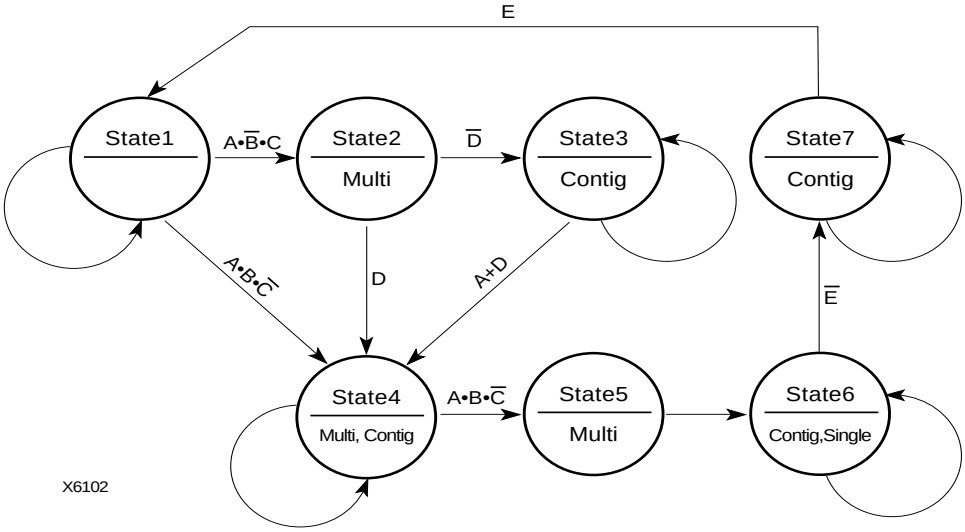

Figure 5-5 State Machine Bubble Diagram

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    --
-- May 1997                                     --
-------------------------------------------------
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

```
////////////////////////////////////////////////
// BINARY.V Version 1.0                       //
// Example of a binary encoded state machine  //
// May 1997                                   //
////////////////////////////////////////////////
module binary (CLOCK, RESET, A, B, C, D, E, SINGLE, MULTI, CONTIG);

input    CLOCK, RESET;
input    A, B, C, D, E;
output   SINGLE, MULTI, CONTIG;

reg      SINGLE, MULTI, CONTIG;
// Declare the symbolic names for states
parameter [2:0]
    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 or posedge RESET)
    begin
        if (RESET == 1'b1)
            CS = S1;
        else
            CS = NS;
    end
    always @ (CS or A or B or C or D or D or E)
    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.

Note Refer to the previous VHDL and Verilog Binary Encoded State Machine examples for the complete Case statement portion of the code.

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//
// May 1997                                     //
////////////////////////////////////////////////

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

input  CLOCK, RESET;
input  A, B, C, D, E;
output SINGLE, MULTI, CONTIG;

reg    SINGLE, MULTI, CONTIG;

// Declare the symbolic names for states
parameter [2:0]
    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 or posedge RESET)
begin
    if (RESET == 1'b1)
        CS = S1;
    else
        CS = NS;
end

always @ (CS or A or B or C or D or D or E)
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. If you are using FPGA

Express, use enumerated type, and avoid using the “when others” construct in the VHDL case statement. This construct can result in a very large state machine.

Note Refer to the previous VHDL and Verilog Binary Encoded State Machine examples for the complete Case statement portion of the code.

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 FPGAs  //
// May 1997                                    //
////////////////////////////////////////////////

module one_hot (CLOCK, RESET, A, B, C, D, E,
              SINGLE, MULTI, CONTIG);

input   CLOCK, RESET;
input   A, B, C, D, E;
output  SINGLE, MULTI, CONTIG;

reg SINGLE, MULTI, CONTIG;

// Declare the symbolic names for states
parameter [6:0]
    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
reg [2:0] CS;
reg [2:0] NS;

// state_vector CS

 always @ (posedge CLOCK or posedge RESET)
 begin
      if (RESET == 1'b1)
          CS = S1;
      else
          CS = NS;
end

always @ (CS or A or B or C or D or D or E)
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 will prompt synthesis tools 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, grey, sequential, etc.

In FPGA Express, FSM encoding is set to “One-Hot” by default. To change this setting, select Synthesis-> Options -> Project Tab. Available options are: One-Hot, Binary, and Zero One-Hot.

In LeonardoSpectrum, FSM encoding is set to “Auto” by default which differs depending on Bit Width of your state machine. To change this setting to a specific value, select Input tab. Available options are: Binary, One-Hot, Random, Gray, and Auto.

In Synplify, the Symbolic FSM Complier option can be accessed from the main GUI. 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 bases with the syn_encoding directive / attribute. Available options are: One-Hot, Gray, Sequential, and Safe.

Summary of Encoding Styles

In the three 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 three 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.

Note Refer to your synthesis tool documentation for instructions on how to extract the state machine and change the encoding style.

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.

- VHDL Example

```
SYNC_PROC: process (CLOCK, RESET)
begin
    if (RESET='1') then
       CS <= s1;
```

- Verilog Example

```
always @ (posedge CLOCK or posedge RESET)
begin
   if (RESET == 1'b 1)
       CS = S1;
```

Alternatively, you can assign an INIT=S attribute to the initial state register to specify the initial state. Refer to your synthesis tool documentation for information on assigning this attribute.

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 Generate Modules

Xilinx FPGAs feature carry logic elements that can be used for optimal implementation of operators and generate modules.

Synthesis tools infer the carry logic automatically when a specific coding style or operator is used.

Adder and Subtracter

Synthesis tools will infer carry logic in Virtex/E/II and Spartan-II devices when an adder and Subtracter is described (+ or - operator).

Multiplier

Synthesis tools will utilize the carry logic by inferring XORCY, MUXCY, and MULT_AND when a multiplier is described.

LeonardoSpectrum features a pipeline multiplier which 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. The following example shows this construct.

- VHDL Example

```
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;
```

- Verilog Example.

```
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
```

Virtex-II devices feature a large amount of 18 X 18-bit twos-complement embedded multipliers. These embedded multipliers are not yet inferred by the synthesis tools. Please refer to the release notes for your synthesis tool to verify support for these new features. At this time you must instantiate the 18 X 18-bit embedded multiplier. Please Refer to the "*Virtex Handbook*" for the HDL instantiation template.

Counters

When describing a counter in HDL, the arithmetic operator '+' will infer the carry chain. The synthesis tools will then infer the MUXCY element for the counter.

```
count <= count + 1; -- This will infer MUXCY
```

This implementation will provide a very effective solution especially for all purpose counters.

Below is an example of a loadable binary counter:

- VHDL Example

```
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, rst)
 begin
   if rst = '1' then
     count <= (others => '0');
   elsif rising_edge(clk) then
       if ld = '1' then
           count <= d;
```

```
          elsif ce = '1' then
              count <= count + '1';
          end if;
      end if;
    end process;
    q <= count;
   end behave;
```

- Verilog Example

```
module counter(d, ld, ce, clk, rst, q);
  input [7:0]  d;
  input ld, ce, clk, rst;
  output [7:0] q;
  reg [7:0]    count;
  always @(posedge clk or posedge rst)
    begin
        if (rst)
          count <= 0;
        else if (ld)
          count <= d;
        else if (ce)
          count <= count + 1;
    end
  assign q = count;
endmodule
```

For application that require faster counters, LFSR can implement high performance and area efficient counters. LFSR will require very minimum logic (only a XOR or XNOR feedback).

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

- VHDL Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity johnson is
    generic (size : integer := 3);
    port (clk   : in std_logic;
        reset : in std_logic;
        qout  : out std_logic_vector(size-1 downto 0));
end johnson;
architecture RTL of johnson is
    signal q : std_logic_vector(size-1 downto 0);
begin  -- RTL
    process(clk, reset)
    begin
      if reset = '1' then
          q <= (others => '0');
      elsif clk'event and clk='1' then
          for i in 1 to size - 1 loop
              q(i) <= q(i-1);
          end loop;  -- i
          q(0) <= not q(size-1);
      end if;
    end process;
    qout <= q;
end RTL;
```

- Verilog Example

```
module johnson (clk, reset, q);
parameter size = 4;
  input     clk, reset;
  output [size-1:0] q;
  reg [size-1:0]    q;
  integer           i;
  always @(posedge clk or posedge reset)
    if (reset)
      q <= 0;
    else
```

```
      begin
        for (i=1;i<size;i=i+1)
            q[i] <= q[i-1];
            q[0] <= ~q[size-1];
      end
endmodule // johnson
```

Comparator

Magnitude comparator '>' or '<' will infer carry chain logic and result in fast implementations in Xilinx devices. Equality comparator '==' will be implemented using LUTs

- VHDL example

```
-- Unsigned 8-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.
module compar(A, B, cmp);
input [7:0] A;
input [7:0] B;
output cmp;
```

```
assign cmp = A >= B ? 1'b1 : 1'b0;
endmodule
```

Encoder and Decoders

Synthesis tools might infer MUXF5 and MUXF6 for encoder and decoder in Virtex/E/II and Spartan-II devices. Virtex-II devices feature additional components, MUXF7 and MUXF8 to use with the encoder and decoder. These components are not yet inferred by the synthesis tools, but you can instantiate them. Reference the "*Virtex Handbook" Design Consideration section* for instantiation examples.

LeonardoSpectrum will infer MUXCY when an if-then-else priority encoder is described in the code. This will result in a faster encoder.

LeonardoSpectrum Priority Encoding HDL Example

- VHDL Example.

```
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 downto 1);
        cond2 : in std_logic_vector(size downto 1);
        data  : in std_logic_vector(size downto 1);
        q     : out std_logic);
end prior;

architecture RTL of prior is
  signal     data_ff,  cond1_ff,  cond2_ff:  std_logic_vector(size
downto 1);
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.

```
module prior(clk, cond1, cond2, data, q);
      parameter size = 8;
      input clk;
      input [1:size] data, cond1, cond2 ;
      output q;
      reg [1:size]   data_ff, cond1_ff, cond2_ff;
      reg q;
      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 // prior
```

Implementing Memory

Virtex/E and Spartan-II FPGAs provide distributed on-chip RAM or ROM memory capabilities. CLB function generators can be configured as ROM (ROM16X1, ROM32X1); edge-triggered, single-port (RAM16X1S, RAM32X1S) RAM; or dual-port (RAM16x1D) RAM. Level sensitive RAMs are not available for the Virtex/E and Spartan-II families.

Virtex-II CLB function generators are much larger and can be configured as larger ROM and edge-triggered, single port and dual port RAM. Available ROM primitive components in Virtex-II are ROM16X1 and ROM32X1. Available single port RAM primitives components in Virtex-II are RAM16X1S, RAM16X2S, RAM16X4S, RAM16X8S, RAM32X1S, RAM32X2S, RAM32X4S, RAM32X8S, RAM64X1S, RAM64X2S, and RAM128X1S. Available dual port RAM primitive components in Virtex-II are RAM16X1D, RAM32X1D, and RAM64X1D.

In addition to distributed RAM and ROM capabilities, Virtex/E/II, and Spartan-II FPGAs provide edge-triggered Block SelectRAM+ in large blocks. Virtex/E and Spartan-II devices provide 4096(4k) bits: RAMB4_Sn and RAMB4_Sm_Sn. Virtex-II devices provide larger

Block SelectRAM+ in 16384 (16k) bits size: RAMB16_Sn and RAMB16_Sm_Sn. Where Sm and Sn are configurable port widths. See the "*Libraries Guide*" for more information on these components.

The edge-triggered capability simplifies system timing and provides better performance for RAM-based designs. This RAM can be used for status registers, index registers, counter storage, constant coefficient multipliers, distributed shift registers, LIFO stacks, latching, or any data storage operation. The dual-port RAM simplifies FIFO designs.

Implementing Block SelectRAM+

Virtex/E/II, and Spartan-II FPGAs 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.

Block SelectRAM+ memory blocks are organized in columns. All Virtex and Spartan-II devices contain two such columns; one along each vertical edge. In Virtex-E, the Block SelectRAM+ column is inserted every 12 CLB columns. In Virtex-EM (Virtex-E with extended memory), the Block SelectRAM+ column is inserted every 4 CLB columns. In Virtex-II, there are at least four Block SelectRAM+ columns and a column is inserted every 12 CLB columns in larger devices.

Instantiating Block SelectRAM+

The following coding examples provide VHDL and Verilog coding styles for FPGA Express, LeonardoSpectrum, and Synplify.

Instantiating Block SelectRAM+ VHDL Example

- FPGA Express and LeonardoSpectrum

  With FPGA Express and LeonardoSpectrum 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 VHDL Code Example below shows how to pass INIT in the VHDL file.

- LeonardoSpectrum

With LeonardoSpectrum you can pass an INIT string in an LeonardoSpectrum command script. The following code sample illustrates this method.

```
set_attribute -instance "inst_ramb4_s4" -name
INIT_00 -type string -value
"1F1E1D1C1B1A19181716151413121110 0F0E0D0C0B0A09
08070605040302 0100"
```

- VHDL Code Example.

```
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
"1F1E1D1C1B1A19181716151413121110 0F0E0D0C0B0A09
08070605040302 0100";
attribute INIT_00 of INST_RAMB4_S8: label is
"1F1E1D1C1B1A19181716151413121110 0F0E0D0C0B0A09
08070605040302 0100";
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;
```

- Synplify

  With Synplify you can instantiate the RAMB* cells by using the Xilinx family library supplied with Synplify. The following code example will illustrate instantiating a RAMB* cell.

```
library IEEE;
use IEEE.std_logic_1164.all;
library virtex;
use virtex.components.all;
library synplify;
use synplify.attributes.all;

entity RAMB4_S8_synp is
    generic (INIT_00, INIT_01 : string :=
```

```
"0000000000000000000000000000000000000000000000000
00000000000000");
    port (WE, EN, RST, CLK : in std_logic;
      ADDR : in std_logic_vector(8 downto 0);
      DI : in std_logic_vector(7 downto 0);
      DO : out std_logic_vector(7 downto 0));
end RAMB4_S8_synp;
architecture XILINX of RAMB4_S8_synp is

component RAMB4_S8
    port (WE, EN, RST, CLK: in STD_LOGIC;
      ADDR: in STD_LOGIC_VECTOR(8 downto 0);
      DI: in STD_LOGIC_VECTOR(7 downto 0);
      DO: out STD_LOGIC_VECTOR(7 downto 0));
end component;
attribute xc_props of u1 : label is "INIT_00=" &
INIT_00 & ",INIT_01=" & INIT_01;
  begin
  U1 : RAMB4_S8
    port map (WE => WE, EN => EN, RST => RST, CLK =>
    CLK, ADDR => ADDR, DI => DI, DO => DO);
end XILINX;

library IEEE;
use IEEE.std_logic_1164.all;

entity block_ram_ex is
port (CLK, WE : in std_logic;
    ADDR : in std_logic_vector(8 downto 0);
    DIN : in std_logic_vector(7 downto 0);
    DOUT : out std_logic_vector(7 downto 0));
end block_ram_ex;

architecture XILINX of block_ram_ex is
component RAMB4_S8_synp
  generic( INIT_00, INIT_01 : string :=
"0000000000000000000000000000000000000000000000000
00000000000000");
port (WE, EN, RST, CLK: in STD_LOGIC;
    ADDR: in STD_LOGIC_VECTOR(8 downto 0);
    DI: in STD_LOGIC_VECTOR(7 downto 0);
    DO: out STD_LOGIC_VECTOR(7 downto 0));
```

```
end component;
begin
U1 : RAMB4_S8_synp
generic map (
INIT_00 =>
"0123456789ABCDEF0123456789ABCDEF0123456789ABCDEF0
23456789ABCDEF",
INIT_01 =>
"FEDCBA9876543210FEDCBA9876543210FEDCBA9876543210F
DCBA9876543210")
port map (WE => WE, EN => '1', RST => '0', CLK =>
CLK, ADDR => ADDR, DI => DIN,
      DO => DOUT);
end XILINX;
```

Instantiating Block SelectRAM+ Verilog Example

Verilog examples of Block SelectRAM+ instantiation are described below.

- FPGA Express

  With FPGA Express the INIT attribute has to be set in the HDL code. See the following 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));
//synopsys attribute
   INIT_00 "1F1E1D1C1B1A1918171615141312111
   00F0E0D0C0B0A0980706050403020100"
   endmodule
```

- 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
"1F1E1D1C1B1A19181716151413121110 0F0E0D0C0B0A09080
60504030201 00"
```

- LeonardoSpectrum 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

   1F1E1D1C1B1A19181716151413121110F0E0D0C0B0A090

   80706050403020100

   endmodule
```

- Synplicity block_ram_ex verilog example.

```
 'include "<synplicity_install>/lib/xilinx/
   virtex.v"
 module block_ram_ex (CLK, WE, ADDR, DIN, DOUT);
 input CLK, WE;
 input [8:0] ADDR;
 input [7:0] DIN;
 output [7:0] DOUT;
 // synthesis translate_off
 defparam

       U0.INIT_00 =

   256'h0123456789ABCDEF0123456789ABCDEF0

   123456789ABCDEF0123456789ABCDEF,

       U0.INIT_01 =

   256'hFEDCBA9876543210FEDCBA9876543210FED

   CBA9876543210FEDCBA9876543210;

    // synthesis translate_on
```

```
RAMB4_S8 U0 (.WE(WE), .EN(1'b1), .RST(1'b0),
   .CLK(CLK),
    .ADDR(ADDR), .DI(DIN), .DO(DOUT)) /*
   synthesis
xc_props="INIT_00=0123456789ABCDEF0123
456789ABCDEF0123456789ABCDEF0123456789ABCDEF,
INIT_01=FEDCBA9876543210FEDCBA9876543210FEDCBA
9876543210FEDCBA9876543210"*;
endmodule
```

Instantiating Block SelectRAM+ in Virtex-II

Virtex-II devices provide 16384-bit data memory and 2048-bit parity memory, totaling to 18Mbit memory in each Block SelectRAM+. These RAMB16_Sn (single port) and RAMB16_Sm_Sn (dual port) blocks are configurable to various width and depth. The "*Virtex Handbook*" provides VHDL and Verilog templates for Virtex-II Block SelectRAM+ instantiations. You can also refer to the "*Libraries Guide*" for a more detailed component and attribute description.

Inferring Block SelectRAM+

The following coding examples provide VHDL and Verilog coding styles for FPGA Express, LeonardoSpectrum, and Synplify. For Virtex/E and Spartan-II devices, the RAMB4_Sn or RAMB4_Sm_Sn will be inferred. For Virtex-II devices, RAMB16_Sn or RAMB16_Sm_Sn will be inferred.

Inferring Block RAM VHDL Example

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 will pointed out in this section.

- FPGA Express

  RAM inference is not supported by FPGA Express.

- LeonardoSpectrum

LeonardoSpectrum can map your memory statements in Verilog or VHDL to the block RAMs on all Virtex devices. The following is a list of the details for block SelectRAM+ in LeonardoSpectrum.

