Synopsys Synthesis and Simulation Design Guide

Getting Started

HDL Coding Hints

Understanding High-Density Design Flow

Designing FPGAs with HDL

Simulating Your Design

Accelerate FPGA Macros with One-Hot Approach

Synopsys Synthesis and Simulation Design Guide — 2.1i *Printed in U.S.A.*

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 the Synopsys FPGA Compiler; and targeted for XC4000, Spartan, and XC5200 devices. However, most of the design methodologies apply to other synthesis tools, as well as to other Xilinx FPGAs. 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. If you use Synopsys tools, refer to the Synopsys 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.*

Additional Resources

For additional information, go to http://support.xilinx.com. The following table lists some of the resources you can access from this

page. You can also directly access some of these resources using the provided URLs.

Resource	Description/URL
Tutorial	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://www.support.xilinx.com/apps/appsweb.htm
Data Book	Pages from *The Programmable Logic Data Book*, which describe device-specific information on Xilinx device characteristics, including read-back, boundary scan, configuration, length count, and debugging http://www.support.xilinx.com/partinfo/databook.htm
Xcell Journals	Quarterly journals for Xilinx programmable logic users http://www.support.xilinx.com/xcell/xcell.htm
Tech Tips	Latest news, design tips, and patch information on the Xilinx design environment http://www.support.xilinx.com/support/techsup/journals/index.htm

Manual Contents

This manual covers the following topics.

- Chapter 1, “Getting Started,” provides a general overview of designing Field Programmable Gate Arrays (FPGAs) with HDLs. This chapter also includes installation requirements and instructions.
- Chapter 2, “HDL Coding Hints,” includes HDL coding hints and design examples to help you develop an efficient coding style.
- Chapter 3, “Understanding High-Density Design Flow,” provides synthesis and Xilinx implementation techniques to increase design performance and utilization.
- Chapter 4, “Designing FPGAs with HDL,” includes coding techniques to help you improve synthesis results.

- Chapter 5, “Simulating Your Design,” describes simulation methods for verifying the function and timing of your designs.
- Appendix A, “Accelerate FPGA Macros with One-Hot Approach,” reprints an article describing one-hot encoding in detail.

Conventions

This manual uses the following typographical and online document conventions. An example illustrates each typographical 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 loc2* . . . *locn;*

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

About this Manual

Additional Resources v
Manual Contents vi

Conventions

Typographical ix
Online Document x

Chapter 1 Getting Started

Introduction 1-1
Advantages of Using HDLs to Design FPGAs 1-2
Designing FPGAs with HDLs 1-3
- Using Verilog 1-3
- Using VHDL 1-3
- Comparing ASICs and FPGAs 1-3
- Using Synthesis Tools 1-4
- Using FPGA System Features 1-4
- Designing Hierarchy 1-4
- Specifying Speed Requirements 1-5

Installing Design Examples and Tactical Software 1-5
- Software Requirements 1-5
- Workstation Requirements 1-5
  - Memory Requirements 1-6
- Disk Space Requirements 1-6
  - Xilinx Internet Site 1-7
- Retrieving Tactical Software and Design Examples 1-9
  - From Xilinx Internet FTP Site 1-9
- Extracting the Files 1-10
  - Extracting .tar.Z File in UNIX 1-11
  - Extracting .zip File in UNIX 1-11
  - Extracting .zip File in MS-DOS 1-11

Directory Tree Structure 1-11
Synopsys Startup File and Library Setup 1-15
Technical Support 1-15
Xilinx World Wide Web Site 1-15
Technical and Applications Support Hotlines 1-15
Xilinx FTP Site 1-16
XDOCS E-mail Server 1-16

Chapter 2 HDL Coding Hints

Comparing Synthesis and Simulation Results 2-1
Omit the Wait for XX ns Statement 2-2
Omit the ...After XX ns or Delay Statement 2-2
Use Case and If-Else Statements 2-3
Order and Group Arithmetic Functions 2-3
Omit Initial Values 2-4
Selecting HDL Formatting Styles 2-4
Selecting a Capitalization Style 2-4
Verilog 2-4
VHDL 2-5
Using Xilinx Naming Conventions 2-5
Matching File Names to Entity and Module Names 2-6
Naming Identifiers, Types, and Packages 2-6
Using Labels 2-6
Labeling Flow Control Constructs 2-7
Using Variables for Constants (VHDL Only) 2-8
Using Constants to Specify Opcode Functions (VHDL) 2-8
Using Parameters for Constants (Verilog Only) 2-8
Using Parameters to Specify Opcode Functions (Verilog) .. 2-9
Using Parameters to Specify Bus Size (Verilog) 2-9
Using Named and Positional Association 2-9
Managing Your Design 2-10
Creating Readable Code 2-10
Indenting Your Code 2-10
Using Empty Lines 2-11
Using Spaces 2-11
Breaking Long Lines of Code 2-12
Adding Comments 2-12
Using Std_logic Data Type (VHDL only) 2-13
Declaring Ports 2-13
Minimizing the Use of Ports Declared as Buffers 2-14
Comparing Signals and Variables (VHDL only) 2-15
Using Signals (VHDL) 2-15

Using Variables (VHDL) 2-16
Using Schematic Design Hints with HDL Designs 2-17
Barrel Shifter Design 2-17
16-bit Barrel Shifter - VHDL 2-17
16-bit Barrel Shifter - Verilog 2-20
16-bit Barrel Shifter with Two Levels of Multiplexers - VHDL 2-23
16-bit Barrel Shifter with Two Levels of Multiplexers - Verilog 2-24
Implementing Latches and Registers 2-26
D Latch Inference 2-27
Converting a D Latch to a D Register 2-28
Resource Sharing 2-31
Gate Reduction 2-38
Preset Pin or Clear Pin 2-41
Register Inference 2-43
Using Clock Enable Pin 2-45
Using If Statements 2-49
Using Case Statements 2-50
Using Nested If Statements 2-51
Inefficient Use of Nested If Statement 2-51
Nested If Example Modified to Use If-Case 2-54
Comparing If Statement and Case Statement 2-58
4-to-1 Multiplexer Design with If Construct 2-59
4-to-1 Multiplexer Design with Case Construct 2-61

Chapter 3 Understanding High-Density Design Flow

Design Flow 3-1
Entering your Design and Selecting Hierarchy 3-5
Design Entry Recommendations 3-5
Use RTL Code 3-5
Carefully Select Design Hierarchy 3-5
Functional Simulation of your Design 3-5
Synthesizing and Optimizing your Design 3-6
Creating a .synopsys_dc.setup File 3-6
Creating a Compile Run Script 3-7
Compiling Your Design 3-7
Modifying your Design 3-7
Compiling Large Designs 3-7
Saving your Compiled Design as an SXNF File 3-7
Setting Timing Constraints 3-8
Naming and Grouping Signals Together 3-8
TNMs 3-8
TIMEGRPs 3-9

TPSYNC Specification 3-9
Specifying Timing Constraints 3-10
Period Constraint 3-10
FROM:TO Style Constraint 3-10
Offset Constraint 3-11
Ignoring Timing Paths 3-11
Controlling Signal Skew 3-12
Timing Constraint Priority 3-12
Evaluating Design Size and Performance 3-14
Using FPGA Compiler to Estimate Device Utilization and Performance 3-14
Using the Report_fpga Command 3-14
Using the Report_timing Command 3-15
Determining Actual Device Utilization and Pre-routed Performance 3-17
Using the Design Manager to Map Your Design 3-17
Using the Command Line to Map Your Design 3-19
Evaluating your Design for Coding Style and System Features 3-21
Tips for Improving Design Performance 3-21
Modifying Your Code 3-21
Using FPGA System Features 3-21
Placing and Routing Your Design 3-22
Decreasing Implementation Time 3-22
Improving Implementation Results 3-24
Multi-Pass Place and Route Option 3-24
Turns Engine Option (UNIX only) 3-24
Re-entrant Routing Option 3-24
Cost-Based Clean-up Option 3-26
Delay-Based Clean-up Option 3-26
Guide Option (not recommended) 3-26
Timing Simulation of your Design 3-26
Creating a Timing Simulation Netlist 3-27
From the Design Manager 3-27
From the Command Line 3-27
Downloading to the Device and In-system Debugging 3-28
Creating a PROM File for Stand-Alone Operation 3-28

Chapter 4 Designing FPGAs with HDL

Introduction 4-1
Using Global Low-skew Clock Buffers 4-2
Inserting Clock Buffers 4-6
Instantiating Global Clock Buffers 4-7

Instantiating Buffers Driven from a Port 4-7
Instantiating Buffers Driven from Internal Logic 4-7
Using Dedicated Global Set/Reset Resource 4-9
Startup State 4-10
Preset vs. Clear (XC4000, Spartan) 4-11
Increasing Performance with the GSR/GR Net 4-11
Design Example without Dedicated GSR/GR Resource 4-12
Design Example with Dedicated GSR/GR Resource 4-17
Design Example with Active Low GSR/GR Signal 4-24
Encoding State Machines 4-26
Using Binary Encoding 4-27
VHDL - Binary Encoded State Machine Example 4-28
Verilog - Binary Encoded State Machine Example 4-30
Using Enumerated Type Encoding 4-33
VHDL- Enumerated Type Encoded State Machine Example 4-34
Verilog - Enumerated Type Encoded State Machine Example 4-35
Using One-Hot Encoding 4-36
VHDL - One-hot Encoded State Machine Example 4-36
Verilog - One-hot Encoded State Machine Example 4-37
Summary of Encoding Styles 4-39
Comparing Synthesis Results for Encoding Styles 4-40
Initializing the State Machine 4-40
Using Dedicated I/O Decoders 4-41
Instantiating LogiBLOX Modules 4-46
Using LogiBLOX in HDL Designs 4-48
Implementing Memory 4-52
Implementing XC4000 and Spartan ROMs 4-52
VHDL - RTL Description of a ROM 4-52
Verilog - RTL Description of a ROM 4-53
Implementing XC4000 Family RAMs 4-55
VHDL - Instantiating RAM 4-55
Verilog - Instantiating RAM 4-56
Using LogiBLOX to Implement Memory 4-57
Implementing Boundary Scan (JTAG 1149.1) 4-61
Instantiating the Boundary Scan Symbol 4-62
VHDL - Boundary Scan 4-62
Verilog - Boundary Scan 4-64
Implementing Logic with IOBs 4-65
XC4000E/EX/XLA/XL/XV and Spartan IOBs 4-65
Inputs 4-66
Outputs 4-66
Slew Rate 4-67

Pull-ups and Pull-downs 4-67
XC4000EX/XLA/XL/XV Output Multiplexer/2-Input Function Generator 4-68
XC5200 IOBs 4-69
Bi-directional I/O 4-70
Inferring Bi-directional I/O 4-70
Instantiating Bi-directional I/O 4-72
Using LogiBLOX to Create Bi-directional I/O 4-75
Specifying Pad Locations 4-78
Moving Registers into the IOB 4-78
Use –pr Option with Map 4-79
Using Unbonded IOBs (XC4000E/EX/XLA/XL/XV and Spartan Only) 4-80
VHDL - 4-bit Shift Register Using Unbonded I/O 4-80
Verilog - 4-bit Shift Register Using Unbonded I/O 4-82
Implementing Multiplexers with Tristate Buffers 4-83
VHDL - Mux Implemented with Gates 4-83
Verilog - Mux Implemented with Gates 4-84
VHDL - Mux Implemented with BUFTs 4-85
Verilog - Mux Implemented with BUFTs 4-86
Using Pipelining 4-88
Before Pipelining 4-88
After Pipelining 4-88
Design Hierarchy 4-89
Using Synopsys with Hierarchical Designs 4-90

Chapter 5 Simulating Your Design

Introduction 5-1
Functional Simulation 5-4
Register Transfer Level (RTL) 5-4
Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation 5-5
Post-NGDBuild (Pre-Map) Gate-Level Simulation 5-6
Post-Map Partial Timing (CLB and IOB Block Delays) 5-6
Creating a Simulation Test Bench 5-7
XSI Pre-implementation Simulation 5-8
XSI Post-implementation Functional Simulation 5-9
Post-NGDBuild Simulation 5-9
Post-Map Simulation 5-9
Post-PAR Simulation (timing simulation) 5-10
Timing Simulation 5-10
Post-Route Full Timing (Block and Net Delays) 5-10
Using VHDL/Verilog Libraries and Models 5-10

Adhering to Industry Standards 5-11
Locating Library Source Files 5-13
Installing and Configuring XSI VHDL Libraries 5-13
Using the UniSim Library 5-15
UniSim Library Structure 5-16
Compiling the UniSim Library 5-17
Instantiating UniSim Library Components 5-18
Using UniSim Libraries with XSI VHDL Designs 5-18
Using the LogiBLOX Library 5-18
Compiling the LogiBLOX Library 5-19
Instantiating LogiBLOX Modules 5-20
Using the LogiCORE Library 5-20
Simulating Global Signals 5-21
Adapting Schematic Global Signal Methodology for VHDL 5-23
Setting VHDL Global Set/Reset Emulation in Functional Simulation 5-24
Global Signal Considerations (VHDL) 5-24
GSR Network Design Cases 5-26
Using VHDL Reset-On-Configuration (ROC) Cell (Case 1A) 5-27
Using ROC Cell Implementation Model (Case 1A) 5-28
ROC Test Bench (Case 1A) 5-29
ROC Model in Four Design Phases (Case 1A) 5-30
Using VHDL ROCBUF Cell (Case 1B) 5-32
ROCBUF Model in Four Design Phases (Case 1B) 5-33
Using VHDL STARTBUF Block (Case 2A and 2B) 5-35
GTS Network Design Cases 5-38
Using VHDL Tristate-On-Configuration (TOC) 5-39
VHDL TOC Cell (Case A1) 5-39
TOC Cell Instantiation (Case A1) 5-40
TOC Test Bench (Case A1) 5-41
TOC Model in Four Design Phases (Case A1) 5-42
Using VHDL TOCBUF (Case B1) 5-44
TOCBUF Model Example (Case B1) 5-44
TOCBUF Model in Four Design Phases (Case B1) 5-46
Using Oscillators (VHDL) 5-47
VHDL Oscillator Example 5-48
Oscillator Test Bench 5-49
Compiling Verilog Libraries 5-50
Compiling Libraries for ModelSim 5-51
Setting Verilog Global Set/Reset 5-51
Defining GSR in a Test Bench 5-52
Designs without a STARTUP Block 5-55

Example 1: XC4000E/L/X, Spartan/XL, or Virtex RTL Functional Simulation (No STARTUP/STARTUP_VIRTEX Block)..... 5-55
Example 2: XC5200 RTL Functional Simulation (No STARTUP Block)..... 5-58
Example 3: XC3000A/L, or XC3100A/L RTL Functional Simulation (No STARTUP Block)..... 5-60
Designs with a STARTUP Block..... 5-62
Example 1: XC4000E/L/X and Spartan/XL Simulation with STARTUP, or Virtex with STARTUP_VIRTEX..... 5-64
Example 2: XC5200 Simulation with STARTUP..... 5-66
Example 3: XC3000A/L and XC3100A/L Designs..... 5-67
Setting Verilog Global Tristate (XC4000, Spartan, and XC5200 Outputs Only)..... 5-67
Defining GTS in a Test Bench..... 5-67
Designs without a STARTUP Block..... 5-68
XC4000E/L/X, Spartan/XL, Virtex and XC5200 RTL Functional Simulation (No STARTUP Block)..... 5-68
Designs with a STARTUP Block..... 5-69
Example 1: XC4000E/L/X, Spartan/XL, Virtex, and XC5200 Simulation (With STARTUP/STARTUP_VIRTEX, GTS Pin Connected)..... 5-69
Example 2: XC4000E/L/X, Spartan/XL, Virtex, and XC5200 Simulation (With STARTUP/STARTUP_VIRTEX, GTS Pin not connected)..... 5-70

Appendix A Accelerate FPGA Macros with One-Hot Approach

Chapter 1

Getting Started

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.

- "Introduction"
- "Advantages of Using HDLs to Design FPGAs"
- "Designing FPGAs with HDLs"
- "Installing Design Examples and Tactical Software"
- "Synopsys Startup File and Library Setup"
- "Technical Support"

Introduction

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. System requirements and installation instructions are also provided.

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 XSI flow; information on installing XSI; 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 Service and Support, and Use the Customer Service and Support page to get answers to your technical questions. You can also use the File Download option to download the latest software patches, tutorials, design files, and documentation.

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. Once 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 traditional gate-level bottleneck. 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 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.

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 now being standardized by the IEEE 1364 Working Group. 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 Synopsys® compilers. The coding strategies 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 are written specifically for Synopsys compilers. 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

Methods used to design ASICs do not always apply to FPGA designs. ASICs have more gate and routing resources than FPGAs. Since

ASICs have a large number of available resources, you can easily create inefficient code that results in a large number of gates. When designing FPGAs, you *must* create efficient code.

Using Synthesis Tools

Synthesis tools, such as the Synopsys FPGA Compiler, have special optimization algorithms for Xilinx FPGAs. Constraints and compiling options perform differently depending on the target device. There are some commands and constraints 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.

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.

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 Synopsys for the current recommendations for preferred module size. As a last resort, use the grouping commands of your synthesizer. 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.

Installing Design Examples and Tactical Software

The information in this section supplements information in the *Synopsys Interface Guide*. Also, read and follow the instructions in the latest version of the *Quick Start Guide for Xilinx Alliance Series*, as well as the current version of the *Alliance Series Install and Release Document*.

This manual includes numerous HDL design examples created with VHDL and Verilog. VHDL is more comprehensive than Verilog, and you many need to spend more time learning how to apply VHDL constructs to synthesis.

Software Requirements

To synthesize, simulate, and implement the design examples in this manual, you should have the following versions of software (see the following table) installed on your system.

Note: The design examples in this manual were compiled with Synopsys 1997.08 and Xilinx M1 software; however, all programs, scripts, and design examples are compatible with the versions in the following table.

Table 1-1 Software Versions

Software	Version
Xilinx Synopsys Interface (XSI)	Alliance Series 2.1 or later
Xilinx Development System	Alliance Series 2.1 or later
Synopsys FPGA Compiler	1997.08 or later

Workstation Requirements

The current release of the software supports the following workstation architectures and operating systems.

- Solaris® 2.5 and 2.51
- Ultra Sparc (or equivalent)

- HP-UX HP 10.2
- HP715 (or equivalent)

Memory Requirements

The values provided in the following table are for typical designs, and include loading the operating system. Additional memory may be required for certain “boundary-case” or unusual designs, as well as for the concurrent operation of other applications (for example, Design Compiler or Verilog XL). Xilinx recommends compiling XC4000EX/ XL designs on the Ultra Sparc, HP715, or equivalent workstations. Although 64 MB of RAM and 64 MB of swap space are required to compile XC4000EX designs, Xilinx recommends that you use 128 MB of RAM and 128 MB of swap space for more efficient processing of your XC4000EX designs.

Table 1-2 Memory Requirements for Workstations

Xilinx Device	RAM	Swap Space
XC3000A/L XC3100A/L XC4000E/L XC4028EX through XC4036EX XC4002XL through XC4028XL XCS (Spartan) XC5200 XC9500 (small devices)	64 MB	64 MB – 128 MB
XC4036XL through XC4062XL XC9500 (large devices)	128 MB	128 MB – 256 MB
XC4085XL XC40125XV	256 MB	256 MB – 512 MB

Disk Space Requirements

Before you install the programs and files, verify that your system meets the requirements listed in the following tables for the applicable options. The disk space requirements listed are approximations and may not exactly match the actual numbers. To significantly reduce the amount of disk space needed, install only the components and documentation that you will actually use.