- Virtex Block RAMs are completely synchronous - both read and write operations are synchronous.
- LeonardoSpectrum infers single port RAMs - RAMs with both read and write on the same address - and, dual port RAMs - RAMs with separate read and write addresses. Currently, LeonardoSpectrum does not infer dual port RAMs that read both read and write address.
- Virtex Block RAMs support RST (reset) and ENA (enable) pins. Currently, LeonardoSpectrum does not infer RAMs which use the functionality of the RST and ENA pins.
- By default, RAMs will be mapped to block SelectRAM+ if possible. You can disable mapping to block SelectRAM+ by setting the attribute block_ram to false.

- LeonardoSpectrum 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;
```

- Synplify

  You can enable the usage of Block SelectRAMs by setting 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 selectRAMs:

  - ENA / ENB pins currently are inaccessible. The 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 RAMs.
  - Initialization of RAMs is not supported.
  - Dual port with Read-Write on a port is not supported.

- 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;
```

Inferring Block RAM Verilog Example

The following coding examples provide Verilog coding styles for FPGA Express, LeonardoSpectrum, and Synplify.

- FPGA Express

FPGA Express does not infer RAMs. All RAMs must be instantiated via primitives or cores.

- LeonardoSpectrum

Refer to the VHDL example in the section above for restrictions in inferring Block SelectRAM+.

- LeonardoSpectrum Verilog Example

```
module ram(din, we, addr, clk, dout);
    parameter data_width=7,
address_width=6,mem_elements=64;
    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];
/* Exemplar attribute mem block_ram FALSE. This
comment sets the block_ram attribute to FALSE on
the signal mem.The block_ram attribute must be
set on the memory signal.*/

    reg [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

```
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
```

Implementing Distributed SelectRAM+

Distributed SelectRAM+ can either be 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 devices only: RAM16X2S, RAM16X4S, RAM16X8S, RAM32X1S, RAM32X2S, RAM32X4S, RAM32X8S, RAM64X1S, RAM64X2S, and RAM128X1S.

- Static synchronous dual-port RAM (RAM16x1D, RAM32x1D)

  Additional dual-port RAM available dual port RAM available for Virtex-II devices only: RAM64X1D.

For more information on distributed SelectRAM, refer to the *"Libraries Guide"*.

Instantiating Distributed SelectRAM+ in VHDL

The following coding examples provide VHDL coding hints for FPGA Express, and LeonardoSpectrum.

- FPGA Express

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.
library IEEE;
use IEEE.std_logic_1164.all;
--use IEEE.std_logic_unsigned.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
component RAM16x1S is
 port (O : out std_logic;
 D : in std_logic;
     A3, A2, A1, A0 : in std_logic;
     WE, WCLK : in std_logic);
end component;
attribute INIT: string;
attribute INIT of U0: label is "FFFF";
attribute INIT of U1: label is "ABCD";
attribute INIT of U2: label is "BCDE";
attribute INIT of U3: label is "CDEF";
begin
U0 : RAM16x1S
   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
   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
```

```
   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
   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;
```

- LeonardoSpectrum

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.

library IEEE;
use IEEE.std_logic_1164.all;
entity ram_16x1s is
   generic (init_val : string := "0000" );
   port (O : out std_logic;
     D : in std_logic;
     A3, A2, A1, A0: in std_logic;
     WE, CLK : in std_logic);
end ram_16x1s;
architecture xilinx of ram_16x1s is
attribute INIT: string;
attribute INIT of u1 : label is init_val;
component RAM16X1S is port (O : out std_logic;
       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);
end component;
begin
U1 : RAM16X1S port map (O => O, WE => WE,  WCLK
   =>
 CLK, D => D, A0 => A0, A1 => A1, A2 => A2, A3 =>
   A3);
end xilinx;

library IEEE;
use IEEE.std_logic_1164.all;
--use IEEE.std_logic_unsigned.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
component ram_16x1s
 generic (init_val: string := "0000");
 port (O : out std_logic;
 D : in std_logic;
       A3, A2, A1, A0 : in std_logic;
       WE, CLK : in std_logic);
end component;
```

```
begin
U0 : ram_16x1s generic map ("FFFF")
 port map (O => o(0), WE => we, CLK => clk,
 D => d(0), A0 => a(0), A1 => a(1),
 A2 => a(2),A3 => a(3));
U1 : ram_16x1s generic map ("ABCD")
 port map (O => o(1), WE => we, CLK => clk,
 D => d(1), A0 => a(0), A1 => a(1),
 A2 => a(2), A3 => a(3));
U2 : ram_16x1s generic map ("BCDE")
 port map (O => o(2), WE => we, CLK => clk,
 D => d(2), A0 => a(0), A1 => a(1),
 A2 => a(2), A3 => a(3));
U3 : ram_16x1s generic map ("CDEF")
 port map (O => o(3), WE => we, CLK => clk,
 D => d(3), A0 => a(0), A1 => a(1),
 A2 => a(2), A3 => a(3));
end xilinx;
```

- Synplify

```
-- This example shows how to create a
-- 16x4s RAM using xilinx RAM16x1S component.
library IEEE;
use IEEE.std_logic_1164.all;
library virtex;
use virtex.components.all;
library synplify;
use synplify.attributes.all;
```

```
entity ram_16x1s is
 generic (init_val : string := "0000" );
 port (O : out std_logic;
      D : in std_logic;
      A3, A2, A1, A0: in std_logic;
      WE, CLK : in std_logic);
end ram_16x1s;

architecture xilinx of ram_16x1s is
attribute xc_props: string;
attribute xc_props of u1 : label is "INIT=" &
   init_val;

begin

U1 : RAM16X1S port map (O => O, WE => WE,  WCLK
   =>
CLK, D => D, A0 => A0, A1 => A1, A2 => A2, A3 =>
   A3);
end xilinx;

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

component ram_16x1s
 generic (init_val: string := "0000");
 port (O : out std_logic;
 D : in std_logic;
       A3, A2, A1, A0 : in std_logic;
       WE, CLK : in std_logic);
end component;

begin
U0 : ram_16x1s generic map ("FFFF")
 port map (O => o(0), WE => we, CLK => clk,
 D =>d(0), A0 => a(0), A1 => a(1),
 A2 => a(2), A3 => a(3));
U1 : ram_16x1s generic map ("ABCD")
 port map (O => o(1), WE => we, CLK => clk,
 D => d(1), A0 => a(0), A1 => a(1),
 A2 => a(2), A3 => a(3));
U2 : ram_16x1s generic map ("BCDE")
 port map (O => o(2), WE => we, CLK => clk,
 D => d(2), A0 => a(0), A1 => a(1),
 A2 => a(2), A3 => a(3));
U3 : ram_16x1s generic map ("CDEF")
```

```
 port map (O => o(3), WE => we, CLK => 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 FPGA Express, Synplify and LeonardoSpectrum.

- FPGA Express/FPGA Compiler II

```
// 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.
// synopsys translate_off
defparam RAM0.INIT="0101", RAM1.INIT="AAAA",
           RAM2.INIT="FFFF", RAM3.INIT="5555";
// synopsys translate_on
assign DATA_BUS = !WE ? DATA_OUT : 4'hz;
// Instantaition of 4 16X1 Synchronous RAMs
// Use the xc_props attribute to pass the INIT
   property
```

```
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)) ;
   /* synopsys attribute 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));
   /* synopsys attribute 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));
   /* synopsys attribute 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));
   /* synopsys attribute 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
```

- 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;

// Instantaition of 4 16X1 Synchronous RAMs

// Use the xc_props attribute to pass the INIT
   property

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

```
// This example shows how to create a
// 16x4 RAM using Xilinx RAM16X1S component.
`include "virtex.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;
// Only for Simulation -- the defparam will not
// synthesize
// Use the defparam for RTL simulation.
// There is no defparam needed for Post P&R
//simulation.
// synthesis translate_off
defparam RAM0.INIT="0101", RAM1.INIT="AAAA",
          RAM2.INIT="FFFF", RAM3.INIT="5555";
// synthesis translate_on
assign DATA_BUS = !WE ? DATA_OUT : 4'hz;
// Instantaition of 4 16X1 Synchronous RAMs
// Use the xc_props attribute to pass the INIT
property
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))
   /* synthesis xc_props="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))
   /* synthesis xc_props="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))
      /* synthesis xc_props="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))
      /* synthesis xc_props="INIT=0101" */;
   endmodule
```

Inferring Distributed SelectRAM+ in VHDL

The following coding examples provide VHDL and Verilog coding styles for FPGA Express, LeonardoSpectrum, and Synplify.

- FPGA Express

  FPGA Express does not infer RAMs.

- LeonardoSpectrum and Synplify

  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;
```

- 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 FPGA Express, Synplify and LeonardoSpectrum.

- FPGA Express

  FPGA Express does not infer RAMs.

- LeonardoSpectrum and Synplify

  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 [(1 << addr_width) 1:0];

always @(posedge clk)
        if (we)
          mem[waddr] = d;

always @(mem or 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 [(1 << 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

The following examples are RTL VHDL and Verilog ROM coding examples.

RTL Description of a ROM VHDL Example

Note LeonardoSpectrum does not infer ROM.

Use the following coding example for FPGA Express and Synplify.

```
--
--  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 ROM Verilog Example

Note LeonardoSpectrum does not infer ROM.

Use the following coding example for FPGA Express and Synplify.

```
/*
 * 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 instantiating 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.

Note Refer to your synthesis tool documentation for the correct syntax.

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. See the VHDL and Verilog RAM examples in the following section for examples of this property using a RAM primitive.

Implementing FIFO

FIFO can be implemented with FPGA RAMs. For more information on implementing FIFO using FPGAs, reference the following Application Notes:

Xilinx provide several Application Notes describing the use of FIFO when implementing FPGAs. Please refer to the following Xilinx Application Notes for more information:

- Xilinx XAPP175: *"High Speed FIFOs in Spartan-II FPGAs"*, application note, v1.0 (11/99) (http://www.xilinx.com/xapp/xapp175.pdf)
- Xilinx XAPP131: *"170MHz FIFOs using the Virtex Block SelectRAM+ Feature"*, v 1.2 (9/99) (http://www.xilinx.com/xapp/xapp131.pdf)

Implementing CAM

Content Addressable Memory (CAM) or associative memory is a storage device which can be addressed by its own contents.

Xilinx provide several Application Notes describing CAM designs in Virtex FPGAs. Please refer to the following Xilinx Application Notes for more information:

- XAPP201: *"An Overview of Multiple CAM Designs in Virtex Family Devices"* v 1.1(9/99) (http://www.xilinx.com/xapp/xapp201.pdf)
- XAPP202: *"Content Addressable Memory (CAM) in ATM Applications"* v 1.1 (9/99) (http://www.xilinx.com/xapp/xapp202.pdf)
- XAPP203: *"Designing Flexible, Fast CAMs with Virtex Family FPGAs"* v 1.1 (9/99) (http://www.xilinx.com/xapp/xapp203.pdf)

- XAPP204: *"Using Block SelectRAM+ for High-Performance Read/Write CAMs"* v1.1 (10/99) (http://www.xilinx.com/xapp/xapp204.pdf)

Using CORE Generator to Implement Memory

If you must instantiate memory, use the CORE Generator to create a memory module larger than 32X1 (16X1 for Dual Port). Implementing memory with the CORE Generator is similar to implementing any module with CORE Generator except for defining the Memory initialization file. Please reference the memory module datasheets that come with every CORE Generator module for specific information on the initialization file.

Implementing Shift Register (Virtex/E/II and Spartan-II)

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) 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 will implement longer shift registers with multiple SRL16 and additional combinatorial logic for multiplexing.

In Virtex-II 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 (MUXF5, MUXF6, and so forth) available in one Virtex-II CLB, up to a 128-bit shift register can be implemented effectively using SRLC16. Current synthesis tools (Synplify 6.0, LeonardoSpectrum 2000.1a2, and FPGA Express / Compiler II 3.5) do not yet infer the SRLC16. For more information, please refer to the *"Virtex Handbook"*.

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. Currently, the shift register components must be instantiated to implement this mode. Future releases of your synthesis tool might support inferencing this feature. Please refer to the latest release of your synthesis tool's documentation for more information.

Inferring SRL16 in VHDL

Use the following coding example for FPGA Express, Synplify and LeonardoSpectrum.

- FPGA Express

```
<PRE>
process (CLK)
begin
      if CLK'event and CLK='1'
        then
        REG <= DIN & REG(15 downto 1);
end if;
DOUT <= REG(0);
end process;
</PRE>
```

```
   --Add a clock enable signal to infer an SRL16E
   --component:
   <PRE>
process (CLK)
begin
        if CLK'event and CLK='1'
```

```
          then
            if CE='1'
            then
        REG <= DIN & REG(15 downto 1);
        end if;
            end if;
  DOUT <= REG(0);
end process;
</PRE>
```

- LeonardoSpectrum and Synplify

LeonardoSpectrum 1999.1h and later will automatically infer the SRL16. In Synplify, the SRL is mapped by default when possible. For LeonardoSpectrum and Synplify, use the following example:

```
-- 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 (input :in std_logic_vector(width - 1
downto 0);
    clk :in std_logic;
    output :out std_logic_vector(width - 1 downto
0));
    attribute clock_node :boolean;
    attribute clock_node of clk : signal is TRUE;
end pipeline_delay;
architecture behav of pipeline_delay is
type my_type is array (0 to cycle -1) of
std_logic_vector(width -1 downto 0);
signal int_sig :my_type;

begin
main :process (clk)
begin
```

```
      if clk'event and clk = '1' then
      int_sig <= input & int_sig(0 to cycle - 2);
      end if;
end process main;
output <= int_sig(cycle -1);
end behav;
```

Inferring SRL16 in Verilog

Use the following coding example for FPGA Express, Synplify and LeonardoSpectrum.

- FPGA Express

```
<PRE>
always @(posedge clk)
begin
      int  = {din, int[15:1]};
end
assign dout = int[0];
</PRE>
<PRE>
always @(posedge clk)
begin
      if (ce)
        int  = {din, int[15:1]};
end
assign dout = int[0];
</PRE>
```

- LeonardoSpectrum and Synplify

```
// 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)
```

```
begin
      if (enable == 1) begin
        for (i = (cycle-1);i >0; i=i-1) shift[i] =
      shift[i-1];
      shift[0] = data_in;
      end
end
assign result = shift[cycle-1];
endmodule
```

Implementing LFSR

The SRL (Shift Register LUT) implements very efficient shift registers and can be used to implement Linear Feedback Shift Registers. Xilinx Application Note XAPP 210 describes the implementation of Linear Feedback Shift Registers (LFSR) using the Virtex SRL macro. 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.

The XApp 210 can be downloaded from the following Xilinx web site.

http://support.xilinx.com/xapp/xapp210.pdf

Implementing Multiplexers

A 4-to-1 multiplexer can be efficiently implemented in a single Virtex/E/II and Spartan-II family slice. The six input signals (four inputs, two select lines) uses a combination of two LUTs and MUXF5 available in every slice. Up to 9 input functions can be implemented with this configuration.

In the 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 slices 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. Please refer to the *"Virtex-II Handbook"* for more information on designing large multiplexer in Virtex-II. This book can be found on the Xilinx website at www.xilinx.com.

In addition, you can use internal tristate 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 tristate 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 the "5-to-1 MUX Implemented with Gates" figure.

Some synthesis tools include commands that allow you to switch between multiplexers with gates or with tristates. Check with your synthesis vendor for more information.

The VHDL and Verilog designs provided at the end of this section show a 5-to-1 multiplexer built with tristate buffers. The tristate buffer version of this multiplexer has one-hot encoded selector inputs and requires five select inputs (SEL<4:0>). The schematic representation of these designs is shown in the "5-to-1 MUX Implemented with Gates" figure.

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
-- May 1997

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
* May 1997 */


module mux_gate (A,B,C,D,E,SEL,SIG);

input A,B,C,D,E;
input [2:0] SEL;
output SIG;
reg SIG;

    always @ (A or B or C or D or SEL)
    case (SEL)
        3'b000:
          SIG=A;
        3'b001:
          SIG=B;
        3'b010:
          SIG=C;
        3'b011:
          SIG=D;
```

```
        3'b100:
          SIG=E;
default: SIG=A;
    endcase
endmodule
```

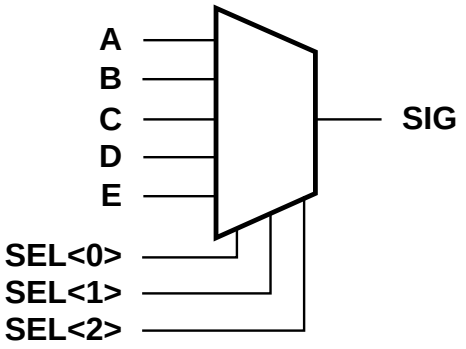

Figure 5-6 5-to-1 MUX Implemented with Gates

Wide MUX Mapped to MUXFs

Synthesis tools will use MUXF5 and MUXF6 to implement wide multiplexers. MUXF5 will be used to map 9-input function and MUXF6 to map up to 18-input function.

Currently, not all of the synthesis tools infer MUXF7 and MUXF8 for Virtex-II wide muxes. Please refer to the your synthesis tool's current documentation for updates on using this function. See the "Virtex-II Handbook" for instantiation examples

Mux Implemented with BUFTs VHDL Example

The following example shows a MUX implemented with BUFTs.

```
-- MUX_TBUF.VHD
-- 5-to-1 Mux Implemented in 3-State Buffers
-- May 1997

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

entity mux_tbuf is
port (SEL: in STD_LOGIC_VECTOR (4 downto 0);
A,B,C,D,E: in STD_LOGIC;
      SIG: out STD_LOGIC);
end mux_tbuf;

architecture RTL of mux_tbuf is
begin

    SIG <= A when (SEL(0)='0') else 'Z';
    SIG <= B when (SEL(1)='0') else 'Z';
    SIG <= C when (SEL(2)='0') else 'Z';
    SIG <= D when (SEL(3)='0') else 'Z';
    SIG <= E when (SEL(4)='0') else 'Z';
end RTL;
```

Mux Implemented with BUFTs Verilog Example

The following example shows a MUX implemented with BUFTs.

```
/* MUX_TBUF.V
 * May 1997 */

module mux_tbuf (A,B,C,D,E,SEL,SIG);

input A,B,C,D,E;
input [4:0] SEL;
output SIG;
reg SIG;

    always @ (SEL or A)
```

```
    begin
      if (SEL[0]==1'b0)
          SIG=A;
      else
          SIG=1'bz;
    end

always @ (SEL or B)
begin
      if (SEL[1]==1'b0)
          SIG=B;
      else
          SIG=1'bz;
    end

always @ (SEL or C)
begin
    if (SEL[2]==1'b0)
        SIG=C;
    else
        SIG=1'bz;
    end

always @ (SEL or D)
begin
    if (SEL[3]==1'b0)
        SIG=D;
    else
        SIG=1'bz;
    end

always @ (SEL or E)
begin
    if (SEL[4]==1'b0)
        SIG=E;
    else
        SIG=1'bz;
    end
endmodule
```

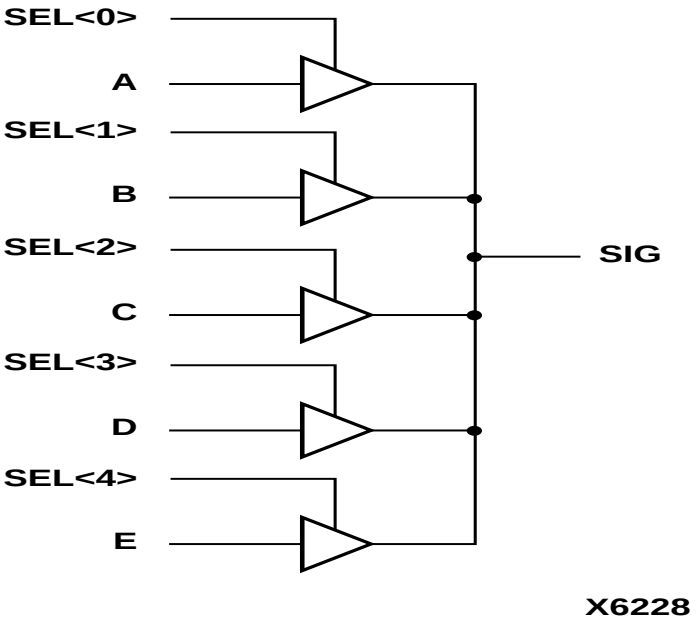

Figure 5-7 5-to-1 MUX Implemented with BUFTs

Using Pipelining

You can use pipelining to dramatically improve device performance. 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 because 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 translate these constraints to Xilinx implementation constraints. Check your synthesis tool documentation for information on multi-cycle paths. 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; and the setup time of the destination register.

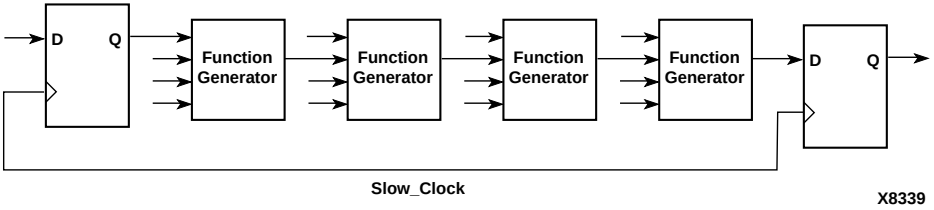

Figure 5-8 Before Pipelining

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; and the setup time of the destination register. In this example, the system clock runs much faster than in the previous example.

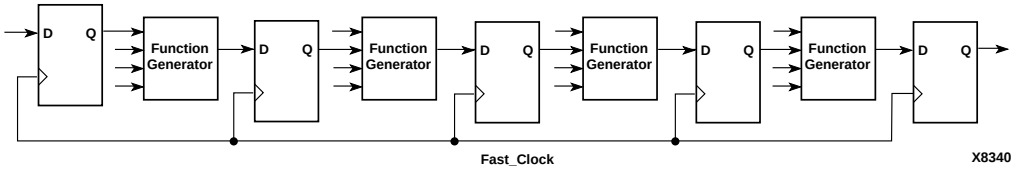

Figure 5-9 After Pipelining

Design Hierarchy

HDL designs can either be synthesized as a flat module or as many small modules. Each methodology has its advantages and disadvantages, but as higher density FPGAs are created, the advantages of hierarchical designs outweigh any disadvantages.

Advantages to building hierarchical designs are as follows.

- Easier and faster verification / simulation
- Allows several engineers to work on one design at the same time
- Speeds up design compilation
- Reduces design time by allowing design module re-use for this and future designs.
- Allows you to produce designs that are easier to understand
- Allows you to efficiently manage the design flow

Disadvantages to building hierarchical designs are as follows.

- Design mapping into the FPGA may not be as optimal across hierarchical boundaries; this can cause lesser device utilization and decreased design performance
- Design file revision control becomes more difficult
- Designs become more verbose

Most of the disadvantages listed above can be overcome with careful design consideration when choosing 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 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 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 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; the time required to complete each optimization run; if the design is worked on by a design team; and 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. Check with your synthesis vendor for guidelines.

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.

Note See your synthesis tool documentation for more information on optimizing logic across hierarchical boundaries and compiling hierarchical designs.

Modular Design and Incremental Design (ECO)

For information on Incremental Design (ECO), please refer to the following Application Notes:

- XAPP165: *"Using Xilinx and Exemplar for Incremental Designing (ECO)"*, application note, v1.0 (8/9/99) (http://www.xilinx.com/xapp/xapp165.pdf).

- XAPP164: *"Using Xilinx and Synplify for Incremental Designing (ECO)"*, application note, v1.0 (8/6/99) (http://www.xilinx.com/xapp/xapp164.pdf).

Xilinx Development Systems feature Modular Design to help you plan and manage large design. Reference the following URL and application note for more information on Modular Design feature:

- Xilinx Modular Design URL:

  http://www.xilinx.com/products/software/moddes/moddes.htm

- XAPP404: "Xilinx Modular Design", application note.

  http://www.xilinx.com/xapp/xapp404.pdf

Chapter 6

Simulating Your Design

This chapter describes the basic HDL simulation flow using the Alliance software. It includes the following sections.

- “Introduction”
- “Adhering to Industry Standards”
- “Simulation Points”
- “VHDL/Verilog Libraries and Models”
- “Compiling HDL Libraries”
- “Running NGD2VHDL and NGD2VER”
- “Understanding the Global Signals for Simulation”
- “Simulating VHDL”
- “Simulating Verilog”
- “Running Simulation”
- “LMG SmartModels”
- “IBIS”
- “STAMP”

Introduction

Increasing design size and complexity, as well as recent 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 are adopted IEEE standards.

The Xilinx implementation tools 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 the powerful place and route software that integrates with your HDL design environment on PC and UNIX workstation platforms.

Adhering to Industry Standards

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-1076-87
Verilog Language	IEEE-STD-1364-95
VITAL Modeling Standard	IEEE-STD-1076.4-95
Standard Delay Format (SDF)	2.1
Std_logic Data Type	IEEE-STD-1164-93

The Xilinx Series software currently supports the Verilog IEEE 1364 Standard, VHDL IEEE Standard 1076.4 for Vital (Vital 95), and SDF version 2.1.

Built-in Verilog support allows you to simulate with Cadence Verilog-XL and other compatible simulators.

VHDL Initiative Towards ASIC Libraries (VITAL) was created to promote the standardization and interchangeability of VHDL libraries and simulators from various vendors. It also defines a standard for timing back-annotation to VHDL simulators.

Most simulator vendors have agreed to use the IEEE-STD 1076.4 VITAL standard for the acceleration of gate-level simulations. Check with your simulator vendor to confirm that this standard is being followed, and to verify proper settings and VHDL packages for this standard. The simulator may also accelerate IEEE-STD-1164, the standard logic package for types.

The Xilinx VHDL libraries are tied to the IEEE-STD-1076.4-95 VITAL standard for simulation acceleration. This VITAL 95 is in turn based

on the IEEE-STD-1076-87 VHDL language. Because of this the Xilinx libraries must be compiled as 1076-87.

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.

Simulation Points

Xilinx supports functional and timing simulation of HDL designs at three primary points in the HDL design flow. There are two additional points at which functional simulation can occur; Functional Post-NGDBuild, and Functional Post-MAP. The second two points are optional. The “Three Primary Simulation Points for HDL Designs” figure below shows the points of the design flow. All five points are described in the following section.

1. Register Transfer Level (RTL) simulation which may include the following:
   - Instantiated UniSim library components
   - LogiBLOX models
   - XilinxCoreLib models (CORE Generator)
2. Post-synthesis functional simulation (Pre-NGDBuild) with one of the following:
   - Gate-level UniSim library components
   - LogiBLOX models
   - XilinxCoreLib models (CORE Generator)

     or

   - Gate-level pre-route SimPrim library components (Post-NGDBuild or Post-MAP))
3. Post-implementation back-annotated timing simulation with the following:
   - SimPrim library components
   - Standard Delay Format (SDF) files

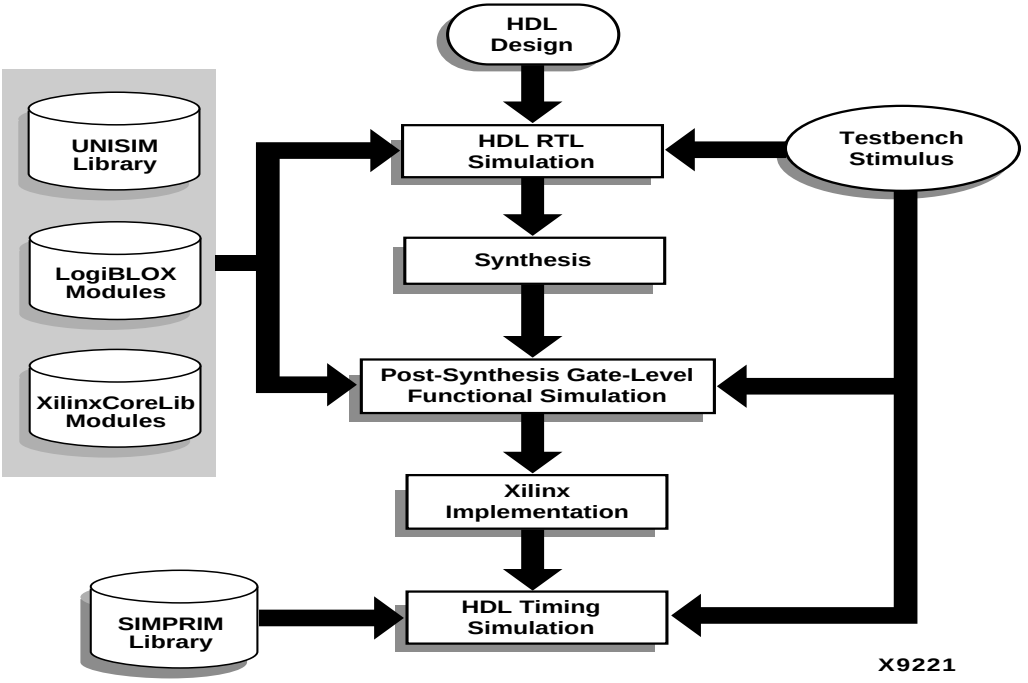

Figure 6-1 Three Primary Simulation Points for HDL Designs

The three primary simulation points can be expanded to allow for two additional post-synthesis simulations, as shown in the following table. These two additional points can be used when the synthesis tool either cannot write VHDL or Verilog, or if the netlist is not in terms of UniSim components.

Table 6-2 Five Simulation Points in HDL Design Flow

Simulation		UniSim	LogiBLOX Models	XilinxCoreLib Models	SimPrim	SDF
1.	RTL	X	X	X		
2.	Post-Synthesis	X	X	X		
3.	Functional Post-NGDBuild (Optional)				X	
4.	Functional Post-MAP (Optional)				X	X

Table 6-2 Five Simulation Points in HDL Design Flow

Simulation		UniSim	LogiBLOX Models	XilinxCoreLib Models	SimPrim	SDF
5.	Post-Route Timing				X	X

These simulation points are described in detail in the following sections. The libraries required to support the simulation flows are described in detail in the "Understanding Global Signals for Simulation" section. The new flows and libraries now support closer 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. After NGDBuild, your design is a netlist containing SimPrims. Although these library changes are fairly transparent, there are two important considerations to keep in mind; one, you must specify different simulation libraries for pre- and post-implementation simulation, and two, there are different gate-level cells in pre- and post-implementation netlists.

For Verilog, the Standard Delay Format (SDF) file is automatically read when the simulator compiles the Verilog simulation netlist. Within the simulation netlist there is the Verilog system task $sdf_annotate which specifies the name of the SDF file to be read.

For VHDL, the user specifies the location of the SDF file. The method for doing so is different depending on the simulator being used. Typically, a command line or GUI switch is used to read in the SDF file.

Register Transfer Level (RTL)

The RTL-level (behavioral) simulation allows the user 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, CORE Generator, or LogiBLOX components. To support these instantiations, Xilinx provides the UniSim, LogiBLOX,

and XilinxCoreLib libraries. The user can instantiate LogiBLOX or CORE Generator components if the user does not want to rely on the module generation capabilities of the synthesis tool, or if the design requires larger memory structures.

A general suggestion for the initial design creation is to keep the code behavioral. Avoid instantiating specific components unless necessary. This allows for more readable code, faster and simpler simulation, code portability (the ability to migrate to different device families) and code reuse (the ability to use the same code in future designs). However, you may find it necessary to instantiate components structurally in order to obtain the desired design structure or performance.

Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation

Most synthesis tools have the ability to write out a post-synthesis HDL netlist for a design. If the VHDL or Verilog netlists are written for UniSim library components, you may then use the netlists to simulate the design and evaluate the synthesis results. However, this method is not supported by Xilinx if the netlists are written in terms of the vendor's own simulation models.

The instantiated LogiBLOX or 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

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 one of the Xilinx simulation netlisters, NGD2VER or NGD2VHDL. NGD2VER and NGD2VHDL create a structural simulation netlist based on the SIMPRIM models.

Like post-synthesis simulation, pre-NGDBuild simulation, this 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 partial timing simulation.

Post-Map Partial Timing (CLB and IOB Block Delays)

You may also perform simulation after mapping the design. Post-Map simulation occurs before placing and routing. This simulation will include 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, NGD2VER or NGD2VHDL is used to create the structural simulation netlist based on SIMPRIM models.

When you run one of the simulation netlister tools, NGD2VER or NGD2VHDL, an SDF file is created. The delays for the design are stored in the SDF file and contains all block or logic delays. However, it will not contain any of the routing delays for the design since the design has not yet been placed and routed. At this point, all block delay values in the SDF file are worst case values. Actual device block delays are generally shorter under normal operating conditions.

Timing Simulation Post-Route Full Timing (Block and Net Delays)

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 will start to see how your design will behave in the 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 NGD2VER or NGD2VHDL created a structural netlist based on SIMPRIM models. However, this netlist will come 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, please refer to the "Understanding the Global Signals for Simulation"" section 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. All delays are worst case values.

Providing Stimulus

Before simulation is performed, you should create a testbench or test fixture to apply the stimulus to the design. A testbench is HDL code written for the simulator that will instantiate the design netlist(s), initialize the design and then apply stimuli to verify the functionality of the design. You can also set up the testbench to display the desired simulation output to a file, waveform or screen.

The testbench has many advantages over interactive simulation methods. For one, it allows repeatable simulation throughout the design process. It also provides documentation of the test conditions.

There are several methods to create a testbench and simulate a design. A testbench can be very simple in structure that sequentially applies stimulus to specific inputs. A testbench may also be very complex, including subroutine calls, stimulus read in from external files, conditional stimulus or other more complex structures.

NGD2VER and NGD2VHDL can optionally create a template testbench or test fixture which may simplify the simulation of the design. You may simply add your stimulus to test for desired outputs. The –tf for NGD2VER or –tb for NGD2VHDL switch will create the test fixture or testbench template. The Verilog test fixture file has a .tv extension and the VHDL test bench file has a .tvhd extension.

Xilinx recommends giving the name *test* to the main module in a Verilog testfixture file. This name is consistent with the name of the test fixture module that is written later in the design flow by NGD2VER during post-NGDBuild, post-MAP, or post-route simulation. If this naming consistency is maintained, you can use the same test fixture file for simulation at all stages of the design flow with minimal modification.

VHDL/Verilog Libraries and Models

The five simulation points listed previously require the UniSim, CORE Generator (XilinxCoreLib), LogiBLOX and SimPrim libraries.

The first point, RTL simulation, is a behavioral description of your design at the register transfer level. RTL simulation is not architecture-specific unless your design contains instantiated UniSim, CORE Generator, or LogiBLOX components. To support these instantiations, Xilinx provides a functional UniSim library, a CORE Generator Behavioral XilinxCoreLib library, and a behavioral LogiBLOX library. You can also instantiate LogiBLOX or 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.

The second simulation point is post-synthesis (pre-NGDBuild) gate-level simulation. If the UniSim library, and CORE Generator or LogiBLOX components are used, then the UniSim as well as CORE Generator and LogiBLOX libraries are used. The synthesis tool must write out the HDL netlist using UniSim primitives. Otherwise, the synthesis vendor will provide it's own post-synthesis simulation library.

The third, fourth, and fifth points (post-NGDBuild, post-map, and post-route) use the SimPrim library. The following table indicates what library is required for each of the five simulation points

Table 6-3 Simulation Phase Library Information

Simulation Point	Compilation Order of Library Required
RTL	UniSim LogiBLOX XilinxCoreLib
Post-Synthesis	UniSim (Device dependent) LogiBLOX XilinxCoreLib
Post-NGDBuild	SimPrim
Post-MAP	SimPrim
Post-Route	SimPrim

Locating Library Source Files

The following table provides information on the location of the simulation library source files, as well as the order for a typical compilation.

Table 6-4 Simulation Library Source Files

Library	Location of Source Files		Compile Order	
	Verilog	VITAL VHDL	Verilog	VITAL VHDL
UniSim 4K Family, Spartan/2/XL and Virtex/E	$XILINX/verilog/src/unisims	$XILINX/vhdl/src/unisims	Compile not required for Verilog-XL; see vendor documentation for other simulators	Required; typical compilation order: unisim_VCOMP.vhd unisim_VPKG.vhd unisim_VITAL.vhd *unisim_VCFG4K.vhd* (optional)
Xilinx-CoreLib (Device Independent)	$XILINX/verilog/src/Xilinx-CoreLib	$XILINX/vhdl/src/Xilinx-CoreLib	Compile not required for Verilog-XL; see vendor documentation for other simulators	Compilation order required; See the vhdl_analyze_order file located in $XILINX/vhdl/src/XilinxCoreLib/ for the required compile order
LogiBLOX (Device Independent)	None; uses SimPrim library	$XILINX/vhdl/src/logiblox	None; uses SimPrim library	Required; typical compilation order: mvlutil.vhd mvlarith.vhd logiblox.vhd
SimPrim (Device Independent)	$XILINX/verilog/src/simprims	$XILINX/vhdl/src/simprims	Not required for Verilog-XL; see vendor documentation for other simulators	Required; typical compilation order: simprim_Vcomponents.vhd simprim_Vpackage.vhd simprim_VITAL.vhd

Using the UniSim Library

The UniSim Library, used for functional simulation only, contains default unit delays. This library includes all the Xilinx Unified Library components that are inferred by most popular synthesis

tools. In addition, the UniSim Library includes components that are commonly instantiated, such as IOs and memory cells. You should use your synthesis tool's module generators (such as LogiBLOX) to generate higher order functions, such as adders, counters, and large RAMs.

UniSim Library Structure

The UniSim library directory structure is different for VHDL and Verilog. There is only one VHDL library for all Xilinx technologies because the implementation differences between architectures are not important for unit delay functional simulation. There are only a few cases where functional differences occur.

In these few cases, configuration statements are used to choose between architectures for the components. One library makes it easy to switch between technologies. It is left up to the user and the synthesis tool to use technology-appropriate cells.

For Verilog, separate libraries are provided for common technologies which share the same functionality. This combined library makes it easy to retarget to other technologies. It is not a requirement to change the library mapping statements when you switch from Virtex to Virtex-E or Spartan to Virtex.

Some synthesis vendors have these macros in their libraries, and can expand them to gates. You can use the HDL output from synthesis to simulate these macros.

The VHDL UniSim Library source files are found in $XILINX/vhdl/src/unisims. The following is a list of VHDL UniSim Library files.