Note: Refer to the *Alliance Series Install and Release Document* for more information on disk space requirements.

Table 1-3 Disk Space Requirements

Software Component	Data	Sol	HP
Xilinx Core Technology	~12 MB	~98 MB	~108 MB
Xilinx Device Data Files (All devices)[a]	~195 MB	~26 MB	~26 MB
Documentation: Online Help Documentation Browser Xilinx Tutorial Files Xilinx Userware	~30 MB total ~17 MB ~1 MB ~4 MB	~10 MB	~10 MB

a. The memory requirements specified are for the installation of all Xilinx devices. You can significantly reduce the amount of disk space required by installing only the files for the devices you want to target.

Table 1-4 XSI Interface Disk Space Requirements

Software Component	Disk Space (MB)
XSI Interface	~105 MB

Xilinx Internet Site

To download the programs and files from the Xilinx Internet Site, you must meet the disk requirements listed in the following table.

Table 1-5 Internet Files

Directory/Location	Description	Compressed File	Directory Size
M1_VHDL_source[a]	All VHDL source code only (no scripts, compilation, or implementation files)	m1_vhdl_src.tar.Z (size: 60 KB) or m1_vhdl_src.zip (size: 68 KB)	271 KB
M1_Verilog_source[a]	All Verilog source code only (no scripts, compilation, or implementation files)	m1_verilog_src.tar.Z (size: 57 KB) or m1_verilog_src.zip (size: 64 KB)	256 KB
M1_HDL_source[a]	All VHDL and Verilog source code only (no scripts, compilation, or implementation files)	m1_hdl_src.tar.Z (size: 110 KB) or m1_hdl_src.zip (size: 129 KB)	497 KB
M1_XSI_VHDL[b]	All VHDL source code, scripts, compilation, and implementation files	m1_xsi_vhdl.tar.Z (size: 6.3 MB) or m1_xsi_vhdl.zip (size: 4.3 MB)	16.1 MB
M1_XSI_Verilog[b]	All Verilog source code, scripts, compilation, and implementation files	m1_xsi_verilog.tar.Z (size: 5.3 MB) or m1_xsi_verilog.zip (size: 3.6 MB)	14.9 MB
M1_XSI_HDL[b]	All VHDL and Verilog source code, scripts, compilation, and implementation files	m1_xsi_hdl.tar.Z (size: 11.5 MB) or m1_xsi_hdl.zip (size: 7.9 MB)	30.8 MB

a. These files are located at ftp://ftp.xilinx.com/pub/applications/3rd party

b. These files are located at ftp://ftp.xilinx.com/pub/swhelp/synopsys, as well as at ftp://ftp.xilinx.com/pub/applications/3rd party

The M1_VHDL_source and M1_Verilog_source directories are smaller than the M1_XSI_VHDL and M1_XSI_Verilog directories because they contain only the VHDL or Verilog source code, and not the compilation and implementation files. If you want to access both Verilog and VHDL examples from this book, you can download either the M1_HDL_source or M1_HDL_XSI directory.

Retrieving Tactical Software and Design Examples

You can retrieve the HDL design examples from the Xilinx Internet Site. If you need assistance retrieving the files, use the information listed in the "Technical Support" section of this chapter to contact the Xilinx Hotline.

You must install the retrieved files on the same system as the Xilinx software and the Synopsys tools. However, do not install the files into the directory with the current release of the software since they may get overwritten during the installation of the next version of the software.

From Xilinx Internet FTP Site

You can retrieve the programs and files from the Xilinx Internet FTP (File Transfer Protocol) site. Alternatively, if you are not familiar with FTP, you can retrieve the files by going to the Xilinx Web site (http://www.xilinx.com), clicking on Service and Support, and using the File Download option. To access the Xilinx FTP Site, you must either have an internet-capable FTP utility available on your machine or a Web browser that has FTP. To use FTP, your machine must be connected to the Internet and you must have permission to use FTP on remote sites. If you need more information on this procedure, contact your system administrator.

To retrieve the programs and files from the Xilinx FTP site, use the following procedure.

1. Go to the directory on your local machine where you want to download the files, as follows.

   cd *directory*

2. Invoke the FTP utility or your Web browser that provides FTP.
3. Connect to the Xilinx FTP site, ftp.xilinx.com as follows.

   ftp> **open ftp.xilinx.com**

or

Enter the following URL.

ftp://ftp.xilinx.com

4. Log into a guest account if the FTP utility or Web browser does not perform this automatically. This account gives you download privileges.

```
Name (machine:user-name): anonymous
Guest login ok, send your complete e-mail
address as the password.
Password: your_email_address
```

5. Go to the following directory.

```
ftp> cd pub/swhelp/synopsys
```

6. If you are using an FTP utility, make sure you are in binary mode.

```
ftp> bin
```

7. Retrieve the appropriate design files as follows.

```
ftp> get design_files.tar.Z
```

or

```
ftp> get design_files.zip
```

or

Select the appropriate file and select a destination directory on your local machine.

8. Extract the files as described in the next section.

Extracting the Files

You must install the retrieved files on the same system as the current release of the Xilinx software and the Synopsys tools. However, do not install the files in the directory with the current software since they may get overwritten during the installation of the next version of the software. The files are stored in the UNIX™ standard tar and compress form, as well as in the PC™ standard zip form. To extract the files, use one of the following procedures.

Note: If the following procedures do not work on your system, consult your system administrator for help on extracting the files.

Extracting .tar.Z File in UNIX

1. Go to the directory where you downloaded the files:

   cd *downloaded_files*

2. Uncompress the files:

   uncompress *design*.**tar.Z**

3. Extract the files:

   tar xvf *design*.**tar**

Extracting .zip File in UNIX

1. Go to the directory where you downloaded the files:

   cd *downloaded_files*

2. Uncompress the files:

   unzip *design*.**zip**

Extracting .zip File in MS-DOS

1. Go to the directory where you downloaded the files:

   cd *downloaded_files*

2. Uncompress the files:

   pkunzip –d *design*.**zip**

Directory Tree Structure

After you have completed the installation, you should have the following directory tree structure:

```
5k_preset
    /VHDL
    /Verilog
/Async_RAM_as_latch
    /VHDL
    /Verilog
/Barrel_SR
    /VHDL
        /Barrel
        /Barrel_Org
    /Verilog
```

```
        /Barrel
        /Barrel_Org
/Bidir_LogiBLOX
    /VHDL
    /Verilog
/Bidir_infer
    /VHDL
    /Verilog
/Bidir_instantiate
    /VHDL
    /Verilog
/Bnd_scan_4k
    /VHDL
    /Verilog
/Bnd_scan_5k
    /VHDL
    /Verilog
/Case_vs_if
    /VHDL
        /Case_ex
        /If_ex
    /Verilog
        /Case_ex
        /If_ex
/Clock_enable
    /VHDL
    /Verilog
/Clock_mux
    /VHDL
    /Verilog
/Constants
    /VHDL
    /Verilog
        /Parameter1
        /Parameter2
/D_latch
    /VHDL
    /Verilog
/D_register
    /VHDL
    /Verilog
/FF_example
    /VHDL
    /Verilog
/GR_5K
```

```
    /VHDL
        /Active_low_GR
        /No_GR
        /Use_GR
    /Verilog
        /Active_low_GR
        /No_GR
        /Use_GR
/GSR
    /VHDL
        /Active_low_GSR
        /No_GSR
        /Use_GSR
    /Verilog
        /Active_low_GSR
        /No_GSR
        /Use_GSR
/Gate_clock
    /VHDL
        /Gate_clock
        /Gate_clock2
    /Verilog
        /Gate_clock
        /Gate_clock2
/Gate_reduce
    /VHDL
        /Synopsys_dw
        /Xilinx_dw
    /Verilog
        /Synopsys_dw
        /Xilinx_dw
/IO_Decoder
    /VHDL
/IO_Decoder
    /Verilog
/LogiBLOX_DP_RAM
    /VHDL
    /Verilog
/LogiBLOX_SR
    /VHDL
    /Verilog
/Mux_vs_3state
    /VHDL
        /Mux_gate
        /Mux_gate16
```

```
            /Mux_tbuf
            /Mux_tbuf16
        /Verilog
            /Mux_gate
            /Mux_gate16
            /Mux_tbuf
            /Mux_tbuf16
/Nested_if
    /VHDL
        /If_case
        /Nested_if
    /Verilog
        /If_case
        /Nested_if
/OMUX_example
    /VHDL
    /Verilog
/RAM_primitive
    /VHDL
    /Verilog
/ROM_RTL
    /VHDL
    /Verilog
/Res_sharing
    /VHDL
        /Res_no_share
        /Res_sharing
        /Res_xdw_no_share
        /Res_xdw_share
    /Verilog
        /Res_no_share
        /Res_sharing
        /Res_xdw_no_share
        /Res_xdw_share1
/Set_and_Reset
    /VHDL
    /Verilog
/Sig_vs_Var
    /VHDL
        /Xor_Sig
        /Xor_Var
/State_Machine
    /VHDL
        /Binary
        /Enum
```

```
            /One_Hot
        /Verilog
            /Binary
            /Enum
            /One_Hot
/Unbonded_IO
    /VHDL
    /Verilog
```

Synopsys Startup File and Library Setup

Follow the procedures in the "Getting Started" chapter of the *Synopsys Interface Guide* for instructions on modifying the Synopsys startup file for the FPGA Compiler.

Technical Support

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

Xilinx World Wide Web Site

http://www.xilinx.com

Click on the Service and Support option on the Xilinx Home Page. Use the Customer Service and Support page to get answers to your technical questions. You can use the Answers Search option to search the Answers database, file download area, application notes, XCELL journals, data sheets, and expert journals.

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 1-6 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

Table 1-6 Technical Support

Location	Telephone	Electronic Mail	Facsimile (Fax)
Germany	49-89-9915-4930	dlhelp@xilinx.com	49-89-904-4748
United Kingdom	44-1932-820821	ukhelp@xilinx.com	44-1932-828522
Corporate Switch-board	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.xilinx.com

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

XDOCS E-mail Server

xdocs@xilinx.com

Include the word “help” in the subject header. This e-mail service provides access to the Customer Service and Support page from the Xilinx World Wide Web Site. On the Xilinx home page, click on Service and Support, and Use the Customer Service and Support page to get answers to your technical questions.

Chapter 2

HDL Coding Hints

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

- "Comparing Synthesis and Simulation Results"
- "Selecting HDL Formatting Styles"
- "Using Schematic Design Hints with HDL Designs"

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.

Comparing Synthesis and Simulation Results

VHDL and Verilog are hardware description and simulation languages that were not originally intended as input 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:

- 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:

- 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.

Use Case and If-Else Statements

You can use If-Else statements, Case statements, or other conditional code to create state machines or other conditional logic. These statements implement the functions differently, however, the simulated designs are identical. The If-Else statement specifies priority-encoded logic and the Case statement specifies parallel behavior. The If-Else statement can, in some cases, result in a slower circuit overall. Refer to the "Comparing If Statement and Case Statement" section of this chapter for more information.

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);
```

Additionally, 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.

The FPGA Compiler 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.

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;
  ```

Selecting HDL Formatting 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.

Selecting a Capitalization Style

Select a capitalization style for your code. Xilinx recommends using a consistent style (lower or upper case) for entity or module names in FPGA designs.

Verilog

For Verilog, the following style is recommended.

- Use lower case letters for entity or module names and reserved words
- Use upper case letters for the following.

- Keywords that are not entity names or reserved words
- Variable, signal, instance, and module names
- Labels
- Libraries, packages, and data types
- Instantiated cells from the XSI Library
- For the names of standard or vendor packages, the style used by the vendor or uppercase letters are used, as shown for IEEE in the following example:

```
library IEEE;
use IEEE.std_logic_1164.all;
signal SIG: UNSIGNED (5 downto 0);
```

VHDL

For VHDL, use lower case for all language constructs from the IEEE-STD 1076. Any inputs defined by you should be upper case. For example, use upper case for the names of signals, instances, components, architectures, processes, entities, variables, configurations, libraries, functions, packages, data types, and sub-types.

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 since 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

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, it 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 Synopsys script file with either a .scr or .script file extension.

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
```

Using Labels

Use labels to group logic. Label all processes, functions, and procedures as shown in the following examples.

- VHDL

  ```
  ASYNC_FF: process (CLK,RST)
  ```

- Verilog

  ```
  always @ (posedge CLK or posedge RST)
  begin: ASYNC_FF
  ```

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.

- 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 Variables for Constants (VHDL Only)

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 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 is easily modified.

Using Constants to Specify Opcode Functions (VHDL)

```
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 for Constants (Verilog Only)

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.

Using Parameters to Specify Opcode Functions (Verilog)

```
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
```

Using Parameters to Specify Bus Size (Verilog)

```
parameter BUS_SIZE = 8;

output ['BUS_SIZE-1:0] OUT;
input ['BUS_SIZE-1:0] X,Y;
```

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));
```

Managing Your Design

As part of your coding specifications, you should include rules for naming, organizing, and distributing your files. Also, use explicit configurations to control the selection of components and architectures that you want to compile, simulate, or synthesize.

Creating Readable Code

Use the recommendations in this section to create code that is easy to read.

Indenting Your Code

Indent blocks of code to align related statements. You should define the number of spaces for each indentation level and specify whether the Begin statement is placed on a line by itself. In the examples in this manual, each level of indentation is four spaces and the Begin statement is on a separate line that is not indented from the previous line of code. The examples below illustrate the indentation style used in this manual.

- VHDL

```
-- 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

```
/* 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: LATCH
        if (GATE == 1'b1)
            Q <= DATA;
    end  // End Latch

endmodule
```

Using Empty Lines

Use empty lines to separate top-level constructs, designs, architectures, configurations, processes, subprograms, and packages.

Using Spaces

Use spaces to make your code easier to read. You can omit or use spaces between signal names as shown in the following examples:

- VHDL

```
process (RST,CLOCK,LOAD,CE)
process (RST, CLOCK, LOAD, CE)
```

- Verilog

```
module test (A,B,C)
module test (A, B, C)
```

Use a space after colons as shown in the following examples:

- VHDL

```
signal QOUT: STD_LOGIC_VECTOR (3 downto 0);
CLK_1: BUFGS port map (I=>CLOCK_IN,O=>CLOCK_OUT);
```

- Verilog

```
begin: CPU_DATA
```

Breaking Long Lines of Code

Break long lines of code at an appropriate point, such as at a comma, a colon, or a parenthesis to make your code easier to read, as illustrated in the following code fragments.

- VHDL

```
U1: load_reg port map
(INX=>A,LOAD=>LD,CLK=>SCLK,OUTX=>B);
```

- Verilog

```
load_reg U1
   (.INX(A), .LOAD(LD), .CLK(SCLK), .OUTX(B));
```

Adding Comments

Add comments to your code to improve readability, reduce debugging time, and make it easier to maintain your code.

- VHDL

```
-- Read Counter (16-bit)
-- Updated 1-25-98 to add Clock Enable, John Doe
-- Updated 1-28-98 to add Terminal Count, Joe Cool

process (RST, CLOCK, CE)
begin
.
.
.
```

- Verilog

```
// Read Counter (16-bit)
// Updated 1-25-98 to add Clock Enable, John Doe
// Updated 1-28-98 to add Terminal Count, Joe Cool

always @ (posedge RST or posedge CLOCK)
begin
.
```

.
.

Using Std_logic Data Type (VHDL only)

The Std_logic (IEEE 1164) type is recommended for synthesis when you use the Synopsys compiler. This type is recommended for hardware descriptions for the following reasons.

- It has nine different values that represent most of the states found in digital circuits.
- Automatically initialized to an unknown value. This 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.
- Easy to perform a 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.

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 effect the functioning 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_SIG” figure and the “Gate Implementation of XOR_VAR” figure.

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;
```

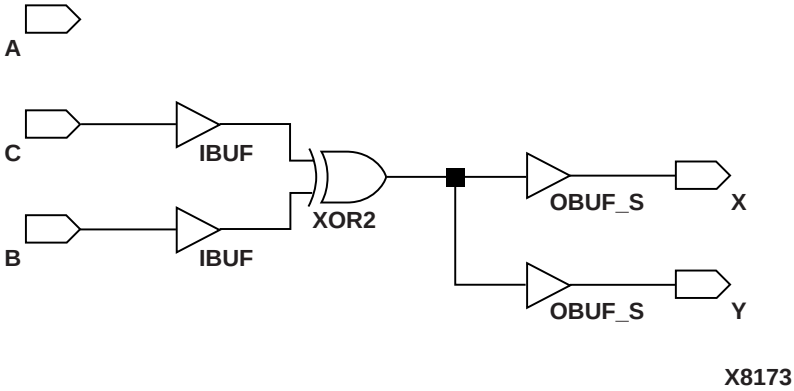

Figure 2-1 Gate implementation of XOR_SIG

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;
```

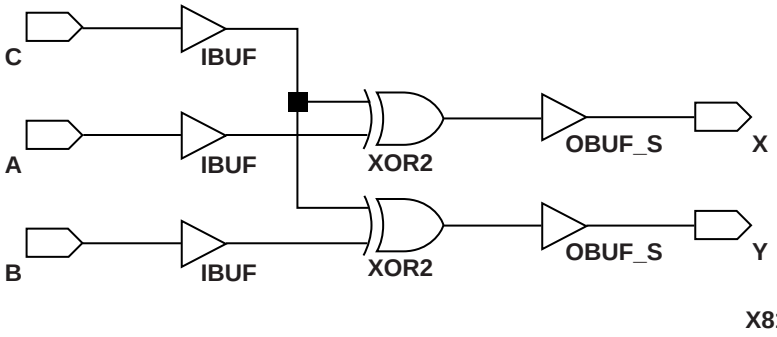

Figure 2-2 Gate Implementation of XOR_VAR

Using Schematic Design Hints with HDL Designs

This section describes how to apply schematic entry design strategies to HDL designs.

Barrel Shifter Design

The schematic version of the barrel shifter design is included in the "Multiplexers and Barrel Shifters in XC3000/XC3100" application note (XAPP 026.001) available on the Xilinx web site at http://www.xilinx.com. In this example, two levels of multiplexers are used to increase the speed of a 16-bit barrel shifter. This design is for XC3000 and XC3100 device families; however, it can also be used for other Xilinx devices.

The following VHDL and Verilog examples show a 16-bit barrel shifter implemented using sixteen 16-to-1 multiplexers, one for each output. A 16-to-1 multiplexer is a 20-input function with 16 data inputs and four select inputs. When targeting an FPGA device based on 4-input lookup tables (such as XC4000 and XC3000 family of devices), a 20-input function requires at least five logic blocks. Therefore, the minimum design size is 80 (16 x 5) logic blocks.