- unisim_VCOMP.vhd (component declaration file)
- unisim_VCOMP52K.vhd (substitutional component declaration file for XC5200 designs)
- unisim_VPKG.vhd (package file)
- unisim_VITAL.vhd (model file)
- unisim_VITAL52K.vhd (additional model file for XC5200 designs)
- unisim_VCFG4K.vhd (configuration file for XC4K edge decoders)

- unisim_VCFG52K.vhd (configuration file for XC5200 internal decoders)

The following is a list of Verilog UniSim Library locations.

- $XILINX/verilog/src/uni3000 (Series 3K)
- $XILINX/verilog/src/unisims (Series 4KE, 4KX, 4KXL, 4KXLA, Spartan, SpartanXL, Spartan-II, Virtex, Virtex-E)
- $XILINX/verilog/src/uni5200 (Series 5200)
- $XILINX/verilog/src/uni9000 (Series 9500)

Note Verilog reserves the names buf, pullup, and pulldown; the Xilinx versions are changed to buff, pullup1, pullup2, or pulldown2, and then mapped to the proper cell during implementation.

Using the LogiBLOX Library

LogiBLOX is a module generator used for modules such as adders, counters, and large memory blocks. Refer to the *LogiBLOX Guide* for more information. You can enter the desired parameters into LogiBLOX and select a HDL model as output. Most LogiBLOX modules contain registers and require global set/reset (GSR) initialization. Since the modules do not contain output buffers going off-chip, the global tristate enable (GTS) initialization does not apply.

LogiBLOX Library Structure

The LogiBLOX library is not a library of modules. It is a set of packages required by the LogiBLOX models that are created "on-the-fly" by the LogiBLOX tool.

The VHDL source files are in $XILINX/vhdl/src/logiblox. The following is a list of the VHDL LogiBLOX library files.

- mvlutil.vhd
- mvlarith.vhd
- logiblox.vhd

  Note For Verilog, the LogiBLOX model is a structural netlist of SimPrim models. Do *not* synthesize this netlist; it is for functional simulation only. For VHDL the netlist is also for simulation. Do *not* synthesize the netlist.

Using the CORE Generator XilinxCoreLib Library

CORE Generator is a graphical interactive design tool for creating high-level modules such as counters, shift registers, RAM and multiplexers. You can customize and pre-optimize the modules to take advantage of the inherent architectural features of the Xilinx FPGA devices, such as Fast Carry Logic for arithmetic functions, and on-chip, dual-port and synchronous RAM. You can also select the appropriate HDL model type as output.

The CORE Generator HDL library models are used for RTL simulation. The models do not use library components for global signals.

CORE Generator Library Structure

The VHDL CORE Generator library source files are found in $XILINX/vhdl/src/XilinxCoreLib.

The Verilog CORE Generator library source files are found in $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.

SimPrim Library Structure

The VHDL SimPrim Library source files are found in $XILINX/vhdl/src/simprims.

The Verilog SimPrim Library source files are found in $XILINX/verilog/src/simprims.

Compiling HDL Libraries

For some simulators, such as NC-Verilog, VCS, VSS, and ModelSim, you may need to compile the HDL libraries before you can use them for design simulations. The advantages of compiling Verilog libraries are speed of execution and economy of memory.

Xilinx provides a perl utility to specifically compile the HDL libraries with most popular simulators. The utility is available at $XILINX/bin/<platform>/compile_hdl.pl

Using the Xilinx Compile Utility

The compile_hdl.pl utility will compile the UNISIMS, LogiBLOX and SIMPRIMS libraries for most popular simulators. For compiling the CoreGEN library, refer to the "Compiling CORE Generator Libraries" section.

To compile libraries using the compile_hdl.pl utility, type the compile script at the command line. Use the following syntax.

`compile_hdl.pl <mtiverilog|mtivhdl|ncver-ilog|vcs|vcsi|vss>`*`[path_name]`*

Based on the simulation software vendor you specify, the script compiles the libraries and places them in the appropriate directory in $XILINX. If you want to place the compiled libraries in a different directory specify a path name. After compiling the libraries, the script also creates and initialization file for the simulator. This file is place in the directory from which you ran the compile script. The naming conventions for initialization files are provided in the following table.

Table 6-5 Initialization File Names

Simulation Software Vendor	Initialization File Names
MTI-VLOG for Verilog	modlesim.ini
MTI-VCOM for VHDL	modlesim.ini
NC-Verilog	cds.lib hdl.var
VSS	.synopsys_vss.setup
VCS	N / A
VCSi	N / A

The initialization file defines the locations of the compiled libraries. When doing a simulation you must provide the initialization file either by copying the file to the directory where the HDL files are to be compiled and the simulation is to be run. Alternatively, the vendor environment variable can be set to the location of your master initialization file. You must set this variable since the installation does not initially declare the path for you.

Compiling CORE Generator Libraries

For information on compiling the CORE Generator Libraries (VHDL and Verilog), please refer to the following web site, http://support.xilinx.com/techdocs/8733.htm.

Running NGD2VHDL and NGD2VER

Xilinx provides programs that will create a netlist file from your VHDL or Verilog NGD file. You can run either netlist from the Design Manager or the command line. Both options are described below.

Creating a Simulation Netlist

You can create a timing simulation netlist from the Design Manager or from the command line, as described in this section.

From the Design Manager

1. Select **Design** → **Options** in the Design Manager.

   or

   Select **Setup** → **Options** in the Flow Engine.

   The Options dialog box appears.

2. Select the Simulation drop-down list in the Options dialog box and select the appropriate simulation netlist type. This will automatically use the appropriate options to create the specified netlist

3. Select the Edit Options button next to the Simulation drop-down list.

   The Simulation Options dialog box appears.

4. Select the General tab.

5. In the General tab, select the applicable options as follows.

   - Format

     Specify the netlist format to use for simulation. The format is usually VHDL or Verilog for synthesis designs.

- Correlate Simulation Data to Input Design

  This option enables signal back annotation to the original compiled netlist. By default, this option is off. Since many of the internal signal and instance names of the design were probably created by the synthesis tool and contain names that have no meaning to the designer, this step is not usually necessary and disabling this feature will decrease the run time of the Alliance Series software.

- Simulation Netlist Name (default is time_sim)

6. Select the VHDL / Verilog tab. There are several options to choose from. For more information on these options, see the *"Design Manager/Flow Engine Guide."*

   - Bring Out Global Set / Reset Net as a Port

     This option creates an external port in the simulation netlist to allow control of the power-on-reset from a port.

   - Bring Out Global Tristate Net as a Port
   - Generate Test Fixture / Testbench File

     This option will create a testbench.

   - Include uselib Directive in Verilog File
   - Generate Pin File
   - Retain Hierarchy in Netlist. (You must also choose the Correlate Simulation Data to Input Design option when choosing this option)
   - Rename Architecture Name to:.

7. Click **OK** in the Simulation Options dialog box.
8. Click **OK** in the Options dialog box.
9. When you implement your design, the Flow Engine produces timing simulation data files.

From the Command Line

Note To display the available options for the programs in this section, enter the program executable name at the prompt without any arguments. For complete descriptions of these options, refer to the *Development System Reference Guide*.

1. Run NGDAnno on your placed and routed .ncd file.

   For back-annotated output (signals correlated to original netlist), enter the following.

   ngdanno -p *design*.**pcf** *design*.**ncd** *design*.**ngm**

   For output that is not back-annotated (faster run time), enter the following.

   ngdanno *design*.**ncd**

2. Run the *NGD2XXX* program for the particular netlist you want to create.

   For VHDL, enter the following.

   ngd2vhdl [options] *design*.**nga**

   For Verilog, enter the following.

   ngd2ver [options] *design*.**nga**

   Note For post-NGDBuild and post-Map simulation, step 1 is skipped. You can move to step 2 if you are doing post-NGDBuild or post-Map simulation.

Enabling ‘X’ Propagation

The "X" propagation option can now be enabled or disabled in timing simulation on memory elements such as flops and memories.

For Verilog, ‘X’ propagation is enabled by default. To disable ‘X’ propagation use the **+no_notifier** option which is native to your simulator. This option disables the toggling of the notifier register argument of the timing check system tasks. By default, the notifier is toggled when there is a timing check violation, and the notifier usually causes a UDP to propagate an ‘X’. Therefore, the +no_notifier option suppresses ‘X’ propagation on timing violations.

For VHDL, this is done with the **-xon** option within NGD2VHDL. The **-xon** option specifies the output behavior when timing violations occur on synchronous elements. If this option is set to equal true, any synchronous elements that violate a setup time trigger "X" on the outputs. If the option is set to equal false, the signal’s previous value is retained. If this option is not set by the user, **-xon** is set to true by default.

Min/Typ/Max Simulation

During simulation you have the option of specifying one delay mode from either minimum delays, typical delays, or maximum delays. However, each field will contain the maximum delays value. The Xilinx Alliance tools state the worst case numbers by default. Currently, it is not possible to generate one SDF file that contains separate values for the min / typ / max fields.

You can specify to get the minimum delay, but all three fields, min / typ / max, will hold the same minimum delays value. To generate a simulation netlist with minimum delays, type the following:

ngdanno -s min *design*.ncd

For VHDL, enter the following:

ngd2vhdl [options] *design*.nga

For Verilog, enter the following:

ngd2ver [options] *design*.nga

Minimum delays may only be available for select FPGA families. The minimum delays are for all speed grades of a device. They are not suggested for use for any worst case analysis (i.e. setup, min clock, max data). These minimum speeds files were created to allow users to check timing between chips on a board. For further details, please see http: / / support.xilinx.com / techdocs / 4422.htm.

Prorating Simulation

Prorating is a linear scaling operation. It applies to existing speed file delays and is applied globally to all delays. The prorating constraints, VOLTAGE and TEMPERATURE, provide a method for determining timing delay characteristics based on known environmental parameters.

The VOLTAGE constraint provides a means of prorating delay characteristics based on the specified voltage. The UCF syntax is as follows:

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.

The TEMPERATURE constraint provides a means of prorating device delay characteristics based on the specified junction temperature. The UCF syntax is as follows:

TEMPERATURE=***value***[C |F| K]

Where *value* is an integer or a real number specifying the temperature. C, K, and F are the temperature units: F is degrees Fahrenheit, K is degrees Kelvin, and C is degrees Celsius, the default.

Note Each architecture has its own specific range of valid operating temperatures and voltages. If the entered temperature or voltage does not fall within the supported range, the constraint is ignored and an architecture-specific default value is used instead.

For simulation, the VOLTAGE and TEMPERATURE constraints will be processed from the UCF file into the PCF file. To generate a simulation netlist using prorating, type the following:

ngdanno -p *design*.pcf *design*.ncd

For VHDL, enter the following:

ngd2vhdl [options] *design*.nga

For Verilog, enter the following:

ngd2ver [options] *design*.nga

Note Do not combine both minimum timing and prorating (-s and -p). Combining both minimum values would override prorating.

Prorating may only be available for select FPGA families, and it is not intended for military and industrial ranges. It is applicable only within the commercial ranges.

Understanding the Global Signals for Simulation

Xilinx FPGAs have register (flip-flops and latches) set / reset circuitry that pulses at the end of the configuration mode. This pulse is automatic and does not need to be programmed. All the flip-flops and latches receive this pulse through a dedicated global GSR (Global Set-Reset) net. The registers either set or reset, depending on how the registers are defined.

It is important to address the built-in reset circuitry behavior in your designs starting with the first simulation to ensure that the simulations agree at the three primary points.

If you do not simulate GSR behavior prior to synthesis and place and route, your RTL and possibly post-synthesis simulations will not initialize to the same state as your post-route timing simulation. As a result, the various design descriptions will not be functionally equivalent and your simulation results will not match. Some synthesis tools can identify, from the behavioral description, the GSR net, and will place the STARTUP module on the net to direct the implementation tools to use the global network. However, other synthesis tools interpret behavioral descriptions literally, and will introduce additional logic into your design to implement a function. Without specific instructions to use device global networks, the Xilinx implementation tools will use general purpose logic and interconnect resources to redundantly build functions already provided by the silicon.

If GSR behavior is not described, the chip will initialize during configuration, and the post-route netlist will include this net that must be driven during simulation. This section includes the methodology to describe this behavior, as well as the GTS behavior for output buffers.

In addition to the set/reset pulse, all output buffers are set to a high impedence state during configuration mode with the dedicated global output tristate enable (GTS) net.

The GSR net requires special handling during synthesis, simulation, and implementation to prevent them from being assigned to normally routed nets, which uses valuable routing resources and degrades design performance. The GSR net receives a reset-on-configuration pulse from the initialization controller, as shown in the following figure.

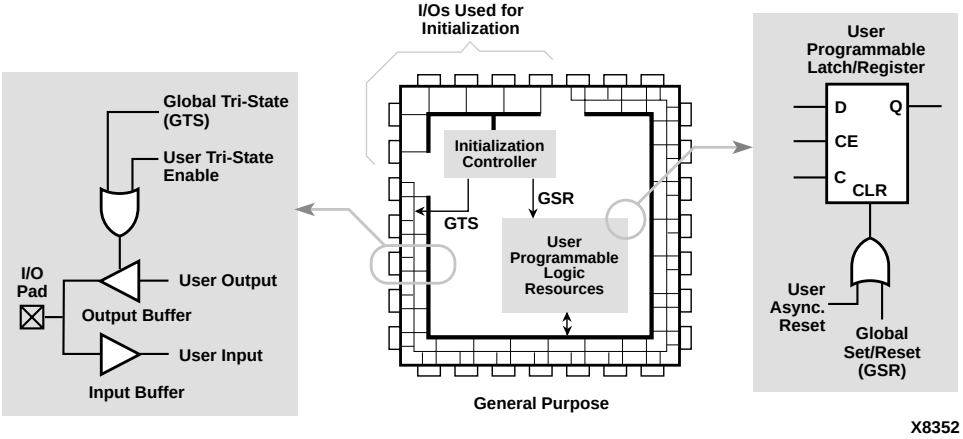

Figure 6-2 Built-in FPGA Initialization Circuitry

This pulse occurs during the configuration mode of the FPGA. However, for ease of simulation, it is usually inserted at time zero of the test bench, before logical simulation is initiated. The pulse width is device-dependent and can vary widely, depending on process voltage and temperature changes. The pulse is guaranteed to be long enough to overcome all net delays on the reset special-purpose net. The parameter for the pulse width is TPOR, as described in *The Programmable Logic Data Book*.

The tristate-on-configuration circuit shown in the "Built-in FPGA Initialization Circuitry" also occurs during the configuration mode of the FPGA. Just as for the reset-on-configuration simulation, it is usually inserted at time zero of the test bench before logical simulation is initiated. The pulse drives all outputs to the tristate condition they are in during the configuration of the FPGA. All general-purpose outputs are affected whether they are regular, tristate, or bi-directional outputs during normal operation. This ensures that the outputs do not erroneously drive other devices as the FPGA is being configured. The pulse width is device-dependent and can vary widely with process and temperature changes. The pulse is guaranteed to be long enough to overcome all net delays on the GTS net. The

generating circuitry is separate from the reset-on-configuration circuit. The pulse width parameter is T_{POR}, as described in *The Programmable Logic Data Book*. Simulation models use this pulse width parameter for determining HDL simulation for global reset and tristate circuitry.

If a global set/reset is desired for behavioral simulation, it must be included in the behavioral code. Any described register in the code must have a common signal that will asynchronously set or reset the register depending on the desired result. Similarly, if a global tristate-state is desired for simulation, it should be described in the code as well.

Simulating VHDL

Defining Global Signals in VHDL

In VHDL designs, any signals that are stimulated or monitored from outside a module must be declared as ports. Global GSR and GTS signals are used to initialize the simulation and require access ports if controlled from the test bench. However, the addition of these ports makes the pre- and post-implementation versions of your design different, and your original test bench is no longer applicable to both versions of your design. Since the port lists for the two versions of your design are different, the socket in the test bench matches only one of them. To address this issue, five new cells are provided for VHDL simulation: ROC, ROCBUF, TOC, TOCBUF, and STARTBUF.

Verilog can simulate internal signals, and these signals are driven directly from the test bench. However, interpretive Verilog (such as Verilog-XL) and compiled Verilog (such as MTI or NC-Verilog) require a different approach for handling the libraries.

The VHDL global signal simulation methodology follows the schematic flow in that there is no need to incorporate any ports into designs for simulators to mimic the device's global reset (GSR) or global tristate (GTS) networks. These signals are not part of the cell's pin list, do not appear in the netlist, and are not implemented in the resulting design. These global signals are mapped into the equivalent signals in the back-end simulation model. Using this methodology with schematic designs, you can fully simulate the silicon's built-in

global networks and implement your design without causing congestion of the general-purpose routing resources and degrading the clock speed.

Setting VHDL Global Set/Reset Emulation in Functional Simulation

When using the VHDL UniSim library, it is important to control the global signals for reset and output tristate enable. If do not control these signals, your timing simulation results will not match your functional simulation results because the initialization differs.

VHDL simulation does not support test bench driven internal global signals. If the test bench drives the global signal, a port is required. Otherwise, the global net must be driven by a component within the architecture.

Also, the register components do not have pins for the global signals because you do not want to wire to these special pre-laid nets. Instead, you want implementation to use the dedicated network on the chip.

The VHDL UniSim library uses special components to drive the local reset and tristate enable signals. These components use the local signal connections to emulate the global signal, and also provide the implementation directives to ensure that the pre-routed wires are used.

You can instantiate these special components in the RTL description to ensure that all functional simulations match the timing simulation with respect to global signal initialization.

For functional simulation, the global reset and output tristate enable signals can be emulated in two ways:

- Instantiating the STARTUP library component. This component is available for the Virtex, VirtexE, Virtex2 and Spartan-II families.
- Using local reset and tristate enable signals in the design. Special implementation directives are put on the nets to move them to special pre-routed nets for global signals.

Global Signal Considerations (VHDL)

The following are important considerations for VHDL simulation, synthesis, and implementation of global signals in FPGAs.

- The global signals have automatically generated pulses that always occur even if the behavior is not described in the front-end description. The back-annotated netlist has these global signals, to match the silicon, even if the source design does not.
- The simulation and synthesis models for registers (flip-flops and latches) and output buffers do not contain pins for the global signals. This is necessary to maintain compatibility with schematic libraries that do not require the pin to model the global signal behavior.
- VHDL does not have a standardized method for handling global signals that is acceptable within a VITAL-compatible library.
- LogiBLOX generates modules that are represented as behavioral models and require a different way to handle the global signal, yet still maintain compatibility with the method used for general user-defined logic and LogiBLOX.
- Intellectual property cores from the CORE Generator are represented as behavioral models and require a different way to handle the global signal, yet still maintain compatibility with the method used for general user-defined logic and LogiBLOX.
- The design is represented at different levels of abstraction during the pre- and post-synthesis and implementation phases of the design process. The solutions work for all three levels and give consistent results.
- The place and route tools must be given special directives to identify the global signals in order to use the built-in circuitry instead of the general-purpose logic.

GSR Network Design Cases

When defining a methodology to control a device's global set/reset (GSR) network, you should consider the following three general cases.