16-bit Barrel Shifter - VHDL

```
-------------------------------------------------
-- VHDL Model for a 16-bit Barrel Shifter       --
```

```
-- barrel_org.vhd                                    --
-- !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! --
-- THIS EXAMPLE IS FOR COMPARISON ONLY               --
-- May 1997                                          --
-- USE barrel.vhd                                    --
-------------------------------------------------------

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

entity barrel_org is
   port (S:in    STD_LOGIC_VECTOR (3 downto 0);
         A_P:in    STD_LOGIC_VECTOR (15 downto 0);
         B_P:out   STD_LOGIC_VECTOR (15 downto 0));
end barrel_org;

architecture RTL of barrel_org is

begin
    SHIFT: process (S, A_P)
    begin
        case S is
            when “0000” =>
               B_P <= A_P;

            when “0001” =>
               B_P(14 downto 0)  <= A_P(15 downto 1);
               B_P(15) <= A_P(0);

            when “0010” =>
               B_P(13 downto 0)  <= A_P(15 downto 2);
               B_P(15 downto 14) <= A_P(1 downto 0);

            when “0011” =>
               B_P(12 downto 0)  <= A_P(15 downto 3);
               B_P(15 downto 13) <= A_P(2 downto 0);

            when “0100” =>
               B_P(11 downto 0)  <= A_P(15 downto 4);
               B_P(15 downto 12) <= A_P(3 downto 0);

            when “0101” =>
               B_P(10 downto 0)  <= A_P(15 downto 5);
               B_P(15 downto 11) <= A_P(4 downto 0);
```

```
            when “0110” =>
               B_P(9 downto 0)   <= A_P(15 downto 6);
               B_P(15 downto 10) <= A_P(5 downto 0);

            when “0111” =>
               B_P(8 downto 0)   <= A_P(15 downto 7);
               B_P(15 downto 9)  <= A_P(6 downto 0);

            when “1000” =>
               B_P(7 downto 0)   <= A_P(15 downto 8);
               B_P(15 downto 8)  <= A_P(7 downto 0);

            when “1001” =>
               B_P(6 downto 0)  <= A_P(15 downto 9);
               B_P(15 downto 7) <= A_P(8 downto 0);

            when “1010” =>
               B_P(5 downto 0)  <= A_P(15 downto 10);
               B_P(15 downto 6) <= A_P(9 downto 0);

            when “1011” =>
               B_P(4 downto 0)  <= A_P(15 downto 11);
               B_P(15 downto 5) <= A_P(10 downto 0);

            when “1100” =>
               B_P(3 downto 0)  <= A_P(15 downto 12);
               B_P(15 downto 4) <= A_P(11 downto 0);

            when “1101” =>
              B_P(2 downto 0)  <= A_P(15 downto 13);
              B_P(15 downto 3) <= A_P(12 downto 0);

           when “1110” =>
              B_P(1 downto 0)  <= A_P(15 downto 14);
              B_P(15 downto 2) <= A_P(13 downto 0);

           when “1111” =>
              B_P(0) <= A_P(15);
              B_P(15 downto 1) <= A_P(14 downto 0);

           when others =>
              B_P <= A_P;
           end case;
    end process; -- End SHIFT
```

```
                end RTL;
```

16-bit Barrel Shifter - Verilog

```
      ////////////////////////////////////////////////
     // BARREL_ORG.V Version 1.0                     //
    // Xilinx HDL Synthesis Design Guide            //
   // Unoptimized model for a 16-bit Barrel Shifter  //
  // THIS EXAMPLE IS FOR COMPARISON ONLY            //
 // Use BARREL.V                                   //
// January 1998                                   //
////////////////////////////////////////////////

module barrel_org (S, A_P, B_P);

    input  [3:0]  S;
    input  [15:0] A_P;
    output [15:0] B_P;

    reg    [15:0] B_P;

    always @ (A_P or S)
        begin
        case (S)
             4'b0000 : // Shift by 0
               begin
               B_P <= A_P;
               end

             4'b0001 : // Shift by 1
               begin
               B_P[15]   <= A_P[0];
               B_P[14:0] <= A_P[15:1];
               end

             4'b0010 : // Shift by 2
               begin
               B_P[15:14] <= A_P[1:0];
               B_P[13:0]  <= A_P[15:2];
               end

             4'b0011 : // Shift by 3
               begin
               B_P[15:13] <= A_P[2:0];
               B_P[12:0]  <= A_P[15:3];
```

```
  end

4'b0100 : // Shift by 4
   begin
   B_P[15:12] <= A_P[3:0];
   B_P[11:0]  <= A_P[15:4];
   end

4'b0101 : // Shift by 5
   begin
   B_P[15:11] <= A_P[4:0];
   B_P[10:0]  <= A_P[15:5];
   end

4'b0110 : // Shift by 6
   begin
   B_P[15:10] <= A_P[5:0];
   B_P[9:0]   <= A_P[15:6];
   end

4'b0111 : // Shift by 7
   begin
   B_P[15:9]  <= A_P[6:0];
   B_P[8:0]   <= A_P[15:7];
   end
4'b1000 : // Shift by 8
   begin
   B_P[15:8]  <= A_P[7:0];
   B_P[7:0]   <= A_P[15:8];
   end

4'b1001 : // Shift by 9
   begin
   B_P[15:7]  <= A_P[8:0];
   B_P[6:0]   <= A_P[15:9];
   end

4'b1010 : // Shift by 10
   begin
   B_P[15:6]  <= A_P[9:0];
   B_P[5:0]   <= A_P[15:10];
   end

4'b1011 : // Shift by 11
   begin
```

```
               B_P[15:5] <= A_P[10:0];
               B_P[4:0]  <= A_P[15:11];
               end

            4'b1100 : // Shift by 12
               begin
               B_P[15:4] <= A_P[11:0];
               B_P[3:0]  <= A_P[15:12];
               end

            4'b1101 : // Shift by 13
               begin
               B_P[15:3]  <= A_P[12:0];
               B_P[2:0]   <= A_P[15:13];
               end

            4'b1110 : // Shift by 14
               begin
               B_P[15:2]  <= A_P[13:0];
               B_P[1:0]   <= A_P[15:14];
               end

            4'b1111 : // Shift by 15
               begin
               B_P[15:1]  <= A_P[14:0];
               B_P[0]     <= A_P[15];
               end

            default :
               B_P        <= A_P;
        endcase
    end

endmodule
```

The following modified VHDL and Verilog designs use two levels of multiplexers and are twice as fast as the previous designs. These designs are implemented using 32 4-to-1 multiplexers arranged in two levels of sixteen. The first level rotates the input data by 0, 1, 2, or 3 bits and the second level rotates the data by 0, 4, 8, or 12 bits. Since you can build a 4-to-1 multiplexer with a single logic block, the minimum size of this version of the design is 32 (32 x 1) logic blocks.

16-bit Barrel Shifter with Two Levels of Multiplexers - VHDL

```
-- BARREL.VHD
-- Based on XAPP 26 (see http://www.xilinx.com)
-- 16-bit barrel shifter (shift right)
-- May 1997

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

entity barrel is
port (S:   in STD_LOGIC_VECTOR(3 downto 0);
A_P: in STD_LOGIC_VECTOR(15 downto 0);
B_P: out STD_LOGIC_VECTOR(15 downto 0));
end barrel;

architecture RTL of barrel is

signal SEL1,SEL2: STD_LOGIC_VECTOR(1 downto 0);
signal C:         STD_LOGIC_VECTOR(15 downto 0);

begin
    FIRST_LVL: process (A_P, SEL1)
    begin
        case SEL1 is
            when “00” => -- Shift by 0
               C <= A_P;

            when “01” => -- Shift by 1
               C(15) <= A_P(0);
               C(14 downto 0) <= A_P(15 downto 1);

            when “10” => -- Shift by 2
               C(15 downto 14) <= A_P(1 downto 0);
               C(13 downto 0) <= A_P(15 downto 2);

            when “11” => -- Shift by 3
               C(15 downto 13) <= A_P(2 downto 0);
               C(12 downto 0) <= A_P(15 downto 3);

            when others =>
               C <= A_P;
        end case;
```

```
    end process; --End FIRST_LVL

SECND_LVL: process (C, SEL2)
    begin
        case SEL2 is
           when “00” => --Shift by 0
               B_P <=  C;

           when “01” => --Shift by 4
               B_P(15 downto 12) <= C(3 downto 0);
               B_P(11 downto 0)  <= C(15 downto 4);

           when “10” => --Shift by 8
               B_P(7 downto 0)   <= C(15 downto 8);
               B_P(15 downto 8)  <= C(7 downto 0);

           when “11” => --Shift by 12
              B_P(3 downto 0)   <= C(15 downto 12);
              B_P(15 downto 4)  <= C(11 downto 0);

           when others =>
               B_P <= C;
        end case;
    end process; -- End SECOND_LVL

    SEL1 <= S(1 downto 0);
    SEL2 <= S(3 downto 2);

end rtl;
```

16-bit Barrel Shifter with Two Levels of Multiplexers - Verilog

```
/***************************************
 * BARREL.V                            *
 * XAPP 26   http://www.xilinx.com     *
 * 16-bit barrel shifter [shift right] *
 * May 1997                            *
 ***************************************/

module barrel (S, A_P, B_P);

input [3:0] S;
input [15:0] A_P;
output [15:0] B_P;
```

```
reg [15:0] B_P;

wire [1:0] SEL1, SEL2;
reg [15:0] C;

assign SEL1 = S[1:0];
assign SEL2 = S[3:2];

    always @ (A_P or SEL1)
           begin
        case (SEL1)
             2'b00 : // Shift by 0
                       begin
                       C <= A_P;
                       end

             2'b01 : // Shift by 1
                       begin
                       C[15] <= A_P[0];
                       C[14:0] <= A_P[15:1];
                       end

             2'b10 : // Shift by 2
                       begin
                       C[15:14] <= A_P[1:0];
                       C[13:0] <= A_P[15:2];
                       end

             2'b11 : // Shift by 3
                       begin
                       C[15:13] <= A_P[2:0];
                       C[12:0] <= A_P[15:3];
                       end

           default :
                       C <= A_P;
        endcase
    end

    always @ (C or SEL2)
    begin
        case (SEL2)
            2'b00 : // Shift by 0
                      begin
```

```
                        B_P <=  C;
                        end

                2'b01 : // Shift by 4
                        begin
                        B_P[15:12] <= C[3:0];
                        B_P[11:0]  <= C[15:4];
                        end

                2'b10 : // Shift by 8
                        begin
                        B_P[7:0] <= C[15:8];
                        B_P[15:8] <= C[7:0];
                        end

                2'b11 : // Shift by 12
                        begin
                        B_P[3:0] <= C[15:12];
                        B_P[15:4] <= C[11:0];
                        end

              default :
                        B_P <= C;
            endcase
        end

    endmodule
```

When these two designs are implemented in an XC4005E-2 device using the Synopsys FPGA compiler, there is a 64% improvement in the gate count (88 occupied CLBs reduced to 32 occupied CLBs) in the barrel.vhd design as compared to the barrel_org.vhd design. Additionally, there is a 19% improvement in speed from 35.58 ns (5 logic levels) to 28.85 ns (4 logic levels).

Implementing Latches and Registers

HDL compilers 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. The XC4000EX/XL and XC5200 FPGAs do have registers that can be configured to act as latches. For these devices, the HDL compiler infers a dedicated latch from incomplete conditional expressions. XC4000E, XC3100A, XC3000A, and Spartan devices do not have latches in their CLBs. For these devices, latches

described in RTL code are implemented with gates in the CLB function generators. For XC4000E or Spartan devices, 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

```
-- 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

```
/* 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
```

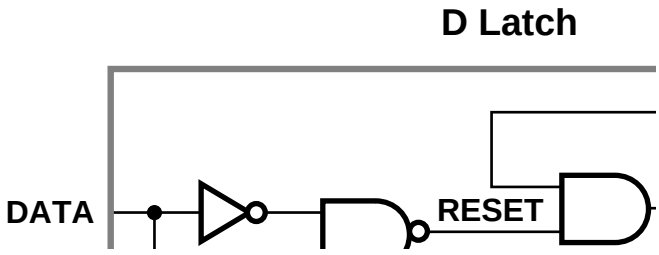

Figure 2-3 D Latch Implemented with Gates

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

```
-- 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

```
/* 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
```

If you are using the Synopsys Design Compiler or FPGA Compiler, you can determine the number of latches that are implemented when your design is read with the following command:

```
hdlin_check_no_latch = "TRUE"
```

When you set this command to true, a warning message is issued when a latch is inferred from a design. Use this command to verify

that a combinatorial design does not contain latches. The default value for this command is false.

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

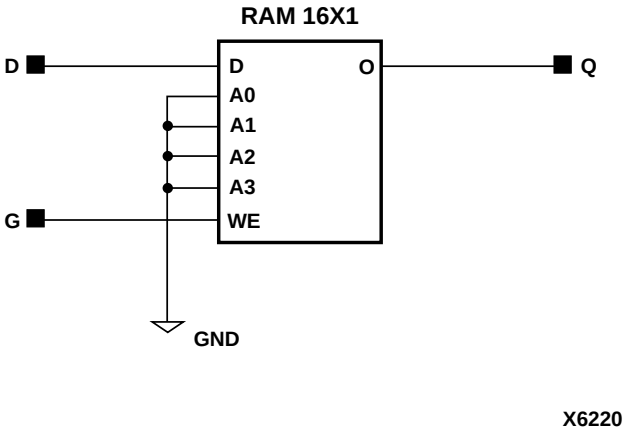

Figure 2-4 D Latch Implemented by Instantiating a RAM

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 2-1 D Latch Implementation Comparison

Comparison	Spartan, XC4000E CLB Latch Implemented with Gates	XC4000EX/ XL/XV, XC5200 CLB Latch	All Spartan and XC4000 Input Latch	XC4000 E/EX/XL/XV 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, best for XC4000E when latch needed in CLB	No hold time or combinatorial loop. FPGAs are register abundant.
Disadvantages	Feedback loop results in hold time requirement, not suggested	Not available in XC4000E or Spartan	Not available in XC5200, input to latch must directly connect to port	Must be instantiated, uses logic resources	Requires change in code to convert latch to register
Area[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. XC4000 and Spartan CLBs have two function generators and two registers; XC5200 CLBs have four function generators and four register / latches.

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 the operator on the same line.

*

+ –

> >= < <=

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

You can implement arithmetic functions (+, –, magnitude comparators) with gates, Synopsys DesignWare functions, or Xilinx DesignWare (XDW) functions. The XDW functions use modules that take advantage of the carry logic in XC4000 family, XC5200 family, and Spartan family CLBs. Carry logic and its dedicated routing increase the speed of arithmetic functions that are larger than 4-bits. To increase speed, use the XDW 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 XDW library automatically occurs 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 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 "Implementation of Resource Sharing" figure.

- VHDL

```
-- 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

```
/* 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
```

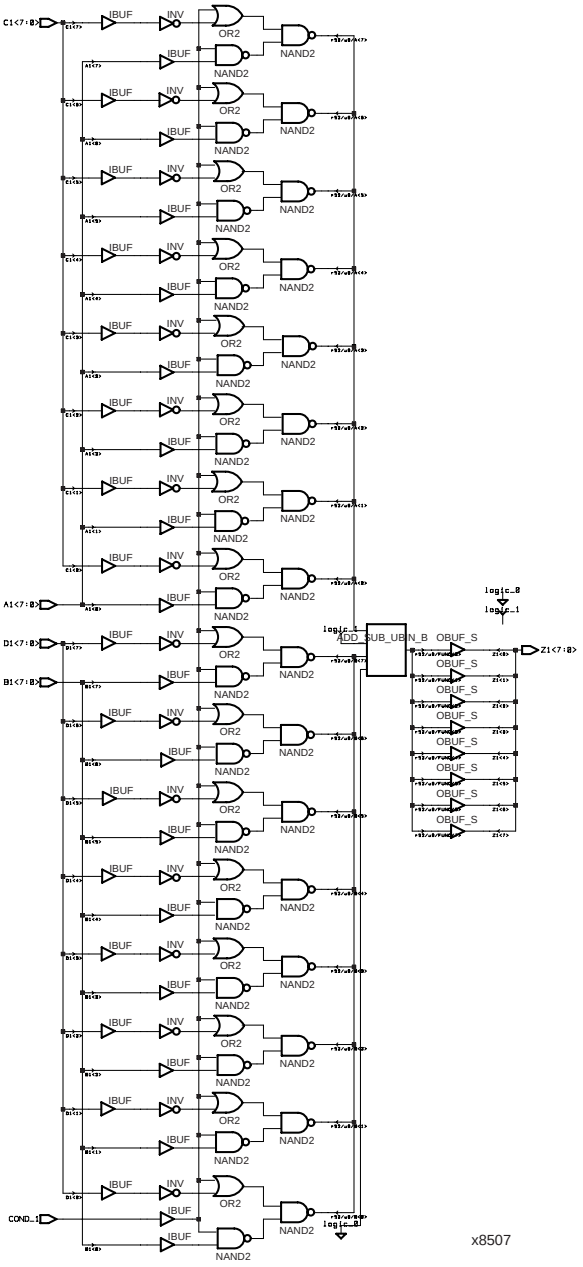

Figure 2-5 Implementation of Resource Sharing

If you disable resource sharing with the Hdl_resource_allocation = none command or if you code the design with the adders in separate processes, the design is implemented using two separate modules as shown in the following figure.

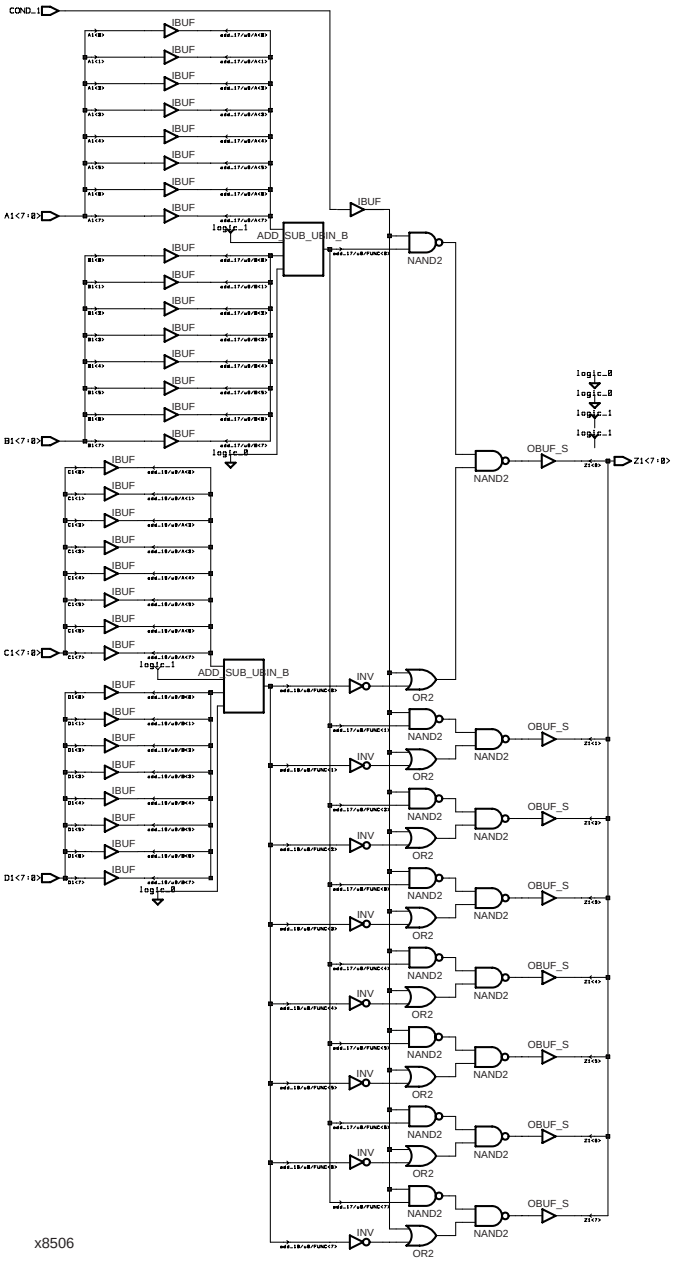

Figure 2-6 Implementation of No Resource Sharing

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. The last column in this table provides CLB and delay information for the same design with resource sharing and without XDW modules.

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

Comparison	Resource Sharing with Xilinx DesignWare	No Resource Sharing with Xilinx DesignWare	Resource Sharing without Xilinx DesignWare	No Resource Sharing without Xilinx DesignWare
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: You can manually specify resource sharing with pragmas. Refer to the appropriate Synopsys reference manual for more information on resource sharing.

Gate Reduction

Use the Synopsys DesignWare library components to reduce the number of gates in your designs. Gate reduction occurs when arithmetic functions are compiled with modules that contain similar functions. Gate reduction does not occur with the XDW library because the underlying logic of the components is not available when the design is compiled. The component logic is created later when the design is implemented through the M1 tools.

In the following design, two instances of the xilinx_dw function are called. To reduce the gate count, the two instances (I_0 and I_1) are grouped together and compiled with the –ungroup_all option. This option allows both instances to be evaluated and optimized together.

- VHDL