Table 6-6 GSR Design Cases

Name	Description
Case 1 Case 1A Case 1B	Reset-On-Configuration pulse only; no user control of GSR Simulation model ROC initializes sequential elements User initializes sequential elements with ROCBUF model and simulation vectors
Case 2 Case 2A Case 2B	User control of GSR after Power-on-Reset External Port driving GSR External Port driving GSR (Virtex and Spartan-II)
Case 3	Don't Care

Note Reset-on-Configuration for FPGAs is similar to Power-on-Reset for ASICs except it occurs during power-up and during configuration of the FPGA.

Case 1 is defined as follows.

- Automatic pulse generation of the Reset-On-Configuration signal
- No control of GSR through a test bench
- Involves initialization of the sequential elements in a design during power-on, or initialization during configuration of the device
- Need to define the initial states of a design's sequential elements, and have these states reflected in the implemented and simulated design
- Two sub-cases
  - In Case 1A, you do not provide the simulation with an initialization pulse. The simulation model provides its own mechanism for initializing its sequential elements (such as the real device does when power is first applied).
  - In Case 1B, you can control the initializing power-on reset pulse from a test bench without a global reset pin on the

FPGA. This case 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 possibly provide further pulses as simulation time progresses (perhaps to simulate cycling power to the device). Although you are providing the reset pulse to the simulation model, this pulse is not required for the implemented device. A reset port is not required on the implemented device, however, a reset port is required in the behavioral code through which your reset pulse can be applied with test vectors during simulation.

Using VHDL Reset-On-Configuration (ROC) Cell (Case 1A)

For Case 1A, the ROC (Reset-On-Configuration) instantiated component model is used. This model creates a one-shot pulse for the global set / reset signal. The pulse width is a generic and can be configured to match the device and conditions specified. The ROC cell is in the post-routed netlist and, with the same pulse width, it mimics the pre-route global set / reset net. The following is an example of an ROC cell.

Note The TPOR parameter from *The Programmable Logic Data Book* is used as the WIDTH parameter.

```
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;
```

Using ROC Cell Implementation Model (Case 1A)

Complementary to the previous VHDL model is an implementation model that guides the place and route tool to connect the net driven by the ROC cell to the special purpose net.

This cell is created during back-annotation if you do not use the –gp or STARTUP block options. It can be instantiated in the front end to match functionality with GSR (in both functional and timing simulation.) During back-annotation, the entity and architecture for the ROC cell is placed in your design's output VHDL file. In the front end, the entity and architecture are in the UniSim Library, requiring only a component instantiation. The ROC cell generates a one-time initial pulse to drive the GSR net starting at time zero for a specified pulse width. You can set the pulse width with a generic in a configuration statement. The default value of the pulse width is 0 ns. This value disables the ROC cell and causes the global set/reset to be held

low. (Active low resets are handled within the netlist itself and need to be inverted before using.)

ROC Test Bench (Case 1A)

With the ROC cell you can simulate with the same test bench used in RTL simulation, and you can control the width of the global set / reset signal in your implemented design. ROC cells require a generic WIDTH value, usually specified with a configuration statement. Otherwise, a generic map is required as part of the component instantiation. You can set the generic with any generic mapping method. Set the width generic after consulting *The Programmable Logic Data Book* for the particular part and mode implemented. For example, an XC4000E part can vary from 10 ms to 130 ms. Use the TPOR parameter in the Configuration Switching Characteristics tables for Master, Slave, and Peripheral modes. The following is the test bench for the ROC example.

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

library UNISIM;
use UNISIM.all;

entity test_ofex_roc is end test_ofexroc;

architecture inside of test_ofex_roc is

Component ex_roc
Port ( CLOCK, ENABLE: in STD_LOGIC;
    CUP, CDOWN: out STD_LOGIC_VECTOR (3 downto 0));
End component;
.
.
.

Begin

UUT: ex_roc port map(. . . .);
.
.
```

```
.
End inside;
```

The best method for mapping the generic is a configuration in your test bench, as shown in the following example.

```
Configuration overall of test_ofexroc is
For inside
      For UUT:ex_roc
        For A
          For U1:ROC use entity UNISIM.ROC (ROC_V)
          Generic map (WIDTH=>52 ns);
        End for;
      End for;
End for;
End overall;
```

This configuration is for pre-NGDBuild simulation. A similar configuration is used for post-NGDBuild simulation. The ROC, TOC, and OSC4 are mapped to the WORK library, and corresponding architecture names may be different. Review the .vhd file created by NGD2VHDL for the current entity and architecture names for post-NGDBuild simulation.

ROC Model in Four Design Phases (Case 1A)

The following figure shows the progression of the ROC model and its interpretation in the four main design phases.

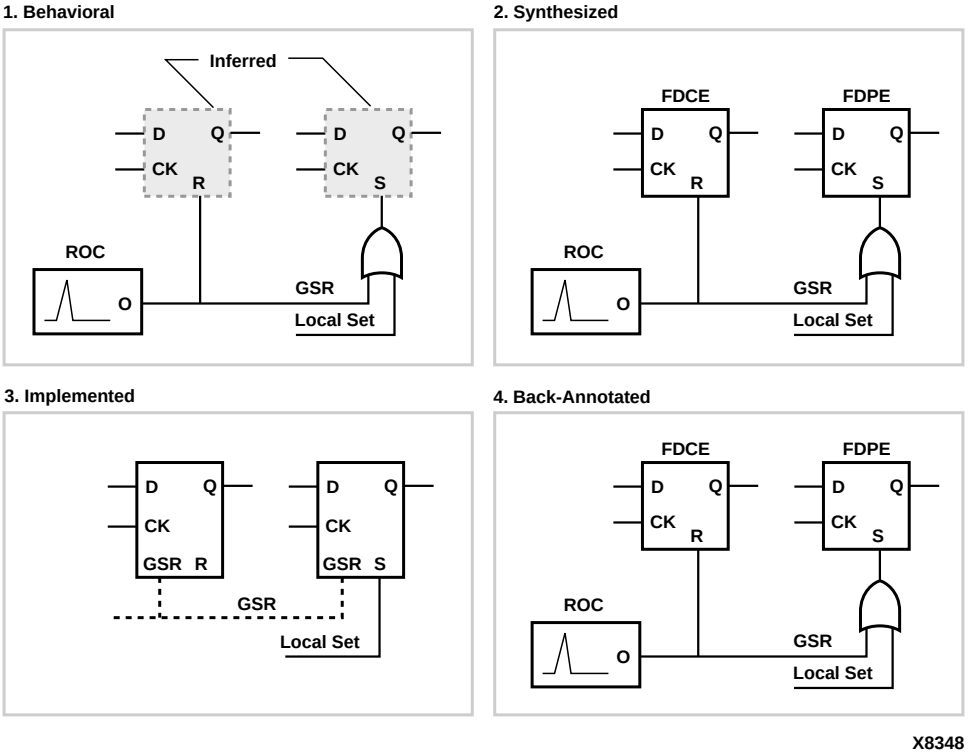

Figure 6-3 ROC Simulation and Implementation

- *Behavioral Phase*—In this phase, the behavioral or RTL description registers are inferred from the coding style, and the ROC cell can be instantiated. If it is not instantiated, the signal is not driven during simulation or is driven within the architecture by code that cannot be synthesized. Some synthesizers infer the local resets that are best for the global signal and insert the ROC cell automatically. When this occurs, instantiation may not be required unless RTL level simulation is needed. The synthesizer may allow you to select the reset line to drive the ROC cell. Xilinx recommends instantiation of the ROC cell during RTL coding because the global signal is easily identified. This also ensures that GSR behavior at the RTL level matches the behavior of the post-synthesis and implementation netlists.

- *Synthesized Phase*—In this phase, inferred registers are mapped to a technology and the ROC instantiation is either carried from the RTL or inserted by the synthesis tools. As a result, consistent global set/reset behavior is maintained between the RTL and synthesized structural descriptions during simulation.
- *Implemented Phase*—During implementation, the ROC is removed from the logical description that is placed and routed as a pre-existing circuit on the chip. The ROC is removed by making the output of the ROC cell appear as an open circuit. Then the implementation tool can trim all the nets driven by the ROC to the local sets or resets of the registers, and the nets are not routed in general purpose routing. 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 this phase, the Xilinx VHDL netlist program assumes all registers are driven by the GSR net; replaces the ROC cell; and rewires it to the GSR nets in the back-annotated netlist. The GSR net is a fully wired net and the ROC cell is inserted to drive it. A similar VHDL configuration can be used to set the generic for the pulse width.

Using VHDL ROCBUF Cell (Case 1B)

For Case 1B, the ROCBUF (Reset-On-Configuration Buffer) instantiated component is used. 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. During the place and route process, this port is removed so it is not implemented on the chip. ROCBUF does not reappear in the post-routed netlist. Instead, you can select an implementation option to add a global set/reset port to the back-annotated netlist. The nets driven by a ROCBUF must be an active High set/reset. A buffer is not necessary since the implementation directive is no longer required.

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;
```

ROCBUF Model in Four Design Phases (Case 1B)

The following figure shows the progression of the ROCBUF model and its interpretation in the four main design phases.

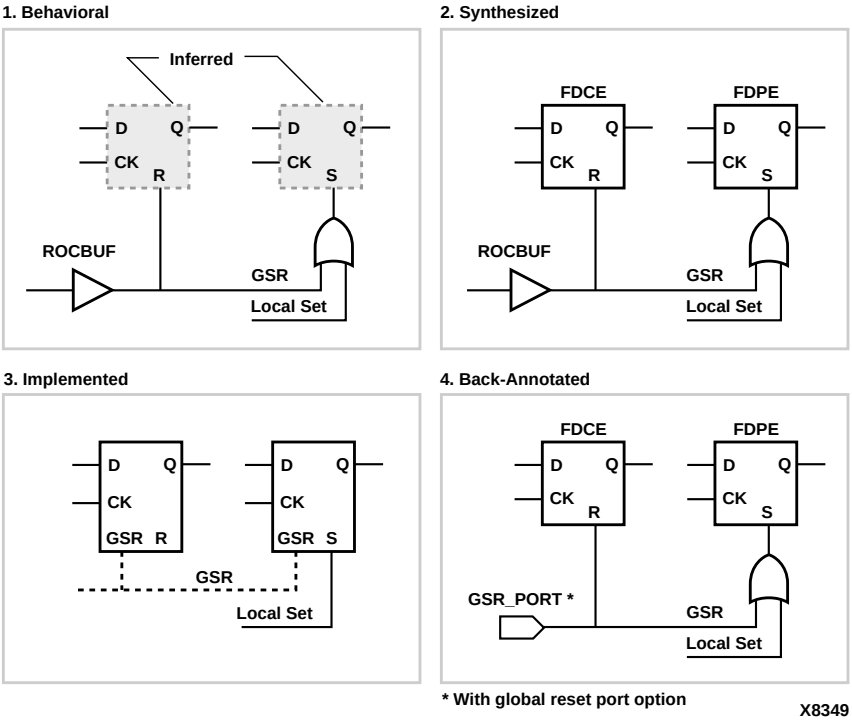

Figure 6-4 ROCBUF Simulation and Implementation

- *Behavioral Phase*—In this phase, the behavioral or RTL description registers are inferred from the coding style, and the ROCBUF cell can be instantiated. If it is not instantiated, the signal is not driven during simulation, or it is driven within the architecture by code that cannot be synthesized. Use the ROCBUF cell instead of the ROC cell when you want test bench control of GSR simulation. Xilinx recommends instantiating the ROCBUF cell during RTL coding because the global signal is easily identified, and you are not relying on a synthesis tool feature that may not be available if ported to another tool. This also ensures that GSR behavior at the

RTL level matches the behavior of the post-synthesis and implementation netlists.

- *Synthesized Phase*—In this phase, inferred registers are mapped to a technology and the ROCBUF instantiation is either carried from the RTL or inserted by the synthesis tools. As a result, consistent global set/reset behavior is maintained between the RTL and synthesized structural descriptions during simulation.
- *Implemented Phase*—During implementation, the ROCBUF is removed from the logical description that is placed and routed as a pre-existing circuit on the chip. The ROCBUF is removed by making the input and the output of the ROCBUF cell appear as an open circuit. Then the implementation tool can trim the port that drives the ROCBUF input, as well as the nets driven by the ROCBUF output. As a result, nets are not routed in general purpose routing. All set/resets for the registers are automatically assumed to be driven by the global set/reset net so data is not lost. You can use a VHDL netlist tool option to add the port back.
- *Back-annotated Phase*—In this phase, the Xilinx VHDL netlist program starts with all registers initialized by the GSR net, and it replaces the ROC cell it would normally insert with a port if the GSR port option is selected. The GSR net is a fully wired net driven by the added GSR port. A ROCBUF cell is not required because the port is sufficient for simulation, and implementation directives are not required

Using VHDL STARTBUF Block (Case 2A)

The STARTUP block is traditionally instantiated to identify the GSR signals for implementation if the global reset or tristate is connected to a chip pin. However, this implementation directive component cannot be simulated, and causes warning messages from the simulator. However, you can use the STARTBUF cell instead, which can be simulated. STARTUP blocks are allowed if the warnings can be addressed or safely ignored.

For Cases 2A, use the STARTBUF cell. This cell provides access to the input and output ports of the STARTUP cell that direct the implementation tool to use the global networks. The input and output port names differ from the names of the corresponding ports of the STARTUP cell. This was done for the following reasons.

- To make the STARTBUF a model that can be simulated with inputs and outputs. The STARTUP cell hangs from the net it is connected to.
- To make one model that works for all Xilinx technologies. The XC4000 and XC5200 families require different STARTUP cells because the XC5200 has a global reset (GR) net and not a GSR. For the Virtex and Spartan families, see Case 2B.

  The mapping to the architecture-specific STARTUP cell from the instantiation of the STARTBUF is done during implementation. The STARTBUF pins have the suffix “IN” (input port) or “OUT” (output port). Two additional output ports, GSROUT and GTSOUT, are available to drive a signal for clearing or setting a design's registers (GSROUT), or for tristating your design's I/Os (GTSOUT).

  The input ports, GSRIN and GTSIN, can be connected either directly or indirectly via combinational logic to input ports of your design. Your design's input ports appear as input pins in the implemented design. The design input port connected to the input port, GSRIN, is then referred to as the device reset port, and the design input port connected to the input port, GTSIN, is referred to as the device tristate port. The following table shows the correspondence of pins between STARTBUF and STARTUP.

Table 6-7 STARTBUF/STARTUP Pin Descriptions

STARTBUF Pin Name	Connection Point	XC4000 STARTUP Pin Name	XC5200 STARTUP Pin Name	Spartan
GSRIN	Global Set/Reset Port of Design	GSR	GR	GSR
GTSIN	Global Tristate Port of Design	GTS	GTS	GTS
GSROUT	All Registers Asynchronous Set/Reset	Not Available For Simulation Only	Not Available For Simulation Only	Not Available For Simulation Only
GTSOUT	All Output Buffers Tristate Control	Not Available For Simulation Only	Not Available For Simulation Only	N/A

Table 6-7 STARTBUF/STARTUP Pin Descriptions

STARTBUF Pin Name	Connection Point	XC4000 STARTUP Pin Name	XC5200 STARTUP Pin Name	Spartan
CLKIN	Port or INternal Logic	CLK	CLK	CLK
Q2OUT	Port Or Internal Logic	Q2	Q2	Q2
Q3OUT	Port Or Internal Logic	Q3	Q3	Q3
OUT	Port Or Internal Logic	Q1Q4	Q1Q4	Q1Q4
DONEINOUT	Port Or Internal Logic	DONEIN	DONEIN	DONEIN

Note Using STARTBUF indicates that you want to access the global set/reset and/or tristate pre-routed networks available in your design's target device. As a result, you must provide the stimulus for emulating the automatic pulse as well as the user-defined set/reset. This allows you complete control of the reset network from the test bench.

The following example shows how to use the STARTBUF cell.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.all;
entity EX_STARTBUF is
    port (CLOCK, ENABLE, DRP, DTP : in std_logic;
          CUP, CDOWN : out std_logic_vector (3 downto 0));
end EX_STARTBUF
architecture A of EX_STARTBUF is
    signal GSR, GSRIN_NET, GROUND, GTS : std_logic;
    signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
    component STARTBUF
        port (GSRIN, GTSIN, CLKIN : in std_logic; GSROUT, GTSOUT,
              DONEINOUT, Q1Q4OUT, Q2OUT, Q3OUT : out std_logic);
    end component;
begin
```

```
    GROUND <= '0';
    GSRIN_NET <= NOT DRP;
    U1 : STARTBUF port map (GSRIN => GSRIN_NET, GTSIN => DTP,
          CLKIN => GROUND, GSROUT => GSR, GTSOUT => GTS);
    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 when (GTS = '0' AND COUNT_UP /= "0000") else
"ZZZZ";
    CDOWN <= COUNT_DOWN when (GTS = '0') else "ZZZZ";
end A;
```

Using VHDL STARTBUF_VIRTEX Block and STARTBUF_SPARTAN2 Block (Case 2B)

The STARTUP_VIRTEX and STARTUP_SPARTAN2 blocks can be instantiated to identify the GSR signals for implementation if the global reset or tristate is connected to a chip pin. However, these cells can not be simulated as there is no simulation model for them.

The VHDL STARTBUF_VIRTEX and STARTBUF_SPARTAN2 blocks can not be used to simulate the GSR signal during any pre-NGDBuild Unisim VHDL simulations.You will not be able to reset the design after simulation time ‘0’, however the design will start up in the

correct state.You can do a pre-NGDBuild UniSim simulation of the GTS signal. Please see the GTS Network Design Cases section.

Note Post-NGDBuild SimPrim VHDL simulation of GSR is supported. To correctly back-annotate a GSR signal, instantiate a STARTUP_VIRTEX, STARTBUF_VIRTEX, STARTUP_SPARTAN2, or STARTBUF_SPARTAN2 symbol and correctly connect the GSR input signal of the component. When back annotated, your GSR signal is correctly connected to the associated registers and RAM blocks

Table 6-8 Virtex/E and Spartan-II STARTBUF/STARTUP Pins

STARTBUF Pin Names	Connection Points	Virtex/E STARTUP Pin Names	Spartan-II STARTUP Pin Names
GSRIN	Global Set / Reset Port of Design	GSR	GSR
GTSIN	Global Tristate Port of Design	GTS	GTS
CLKIN	Port or Internal Logic	CLK	CLK
GTSOUT	All Output Buffers Tristate Control	N / A	N / A

Xilinx recommends that you use the local routing for Virtex devices as opposed to using the dedicated GSR. If the design resources are available using this method will provide better performance.

If you do not plan on bringing the GSR pin out to a device pin, but want to have access to it for simulation you can use the ROC or ROCBUF with any device including Virtex, Virtex-E and and Spartan-II.

GTS Network Design Cases

Just as for the global set/reset net there are three cases for using your device's output tristate enable (GTS) network, as shown in the following table.

Table 6-9 GTS Design Cases

Name	Description
Case A Case A1 Case A2	Tristate-On-Configuration only; no user control of GTS Simulation Model TOC tristates output buffers during configuration or power-up User initializes sequential elements with TOCBUF model and simulation vectors
Case B Case B1 Case B2	User control of GTS after Tristate-On-Configuration External PORT driving GTS External Port driving GTS (Virtex and Spartan-II
Case C	Don't Care

Case A is defined as follows.

- Tristating of output buffers during power-on or configuration of the device
- Output buffers are tristated and reflected in the implemented and simulated design
- Two sub-cases
  - In Case A1, you do not provide the simulation with an initialization pulse. The simulation model provides its own mechanism for initializing its sequential elements (such as the real device does when power is first applied).
  - In Case A2, you can control the initializing Tristate-On-Configuration pulse. This case is applicable when system-level issues make your design's configuration synchronous with an off-chip event. In this case, you provide a pulse to tristate the output buffers at the start of simulation time, and possibly provide further pulses as simulation time progresses (perhaps to simulate cycling power to the device). Although you are providing the Tristate-On-Configuration pulse to the simulation model, this pulse is not required for the implemented device. A Tristate-On-Configuration port is not

required on the implemented device, however, a TOC port is required in the behavioral code through which your TOC pulse can be applied with test vectors during simulation.

Using VHDL Tristate-On-Configuration (TOC)

The TOC cell is created if you do not use the –tp or STARTUP block options. The entity and architecture for the TOC cell is placed in the design's output VHDL file. The TOC cell generates a one-time initial pulse to drive the GSR net starting at time '0' for a user-defined pulse width. The pulse width can be set with a generic. The default WIDTH value is 0 ns, which disables the TOC cell and holds the tristate enable low. (Active low tristate enables are handled within the netlist itself; you must invert this signal before using it.)

The TOC cell enables you to simulate with the same test bench as in the RTL simulation, and also allows you to control the width of the tristate enable signal in your implemented design.

The TOC components require a value for the generic WIDTH, usually specified with a configuration statement. Otherwise, a generic map is required as part of the component instantiation.

You may set the generic with any generic mapping method you choose. Set the WIDTH generic after consulting *The Programmable Logic Data Book* for the particular part and mode you have implemented. For example, an XC4000E part can vary from 10 ms to 130 ms. Use the TPOR (Power-On Reset) parameter found in the Configuration Switching Characteristics tables for Master, Slave, and Peripheral modes.

VHDL TOC Cell (Case A1)

For Case A1, use the TOC (Tristate-On-Configuration) instantiated component. This component creates a one-shot pulse for the global Tristate-On-Configuration signal. The pulse width is a generic and can be selected to match the device and conditions you want. The TOC cell is in the post-routed netlist and, with the same pulse width set, it mimics the pre-route Tristate-On-Configuration net.

TOC Cell Instantiation (Case A1)

The following is an example of how to use the TOC cell.

Note The TPOR parameter from *The Programmable Logic Data Book* is used as the WIDTH parameter in this example.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.all;
entity EX_TOC is
    port (CLOCK, ENABLE : in std_logic;
          CUP, CDOWN : out std_logic_vector (3 downto 0));
end EX_TOC;
architecture A of EX_TOC is
    signal GSR, GTS : std_logic;
    signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
    component ROC
        port (O : out std_logic);
    end component;
    component TOC
        port (O : out std_logic);
    end component;
begin
    U1 : ROC port map (O => GSR);
    U2 : TOC port map (O => GTS);
    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 when (GTS = '0' AND COUNT_UP /= "0000") else
"ZZZZ";
    CDOWN <= COUNT_DOWN when (GTS = '0') else "ZZZZ";
end A;
```

TOC Test Bench (Case A1)

The following is the test bench for the TOC example.

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

library UNISIM;
use UNISIM.all;

entity test_ofex_toc is end test_ofextoc;

architecture inside of test_ofex_toc is


Component ex_toc
Port ( CLOCK, ENABLE: in STD_LOGIC;
    CUP, CDOWN: out STD_LOGIC_VECTOR (3 downto 0));
End component;
.
.
.
Begin

UUT: ex_toc port map(. . . .);
.
.
.
End inside;
```

The best method for mapping the generic is a configuration in the test bench, as shown in the following example.

```
Configuration overall of test_ofextoc is
For inside
      For UUT:ex_toc
```

```
        For A
          For U1:TOC use entity UNISIM.TOC (TOC_V)

          Generic map (WIDTH=>52 ns);
        End for;
      End for;
End for;
End overall;
```

This configuration is for pre-NGDBuild simulation. A similar configuration is used for post-NGDBuild simulation. The ROC, TOC, and OSC4 are mapped to the WORK library, and corresponding architecture names may be different. Review the .vhd file created by NGD2VHDL for the current entity and architecture names for post-NGDBuild simulation.

TOC Model in Four Design Phases (Case A1)

The following figure shows the progression of the TOC model and its interpretation in the four main design phases.

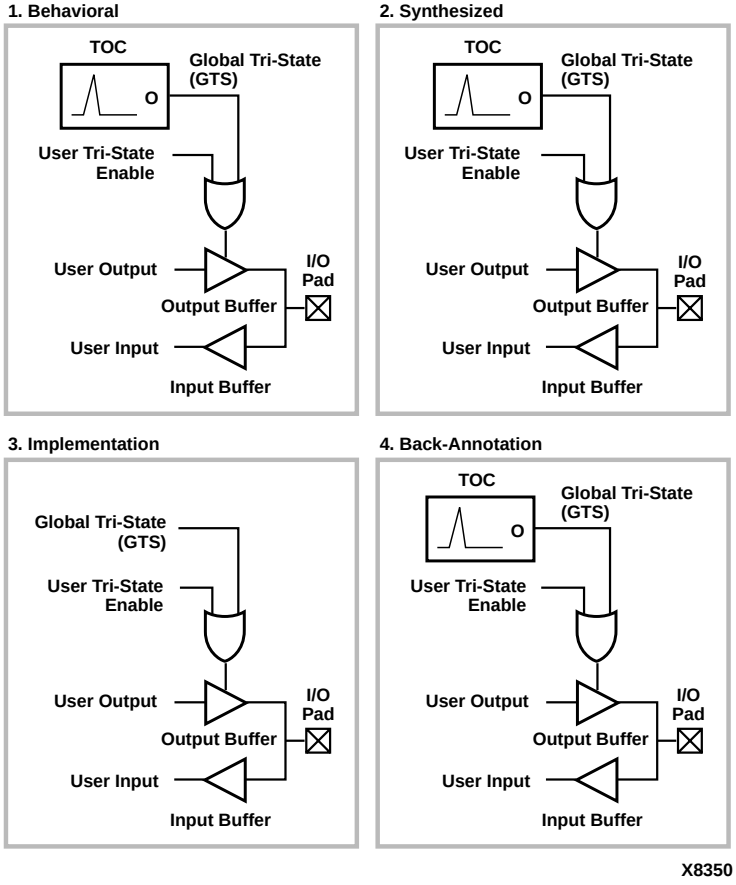

Figure 6-5 TOC Simulation and Implementation

- *Behavioral Phase*—In this phase, the behavioral or RTL description of the output buffers are inferred from the coding style. The TOC cell can be instantiated. If it is not instantiated, the GTS signal is not driven during simulation or is driven within the architecture by code that cannot be synthesized. Some synthesizers can infer which of the local output tristate enables is best for the global signal, and will insert the TOC cell automatically so instantiation may not be required unless RTL level simulation is desired. The synthesizer may also allow you to select the output tristate enable

line you want driven by the TOC cell. Instantiation of the TOC cell in the RTL description is recommended because you can immediately identify what signal is the global signal, and you are not relying on a synthesis tool feature that may not be available if ported to another tool.

- *Synthesized Phase*—In this phase, the inferred registers are mapped to a device, and the TOC instantiation is either carried from the RTL or is inserted by the synthesis tools. This results in maintaining consistent global output tristate enable behavior between the RTL and the synthesized structural descriptions during simulation.
- *Implemented Phase*—During implementation, the TOC is removed from the logical description that is placed and routed because it is a pre-existing circuit on the chip. The TOC is removed by making the input and output of the TOC cell appear as an open circuit. This allows the router to remove all nets driven by the TOC cell as if they were undriven nets. The VHDL netlist program assumes all output tristate enables are driven by the global output tristate enable so data is not lost.
- *Back-annotation Phase*—In this phase, the VHDL netlist tool re-inserts a TOC component for simulation purposes. The GTS net is a fully wired net and the TOC cell is inserted to drive it. You can use a configuration similar to the VHDL configuration for RTL simulation to set the generic for the pulse width.

Using VHDL TOCBUF (Case A2)

For Case A2, use the TOCBUF (Tristate-On-Configuration Buffer) instantiated component model. This model creates a buffer for the global output tristate enable signal. You now have an input port on the buffer to drive the global set reset line. The implementation model directs the place and route tool to remove the port so it is not implemented on the actual chip. The TOCBUF cell does not reappear in the post-routed netlist. Instead, you can select an option on the implementation tool to add a global output tristate enable port to the back-annotated netlist. A buffer is not necessary because the implementation directive is no longer required.

TOCBUF Model Example (Case A2)

The following is an example of the TOCBUF model.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.all;
entity EX_TOCBUF is
    port (CLOCK, ENABLE, SRP, STP : in std_logic;
          CUP, CDOWN : out std_logic_vector (3 downto 0));
end EX_TOCBUF;
architecture A of EX_TOCBUF is
    signal GSR, GTS : 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;
    component TOCBUF
        port (I : in std_logic;
              O : out std_logic);
    end component;
begin
    U1 : ROCBUF port map (I => SRP, O => GSR);
    U2 : TOCBUF port map (I => STP, O => GTS);
    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') the
                COUNT_DOWN <= COUNT_DOWN - "0001";
            end if;
    end process DOWN_COUNTER;
```

```
    CUP <= COUNT_UP when (GTS = '0' AND COUNT_UP /= "0000") else
"ZZZZ";
    CDOWN <= COUNT_DOWN when (GTS = '0') else "ZZZZ";
end A;
```

TOCBUF Model in Four Design Phases (Case A2)

The following figure shows the progression of the TOCBUF model and its interpretation in the four main design phases.

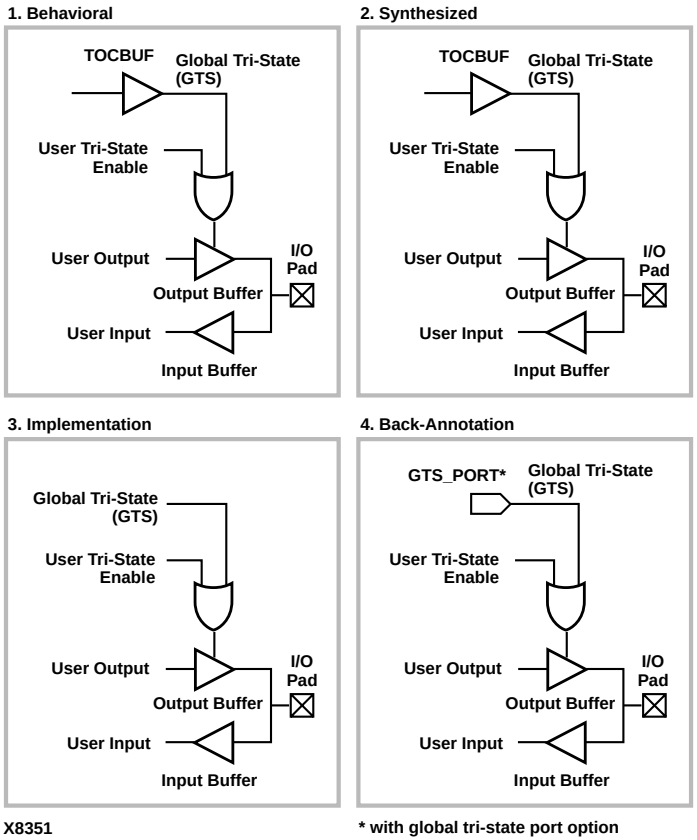

Figure 6-6 TOCBUF Simulation and Implementation

- *Behavioral Phase*—In this phase, the behavioral or RTL description of the output buffers are inferred from the coding style and may

be inserted. You can instantiate the TOCBUF cell. If it is not instantiated, the GTS signal is not driven during simulation or it is driven within the architecture by code that cannot be synthesized. Some synthesizers can infer the local output tristate enables that make the best global signals, and will insert the TOCBUF cell automatically. As a result, instantiation may not be required unless you want RTL level simulation. The synthesizer can allow you to select the output tristate enable line you want driven by the TOCBUF cell. Instantiation of the TOCBUF cell in the RTL description is recommended because you can immediately identify which signal is the global signal and you are not relying on a synthesis tool feature that may not be available if ported to another tool.

- *Synthesized Phase*—In this phase, the inferred output buffers are mapped to a device and the TOCBUF instantiation is either carried from the RTL or is inserted by the synthesis tools. This maintains consistent global output tristate enable behavior between the RTL and the synthesized structural descriptions during simulation.
- *Implemented Phase*—In this phase, the TOCBUF is removed from the logical description that is placed and routed because it is a pre-existing circuit on the chip.

  The TOCBUF is removed by making the input and output of the TOCBUF cell appear as an open circuit. This allows the router to remove all nets driven by the TOCBUF cell as if they were undriven nets. The VHDL netlist program assumes all output tristate enables are driven by the global output tristate enable so data is not lost.
- *Back-annotated Phase*—In this phase, the TOCBUF cell does not reappear in the post-routed netlist. Instead, you can select an option in the implementation tool to add a global output tristate enable port to the back-annotated netlist. A buffer is not necessary because the implementation directive is no longer required. If the option is not selected, the VHDL netlist tool re-inserts a TOCBUF component for simulation purposes. The GTS net is a fully wired net and the TOCBUF cell is inserted to drive it. You can use a configuration similar to the VHDL configuration used for RTL simulation to set the generic for the pulse width.

Using VHDL STARTBUF Block (Case B1)

The STARTBUF block can be simulated in cases where the STARTUP block causes warning messages from the simulator. The STARTUP block is normally instantiated to identify the GR, PRLD, GSR or GTS signals for implementation if the global reset or tristate is connected to a chip pin. However, this implementation directive cannot be simulated. STARTUP blocks should only be used in cases where error messages can be addressed or safely ignored.

For Case B1use the STARTBUF cell. This cell provides access to the input and output ports of the STARTUP cell that direct the implementation tool to use the global networks. The input and output port names differ from the names of the corresponding ports of the STARTUP cell. This was done for the following reasons.

- To make the STARTBUF a model that can be simulated with inputs and outputs. The STARTUP cell hangs from the net it is connected to.
- To make one model that works for all Xilinx technologies. The XC4000 and XC5200 families require different STARTUP cells because the XC5200 has a global reset (GR) net and not a GSR.

The mapping to the architecture-specific STARTUP cell from the instantiation of the STARTBUF is done during implementation. The STARTBUF pins have the suffix “IN” (input port) or “OUT” (output port). Two additional output ports, GSROUT and GTSOUT, are available to drive a signal for clearing or setting a design's registers (GSROUT), or for tristating your design's I/Os (GTSOUT).

The input ports, GSRIN and GTSIN, can be connected either directly or indirectly via combinational logic to input ports of your design. Your design's input ports appear as input pins in the implemented design. The design input port connected to the input port, GSRIN, is then referred to as the device reset port, and the design input port connected to the input port, GTSIN, is referred to as the device tristate port. Please refer to the above *“STARTBUF/STARTUP Pin Descriptions”* table, which shows the correspondence of pins between STARTBUF and STARTUP.

Note Using STARTBUF indicates that you want to access the global set/reset and/or tristate pre-routed networks available in your design's target device. As a result, you must provide the stimulus for emulating the automatic pulse as well as the user-defined set/reset.

This allows you complete control of the reset network from the test bench.

For a VHDL design example of using the global tristate network please refer to the *"Using VHDL STARBUF Block (Case 2a)"*, in the GSR Network Design Cases section.

Using VHDL STARTBUF_VIRTEX Block and STARTBUF_SPARTAN2 Block (Case B2)

The STARTUP_VIRTEX and STARTUP_SPARTAN2 blocks can be instantiated to identify the GTS signal for implementation if the global reset or tristate is connected to a chip pin. However, these cells can not be simulated as there is no simulation model for them.

The VHDL STARBUF_VIRTEX and STARBUF_SPARTAN2 blocks can do a pre-NGDBuild UniSim simulation of the GTS signal. You can also correctly back-annotate a GTS signal by instantiating a STARTUP_VIRTEX, STARTBUF_VIRTEX, STARTUP_SPARTAN2, or STARTBUF_SPARTAN2 symbol and correctly connect the GTS input signal of the component.

Note You can not simulate the GSR signal during any pre-NGDBuild Unisim VHDL simulations.You will not be able to reset the design after simulation time '0', however the design will start up in the correct state.

See the following table for Virtex and Spartan-II correspondence of pins between STARTBUF and STARTUP.

Table 6-10 Virtex/E and Spartan2 STARTBUF/STARTUP Pins

STARTBUF Pin Names	Connection Points	Virtex/E STARTUP Pin Names	Spartan-II STARTUP Pin Names
GSRIN	Global Set / Reset Port of Design	GSR	GSR
GTSIN	Global Tristate Port of Design	GTS	GTS

Table 6-10 Virtex/E and Spartan2 STARTBUF/STARTUP Pins

STARTBUF Pin Names	Connection Points	Virtex/E STARTUP Pin Names	Spartan-II STARTUP Pin Names
CLKIN	Port of Internal Logic	CLK	CLK
GTSOUT	All Output Buffers Tristate Control	N/A	N/A

STARTBUF_VIRTEX Model Example (Case B2)

The following is an example of the STARTBUF_VIRTEX model.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
library UNISIM;
use UNISIM.all;
entity EX_STARTBUF is
  port (CLOCK, ENABLE, RESET, STP : in std_logic;
    CUP, CDOWN : out std_logic_vector (3 downto 0));
end EX_STARTBUF;

architecture A of EX_STARTBUF is
  signal GTS_sig : std_logic;
  signal COUNT_UP, COUNT_DOWN : std_logic_vector (3 downto 0);
  signal ZERO : std_ulogic := '0';

component STARTBUF_VIRTEX
      port (GSRIN, GTSIN, CLKIN : in std_logic; O : out std_logic);
end component;

begin
  U1 :STARTBUF_VIRTEX port map (GTSIN=>STP,GSRIN=>ZERO, CLKIN=>ZERO
      GTSOUT=>GTS_sig);
UP_COUNTER : process (CLOCK, ENABLE, RESET)

  begin
    if (RESET = '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, RESET, COUNT_DOWN)
  begin
    if (RESET = '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 when (GTS_sig='0' AND COUNT_UP /= "0000") else
"ZZZZ";
  CDOWN <= COUNT_DOWN when (GTS_sig = '0') else "ZZZZ";
end A;
```

Simulating Special Components in VHDL

The following section provides a description and examples of using special components such as the Block RAM for Virtex.

Boundary Scan and Readback

The Boundary Scan and Readback circuitry can not be simulated at this time. Efforts are being made to create models for these components and should be available in the near future.

Differential I/O (LVDS, LVPECL)

The inputs of the differential pair are currently modeled with only the positive side. Whereas, the outputs have both pairs, positive and negative. For details, please see http://support.xilinx.com/techdocs/8187.htm.

The following is an example of Differential I/O.

```
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 a LUT

The LUT (look-up table) component is initialized for simulation by a generic mapping to the INIT attribute.