```
-- GATE_REDUCE.VHD
-- May 1997

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

entity gate_reduce is
    port (A_CHECK:   in STD_LOGIC;
          B_CHECK:   in STD_LOGIC;
          RESET:  in STD_LOGIC;
          CLOCK:      in STD_LOGIC;
          CLKEN:    in STD_LOGIC;
          A_TICK:   in STD_LOGIC;
          B_TICK:   in STD_LOGIC;
          ST_A:      out STD_LOGIC;
          ST_B:      out STD_LOGIC);
end gate_reduce;

architecture XILINX of gate_reduce is

   component xilinx_dw
      Port (CHECK    : in    STD_LOGIC;
            RST      : in    STD_LOGIC;
            CLK      : in    STD_LOGIC;
            CLK_EN   : in    STD_LOGIC;
            TICK     : in    STD_LOGIC;
            ST       : out   STD_LOGIC );
   end component;

begin

   I_0 : xilinx_dw
      port Map (CHECK=>A_CHECK, RST=>RESET, CLK=>CLOCK, CLK_EN=>CLKEN,
                TICK=>A_TICK, ST=>ST_A);
   I_1 : xilinx_dw
      port Map (CHECK=>B_CHECK, RST=>RESET, CLK=>CLOCK, CLK_EN=>CLKEN,
                TICK=>B_TICK, ST=>ST_B);

end XILINX;
```

- Verilog

```
 * GATE_REDUCE.V
 * December 1997                          */

module gate_reduce
               (A_CHECK, B_CHECK,
                RESET, CLOCK, CLKEN,
                A_TICK, B_TICK,
                ST_A, ST_B);

input A_CHECK, B_CHECK;
input RESET, CLOCK, CLKEN;
input A_TICK, B_TICK;
output ST_A, ST_B;

   xilinx_dw I_0 (.CHECK(A_CHECK), .RST(RESET), .CLK(CLOCK),
                  .CLK_EN(CLKEN),  .TICK(A_TICK), .ST(ST_A));
   xilinx_dw I_1 (.CHECK(B_CHECK), .RST(RESET), .CLK(CLOCK),
                  .CLK_EN(CLKEN), .TICK(B_TICK), .ST(ST_B));

endmodule
```

The following table provides a comparison of the number of CLBs used and the delay for the gate reduction designs using the Synopsys DesignWare library and the XDW library. Fewer CLBs are used when the Synopsys DesignWare library is used because the gates are reduced by flattening and compiling the two instances together.

Table 2-3 Synopsys DesignWare/Xilinx DesignWare

DesignWare Library	Synopsys FPGA Report	PAR Report	Timing Analyzer (CLB Levels)
Synopsys Design-Ware Library	47 CLBs	47 Occupied CLBs	C2S[a]: 20.017 (6) P2S[b]: 20.819 (5) C2P[c]: 12.278 (2)

Table 2-3 Synopsys DesignWare/Xilinx DesignWare

DesignWare Library	Synopsys FPGA Report	PAR Report	Timing Analyzer (CLB Levels)
Xilinx DesignWare Library	68 CLBs	72 Occupied CLBs	C2S[d]: 24.495 (9) P2S[e]: 19.150 (5) C2P[f]: 12.137 (2)

a. ClockToSetup

b. PadToSetup

c. ClockToPad

d. ClockToSetup

e. PadToSetup

f. ClockToPad

Note: Use the following Synopsys commands to reduce the compile time when compiling to reduce area.

```
dc_shell> set_resource_implementation=area_only
dc_shell> set_resource_allocation=area_only
```

These commands reduce the compile time when optimizing for area without changing the results.

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.

The XC3000 family and XC5200 family FPGAs have an asynchronous reset pin on the CLB registers, but do not have a dedicated preset pin. An asynchronous preset can be inferred by Synopsys but is built by connecting one inverter to the D input and connecting a second inverter to the Q output of a register. In this case, an asynchronous preset is created when the asynchronous reset is activated. This may require additional logic and increase delays. If possible, the inverters are merged with existing logic connected to the register input or output.

You can configure FPGA CLB registers to have either a preset pin or a clear pin. You cannot configure the CLB register for both pins. 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. If a register is described with an asynchronous set and reset, the Synopsys compiler reports the following error message 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 Synopsys netlist, NGDBuild issues the following error message:

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

An XC4000 CLB is shown in the following figure.

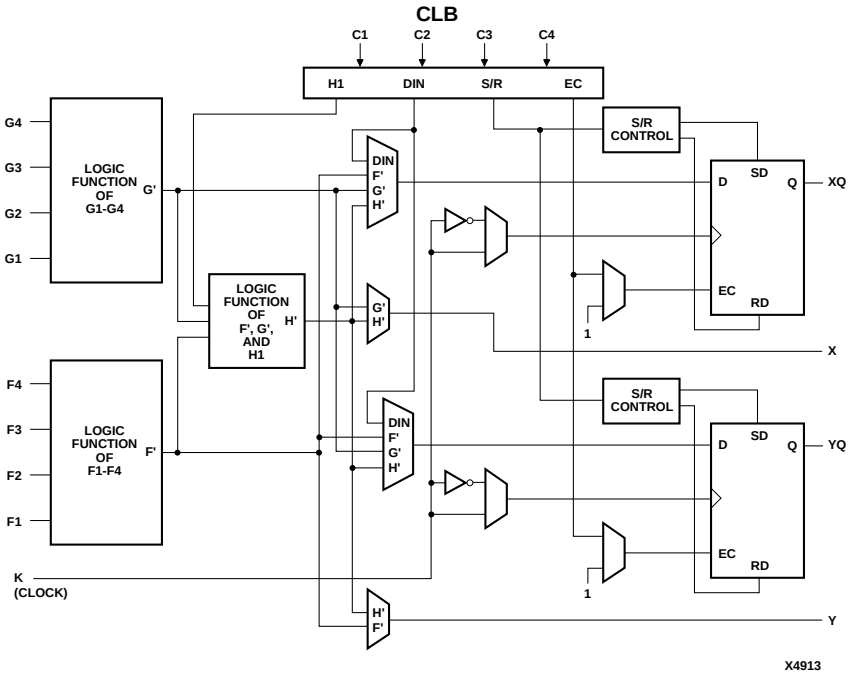

Figure 2-7 XC4000 Configurable Logic Block

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

Register Inference

- VHDL

```
-- 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 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 and 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
```

Using Clock Enable Pin

Use the CLB clock enable pin instead of gated clocks in your designs. Gated clocks can introduce glitches, increased clock delay, clock skew, as well as other undesirable effects. The first two examples in this section (VHDL and Verilog) illustrate a design that uses a gated clock. The “Implementation of Gated Clock” figure 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 “Implementation of Clock Enable” figure shows this design implemented with gates.

- VHDL

```
-- GATE_CLOCK.VHD
-- 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 <= not((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

```
/* Gated Clock Example
 * GATE_CLOCK.V
 * 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
```

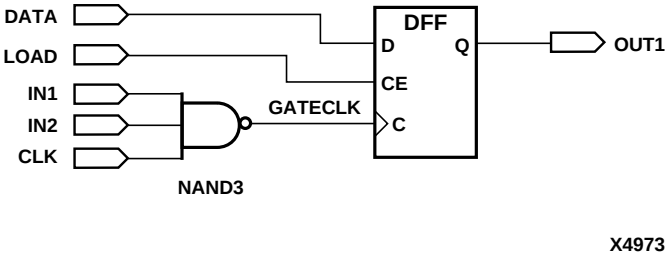

Figure 2-8 Implementation of Gated Clock

- VHDL

```
-- 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

```
/* 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
```

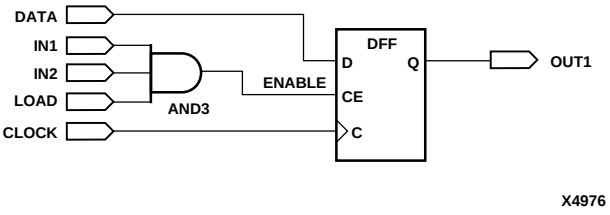

Figure 2-9 Implementation of Clock Enable

Using If Statements

The VHDL syntax for If statements is as follows:

```
if condition then
      sequence_of_statements;
{elsif condition then
      sequence_of_statements;}
else
      sequence_of_statements;
end if;
```

The Verilog syntax for If statements is as follows:

```
if (condition)
   begin
        sequence of statements;
   end
{else if (condition)
   begin
        sequence of statements;
   end}
else
   begin
        sequence of statements;
   end
```

Use If statements to execute a sequence of statements based on the value of a condition. The If statement checks each condition in order until the first true condition is found and then executes the statements associated with that condition. Once a true condition is found and the statements associated with that condition are executed, the rest of the If statement is ignored. If none of the conditions are true, and an Else clause is present, the statements associated with the Else

are executed. If none of the conditions are true, and an Else clause is not present, none of the statements are executed.

If the conditions are not completely specified (as shown below), a latch is inferred to hold the value of the target signal.

- VHDL

```
if (L = '1') then
      Q <= D;
end if;
```

- Verilog

```
if (L==1'b1)
      Q=D;
```

To avoid a latch inference, specify all conditions, as shown here.

- VHDL

```
if (L = '1') then
      Q <= D;
else
      Q <= '0';
end if;
```

- Verilog

```
if (L==1'b1)
     Q=D;
else
     Q=0;
```

Using Case Statements

The VHDL syntax for Case statements is as follows:

```
case expression is
      when choices =>
            {sequence_of_statements;}
      {when choices =>
            {sequence_of_statements;}}
      when others =>
            {sequence_of_statements;}
end case;
```

The Verilog syntax for Case statements is as follows:

```
case (expression)
   choices:   statement;
   {choices:  statement;}
   default:   statement;
endcase
```

Use Case statements to execute one of several sequences of statements, depending on the value of the expression. When the Case statement is executed, the given expression is compared to each choice until a match is found. The statements associated with the matching choice are executed. The statements associated with the Others (VHDL) or Default (Verilog) clause are executed when the given expression does not match any of the choices. The Others or Default clause is optional, however, if you do not use it, you must include all possible values for expression. Also, each Choices statement must have a unique value for the expression.

Using Nested If Statements

Improper use of the Nested If statement can result in an increase in area and longer delays in your designs. Each If keyword specifies priority-encoded logic. To avoid long path delays, do not use extremely long Nested If constructs as shown in the following VHDL / Verilog examples. These designs are shown implemented in gates in the “Implementation of Nested If” figure. Following these examples are VHDL and Verilog designs that use the Case construct with the Nested If to more effectively describe the same function. The Case construct reduces the delay by approximately 3 ns (using an XC4005E-2 part). The implementation of this design is shown in the “Implementation of If-Case” figure.

Inefficient Use of Nested If Statement

- VHDL

```
-- NESTED_IF.VHD
-- May 1997

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity nested_if is
```

```
    port (ADDR_A:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          ADDR_B:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          ADDR_C:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          ADDR_D:   in std_logic_vector (1 downto 0); -- ADDRESS Code
          RESET:    in std_logic;
          CLK :     in std_logic;
          DEC_Q:   out std_logic_vector (5 downto 0)); -- Decode OUTPUT
end nested_if;

architecture xilinx of nested_if is
begin

---------------- NESTED_IF PROCESS --------------
     NESTED_IF: process (CLK)
     begin
         if (CLK'event and CLK = '1') then
             if (RESET = '0') then
                 if (ADDR_A = "00") then
                     DEC_Q(5 downto 4) <= ADDR_D;
                     DEC_Q(3 downto 2) <= "01";
                     DEC_Q(1 downto 0) <= "00";
                     if (ADDR_B = "01") then
                         DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
                         DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                         if (ADDR_C = "10") then
                             DEC_Q(5 downto 4) <= unsigned(ADDR_D) + '1';
                             if (ADDR_D = "11") then
                                 DEC_Q(5 downto 4) <= "00";
                             end if;
                         else
                             DEC_Q(5 downto 4) <= ADDR_D;
                         end if;
                     end if;
                 else
                     DEC_Q(5 downto 4)  <= ADDR_D;
                     DEC_Q(3 downto 2)  <= ADDR_A;
                     DEC_Q(1 downto 0)  <= unsigned(ADDR_B) + '1';
                 end if;
             else
                 DEC_Q <= "000000";
             end if;
         end if;
    end process;
end xilinx;
```

- Verilog

```
    ////////////////////////////////////
   // NESTED_IF.V                    //
  // Nested If vs. Case Design Example  //
 // August 1997                      //
////////////////////////////////////

module nested_if (ADDR_A, ADDR_B, ADDR_C, ADDR_D, RESET, CLK, DEC_Q);

    input  [1:0]  ADDR_A ;
    input  [1:0]  ADDR_B ;
    input  [1:0]  ADDR_C ;
    input  [1:0]  ADDR_D ;
    input         RESET, CLK ;
    output [5:0]  DEC_Q ;

    reg    [5:0]  DEC_Q ;

    //  Nested If Process  //
    always @ (posedge CLK)
        begin
        if (RESET == 1'b1)
            begin
            if (ADDR_A == 2'b00)
                begin
                DEC_Q[5:4] <= ADDR_D;
                DEC_Q[3:2] <= 2'b01;
                DEC_Q[1:0] <= 2'b00;
                if (ADDR_B == 2'b01)
                    begin
                    DEC_Q[3:2] <= ADDR_A + 1'b1;
                    DEC_Q[1:0] <= ADDR_B + 1'b1;
                    if (ADDR_C == 2'b10)
                        begin
                        DEC_Q[5:4] <= ADDR_D + 1'b1;
                        if (ADDR_D == 2'b11)
                            DEC_Q[5:4] <= 2'b00;
                        end
                    else
                        DEC_Q[5:4] <= ADDR_D;
                    end
                end
            else
                DEC_Q[5:4] <= ADDR_D;
                DEC_Q[3:2] <= ADDR_A;
                DEC_Q[1:0] <= ADDR_B + 1'b1;
```

```
        end
    else
        DEC_Q <= 6'b000000;
    end

endmodule
```

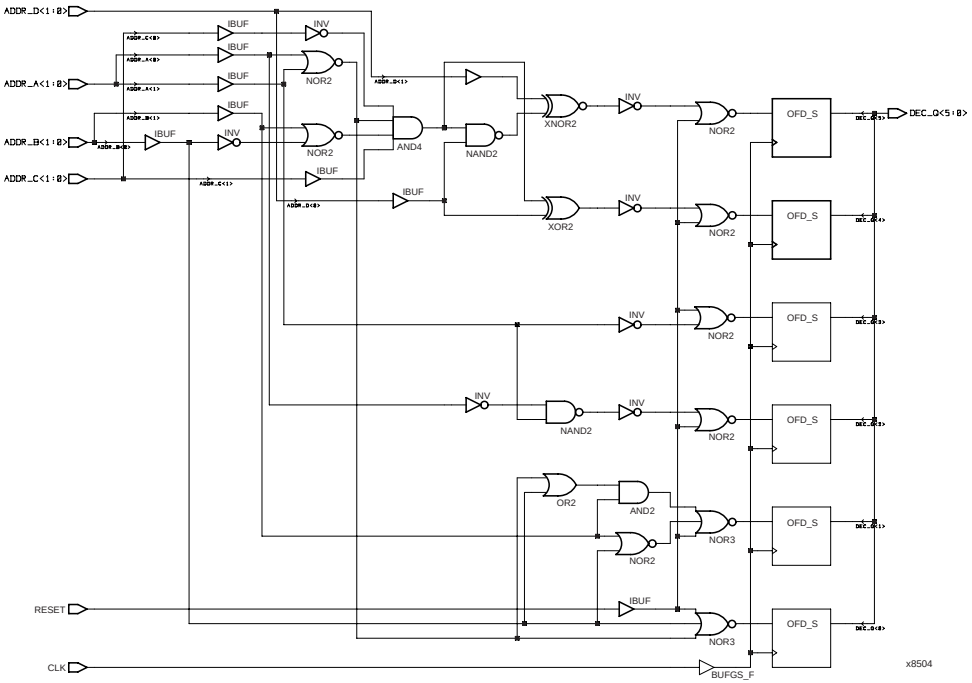

Figure 2-10 Implementation of Nested If

Nested If Example Modified to Use If-Case

- VHDL

```
-- IF_CASE.VHD
-- May 1997

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity if_case is
```

```
     port (ADDR_A:   in std_logic_vector (1 downto 0); -- ADDRESS Code
           ADDR_B:   in std_logic_vector (1 downto 0); -- ADDRESS Code
           ADDR_C:   in std_logic_vector (1 downto 0); -- ADDRESS Code
           ADDR_D:   in std_logic_vector (1 downto 0); -- ADDRESS Code
           RESET:    in std_logic;
           CLK :     in std_logic;
           DEC_Q:   out std_logic_vector (5 downto 0)); -- Decode OUTPUT
end if_case;

architecture xilinx of if_case is
signal ADDR_ALL : std_logic_vector (7 downto 0);
begin

----concatenate all address lines ----------------------
ADDR_ALL <= (ADDR_A & ADDR_B & ADDR_C & ADDR_D) ;

--------Use 'case' instead of 'nested_if' for efficient gate netlist------
    IF_CASE: process (CLK)
    begin
        if (CLK'event and CLK = '1') then
            if (RESET = '0') then
                case ADDR_ALL is
                    when "00011011" =>
                        DEC_Q(5 downto 4) <= "00";
                        DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
                        DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                    when "000110--" =>
                        DEC_Q(5 downto 4) <= unsigned(ADDR_D) + '1';
                        DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
                        DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                    when "0001----" =>
                        DEC_Q(5 downto 4) <= ADDR_D;
                        DEC_Q(3 downto 2) <= unsigned(ADDR_A) + '1';
                        DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                    when "00------" =>
                        DEC_Q(5 downto 4) <= ADDR_D;
                        DEC_Q(3 downto 2) <= "01";
                        DEC_Q(1 downto 0) <= "00";
                    when others     =>
                        DEC_Q(5 downto 4) <= ADDR_D;
                        DEC_Q(3 downto 2) <= ADDR_A;
                        DEC_Q(1 downto 0) <= unsigned(ADDR_B) + '1';
                end case;
            else
                DEC_Q <= "000000";
```

```
            end if;
        end if;
    end process;
end xilinx;
```

- Verilog

```
    ////////////////////////////////////
   // IF_CASE.V                        //
  // Nested If vs. Case Design Example  //
 // August 1997                      //
////////////////////////////////////

module if_case (ADDR_A, ADDR_B, ADDR_C, ADDR_D, RESET, CLK, DEC_Q);

    input  [1:0]  ADDR_A ;
    input  [1:0]  ADDR_B ;
    input  [1:0]  ADDR_C ;
    input  [1:0]  ADDR_D ;
    input         RESET, CLK ;
    output [5:0]  DEC_Q ;

    wire   [7:0]  ADDR_ALL ;
    reg    [5:0]  DEC_Q ;

    // Concatenate all address lines //
    assign ADDR_ALL = {ADDR_A, ADDR_B, ADDR_C, ADDR_D} ;

    // Use 'case' instead of 'nested_if' for efficient gate netlist //
    always @ (posedge CLK)
        begin
        if (RESET == 1'b1)
            begin
                casex (ADDR_ALL)
                    8'b00011011: begin
                                 DEC_Q[5:4] <= 2'b00;
                                 DEC_Q[3:2] <= ADDR_A + 1;
                                 DEC_Q[1:0] <= ADDR_B + 1'b1;
                                 end
                    8'b000110xx: begin
                                 DEC_Q[5:4] <= ADDR_D + 1'b1;
                                 DEC_Q[3:2] <= ADDR_A + 1'b1;
                                 DEC_Q[1:0] <= ADDR_B + 1'b1;
                                 end
                    8'b0001xxxx: begin
                                 DEC_Q[5:4] <= ADDR_D;
```

```
                                DEC_Q[3:2] <= ADDR_A + 1'b1;
                                DEC_Q[1:0] <= ADDR_B + 1'b1;
                                end
                    8'b00xxxxxx: begin
                                DEC_Q[5:4] <= ADDR_D;
                                DEC_Q[3:2] <= 2'b01;
                                DEC_Q[1:0] <= 2'b00;
                                end
                    default:    begin
                                DEC_Q[5:4] <= ADDR_D;
                                DEC_Q[3:2] <= ADDR_A;
                                DEC_Q[1:0] <= ADDR_B + 1'b1;
                                end
                endcase
            end
        else
            DEC_Q <= 6'b000000;
        end

endmodule
```

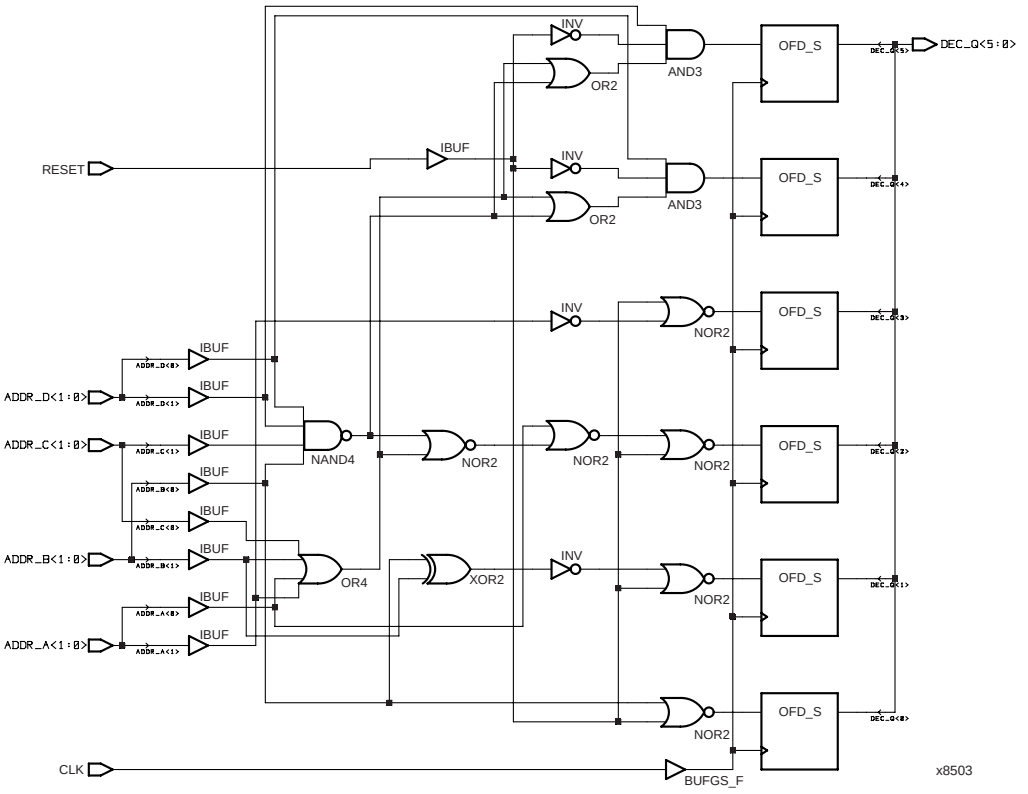

Figure 2-11 Implementation of If-Case

Comparing If Statement and Case Statement

The If statement produces priority-encoded logic and the Case statement creates parallel 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.

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

```
-- 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

```
     // 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
```

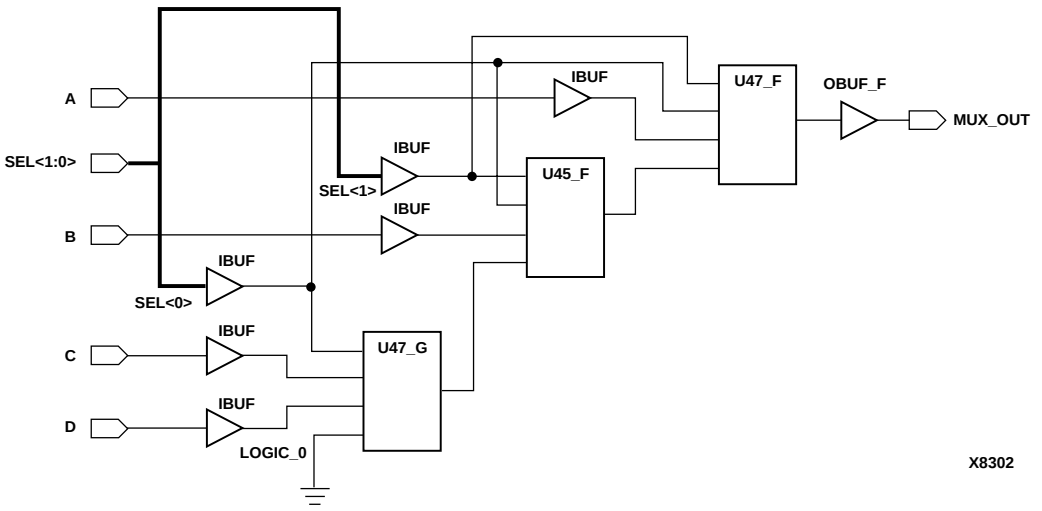

Figure 2-12 If_Ex Implementation

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 (using the Synopsys FPGA compiler). In this case,

design the multiplexer using the Case construct because fewer resources are used and the delay path is shorter.

4-to-1 Multiplexer Design with Case Construct

- VHDL

```
-- 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

```
      ////////////////////////////////////////
     // 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
```

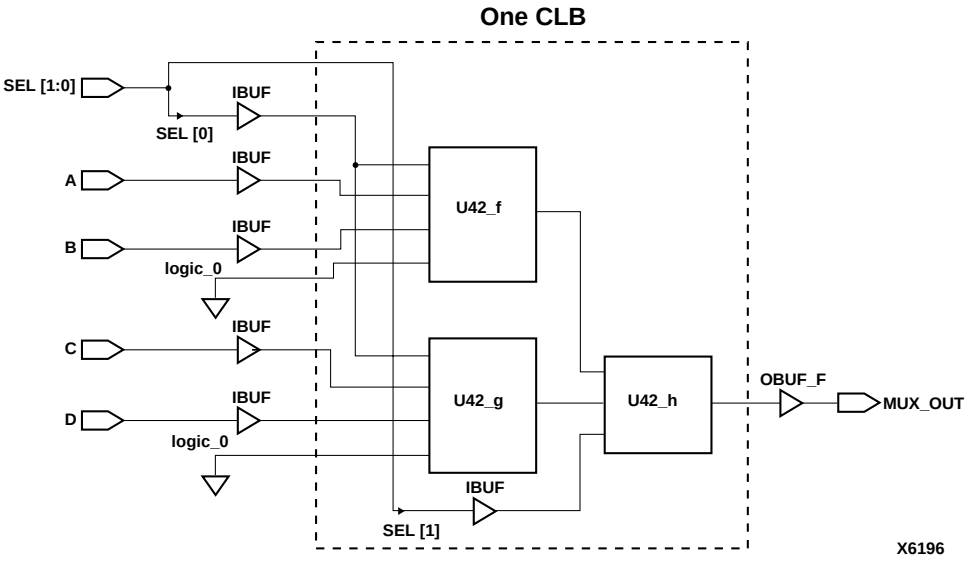

Figure 2-13 Case_Ex Implementation

Chapter 3

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.

- "Design Flow"
- "Entering your Design and Selecting Hierarchy"
- "Functional Simulation of your Design"
- "Synthesizing and Optimizing your Design"
- "Setting Timing 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

The steps in the design flow are shown in the "Design Flow" figure. The "Design Flow Using the Design Manager" figure and the "Design Flow using the Command Line" figure show the commands used and the files created with the Xilinx Design Manager and the command line, respectively.

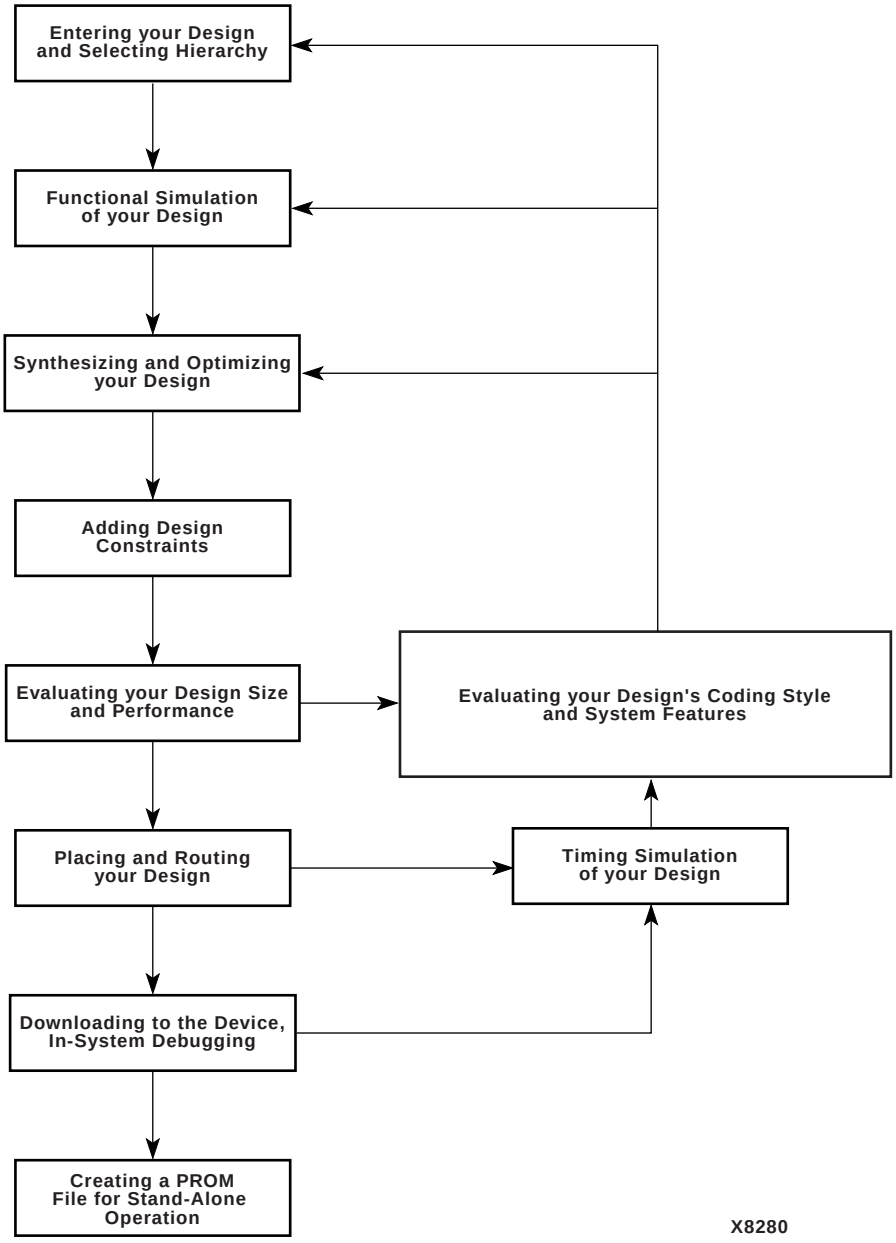

Figure 3-1 Design Flow

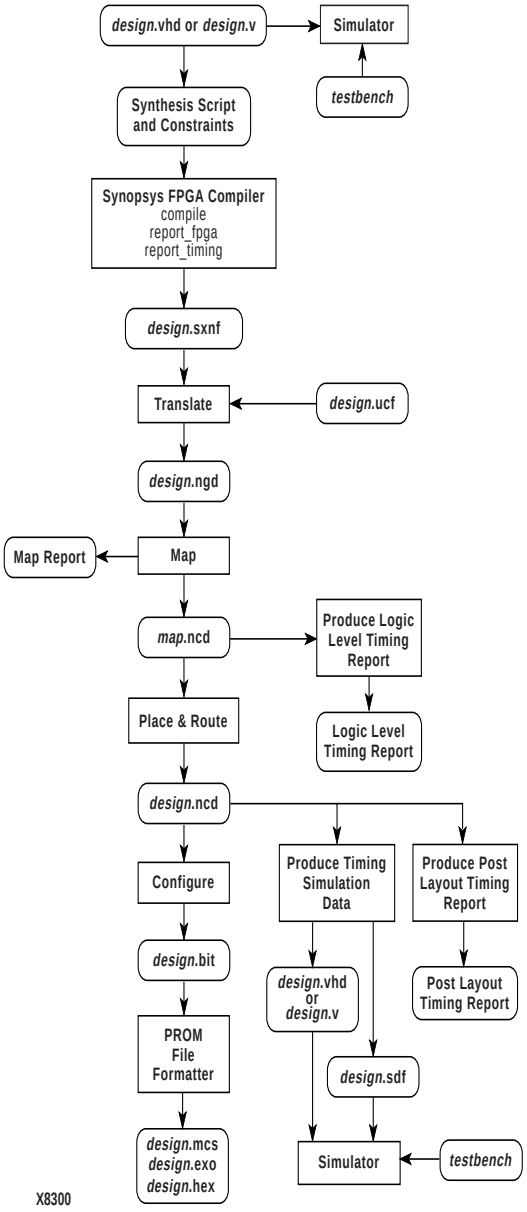

Figure 3-2 Design Flow Using the Design Manager

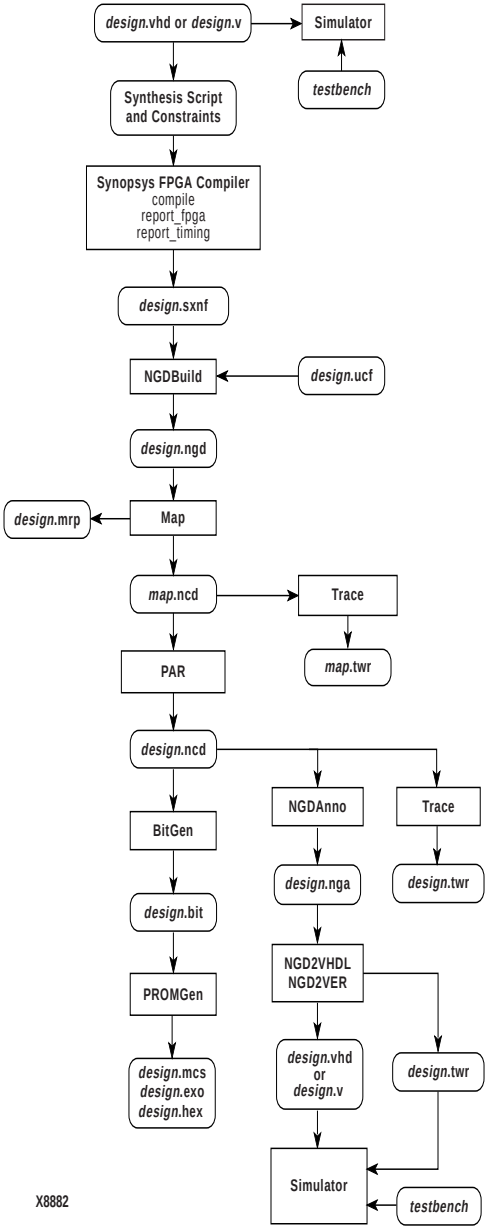

Figure 3-3 Design Flow using the Command Line

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.

Use RTL Code

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

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

Carefully Select Design Hierarchy

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

- Improves simulation and synthesis results
- Modular designs are easier to debug and modify
- 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 the Synopsys compiler documentation for more information on compilation options and suggestions.

Creating a .synopsys_dc.setup File

Before you can compile your design, you must create a .synopsys_dc.setup file to specify compiler defaults and to point to the applicable implementation libraries.

To create a .synopsys_dc.setup file, perform the following steps.

1. Copy the template.synopsys_dc.setup_fc file located in $XILINX/synopsys/examples to your project directory, and rename it .synopsys_dc.setup:

   cd *project_directory*

   cp $XILINX/synopsys/examples/
   template.synopsys_dc.setup_fc .synopsys_dc.setup

   The setup file uses the $XILINX and $SYNOPSYS environment variables to set up the correct search path for the libraries. To use this file, these environment variables must be defined on your system.

2. You can append the applicable device libraries to the .synopsys_dc.setup file using the synlibs command:

   synlibs –fc *target_device* **>> .synopsys_dc.setup**

Creating a Compile Run Script

The next step is creating a compile run script. However, if you are new to the Synopsys tools, you may want to use the graphical Design Analyzer to compile your design instead of using a run script. Because the iterative design compilation process can be tedious with the graphical interface, you may want to use a compile run script to speed up the design process.

The template.fpga.script.4kex sample run script is provided in $XILINX/synopsys/examples. You can copy this file into your project directory and modify it for your design. To copy this file:

cd *project_directory*

cp $XILINX/synopsys/examples/template.fpga.script.4kex *design*.**script**

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

For large designs, you may want to compile your design in increments to decrease run times. Compile some or all levels of hierarchy with separate compile commands and save the output files as .db files. When the top level code is synthesized, you can read in the compiled .db files.

Saving your Compiled Design as an SXNF File

After your design is successfully compiled, write out a Synopsys XNF file (.sxnf) for input to the Xilinx software. Make sure this file has the .sxnf extension. If it does not have this extension, it may not be correctly identified by the M1 translation process, and may result in unexpanded block errors.

Setting Timing Constraints

You can define timing specifications for your design in the User Constraints File (UCF). 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. For more information on timing specifications in the UCF file, refer to the *Quickstart Guide for Xilinx Alliance Series*, the *Libraries Guide*, and the Answers Database on the Xilinx Web site (http://www.xilinx.com).

Naming and Grouping Signals Together

You can name and group signals with TNMs (Timing Names) or with TIMEGRPs (Time groups). TNMs and TIMEGRPs are placed on these start and endpoints: ports, registers, latches, or synchronous RAMs. The new specification, TPSYNC, allows you to define an asynchronous node for a timing specification.

TNMs

Timing Names are used to identify a port, register, latch, RAM or groups of these components for timing specifications. TNMs are specified from a UCF with the following syntax.

`INST` *Instance_Name* `TNM=`*TNM_Name*;

Instance_Name is the name given to the port, register, latch or RAM in your design. The instance names for any port or instantiated component are provided by you in your HDL code. Inferred flip-flops and latch names may usually be determined from the Synopsys log files. TNM_Name is the arbitrary name you give the timing group.

You can include several of these statements in the UCF file with a common TNM_NAME to group elements for a timing specification as follows.