The following is an example in which a LUT is initialized.

```
entity lut_ex is
port (LUT1_IN, LUT2_IN : in std_logic_vector(1
   downto 0);
LUT1_OUT, LUT2_OUT : out std_logic_vector(1 downto
   0));
end entity lut_ex;
architecture lut_arch of lut_ex is
component LUT1
generic (INIT: std_logic_vector(1 downto 0) :=
   "10");
port (O : out std_logic;
I0 : in std_logic);
```

```
end component;
component LUT2
generic (INIT: std_logic_vector(3 downto 0) :=
   “0000”);
port (O : out std_logic;
I0, I1: in std_logic);
end component;
begin
-- LUT1 used as an inverter
U0: LUT1 generic map (INIT => “01”)
port map (O => LUT1_OUT(0), I0 => LUT1_IN(0));
-- LUT1 used as a buffer
U1: LUT1 generic map (INIT => “10”)
port map (O => LUT1_OUT(1), I0 => LUT1_IN(1));
--LUT2 used as a 2-input AND gate
U2: LUT2 generic map (INIT => “1000”)
port map (O => LUT2_OUT(0), I1 => LUT2_IN(1), I0 =>
   LUT2_IN(0));
--LUT2 used as 2-input NAND gate
U3: LUT2 generic map (INIT => “0111”)
port map (O => LUT2_OUT(1), I1 => (LUT2_IN(1), I0
   => LUT2_IN(0));
end lut_arch;
```

Simulating Virtex Block RAM

By Default the Virtex Block RAMs will come up initialized to zero in all data locations starting at time zero. For a post-NGDBuild, post-MAP, or Post-PAR (timing) simulation the Block RAMs will initialize to the value the user specifies in the UCF, or if an INIT value was given in the input design file to NGDBuild. For a pre-synthesis or post-synthesis (Pre-NGDBuild) functional simulation you must use a configuration statement to apply an initial value to the Block RAM.

The following is an example of using a configuration statement to apply an initial value to a block RAM.

```
LIBRARY ieee;
use IEEE.std_logic_1164.all;
Library UNISIM;
use UNISIM.vcomponents.all;
entity ex_blkram is
  port(CLK, EN, RST, 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));
end;

architecture struct of ex_blkram 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;

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

end struct;
```

Block RAM Testbench

The following is a Block RAM testbench coding example.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity ex_blkram_tb is
end;
architecture tb of ex_blkram_tb is
component ex_blkram
    port(CLK, EN, RST, WE : in std_logic;
      DI : in std_logic_vector(15 downto 0);
      ADDR : in std_logic_vector(11 downto 0);
```

```
                DORAMB4_S4 : out std_logic_vector(3 downto
   0));
end component;
constant CLK_PERIOD : time := 100 ns;
signal CLK_TB, EN_TB, RST_TB, WE_TB, TRST_TB,
std_logic;
signal DI_TB : std_logic_vector(15 downto 0);
signal ADDR_TB : std_logic_vector(11 downto 0);
signal DORAMB4_S4_TB : std_logic_vector(3 downto
   0);
begin
uut : ex_blkram port map (....);
.
.
.
end tb;

configuration cfg_ex_blkram_tb of ex_blkram_tb is
    for tb
    end for;
end cfg_ex_blkram_tb;
```

The best method for mapping the generic block RAM is in a configuration. The configuration can be all in the test bench or as shown in these examples. The configuration is in the following separate configuration file. This can all be contained in the testbench. If it is in a separate file then it must be compiled last. Run the simulation on the configuration in order for the generic mapping to apply.

Block RAM Configuration

The following is a block RAM configuration example.

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

configuration cfg_ex_blkram_tb of ex_blkram_tb is
for tb
    for uut : ex_blkram use entity work.ex_blkram(struct);
      for struct
```

```
        for INST_RAMB4_S4 : RAMB4_S4 use entity
unisim.RAMB4_S4(RAMB4_S4_V)
generic map (INIT_00 =>
X"1F1E1D1C1B1A19181716151413121110 0F0E0D0C0B0A09080706050403020100
",
    INIT_01 =>
X"3F3E3D3C3B3A39383736353433231302F2E2D2C2B2A29282726252423222120
",
    INIT_02=>
X"5F5E5D5C5B5A59585756555453525150 4F4E4D4C4B4A49484746454443424140
",
.
.
.

    INIT_08=>
X"1F1E1D1C1B1A19181716151413121110 0F0E0D0C0B0A09080706050403020100
",
    INIT_09=>
X"3F3E3D3C3B3A39383736353433231302F2E2D2C2B2A29282726252423222120
",
    INIT_0A=>
X"5F5E5D5C5B5A59585756555453525150 4F4E4D4C4B4A49484746454443424140
",
.

.

.

    INIT_0F=>
X"FFFEFDFCFBFAF9F8F7F6F5F4F3F2F1F0 EFEEEDECEBEAE9E8E7E6E5E4E3E2E1E0
");
          end for;
        end for;
      end for;
    end for;
end cfg_ex_blkram_tb;
```

Simulating the Virtex Clock DLL

When Functionally simulating the Virtex Clock DLL generic maps will have to be used when specifying the Clock Divide and Duty Correction values. All other aspects of the CLKDLL will be simulated properly. By default the Clock Divide is set to 2 and Duty Correction is set to true. This example will set the Clock Divide to 4, and set the Duty Correction to False. In the testbench the clock is set to a non-50% duty cycle to see Duty correction in the simulation if the generic is not set to false.

Note You must use a UCF file to pass the Clock Divide and Duty Correction values to the Xilinx implementation tools. Depending on the synthesis tool INIT attributes may be used too.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
Library UNISIM;
use UNISIM.vcomponents.all;
entity clkdlls is
   port(CLK_LF, RST_LF : in std_logic;
     CLK90_LF, CLK180_LF : out std_logic;
     CLK270_LF, CLK2X_LF : out std_logic;
     CLKDV_LF, LOCKED_LF : out std_logic;
     LFCount   : out std_logic_vector(3 downto 0));
end;
architecture struct of clkdlls is
component CLKDLL
   port (CLKIN   : in std_logic;
     CLKFB   : in std_logic;
     RST     : in std_logic;
     CLK0    : out std_logic;
     CLK90   : out std_logic;
     CLK180  : out std_logic;
     CLK270  : out std_logic;
     CLK2X   : out std_logic;
     CLKDV   : out std_logic;
     LOCKED  : out std_logic);
end component;
component 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;
signal COUNT: integer range 0 to 15 := 0;
signal sigCLK_LF, sigCLK0_LF, sigCLKFB_LF, CLK0_LF
   : std_logic;
signal sigLFCount : std_logic_vector (3 downto 0);

begin
INST_IBUFGLF : IBUFG port map (I => CLK_LF, O =>
   sigCLK_LF);
INST_BUFGLF : BUFG port map (I => sigCLK0_LF, O =>
   sigCLKFB_LF);
INST_CLKDLL : CLKDLL port map (CLKIN => sigCLK_LF,
   CLKFB => sigCLKFB_LF,
RST   => RST_LF, CLK0  => sigCLK0_LF, CLK90 =>
   CLK90_LF,
CLK180 => CLK180_LF, CLK270 => CLK270_LF, CLK2X
   =>CLK2X_LF,
CLKDV => CLKDV_LF, LOCKED => LOCKED_LF);
CLK0_LF <= sigCLK0_LF;

procCLKDLLCount: process (CLK0_LF)

begin
   if (CLK0_LF'event and CLK0_LF = '1') then
     sigLFCount <= sigLFCount + "0001";
   end if;
LFCount <= sigLFCount;
end process;
end struct;
```

Clock DLL Testbench

The following is an example of a clock DLL testbench.

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

```
entity clkdlls_tb is
end;
architecture tb of clkdlls_tb is
component clkdlls
   port(CLK_LF, RST_LF : in std_logic;
     CLK90_LF, CLK180_LF : out std_logic;
     CLK270_LF, CLK2X_LF : out std_logic;
     CLKDV_LF, LOCKED_LF : out std_logic);
end component;
signal CLK_LF_TB, RST_LF_TB : std_logic;
signal CLK90_LF_TB, CLK180_LF_TB, CLK270_LF_TB :
   std_logic;
signal CLK2X_LF_TB, CLKDV_LF_TB, LOCKED_LF_TB :
   std_logic;
constant CLK_PERIOD_LF : time :=40 ns;
begin
-- Define CLKLF
   procCLKLF : process (CLK_LF_TB)
     begin
      if (CLK_LF_TB = '1') then
         CLK_LF_TB <= '0' after CLK_PERIOD_LF*1/4;
      elsif (CLK_LF_TB = '0') then
         CLK_LF_TB <= '1' after CLK_PERIOD_LF*3/4;
      else
         CLK_LF_TB <= '1' after CLK_PERIOD_LF*1/4;
      end if;
   end process;


     -- Define RST_LF_TB
RST_LF_TB <= '1',
  '0' after 400 ns,
  '1' after 800 ns,
  '0' after 1200 ns;


uut_clkdlls : clkdlls port map (
   CLK_LF    => CLK_LF_TB, RST_LF    => RST_LF_TB,
   CLK90_LF  => CLK90_LF_TB, CLK180_LF =>
   CLK180_LF_TB,
```

```
   CLK270_LF => CLK270_LF_TB, CLK2X_LF  =>
   CLK2X_LF_TB,
   CLKDV_LF  => CLKDV_LF_TB, LOCKED_LF =>
   LOCKED_LF_TB);
end tb;
configuration cfg_clkdlls_tb of clkdlls_tb is
   for tb
   end for;
end cfg_clkdlls_tb;
```

Clock DLL Configuration

The configuration will set the clock divide to 4 and turn off duty cycle correction. To see duty cycle correction in simulation simply don't set the generic, as in the commented line.

```
library IEEE;
use IEEE.std_logic_1164.all;
library UNISIM;
use UNISIM.vcomponents.all;
configuration cfg_clkdlls_tb of clkdlls_tb is
for tb
for uut_clkdlls : clkdlls use entity
   work.clkdlls(struct);
for struct
for all : clkdll use entity unisim.clkdll(clkdll_v)
generic map (DUTY_CYCLE_CORRECTION => FALSE,
CLKDV_DIVIDE => 4.0);
--  generic map (CLKDV_DIVIDE => 4.0);
end for;
end for;
end for;
end for;
end cfg_clkdlls_tb;
```

Using Oscillators

Oscillator output can vary within a fixed range. This cell is not included in the SimPrim library because you cannot drive global signals in VHDL designs. Schematic simulators can define and drive global nets so the cell is not required. Verilog has the ability to drive nets within a lower level module as well. Therefore the oscillator cells are only required in VHDL. After back-annotation, their entity and

architectures are contained in your design's VHDL output. For functional simulation, they can be instantiated and simulated with the UniSim Library.

The period of the base frequency must be set in order for the simulation to proceed, since the default period of 0 ns disables the oscillator. The oscillator's frequency can vary significantly with process and temperature.

Before you set the base period parameter, consult *The Programmable Logic Data Book* for the part you are using. For example, the section in *The Programmable Logic Data Book* for the XC4000 Series On-Chip Oscillator states that the base frequency can vary from 4MHz to 10 MHz, and is nominally 8 MHz. This means that the base period generic "period_8m" in the XC4000E OSC4 VHDL model can range from 250ns to 100ns. An example of this follows.

Oscillator VHDL Example

The following is an example of an Oscillator VHDL component.

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

library UNISIM;
use UNISIM.all;

entity test1 is
port (DATAIN: in STD_LOGIC;
DATAOUT: out STD_LOGIC);
end test1;

architecture inside of test1 is

signal RST: STD_LOGIC;

component ROC
port(O: out STD_LOGIC);
end component;

component OSC4
port(F8M: out STD_LOGIC);
end component;
```

```
signal internalclock: STD_LOGIC;
begin
U0: ROC port map (RST);

U1: OSC4 port map (F8M=>internalclock);

process(internalclock)
begin
if (RST='1') then
DATAOUT <= '0';

elsif(internalclock'event and internalclock='1')
   then
DATAOUT <= DATAIN;

end if;

end process;

end inside;
```

Oscillator Test Bench

The following is an example of an Oscillator test bench.

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

library UNISIM;
use UNISIM.all;

entity test_oftest1 is end test_oftest1;

architecture inside of test_oftest1 is

component test1
port(DATAIN: in STD_LOGIC;
DATAOUT: out STD_LOGIC);
end component;

signal userdata, userout: STD_LOGIC;
```

```
begin

UUT: test1 port
   map(DATAIN=>userdata,DATAOUT=>userout);

myinput: process
begin
userdata <= '1';
wait for 299 ns;
userdata <= '0';
wait for 501 ns;
end process;

end inside;

configuration overall of test_oftest1 is
for inside
        for UUT:test1
              for inside
                  for U0:ROC use entity
   UNISIM.ROC(ROC_V)
                  generic map (WIDTH=> 52 ns);
                  end for;

                  for U1:OSC4 use entity
   UNISIM.OSC4(OSC4_V)
                  generic map (PERIOD_8M=> 25 ns);
                  end for;
              end for;
        end for;
end for;
end overall;
```

This configuration is for pre-NGDBuild simulation. A similar configuration is used for post-NGDBuild simulation. The ROC, TOC, and OSC4 are mapped to the WORK library, and corresponding architecture names may be different. Review the .vhd file created by NGD2VHDL for the current entity and architecture names for post-NGDBuild simulation.

Simulating Verilog

Defining Global Signals in Verilog

To specify the global set/reset or global reset, you must first define them in the $XILINX/verilog/src/glbl.v module. The VHDL UniSims library contains the ROC, ROCBUF, TOC, TOCBUF, and STARTBUF cells to assist in VITAL VHDL simulation of the global set/reset and tristate 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.

The following is the definition of the glbl.v file.

```
`timescale 1 ns / 1 ps
module glbl();
wire GR;
wire GSR;
wire GTS;
wire PRLD;
endmodule
```

Defining GSR/GTS in a Test Bench

There are two cases to consider when defining a GSR or GTS in a test bench: designs without a STARTUP block and designs with a STARTUP block.

Note The terms “test bench” and “test fixture” are used synonymously throughout this manual.

Designs Without a Startup Block

When you use the UniSim libraries for RTL simulation, you must set the value of the appropriate Verilog global signals (glbl.GSR or

glbl.GTS) to the name of the GSR or GTS net, qualified by the appropriate scope identifiers.

The global set/reset net is present in your implemented design even if you do not instantiate the STARTUP block in your design. The function of STARTUP is to give you the option to control the global reset net from an external pin. The following example should be added to your design code and test fixture to set the GSR and GTS pin for XC4000XLA, Spartan/XL, or Virtex devices:

```
reg GSR;
assign glbl.GSR = GSR;
reg GTS;
assign glbl.GTS = GTS;
initial begin
GSR = 1; GTS = 1;
#100 GSR = 0; GTS = 0;
end
```

Example 1: No STARTUP With GSR Defined

The following design shows how to drive the GSR signal in a testfixture file at the beginning of a pre-NGDBuild Unified Library functional simulation.

In the design code, declare the GSR as a Verilog wire. The GSR will not be specified in the port list for the module. Describe the GSR to reset or set every inferred register or latch in your design. GSR does not need to be connected to any instantiated registers or latches, as shown in the following 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 the 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.

In the test fixture file, set GSR to test.uut.GSR (the name of the global set / reset signal, qualified by the name of the design instantiation instance name and the test fixture instance name). Since there is no STARTUP block, a connection to GSR is made in the testfixture via an assign statement. See the following example:

```
`timescale 1 ns / 1 ps
module test;
reg CLK, D;
wire Q;
wire [3:0] COUT;
reg GSR;
assign glbl.GSR = GSR;
assign test.uut.GSR = GSR;
my_counter uut (.CLK (CLK), .D (D), .Q (Q), .COUT
   (COUT));
initial begin
  $timeformat(-9,1,"ns",12);
  $display("\t   T C G D Q C");
  $display("\t   i L S     O");
  $display("\t   m K R     U");
  $display("\t   e         T");
  $monitor("%t %b %b %b %b %h", $time, CLK, GSR, D,
   Q, COUT);
end
initial begin
    CLK = 0;
    forever #25 CLK = ~CLK;
end
```

```
initial begin
    #0 {GSR, D} = 2'b11;
    #100 {GSR, D} = 2'b10;
    #100 {GSR, D} = 2'b00;
    #100 {GSR, D} = 2'b01;
    #100 $finish;
end
endmodule
```

Designs with a STARTUP Block

For RTL simulation using the UniSim libraries, asserting global set/reset and global tri-state when the STARTUP block is specified in the design is similar to asserting global set/reset and global tristate without a STARTUP block in the design. See the "User-Controlled GSR" figure.

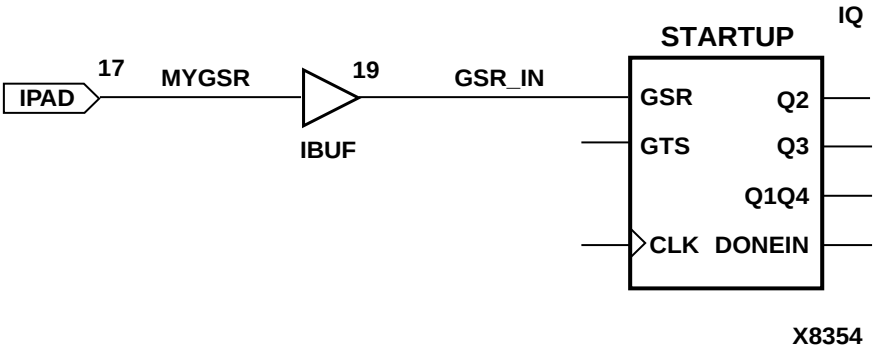

Figure 6-7 User-Controlled GSR

To set the GSR pin to set an external input port, the testfixture would be written as the following:

```
reg MYGSR;
initial begin
MYGSR = 1;
#100 MYGSR = 0;
end
```

You must omit the assign statement for the global signal. This is because a the global signal, glbl.GSR, is defined within the STARTUP block to make the connection between the user logic and the global GSR net embedded in the UniSim models for RTL simulation. For post-NGDBuild, GSR is connected in the netlist created by NGD2VER. Retaining the assign definition causes a possible conflict with these connections.

Example 1: STARTUP with GSR Pin Connector

In the following Verilog code, GSR is listed as a top-level port. Synthesis sees a connection of GSR to the STARTUP and as well to the behaviorally described counter. Although this is correct in the hardware, it is actually an implicit connection, and GSR is only listed as a connection to the STARTUP in the implementation netlist.

```
module my_counter (MYGSR, CLK, D, Q, COUT);
input MYGSR, CLK, D;
output Q;
output [3:0] COUT;

reg [3:0] COUT;

always @(posedge MYGSR or posedge CLK)
  begin
    if (MYGSR == 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));
STARTUP U1 (.GSR (MYGSR), .GTS (1'b0), .CLK
   (1'b0));
endmodule
```

The following is an example of controlling the global set/reset signal by driving the external MYGSR input port in a test fixture file at the

beginning of an RTL or post-synthesis functional simulation when there is a STARTUP block.

The global set/reset control signal should be toggled High, then Low in an initial block.