```
NET DATA TNM=INPUT_PORTS;

NET SELECT TNM=INPUT_PORTS;
```

The above example takes two ports, DATA and SELECT, and gives them the common timing name INPUT_PORTS.

TIMEGRPs

Time Groups are another method for specifying a group of components for timing specifications.

Time groups use existing TNMs or TIMEGRPs to create new groups or to define new groups based on the output net that the group sources. There are several methods to create TIMEGRPs in the UCF file, as follows.

```
TIMEGRP TIMEGRP_Name=TNM1:TNM2;
```

```
TIMEGRP TIMEGRP_Name=TNM3:EXCEPT:TNM_4;
```

The Xilinx software recognizes the following global timing names.

- FFS—All flips-flop in your design
- PADS—All external ports in your design
- RAMS—All synchronous RAMs in your design
- LATCHES—All Latches in your design

The following time group specifies the group name, FAST_FFS, which consists of all flip-flops in your design except for the ones with the TNM or TIMEGRP SLOW_FFS attribute.

```
TIMEGRP FAST_FFS=FFS:EXCEPT:SLOW_FFS;
```

TPSYNC Specification

In the latest version of the Xilinx software, you can define any node as a source or destination for a timing specification with the TPSYNC keyword. In synthesis designs, it is usually difficult to identify the net names for asynchronous paths of inferred logic. These net names can change from compile to compile, so it is not recommended to use this specification with inferred logic. However, with instantiated logic, the declared SIGNAL or WIRE name usually remains intact in the netlist and does not change from compile to compile. The UCF syntax is as follows.

```
NET Net_Name TPSYNC=TPSYNC_Name;
```

In the following NET statement, the TPSYNC is attached to the output net of a 3-state buffer, BUS3STATE. If a TPSYNC is attached to a net, then the source of the net is considered to be the endpoint (in

this case, the 3-state buffer itself). The subsequent TIMESPEC statement can use the TPSYNC name just as it uses a TNM name.

```
NET BUS3STATE TPSYNC=bus3;

TIMESPEC TSNewSpc3=FROM:PAD(ENABLE_BUS):TO:bus3:20ns;
```

Specifying Timing Constraints

Once your design signals are specified with TNMs, TIMEGRPs, or global timing names, you can place a specification on the design paths. There are a few methods for specifying these timing paths and different specifications have different priorities.

Period Constraint

The Period constraint specifies a clock period or clock speed on a net or clock port. The Xilinx tools attempt to meet all Pad to Setup requirements, as well as all Clock to Setup delays for registers clocked by the specified clock net. This is equivalent to the create_clock command from a Synopsys script. Following are the two methods for specifying a period constraint.

NET *Clock_Name* PERIOD = *Clock_Period* ;

or

NET *Clock_Name* TNM=*TNM_Name*;

TIMESPEC *TIMESPEC_Name* = PERIOD *TNM_Name Clock_Period*;

The following example specifies that the CLOCK port has a period of 50nS. All input paths to flip-flops clocked with this port are designated to operate at 50nS.

```
NET CLOCK PERIOD = 50;
```

FROM:TO Style Constraint

Specific paths can be specified with a FROM:TO style timing specification. These constraints are specified using global timing names, TNMs, TIMEGRPs, or TPSYNCs to connect the source and destination of the timing path, as well as the desired maximum delay of the path. The equivalent Synopsys command is set_max_delay. A UCF example is as follows.

TIMESPEC *TIMESPEC_Name* =

FROM:*Source_Name*:TO:*Desination_Name*:*Delay_Value*;

TIMESEPC_Name is specified with the TS identifier followed by a number, such as TS01.

The following example specifies a new timespec with the identifier TS01 so that all paths that are sourced by a port and end at a register grouped with the name DATA_FLOPS have a delay less than 30nS.

```
TIMESPEC TS01 = FROM:PADS:TO:DATA_FLOPS:30;
```

Offset Constraint

The OFFSET constraint is applied to a port defined in your code. It defines the delay of a signal relative to a clock, and is only valid for registered data paths. The OFFSET constraint specifies the signal delay external to the chip, allowing the implementation tools to automatically adjust relevant internal delays (CLK buffer and distribution delays) to accommodate the external delay specified with this constraint. This constraint is the equivalent of the set_input_delay and set_output_delay in Synopsys.

NET *Port_Name* OFFSET = {IN | OUT} *Time* {BEFORE | AFTER} *Clock_Name* ;

IN | OUT specifies that the offset is calculated with respect to an input IOB or an output IOB.

For a bidirectional IOB, the IN | OUT syntax lets you specify the flow of data (input or output) on the IOB. BEFORE | AFTER indicates whether data is to arrive (input) or leave (output) the device before or after the clock input.

The following example specifies that the data on the output port, DATA_OUT, arrive on the output pin 20nS after the edge of the clock signal, CLOCK, arrives.

```
NET DATA_OUT OFFSET = OUT 20 AFTER CLOCK;
```

Ignoring Timing Paths

When a timespec is issued for a path that is not timing-critical, you can specify to ignore this path for one or all timing specifications. A TIG (Timing IGnore) can be specified on these particular nets. The Synopsys equivalent is set_false_path. The UCF syntax is as follows.

NET *Signal_Name* TIG=*TIMESPEC_Name* ;

To ignore all timing constraints for a signal:

```
NET Signal_Name TIG;
```

To ignore an entire timing constraint:

```
TIG=TIMESPEC_Name;
```

In the following example, the SLOW_PATH net is set to ignore the timing constraint with the name TS01.

```
NET SLOW_PATH TIG=TS01;
```

Controlling Signal Skew

You can control the maximum allowed skew in your designs. The maximum skew (MAXSKEW) is the difference between the longest and shortest driver-to-load connection delays for a given net. The maximum and minimum delays are determined using worst case maximum delay values for each path. While this specification can not guarantee that this maximum skew value is achieved in the actual device, it allows the software to minimize the amount of skew on the specified signal. This specification is useful for high-fanout nets when all available global buffers have been used for other critical signals. An example of the UCF syntax for this specification follows.

```
NET Signal_Name MAXSKEW=Skew_Value ;
```

The following example specifies that the CLOCK_ENABLE signal, should not have a skew value greater than 4nS.

```
NET CLOCK_ENABLE MAXSKEW=4;
```

Timing Constraint Priority

Timing constraints can be assigned priorities when paths are overlapped by multiple timing constraints. Priorities can be directly specified to a timing constraint as follows.

```
TIMESPEC TIMESPEC_Name = FROM Group1 TO Group2
Delay_Value PRIORITY Priority_Level;
```

The lower the priority_level, the higher the precedence.

The following example sets a timespec where the source is a time group labeled THESE_FFS and the destination is labeled THOSE_FFS, with a delay value of 25nS and a priority level of 2.

```
TIMESPEC TS04=FROM THESE_FFS TO THOSE_FFS 25
PRIORITY 2;
```

Layout constraints also have an inherent precedence that is based on the type of constraint and the site description provided to the tools. If two constraints are of the same priority and cover the same path, then the last constraint in the constraint file overrides any other constraints that overlap.

Layout constraint priority is shown in the following table.

Table 3-1 Precedence of Constraints

Across Constraint Sources	
Highest Priority	Physical Constraint File (PCF)
	User Constraint File (UCF)
Lowest Priority	Input Netlist / Netlist Constraint File (NCF)
Within Constraint Sources	
Highest Priority	TIG (Timing Ignore)
	FROM:USER1:THRU:USER_T:TO:USER2 Specification (USER1 and USER2 are user-defined groups)
	FROM:USER1:THRU:USER_T:TO:FFS Specification or FROM:FFS:THRU:USER_T:TO:USER2 Specification (FFS is any pre-defined group)
	FROM:FFS:THRU:USER_T:TO:FFS Specification
	FROM:USER1:TO:USER2 Specification
	FROM:USER1:TO:FFS Specification or FROM:FFS:TO:USER2 Specification
	FROM:FFS:TO:FFS specification

Table 3-1 Precedence of Constraints

	Period specification
Lowest Priority	"Allpaths" type constraints

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 the FPGA Compiler reporting options. After your design is mapped, 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. Also, at this point, it is important to verify that your design will perform as specified.

Using FPGA Compiler to Estimate Device Utilization and Performance

Use the FPGA Compiler reporting options to estimate device utilization and performance.

Using the Report_fpga Command

After compiling, use the following command to obtain a report of device resource utilization.

```
report_fpga
```

This report lists the compiled cells in your design and how your design is mapped in the FPGA. These reports are generally accurate for the XC4000 and Spartan family because the Synopsys tools create the logic from your code and map your design into the FPGA. However, these reports are not as accurate if you are targeting devices other than the XC4000 family; if you are using the Design Compiler; or if you are not using the mapping feature of the FPGA Compiler. Also, any instantiated components, such as LogiBLOX modules, EDIF files, XNF files, or other components that Synopsys does not recognize during compilation, are not included in the report file. If

you include these type of 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.

The following is a sample report file generated by the report_fpga command.

```
****************************************
Report : fpga
Design : demo_board
Version: 1997.08
Date   : Tue Jan 27 10:53:02 1998
****************************************

  Xilinx FPGA Design Statistics
  -----------------------------

    FG Function Generators:           55
    H Function Generators:             4
    Number of CLB cells:              38
    Number of Hard Macros and
        Other Cells:                  10
    Number of CLBs in
        Other Cells:                   0
    Total Number of CLBs:             38

    Number of Ports:                  30
    Number of Clock Pads:              0
    Number of IOBs:                   30

    Number of Flip Flops:             28
    Number of 3-State Buffers:         0

    Total Number of Cells:            78
```

Using the Report_timing Command

Use the following command to obtain a timing report with estimated data path delays.

```
report_timing
```

This 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 modules, EDIF files, XNF files, or other components that Synopsys does not recognize during compilation.

The following is a sample report file generated by the report_timing command.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : demo_board
Version: 1997.08
Date   : Tue Jan 27 10:53:02 1998
****************************************

Operating Conditions: WCCOM   Library: xprim_4003e-2
Wire Loading Model Mode: top

Design           Wire Loading Model      Library
------------------------------------------------
demo_board             4003e-2_avg       xprim_4003e-2


  Startpoint: control_logic/U141
              (falling edge-triggered flip-flop)
  Endpoint: SEG7_A<1> (output port)
  Path Group: (none)
  Path Type: max

  Point                                    Incr       Path
  -----------------------------------------------------------
  control_logic/U141/K (clb_4000)          0.00       0.00 f
  control_logic/U141/YQ (clb_4000)         6.19       6.19 f
  control_logic/SEG7_IN<0> (control)       0.00       6.19 f
  s7d1/Q<0> (seg7decoder_0)                0.00       6.19 f
  s7d1/U80/X (clb_4000)                    2.68       8.87 r
  s7d1/U78/PAD (iob_4000)                  7.30      16.17 r
  s7d1/A (seg7decoder_0)                   0.00      16.17 r
  SEG7_A<1> (out)                          0.00      16.17 r
  data arrival time                                  16.17
```

```
-------------------------------------------------------
(Path is unconstrained)
```

Determining Actual Device Utilization and Pre-routed Performance

To determine if your design fits the specified device, you must map it with the Map program. The generated report file *design_name*.mrp contains the implemented device utilization information. 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 Guide.*

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

   `xilinx`

2. To create a new project, select the .sxnf file generated by the FPGA Compiler 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 FPGA Compiler designs. Also, do not use the guided map option with synthesis 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 on 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.

Using the Command Line to Map Your Design

1. Translate your design as follows.

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

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 Device Summary section of a Map report.

```
Design Summary
--------------
   Number of errors:        0
   Number of warnings:      3
   Number of CLBs:             39 out of   100   39%
      CLB Flip Flops:      32
      4 input LUTs:        66
      3 input LUTs:         5
   Number of bonded IOBs:      30 out of    61   49%
      IOB Flops:            0
      IOB Latches:          0
   Number of secondary CLKs:    1 out of     4   25%
   Number of oscillators:       1
```

```
                        Number of STARTUPs:          1
                        Number of READCLKs:          1
                        Number of READBACKs:         1
                        Number of MD0 pads:          1
                        Number of MD1 pads:          1
                    Total equivalent gate count for design: 1538
                    Additional JTAG gate count for IOBs:    1536
```

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 =  121.600 nS
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

- Reduce levels of logic to improve timing
- Redefine hierarchical boundaries to help the compiler optimize design logic
- Pipeline
- Logic replication
- Use of LogiBLOX or Coregen 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 it's own set of system features. Review the current version of *The Programmable Logic Data Book* for the system features available for the device you are targeting.

- Use global set / reset and global tri-state nets to reduce routing congestion and improve design performance

- Place the highest fanout signals on the global buffers
- Modify large multiplexes to use tri-state buffers
- Use one-hot encoding for large or complex state machines
- Use I/O registers where possible
- Use I/O decoders where possible

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 less-than-optimal placement and routing.

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

     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.

   - 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).

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. Once 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.

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 with 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 (not recommended)

This option is generally not recommended for synthesis-based designs. 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.

Timing Simulation of your Design

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.

Note: Refer to the "Simulating Your Design" chapter for more information on design simulation.

Creating a Timing 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 **Setup** → **Options** in the Flow Engine.

   The Options dialog box appears.

2. Select the Produce Timing Simulation Data button in the Optional Targets field.

3. Select the simulator you want to use from the pull down menu next to the Simulation Program Option Template field.

4. To make additional customizations, select the Edit Template button next to the Simulator pulldown menu.

5. Click **OK** in the Implementation Template dialog box.

6. Click **OK** in the Options dialog box.

7. 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 NGD2*XXX* 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**

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 and / 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 4

Designing FPGAs with HDL

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

- "Introduction"
- "Using Global Low-skew Clock Buffers"
- "Using Dedicated Global Set/Reset Resource"
- "Encoding State Machines"
- "Using Dedicated I/O Decoders"
- "Instantiating LogiBLOX Modules"
- "Implementing Memory"
- "Implementing Boundary Scan (JTAG 1149.1)"
- "Implementing Logic with IOBs"
- "Implementing Multiplexers with Tristate Buffers"
- "Using Pipelining"
- "Design Hierarchy"

Introduction

Xilinx FPGAs 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 XC4000 family, XC5200 family, and Spartan family FPGAs contain system-oriented features such as the following.

- Global low-skew clock or signal distribution network
- Wide edge decoders (XC4000 family only)

- On-chip RAM and ROM (XC4000 family and Spartan)
- 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.

Using Global Low-skew 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. When you use the Insert Pads command, the FPGA Compiler automatically inserts a BUFG generic clock buffer whenever an input signal drives a clock signal. 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. Each XC4000E/L and Spartan device contains four primary and four secondary global buffers that share the same routing resources. XC4000EX/XLA/XL/XV devices have sixteen global buffers; each buffer has its own routing resources. XC5200 devices have four dedicated global buffers in each corner of the device.

XC4000 EX/XLA/XL/XV devices have two different types of global buffer, Global Low-Skew Buffers (BUFGLS) and Global Early Buffers (BUFGE). Global Low-Skew buffers 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 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 Global early and Global Low-Skew Buffers share a single pad, a single IPAD can drive a BUFGE, BUFGLS or both in parallel. The parallel configuration is especially useful for clocking the fast capture latches of the device. Since the Global Early and Global Low-Skew Buffers share a common input, they cannot be driven by two unique 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 Global Low-Skew Buffer.
- If you want a faster clock path, use a BUFG. Initially, the software will try to use a Global Low-Skew Buffer. 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 Global Low-Skew buffer, use a Global Early Buffer.

Note: For more information on using the XC4000 EX / XLA / XL / XV device family global buffers, refer to the online *The Programmable Logic Data Book* or the Xilinx web site at http: / / www.xilinx.com.

For XC4000E / L and Spartan devices, you can use secondary global buffers (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_F cell. You can use primary global buffers (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.

XC4000E / L and 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.

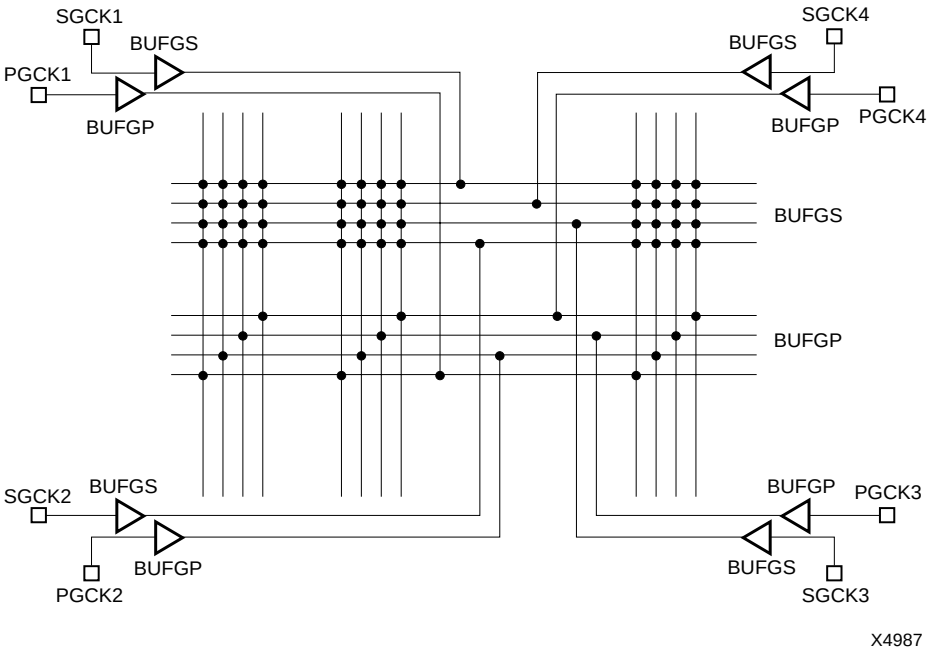

Figure 4-1 Global Buffer Routing Resources (XC4000E, Spartan)

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.

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.

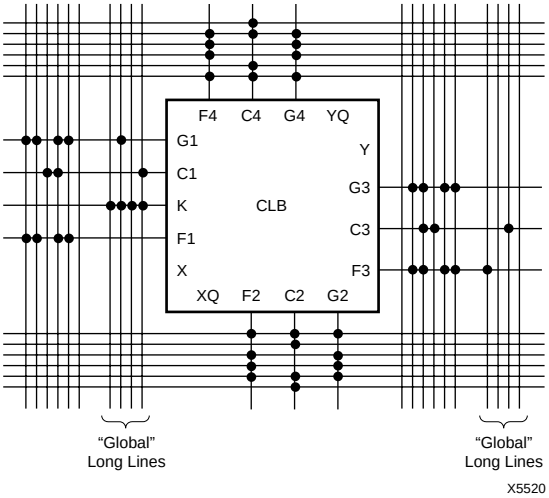

Figure 4-2 Global Longlines Resource CLB Connections

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

Note: Refer to the *Synopsys Interface Guide* for more information on inserting I/O buffers and clock buffers.

Synopsys tools automatically insert a secondary global clock buffer on all input ports that drive a register's clock pin or a gated clock signal. To disable the automatic insertion of clock buffers and specify the ports that should have a clock buffer, perform the following steps.

1. In the Synopsys Compiler, ports that drive gated clocks or a register's clock pin are assigned a clock attribute. Remove this attribute from ports tagged with the clock attribute by typing:

   set_pad_type –no_clock "*"

2. Assign a clock attribute to the input ports that should have a BUFGS as follows:

   set_pad_type –clock {*input ports***}**

3. Enter the following commands:

   set_port_is_pad "*"
   insert_pads

   The Insert Pads command causes the FPGA Compiler to automatically insert a clock buffer to ports tagged with a clock attribute.

To insert a global buffer other than a BUFGS, such as a BUFGP, use the following commands.

1. Use the following command on all ports with inferred buffers.

   set_port_is_pad "*"

2. Specify the buffer as follows.

   set_pad_type –exact BUFGP_F {*port_list***}**

   You can replace BUFGP_F with another supported buffer for the device you are targeting. Refer to the *Synopsys Interface Guide* for a list of supported buffers. Port_list specifies the port(s) for the buffer.

3. Generate the buffers for the device as follows.

   insert_pads

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 Synopsys script. If you do instantiate global buffers that are connected to a port, check your Synopsys script to make sure the Set Port Is Pad command is not specified for the buffer.

set_port_is_pad
{*list_of_all_ports_except_instantiated_buffer_port*}

insert_pads

or

set_port_is_pad “*”

remove_attribute {*instantiated_buffer_port***} port_is_pad**

insert_pads

Instantiating Buffers Driven from Internal Logic

You must instantiate a global buffer in your code in order 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 are examples of instantiating a BUFGS for an internal multiplexed clock circuit. A Set Dont Touch attribute is added to the instantiated component.

- VHDL

```
-----------------------------------------------
-- CLOCK_MUX.VHD                             --
-- This is an example of an instantiation of --
-- global buffer (BUFGS) from an internally  --
-- driven signal, a multiplexed clock.       --
-- July 1997                                 --
-----------------------------------------------
```

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

end XILINX;
```

- Verilog

```
     // CLOCK_MUX.V                                  //
    // This is an example of an instantiation of //
   // global buffer (BUFGS) from an internally  //
  // driven signal, a multipled clock.         //
 // September 1997                            //
//////////////////////////////////////////////

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)
    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

XC4000 and Spartan 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 effects every flip-flop in

your design, use the GSR net to increase design performance and reduce routing congestion.

The XC5200 family has a dedicated Global Reset (GR) net that resets all device registers. As in the XC4000 and Spartan devices, the STARTUP or STARTBUF (VHDL) block must be instantiated in your code in order to access this resource. The XC3000A devices also have dedicated Global Reset (GR) that is connected to a dedicated device pin (see device pinout). Since this resource is always active, you do not need to do anything to activate this feature.

For XC4000, Spartan, and XC5200 devices, the Global Set/Reset (GSR or GR) signal is, by default, set to active high (globally resets device when logic equals 1). For an active low reset, you can instantiate an inverter in your code to invert the global reset signal. The inverter is absorbed by the STARTUP block and does not use any device resources (function generators). Even though the inverted signal may be behaviorally described in your code, Xilinx recommends instantiating the inverter to prevent the mapping of the inverter into a CLB function generator, and subsequent delays to the reset signal and unnecessary use of device resources Also make sure you put a Don't Touch attribute on the instantiated inverter before compiling your design. If you do not add this attribute, the inverter may get mapped into a CLB function generator.

Note: For more information on simulating the Global Set/Reset, see the "Simulating Your Design" chapter.

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. Since 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: For the XC3000 family and the XC5200 family, all flip-flops and latches are reset to zero after configuration.

Preset vs. Clear (XC4000, Spartan)

The XC4000 family flip-flops are configured as either preset (asynchronous set) or clear (asynchronous reset). 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 flip-flop from the Synopsys run script, as follows.

```
set_attribute "cell" fpga_xilinx_init_state –
type string "S"
```

I/O flip-flops and latches 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=S attribute.

Note: Refer to the *Synopsys Interface Guide* for information on changing the initial state of registers that do not use the Preset or Clear pins.

Increasing Performance with the GSR/GR 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/GR 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/GR net from HDL code. To utilize the GSR net, you must instantiate the STARTUP or STARTBUF block (VHDL), as shown in the "No_GSR Implemented with Gates" figure.

Design Example without Dedicated GSR/GR 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.

- VHDL - No GSR

```
-- 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;
```

- VHDL - No GR

```
-- NO_GR.VHD Example
-- The signal RESET initializes all registers
-- Without the use of the dedicated Global Reset routing
-- December 1997

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

entity no_gr 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_gr;

architecture XILINX of no_gr 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 XILINX;
```

- Verilog - No GSR

```
/* NO_GSR Example
 * Synopsys HDL Synthesis Design Guide for FPGAs
 * 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
```

- Verilog - No GR

```
/* NO_GR.V Example
 * The signal RESET initializes all registers
```

```
 * Aug 1997 */

module no_gr ( 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
```

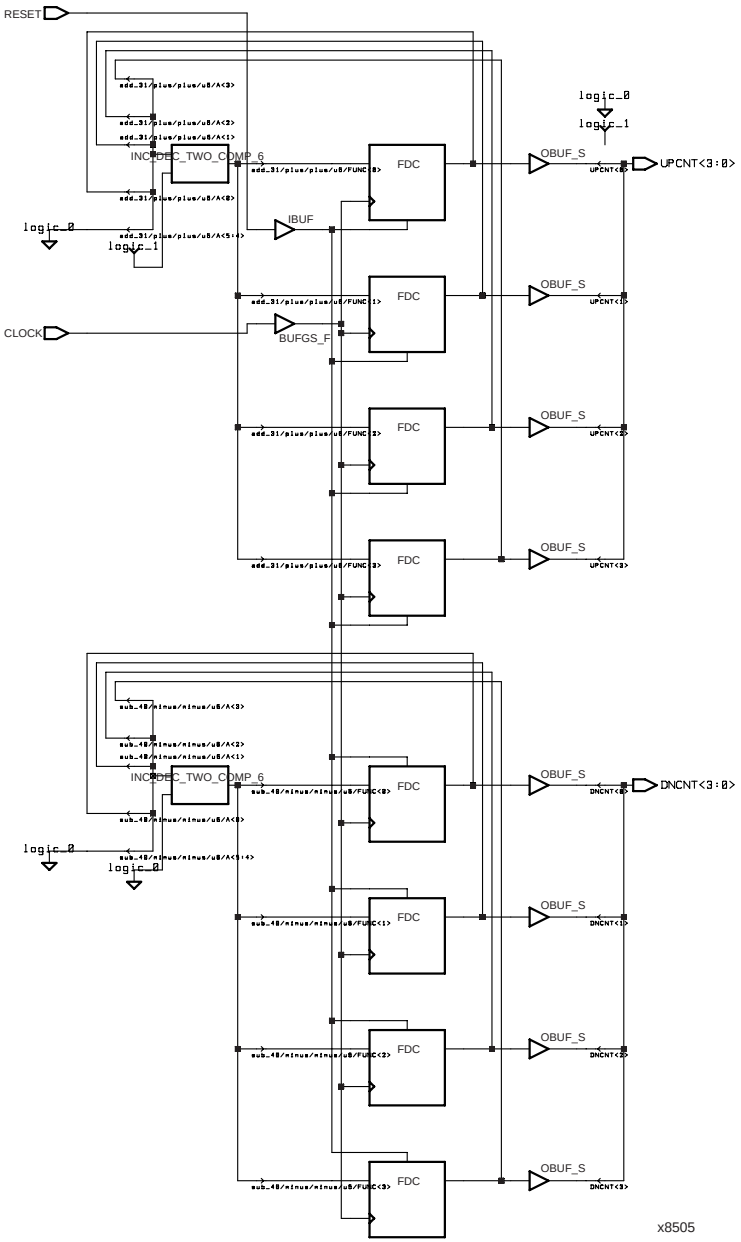

Figure 4-3 No_GSR Implemented with Gates

Design Example with Dedicated GSR/GR Resource

To reduce routing congestion and improve the overall performance of the reset net in the No_GSR and No_GR designs, use the dedicated GSR or GR net instead of the general purpose routing. Instantiate the STARTUP or STARTBUF block in your design and use the GSR or GR pin on the STARTUP block (or the GSRIN pin on the STARTBUF block) to access the global reset net. The modified designs (Use_GSR and Use_GR) are included at the end of this section. The Use_GSR design implemented with gates is shown in "Active_Low_GSR Implemented with Gates" figure.

In XC4000 and Spartan designs, 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=S attribute to the flip-flop (described later in this section) from the Synopsys run script.

In XC5200 family designs, the GR resets all flip-flops in the device to a logic zero. If a flip-flop is described as asynchronous preset to a logic 1, Synopsys automatically infers a flip-flop with a synchronous preset, and the M1 software puts an inverter on the input and output of the device to simulate a preset.

The Use_GSR and Use_GR designs explicitly state that the down-counter resets to all ones, therefore, asserting the reset net causes this counter to reset to a default of all zeros. You can use the INIT = S attribute to prevent this reset to zeros.

- Attach the INIT=S attribute to the down-counter flip-flops as follows:

  `set_attribute` *cell name* **`fpga_xilinx_init_state –type string "S"`**

Note: The "\" character represents a continuation marker.

This command allows you to override the default clear (or Low) state when your code does not specify a preset condition. However, since attributes are assigned outside the HDL code, the code no longer accurately represents the behavior of the design.

Note: Refer to the *Synopsys Interface Guide* for more information on assigning attributes.

The STARTUP block must not be optimized during the synthesis process. Add a Don't Touch attribute to the STARTUP or STARTBUF block before compiling the design as follows:

set_dont_touch *cell_instance_name*

- VHDL - Use_GSR (XC4000 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;
```

- VHDL - Use_GR

```
------------------------------------------------
-- USE_GR.VHD Version 1.0                      --
-- Xilinx HDL Synthesis Design Guide           --
-- The signal RESET initializes all registers --
-- Using the global reset resources since      --
-- STARTBUF block was added                    --
-- December 1997                               --
------------------------------------------------

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

entity use_gr 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_gr;

architecture XILINX of use_gr 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: process(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;
```

- Verilog - Use_GSR

```
//////////////////////////////////////////////
// USE_GSR.V Version 1.0                    //
// Xilinx HDL Synthesis Design Guide        //
// 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
```

- Verilog - Use_GR

```
      ////////////////////////////////////////////
     // USE_GR.V Version 1.0                     //
    // Xilinx HDL Synthesis Design Guide        //
   // The signal RESET initializes all registers //
  // Using the global reset resources since     //
 // STARTUP block instantiation was added      //
// December 1997                              //
////////////////////////////////////////////

module use_gr ( 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 (.GR(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
```

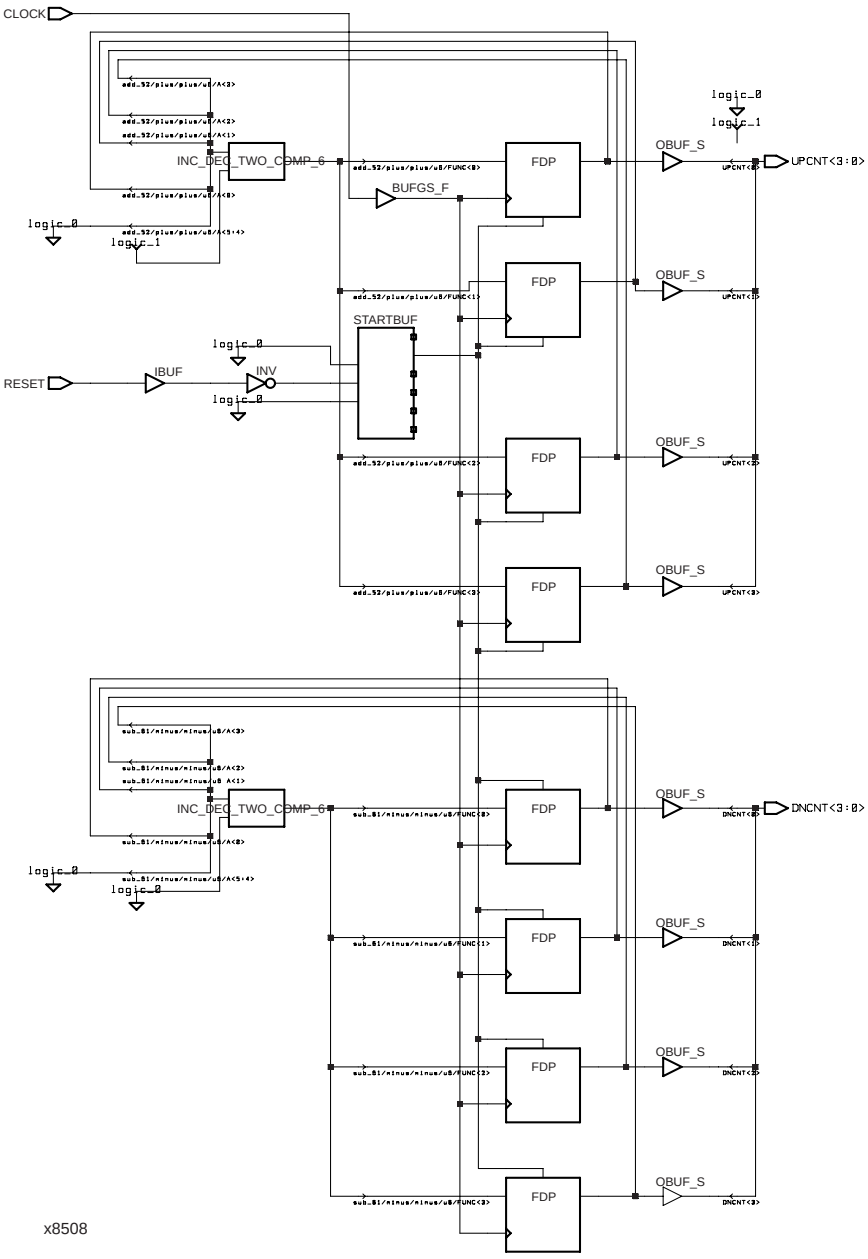

Figure 4-4 Active_Low_GSR Implemented with Gates

Design Example with Active Low GSR/GR 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_dont_touch attribute is added to the Synopsys 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. VHDL and Verilog Active_Low_GSR designs are shown following.

- VHDL - Active_Low_GSR

```
--------------------------------------------------
-- ACTIVE_LOW_GSR.VHD Version 1.0                --
-- Xilinx HDL Synthesis Design Guide             --
-- 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;
```

- Verilog - Active_low_GSR

```
     ////////////////////////////////////////////////
    // ACTIVE_LOW_GSR.V Version 1.0                 //
   // Xilinx HDL Synthesis Design Giude            //
  // The signal RESET is inverted before being     //
 // connected to the GSR pin of the STARTUP block  //
// The inverter will be absorbed by STARTUP in M1 //
```

```
   // Inverter is instantiated to avoid being mapped //
  // into a LUT by Synopsys                          //
 // 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
```

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 an identical 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.

Note: The bold text in each of the three examples indicates the portion of the code that varies depending on the method used to encode the state machine.

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.

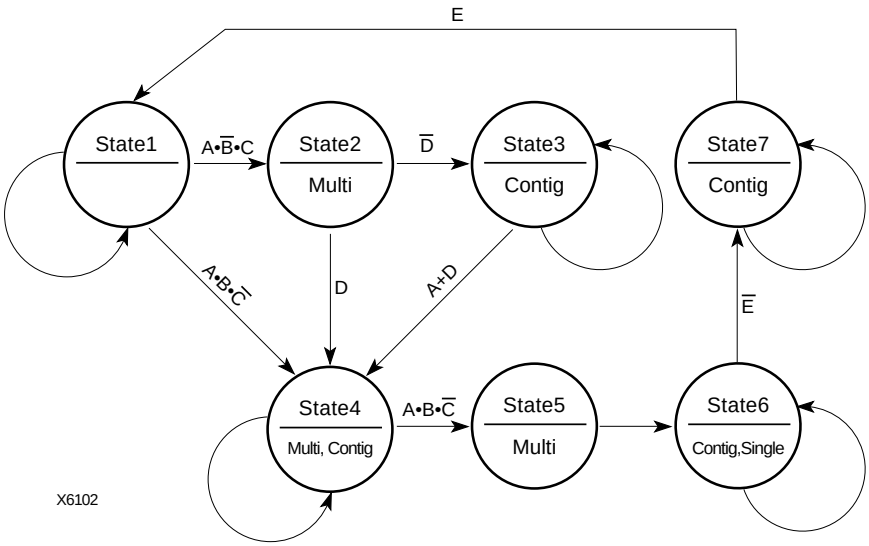

Figure 4-5 State Machine Bubble Diagram

VHDL - Binary Encoded State Machine Example

```
-------------------------------------------------
-- BINARY.VHD Version 1.0                       --
-- Example of a binary encoded state machine    --
-- Xilinx HDL Synthesis Design Guide for FPGAs --
-- 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;
```

Verilog - Binary Encoded State Machine Example

```
    ////////////////////////////////////////////////
   // BINARY.V Version 1.0                        //
  // Example of a binary encoded state machine   //
 // Xilinx HDL Synthesis Design Guide for FPGAs //
// 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] //synopsys enum STATE_TYPE
    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] /* synopsys enum STATE_TYPE */ CS;
reg [2:0] /* synopsys enum STATE_TYPE */ NS;

// synopsys 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) //synopsys full_case
            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. You can explicitly declare state vectors or you can allow the Synopsys tool to determine the vectors. Synopsys 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 the Synopsys compiler to select the encoding style that results in the lowest gate count when the design is synthesized.

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

VHDL- Enumerated Type Encoded State Machine 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';
   .
   .
   .
```

Verilog - Enumerated Type Encoded State Machine Example

```
     ////////////////////////////////////////////////
    // ENUM.V Version 1.0                          //
   // Example of an enumerated encoded state machine //
  // Xilinx HDL Synthesis Design Guide for FPGAs    //
 // 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] //synopsys enum STATE_TYPE
    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] /* synopsys enum STATE_TYPE */ CS;
reg [2:0] /* synopsys enum STATE_TYPE */ NS;

// synopsys 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) //synopsys full_case
           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

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.

Note: Refer to the previous VHDL and Verilog Binary Encoded State Machine examples for the complete Case statement portion of the code. See the "Accelerate FPGA Macros with One-Hot Approach" appendix for a detailed description of one-hot encoding and its applications.