```
reg MYGSR;
initial begin
MYGSR = 1; // To reset/set the device
#100 MYGSR = 0; // To deactivate GSR
end
```

In addition, the global signal, glbl.GSR, is defined within the STARTUP block to make the connection between the user logic and the global GSR net embedded in the UniSim models for RTL simulation. For post-NGDBuild functional simulation, post-Map timing simulation, and post-route timing simulation, GSR is connected in the Verilog netlist that is created by NGD2VER.

Example 2: STARTUP with GTS Pin Connected

In the following figure, MYGTS is an external user signal that controls GTS.

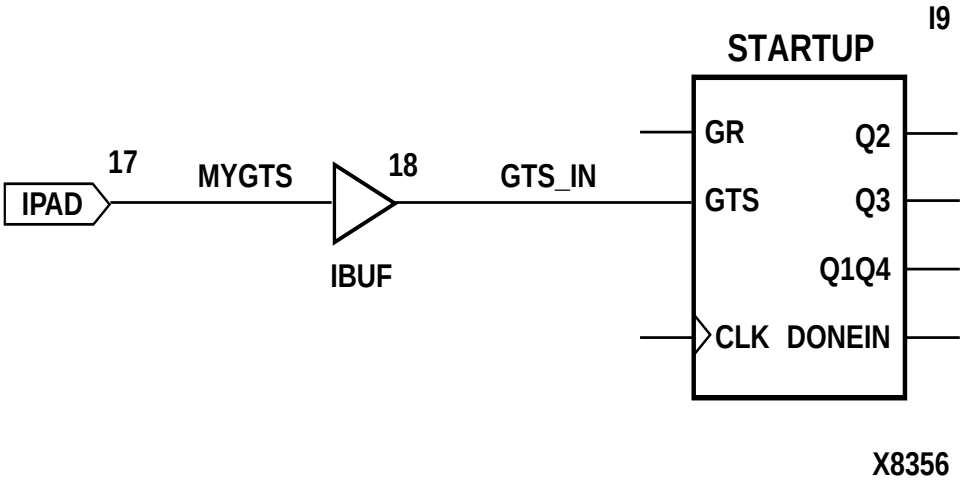

Figure 6-8 User-Controlled GTS

The following is an example of controlling the global tristate signal by driving the external MYGTS input port in a test fixture file at the beginning of an RTL or post-synthesis functional simulation when there is a STARTUP block. The global GTS model in the UniSim simulation models for output buffers (OBUF, OBUFT, and so on).

The global tristate control signal should be toggled High, then Low in an initial block.

```
reg MYGTS;

initial begin
MYGTS = 1; // To tristate the device;
#100 MYGTS = 0;     // To deactivate GTS
end
```

Example 3: STARTUP with GTS Pin Not Connected

A Verilog global signal called glbl.GTS is defined within the STARTUP / STARTUP_VIRTEX block to make the connection between the user logic and the global GTS net embedded in the Unified models. For post-NGDBuild functional simulation, post-map timing

simulation, and post-route timing simulation, glbl.GTS is defined in the Verilog netlist that is created by NGD2VER.

```
reg GTS;
assign glbl.GTS = GTS;

initial begin
GTS = 1; // To tristate the device;
#100 GTS = 0;// To deactivate GTS
end
```

Simulating Special Components in Verilog

The following section provides a description and examples of simulating special components for Virtex.

Boundary Scan and Readback

The Boundary Scan and Readback circuitry can not be simulated at this time. Efforts are being made to create models for these components and should be available in the near future.

Differential I/O (LVDS, LVPECL)

The inputs of the differential pair are currently modeled with only the positive side. Whereas, the outputs have both pairs, positive and negative. For details, please see http://support.xilinx.com/techdocs/8187.htm.

The following is an example of Differential I/O.

```
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
```

LUT

For simulation, the INIT attribute passed by the defparam statement is used to initialize contents of the LUT.

The following is an example of the defparam statement being used to initialize the contents of a LUT.

```
module lut_ex (LUT1_OUT, LUT1_IN);
input  [1:0] LUT1_IN;
output [1:0] LUT1_OUT;

// For RTL simulation only.
// The defparam will not synthesize.

// synopsys translate_off
defparam U0.INIT = 2'b01;
defparam U1.INIT = 2'b10;
// synopsys translate_on

// LUT1 used as an inverter
LUT1 U0 (.O (LUT1_OUT[0]), .I0 (LUT1_IN[0]));

// LUT1 used as a buffer
LUT1 U1 (.O (LUT1_OUT[1]), .I0 (LUT1_IN[1]));

endmodule
```

However, passing the INIT attribute in this manner does not initialize the contents for synthesis. All synthesis tools have their own mechanism for passing attributes to the implementation netlist. For referneces on today's popular synthesis tools, refer to the *LUT Instantiation and Initialization for Synthesis* table.

Table 6-11 LUT Instantiation and Initialization for Synthesis

Synthesizer	
FPGA Express	http://support.xilinx.com/techdocs/5334.htm
Synplify	http://support.xilinx.com/techdocs/1992.htm
Leonardo Spectrum	http://support.xilinx.com/techdocs/8207.htm

SRL16

For simulation, the INIT attribute passed by the defparam statement is used to initialize contents of the SRL16.

The following is an example of the defparam statement being used to initialize the contents of a SRL16.

```
module srl16_ex (CLK, DIN, QOUT);
input CLK, DIN;
output QOUT;

// For RTL simulation only.
// The defparam will not synthesize.

// synopsys translate_off
defparam U0.INIT = 16'hAAAA;
// synopsys translate_on

// Static length - 16-bit SRL
SRL16 U0 (.D (DIN), .Q (QOUT), .CLK (CLK),
  .A0 (1'b1), .A1 (1'b1), .A2 (1'b1), .A3 (1'b1));
endmodule
```

However, passing the INIT attribute in this manner does not initialize the contents for synthesis. Please refer to your synthesis vendor's documentation since all synthesis tools have their own mechanism for passing attributes to the implementation netlist.

BlockRAM

For simulation, the INIT_0x attributes passed by the defparam statement are used to initialize contents of the BlockRAM.

```
module bram512x4 (CLK, DATA_BUSA, ADDRA, WEA,
   DATA_BUSB, ADDRB, WEB);
input [9:0] ADDRA, ADDRB;
input CLK, WEA, WEB;
inout [3:0] DATA_BUSA, DATA_BUSB;

wire [3:0] DOA, DOB;

assign DATA_BUSA = !WEA ? DOA : 4'hz;
assign DATA_BUSB = !WEB ? DOB : 4'hz;
```

```
// For RTL simulation only. The defparam will not
   synthesize.

// synopsys translate_off
defparam
U0.INIT_00 =
   256'h5555aaaa5555aaaa5555aaaa5555aaaa5555aaaa,
U0.INIT_01 =
   256'h5555aaaa5555aaaa5555aaaa5555aaaa5555aaaa;
// synopsys translate_on

RAMB4_S4_S4 U0 (.DOA (DOA), .DOB (DOB),
  .ADDRA (ADDRA), .DIA (DATA_BUSA), .ENA (1'b1),
  .CLKA (CLK), .WEA (WEA), .RSTA (1'b0),
  .ADDRB (ADDRB), .DIB (DATA_BUSB), .ENB (1'b1),
  .CLKB (CLK), .WEB (WEB), .RSTB (1'b0));
endmodule
```

However, passing the INIT_0x attributes in this manner does not initialize the memory contents for synthesis since all synthesis tools have their own mechanism for passing attributes to the implementation netlist. For references on today's synthesis tools, refer to the *BlockRAM Instantiation and Initialization for Synthesis* table.

Table 6-12 BlockRAM Instantiation and Initialization for Synthesis

Synthesizer	
FPGA Express	http://support.xilinx.com/techdocs/4392.htm
Synplify	http://support.xilinx.com/techdocs/2022.htm
Leonardo Spectrum	http://support.xilinx.com/techdocs/7947.htm

Another method for passing the INIT_0x attributes to the Alliance tools is through the use of a UCF file. For example, the following statement defines the initialization string for the code example above.

```
INST U0 INIT_00 =
   5555aaaa5555aaaa5555aaaa5555aaaa5555aaaa;
INST U0 INIT_01 =
   5555aaaa5555aaaa5555aaaa5555aaaa5555aaaa;
```

The value of the INIT_0x string is a hexadecimal number that defines the initialization string.

CLKDLL

The duty cycle of the CLK0 output is 50-50 unless DUTY_CYCLE_CORRECTION attribute is set to FALSE, in which case the duty cycle is the same as that of the CLKIN.

The frequency of CLKDV is determined by the value assigned to the CLKDV_DIVIDE attribute. The default is 2.

The STARTUP_WAIT is not implemented in the model. Monitor the LOCK signal and use it to trigger the release of the GSR signal.

```
module clkdll_ex (CLKIN_P, RST_P, CLK0_P, CLK90_P,
   CLK180_P,
CLK270_P, CLK2X_P, CLKDV_P, LOCKED_P);
input CLKIN_P, RST_P;
output  CLK0_P, CLK90_P, CLK180_P, CLK270_P,
   CLK2X_P;
output  CLKDV_P;
// Active high indication that DLL is LOCKED to
   CLKIN
output  LOCKED_P;

wire CLKIN, CLK0;

// Input buffer on the clock
IBUFG U0 (.I (CLKIN_P), .O (CLKIN));

// GLOBAL CLOCK BUFFER on the delay compensated
   output
BUFG U2 (.I (CLK0), .O (CLK0_P));

// For RTL simulation only. The defparam will not
   synthesize.
// synopsys translate_off
// CLK0 divided by 1.5 2.0 2.5 3.0 4.0 5.0 8.0 or
   16.0
defparam DLL0.CLKDV_DIVIDE = 4.0;
defparam DLL0.DUTY_CYCLE_CORRECTION = "FALSE";
// synopsys translate_on
```

```
// Instantiate the DLL primitive cell
CLKDLL DLL0 (.CLKIN (CLKIN), .CLKFB(CLK0_P), .RST
   (RST_P),
  .CLK0 (CLK0), .CLK90 (CLK90_P), .CLK180
   (CLK180_P),
 .CLK270 (CLK270_P), .CLK2X (CLK2X_P), .CLKDV
   (CLKDV_P),
  .LOCKED (LOCKED_P));
endmodule
```

However, passing the CLKDLL attributes in this manner does not initialize the contents for synthesis. Please refer to your synthesis vendor's documentation since all synthesis tools have their own mechanism for passing attributes to the implementation netlist.

Another method for passing the CLKDLL attributes to the Alliance tools is through the use of an UCF file. For example, the following statement defines the initialization string for the code example above.

```
INST DLL0 CLKDV_DIVIDE = 4;
INST DLL0 DUTY_CYCLE_CORRECTION = FALSE;
```

Running Simulation

When simulating, 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 *test* to the main module in the test fixture file. This name is consistent with the name of the test fixture module that is written later in the design flow by NGD2VER during post-NGDBuild, post-MAP, or post-route simulation. If this naming consistency is maintained, you can use the same test fixture file for simulation at all stages of the design flow with minimal modification.

ModelSim Vcom

The following is information regarding ModelSim Vcom.

Using Shared Pre-Compiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using VCS / VCSi. See the *"Compiling HDL Libraries"* section for instruction on how to compile the Xilinx Verilog libraries.

Depending on the makeup of the design (LogiBLOX, Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the command-line

vcom -work *work_macro1.vhd logiblox_macro.vhd top_level.vhd testbench.vhd testbench_cfg.vhd*

For timing simulation or post-Ngd2vhdl, the Simprims-based libraries are used. Specify the following at the command-line:

vcom -work *work_design.vhd testbench.vhd*

VSS

The following is information regarding VSS.

Using Shared Pre-Compiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using VSS vhdlan. See the *"Compiling HDL Libraries"* section for instruction on how to compile the Xilinx VHDL libraries.

Depending on the makeup of the design (LogiBLOX, Xilinx instantiated components, or CORE Generator components), for RTL simulation, specify the following at the command-line.

```
vhdlan -i macro1.vhd
vhdlan -i logiblox_macro.vhd
vhdlan -i top_level.vhd
vhdlan -i testbench.vhd
vhdlan -i testbench_cfg.vhd
```

For timing simulation or post-Ngd2vhdl, the Simprims-based libraries are used. Specify the following at the command-line.

```
vhdlan -i design.vhd
vhdlan -i testbench.vhd
```

Note Make sure the WORK directory is created and that the .synopsys_vss.setup file exists and points to this directory.

For more information and a tutorial on running the VSS simulator, go to Synopsys tutorial at http://support.xilinx.com/support/techsup/tutorials.

Verilog-XL

Depending on the makeup of the design (LogiBLOX, Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the command-line.

```
verilog -y $XILINX/verilog/src/unisims -y $XILINX/verilog/src/simprims
+incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v
<testfixture>.v <design>.v
```

The –y switch points the simulator to the HDL models.

For timing simulation or post-NGD2VER, the Simprims-based libraries are used. Specify the following at the command-line:

```
verilog -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \
+libext+.v <testfixture>.v <design>.v
```

For more information on specifying Xilinx SimPrims library using the -ul switch with NGD2VER instead of using the -y switch in Verilog-XL, go to http://support.xilinx.com/techdocs/3167.htm.

Note You do not need to compile the libraries for Verilog-XL because it uses an interpretive compilation of the libraries.

NC-Verilog

There are two methods to run simulation with NC-Verilog.

1. Using library source files with compile time options (similar to Verilog-XL).
2. Using shared pre-compiled libraries.

Using Library Source Files With Compile Time Options

Depending on the makeup of the design (LogiBLOX, Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the command-line:

```
ncxlmode +libext+.v -y $XILINX/verilog/src/unisims -y $XILINX/verilog/
src/simprims +incdir+$XILINX/verilog/src $XILINX/verilog/src/glbl.v
```

```
<testfixture>.v <design>.v
```

For timing simulation or post-NGD2VER, the Simprims-based libraries are used. Specify the following at the command-line.

```
ncxlmode -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v
+libext+.v <testfixture>.v time_sim.v
```

Using Shared Pre-Compiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using NC-Verilog. See the *"Compiling HDL Libraries"* section for instruction on how to compile the Xilinx Verilog libraries.

Depending on the makeup of the design (LogiBLOX, 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, <compiled_lib_dir>/simprims_ver =>
xilinxcorelib_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 simulation or post-Ngd2ver, the Simprims-based libraries are used. 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
```

For more information on how to back-annotate the SDF file for timing simulation, go to http://support.xilinx.com/techdocs/947.htm.

VCS/VCSi

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 the exact same simulation results. VCSi is invoked using the **`vcsi`** command rather than the **`vcs`**. command

There are two methods to run simulation with VCS/VCSi.

1. Using library source files with compile time options (similar to Verilog-XL).
2. Using shared pre-compiled libraries.

Using Library Source Files With Compile Time Options

Depending on the makeup of the design (LogiBLOX, Xilinx instantiated primitives, or CORE Generator components), for RTL simulation, specify the following at the command-line.

```
vcs -y $XILINX/verilog/src/unisims -y $XILINX/verilog/src/simprims
incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v
-Mupdate -R <testfixture>.v <design>.v
```

For timing simulation or post-NGD2VER, the Simprims-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 .

The -Mupdate option enables incremental compilation. Modules will be recompiled because of one of the following reasons:

1. Target of a hierarchical reference has changed.
2. Some compile time constant such as a parameter has changed.
3. Ports of a module instantiated in the module has changed.

4. Module inlining. For example, merging, internally in VCS, of a group of module definitions into a larger module definition which leads to faster simulation. These affected modules are again recompiled. This is done only once.

For more information on how to back-annotate the SDF file for timing simulation, go to http://support.xilinx.com/techdocs/6349.htm.

Using Shared Pre-Compiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before using VCS/VCSi. See the *"Compiling HDL Libraries"* section for instruction on how to compile the Xilinx Verilog libraries.

Depending on the makeup of the design (LogiBLOX, 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 $XILINX/verilog/src/simprims
-Mlib=<compiled_dir>/xilinxcorelib_ver +incdir+$XILINX/verilog/src
+libext+.v $XILINX/verilog/src/glbl.v -R <testfixture>.v <design>.v
```

For timing simulation or post-NGD2VER, the Simprims-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 -Mupdate option enables incremental compilation. Modules will be recompiled because of one of the following reasons:

1. Target of a hierarchical reference has changed.
2. Some compile time constant such as a parameter has changed.
3. Ports of a module instantiated in the module has changed.
4. Module inlining. For example, merging, internally in VCS, of a group of module definitions into a larger module definition which leads to faster simulation. These affected modules are again recompiled. This is done only once.

For more information on how to back-annotate the SDF file for timing simulation, go to http://support.xilinx.com/techdocs/6349.htm.

ModelSim Vlog

There are two methods to run simulation with ModelSim Vlog.

1. Using library source files with compile time options (similar to Verilog-XL).
2. Using shared pre-compiled libraries.

Using Library Source Files With Compile Time Options

Depending on the makeup of the design (LogiBLOX, 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 -y $XILINX/verilog/src/simprims
+incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v -incr

<testfixture>.v <design>.v

vsim <testfixture> glbl
```

For timing simulation or post-NGD2VER, the Simprims-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 Pre-Compiled Libraries

Simulation Libraries have to be compiled to *compiled_lib_dir* before usingModelSim Vlog. See the *"Compiling HDL Libraries"* section for instruction on how to compile the Xilinx Verilog libraries.

Depending on the makeup of the design (LogiBLOX, 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 simulation or post-NGD2VER, the Simprims-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.

LMG SmartModels

The Synopsys Logic Modeling Group (LMG) distributes SWIFT SmartModels for a large number of Xilinx devices. Instead of simulating devices at the gate level, SmartModels represent the FPGA as "black boxes" that accept input stimulus and respond with appropriate output behavior. Such behavioral models are well suited for distribution in object code form because they provide improved performance over gate-level models.

IBIS

The Xilinx IBIS models provide information on I/O characteristics. The IBIS models can be used for the following.

1. To model best case and worst case models by using the min, max current w/the proper min, max ramp rates.
2. To model SSO (Simultaneous Switching Output). These are mainly the package inductance, other associated parasitics and the number of buffers switching. IBIS specifies R, L and C in matrix format and t he use of a matrix for the inductance accounts for the "loop"inductance i.e. the mutuals between the pins. Specifying the mutual inductance is necessary to account for SSO event simulation.
3. To (v2.1) model RTC (Rise Time Controlled), GTO (Gradual Turn on) or Slew rate controlled outputs. The se are defined under [Rising Waveform] and [Falling Waveform] keywords.

4. To model ground bounce. IBIS contains the package parasitic information necessary to simulate ground bounce. Even though the data is available within the model file, not all simulators may be able to use it to simulate ground bounce. Refer to your respective simulator for support.

The Xilinx IBIS models are available for download at:

ftp://ftp.xilinx.com/pub/swhelp/ibis/

STAMP

The Xilinx 3.1i development system supports Stamp Model Generation. This feature supports the use of board level Static Timing Analysis tools, such as Mentor Graphics' Tau and Viewlogic's Blast. With these tools, users of Xilinx programmable logic products can accelerate board level design verification.

Using the -stamp switch in the Xilinx program Trace, will write out the stamp models.

For more information on creating the STAMP files, options to use in Trace, and integrating it with Tau and Blast, please see the Application note at http://support.xilinx.com/xapp/xapp166.pdf