VHDL - One-hot Encoded State Machine 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;
              .
              .
              .
```

Verilog - One-hot Encoded State Machine 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] //synopsys enum STATE_TYPE
    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] /* synopsys enum STATE_TYPE */ CS;
reg [2:0] /* synopsys enum STATE_TYPE */ NS;

// synopsys 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) //synopsys full_case
            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
        .
        .
        .
```

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, the Synopsys 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. The Synopsys enum.script file contains the commands you need to convert an enumerated types encoded state machine to a one-hot encoded state machine.

Note: Refer to the Synopsys 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 XC4005EPC84-2 device

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

Table 4-1 State Machine Encoding Styles Comparison (XC4005E-2)

Comparison	One-Hot	Binary	Enum (One-hot)
Occupied CLBs	6	9	6
CLB Flip-flops	6	3	7
PadToSetup	9.4 ns (3[a])	13.4 ns (4)	9.6 ns (3)
ClockToPad	15.1 ns (3)	15.1 ns (3)	14.9 ns (3)
ClockToSetup	13.0 ns (4)	13.9 ns (4)	10.1 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 Synopsys 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

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

- Verilog

```
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.

```
set_attribute "CS_reg<0>" \ fpga_xilinx_init_state –type string
"S"
```

Note: The "\" character in this command represents a continuation marker.

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).

Using Dedicated I/O Decoders

The periphery of XC4000 family devices has four wide decoder circuits at each edge. The inputs to each decoder are any of the IOB signals on that edge plus one local interconnect per CLB row or column. Each decoder generates a High output (using a pull-up resistor) when the AND condition of the selected inputs or their complements is true. The decoder outputs drive CLB inputs so they can be combined with other logic or can be routed directly to the chip outputs.

To implement XC4000 family edge decoders in HDL, you must instantiate edge decoder primitives. The primitive names you can use vary with the synthesis tool you are using. Using the Synopsys tools, you can instantiate the following primitives: DECODE1_IO, DECODE1_INT, DECODE4, DECODE8, and DECODE16. These primitives are implemented using the dedicated I/O edge decoders. The XC4000 family wide decoder outputs are effectively open-drain and require a pull-up resistor to take the output High when the specified pattern is detected on the decoder inputs. To attach the pull-up resistor to the output signal, you must instantiate a PULLUP component.

The following VHDL example shows how to use the I/O edge decoders by instantiating the decode primitives from the XSI library. Each decoder output is a function of ADR (IOB inputs) and CLB_INT (local interconnects). The AND function of each DECODE output and

Chip Select (CS) serves as the source of a flip-flop Clock Enable pin. The four edge decoders in this design are placed on the same device edge. The "Schematic Block Representation of I/O Decoder" figure shows the schematic block diagram representation of this I/O decoder design.

VHDL - Using Dedicated I/O Decoders Example

```
--Edge Decoder
--A XC4000 LCA has special decoder circuits at each edge. These decoders
--are open-drained wired-AND gates. When one or more of the inputs (I) are
--Low output(O) is Low. When all of the inputs are High, the output is --
--High.A pull-up resistor must be connected to the output node to achieve
--a true logic High.

Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity io_decoder is
  port (ADR: in std_logic_vector (4 downto 0);
        CS: in std_logic;
        DATA: in std_logic_vector (3 downto 0);
        CLOCK: in std_logic;
        QOUT: out std_logic_vector (3 downto 0));
end io_decoder;

architecture STRUCTURE of io_decoder is

COMPONENT DECODE1_IO
  PORT ( I: IN std_logic;
         O: OUT std_logic );
END COMPONENT;

COMPONENT DECODE1_INT
  PORT ( I: IN std_logic;
         O: OUT std_logic );
END COMPONENT;

COMPONENT DECODE4
  PORT ( A3, A2, A1, A0: IN std_logic;
         O: OUT std_logic );
END COMPONENT;

COMPONENT PULLUP
```

```
  PORT ( O: OUT std_logic );
END COMPONENT;

---- Internal Signal Declarations ----------------------
signal DECODE, CLKEN, CLB_INT: std_logic_vector (3 downto 0);
signal ADR_INV, CLB_INV: std_logic_vector (3 downto 0);
begin

ADR_INV <= not ADR (3 downto 0);
CLB_INV <= not CLB_INT;

----- Instantiation of Edge Decoder: Output "DECODE(0)" ---------------
    A0: DECODE4 port map (ADR(3), ADR(2), ADR(1), ADR_INV(0), DECODE(0));

    A1: DECODE1_IO port map (ADR(4), DECODE(0));

    A2: DECODE1_INT port map (CLB_INV(0), DECODE(0));

    A3: DECODE1_INT port map (CLB_INT(1), DECODE(0));

    A4: DECODE1_INT port map (CLB_INT(2), DECODE(0));

    A5: DECODE1_INT port map (CLB_INT(3), DECODE(0));

    A6: PULLUP port map (DECODE(0));

----- Instantiation of Edge Decoder: Output "DECODE(1)" ---------------
    B0: DECODE4 port map (ADR(3), ADR(2), ADR_INV(1), ADR(0), DECODE(1));

    B1: DECODE1_IO port map (ADR(4), DECODE(1));

    B2: DECODE1_INT port map (CLB_INT(0), DECODE(1));

    B3: DECODE1_INT port map (CLB_INV(1), DECODE(1));

    B4: DECODE1_INT port map (CLB_INT(2), DECODE(1));

    B5: DECODE1_INT port map (CLB_INT(3), DECODE(1));

    B6: PULLUP port map (DECODE(1));

----- Instantiation of Edge Decoder: Output "DECODE(2)" ---------------
    C0: DECODE4 port map (ADR(3), ADR_INV(2), ADR(1), ADR(0), DECODE(2));

    C1: DECODE1_IO port map (ADR(4), DECODE(2));
```

```
    C2: DECODE1_INT port map (CLB_INT(0), DECODE(2));

    C3: DECODE1_INT port map (CLB_INT(1), DECODE(2));

    C4: DECODE1_INT port map (CLB_INV(2), DECODE(2));

    C5: DECODE1_INT port map (CLB_INT(3), DECODE(2));

    C6: PULLUP port map (DECODE(2));

----- Instantiation of Edge Decoder: Output "DECODE(3)" ---------------
    D0: DECODE4 port map (ADR_INV(3), ADR(2), ADR(1), ADR(0), DECODE(3));

    D1: DECODE1_IO port map (ADR(4), DECODE(3));

    D2: DECODE1_INT port map (CLB_INT(0), DECODE(3));

    D3: DECODE1_INT port map (CLB_INT(1), DECODE(3));

    D4: DECODE1_INT port map (CLB_INT(2), DECODE(3));

    D5: DECODE1_INT port map (CLB_INV(3), DECODE(3));

    D6: PULLUP port map (DECODE(3));

-----CLKEN is the AND function of CS & DECODE--------

CLKEN(0) <= CS and DECODE(0);
CLKEN(1) <= CS and DECODE(1);
CLKEN(2) <= CS and DECODE(2);
CLKEN(3) <= CS and DECODE(3);

--------Internal 4-bit counter --------------
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
                CLB_INT <= CLB_INT + 1;
        end if;
    end process;

-------"QOUT(0)" Data Register Enabled by "CLKEN(0)"-----
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
```

```
          if (CLKEN(0) = '1') then
              QOUT(0) <= DATA(0);
          end if;
        end if;
    end process;

-------"QOUT(1)" Data Register Enabled by "CLKEN(1)"-----
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
          if (CLKEN(1) = '1') then
              QOUT(1) <= DATA(1);
          end if;
        end if;
    end process;

-------"QOUT(2)" Data Register Enabled by "CLKEN(2)"-----
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
          if (CLKEN(2) = '1') then
              QOUT(2) <= DATA(2);
          end if;
        end if;
    end process;

-------"QOUT(3)" Data Register Enabled by "CLKEN(3)"-----
    process (CLOCK)
        begin
        if (CLOCK'event and CLOCK='1') then
          if (CLKEN(3) = '1') then
              QOUT(3) <= DATA(3);
          end if;
        end if;
    end process;

end STRUCTURE;
```

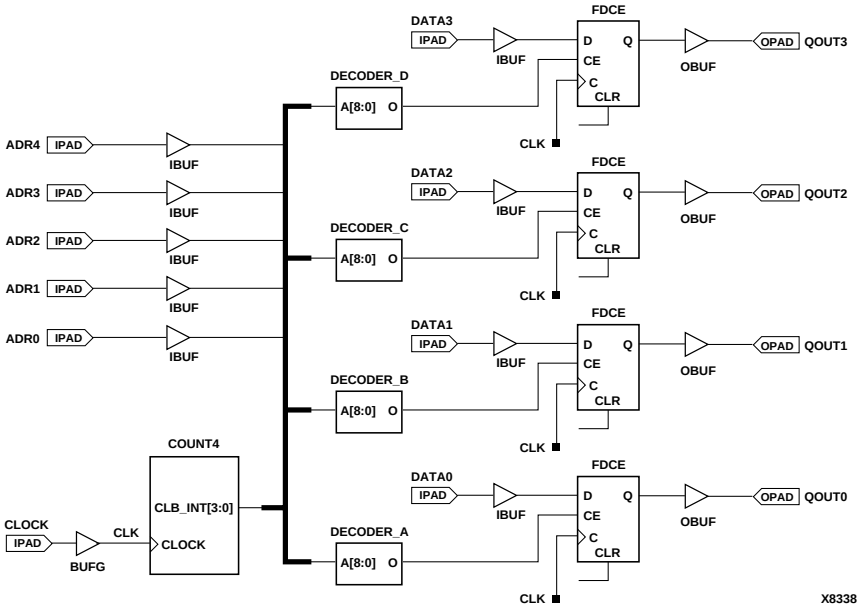

Figure 4-6 Schematic Block Representation of I/O Decoder

Note: In the previous figure, the pull-up resistors are inside the Decoder blocks.

Instantiating LogiBLOX Modules

Note: Refer to the *LogiBLOX Guide* for detailed instructions on using LogiBLOX.

Synopsys can infer arithmetic DesignWare modules from VHDL or Verilog code for these operators: +, –, <, <=, >, >=, =, +1, –1. These adders, subtracters, comparators, incrementers, and decrementers use FPGA dedicated device resources, such as carry logic, to improve the speed and area of designs. For bus widths greater than four, DesignWare modules are generally faster unless multiple instances of the same function are compiled together. For more information on the DesignWare libraries, refer to the *Synopsys Interface Guide.*

If you want to use a module that is not in the DesignWare libraries, or if you want better performance or area, you can use LogiBLOX to create components that can be instantiated in your code. A simulation

model is also created so that RTL simulation can be performed before your design is compiled.

LogiBLOX is a graphical tool that allows you to select from several arithmetic, logic, I/O, sequential, and data storage modules for inclusion in your HDL design. Use LogiBLOX to instantiate the modules listed in the following table.

Table 4-2 LogiBLOX Modules

Module	Description
Arithmetic	
Accumulator	Adds data to or subtracts it from the current value stored in the accumulator register
Adder/Subtracter	Adds or subtracts two data inputs and a carry input
Comparator	Compares the magnitude or equality of two values
Counter	Generates a sequence of count values
Logic	
Constant	Forces a constant value onto a bus
Decoder	Routes input data to 1-of-n lines on the output port
Multiplexer	Type 1, Type 2 - Routes input data on 1-of-n lines to the output port
Simple Gates	Type 1, Type 2, Type 3 - Implements the AND, INVERT, NAND, NOR, OR, XNOR, and XOR logic functions
Tristate	Creates a tri-stated internal data bus
I/O	
Bi-directional Input/Output	Connects internal and external pin signals
Pad	Simulates an input/output pad
Sequential	
Clock Divider	Generates a period that is a multiple of the clock input period
Counter	Generates a sequence of count values
Shift Register	Shifts the input data to the left of right
Storage	

Table 4-2 LogiBLOX Modules

Module	Description
Data Register	Captures the input data on active clock transitions
Memory: ROM, RAM, SYNC_RAM, DP_RAM	Stores information and makes it readable

Using LogiBLOX in HDL Designs

1. Before using LogiBLOX, verify the following.
   - Xilinx software is correctly installed
   - Environment variables are set correctly
   - Your display environment variable is set to your machine's display
2. To run LogiBLOX, enter the following command.

   ```
   lbgui
   ```

   The LogiBLOX Setup Window appears after the LogiBLOX module generator is loaded. This window allows you to name and customize the module you want to create.
3. Select the Vendor tab in the Setup Window. Select Synopsys in the Vendor Name field to specify the correct bus notation for connecting your module.

   Select the Project Directory tab.Enter the directory location of your Synopsys project in the LogiBLOX Project Directory field.

   Select the Device Family tab. Select the target device for your design in the Device Family field.

   Select the Options tab and select the applicable options for your design as follows.

   - Simulation Netlist

     This option allows you to create simulation netlists of the selected LogiBLOX module in different formats. You can choose one or more of the outputs listed in the following table.

Table 4-3 Simulation Netlist Options

Option	Description
Behavioral VHDL netlist	Generates a simulation netlist in behavioral VHDL; output file has a .vhd extension.
Gate level EDIF netlist	Generates a simulation netlist in EDIF format; output file has an .edn extension.
Structural Verilog netlist	Generates a simulation netlist in structural Verilog; output file has a .v extension.

- Component Declaration

  This option creates instantiation templates in different formats that can be copied into your design. You can select none, one, or both of the following options.

Table 4-4 Component Declaration Options

Option	Description
VHDL template	Generates a LogiBLOX VHDL component declaration / instantiation template that is copied into your VHDL design when a LogiBLOX module is instantiated. The output file has a .vhi extension.
Verilog template	Generates a LogiBLOX Verilog module definition / instantiation template that is copied into your Verilog design when a LogiBLOX module is instantiated. The output file has a .vei extension.

- Implementation Netlist

  Select NGC File to generate an implementation netlist in Xilinx NGD binary format. You must select this option when instantiating LogiBLOX symbols in an HDL design. The

output file has an .ngc extension and can be used as input to NGDBuild.

- LogiBLOX DRC

  Select the Stop Process on Warning option to stop module processing if any warning messages are encountered during the design process.

For example, if you have a Verilog design, and you are simulating with Verilog-XL, select Structural Verilog netlist, Verilog template, NGC File, and Stop Process on Warning. For a VHDL design and simulating with Synopsys VSS, select Behavioral VHDL, VHDL template, NGC File, and Stop Process on Warning.

Select OK.

4. Enter a name in the Module Name field in the Module Selector Window.

   Select a base module type from the Module Type field.

   Select a bus width from the Bus Width field.

   Customize your module by selecting pins and specifying attributes.

   After you have completed module specification, select OK.

   This initiates the generation of a component instantiation declaration, a behavioral model, and an implementation netlist.

5. Copy the module declaration / instantiation into your design. The template file created by LogiBLOX is *module_name*.vhi (VHDL) or *module_name*.vei (Verilog), and is saved in the project directory as specified in the LogiBLOX setup.

6. Complete the signal connections of the instantiated module to the rest of your design.

Note: For more information on simulation, refer to the “Simulating Your Design” chapter.

7. Create a Synopsys implementation script. Add a Set_dont_touch attribute to the instantiated LogiBLOX module, and compile your design.

Note: Logiblox_instance_name is the name of the instantiated module in your design. Logiblox_name is the LogiBLOX component that corresponds to the .ngc file created by LogiBLOX.

`set_dont_touch` *logiblox_instance_name*

`compile`

Also, if you have a Verilog design, use the Remove Design command before writing the .sxnf netlist.

Note: If you do not use the Remove Design command, Synopsys may write an empty .sxnf file. If this occurs, the Xilinx software will trim this module/component and all connected logic.

`remove_design` *logiblox_name*

`write –format xnf –hierarchy –output` *design.sxnf*

8. Compile your design and create a .sxnf file. You can safely ignore the following error messages.

   `Warning: Can't find the design in the library` *WORK*. `(LBR-1)`

   `Warning: Unable to resolve reference` *LogiBLOX_name* `in` *design_name*`. (LINK-5)`

   `Warning: Design` *design_name* `has` *1* `unresolved references. For more detailed information, use the "link" command. (UID-341)`

9. Implement your design with the Xilinx tools. Verify that the .ngc file created by LogiBLOX is in the same project directory as the Synopsys netlist.

   You may get the following warnings during the NGDBuild and mapping steps. These messages are issued if the Xilinx software can not locate the corresponding .ngc file created by LogiBLOX.

   `Warning: basnu - logical block` *LogiBLOX_instance_name* `of type` *LogiBLOX_name* `is unexpanded. Logical Design DRC complete with 1 warning(s).`

   If you get this message, you will get the following message during mapping.

   `ERROR:basnu - logical block` *LogiBLOX_instance_name* `of type` *LogiBLOX_name* `is unexpanded. Errors detected in general drc.`

If you get these messages, first verify that the .ngc file created by LogiBLOX is in the project directory. If the file is there, verify that the module is properly instantiated in the code.

10. To simulate your post-layout design, convert your design to a timing netlist and use the back-annotation flow applicable to Synopsys.

Note: For more information on simulation, refer to the "Simulating Your Design" chapter.

Implementing Memory

XC4000E/EX/XL/XLA and Spartan FPGAs provide distributed on-chip RAM or 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 family. The edge-triggered capability simplifies system timing and provides better performance for RAM-based designs. This distributed 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.

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 XC4000 and Spartan ROMs

ROMs can be implemented in Synopsys as follows.

- Use RTL descriptions of ROMs
- Instantiate 16x1 and 32x1 ROM primitives
- Use LogiBLOX to implement any other ROM size

VHDL and a Verilog examples of an RTL description of a ROM follow.

VHDL - RTL Description of a ROM

```
--
--  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;
```

Verilog - RTL Description of a ROM

```
/*
 * 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, Synopsys creates ROMs from random logic gates that are implemented using function generators.

Another method for implementing ROMs in your Synopsys design is to instantiate the 16x1 or 32x1 ROM primitives. To define the ROM value use the Set_attribute command as follows.

`set_attribute` *instance_name* **`xnf_init`** *rom_value* –**`type string`**

This attribute allows Synopsys to write the ROM contents to the netlist file so the Xilinx tools can initialize the ROM. Rom_value should be specified in hexadecimal values. See the VHDL and Verilog

RAM examples in the following section for examples of this attribute using a RAM primitive.

Implementing XC4000 Family RAMs

Note: Do not use RTL descriptions of RAMs in your code because they do not compile efficiently and can cause combinatorial loops.

You can implement RAMs in your HDL code as follows.

- Instantiate 16x1 and 32x1 RAM primitives (RAM16X1, RAM32X1, RAM16X1S, RAM32X1S, RAM16X1D)
- Use LogiBLOX to implement any other RAM size

When implementing RAM in XC4000 and Spartan designs, Xilinx recommends using the synchronous write, edge-triggered RAM (RAM16X1S, RAM32X1S, or RAM16X1D) instead of the asynchronous-write RAM (RAM16X1 or RAM32X1) to simplify write timing and increase RAM performance.

Examples of an instantiation of edge-triggered RAM primitives are provided in the following VHDL and Verilog designs. As with ROMs, initial RAM values can be specified from within a Synopsys Script as follows.

`set_attribute` *instance_name* **`xnf_init`** *ram_value* **`–type string`**

Ram_value is specified in hexadecimal values.

VHDL - Instantiating RAM

```
-----------------------------------------
-- RAM_PRIMITIVE.VHD                   --
-- Example of instantiating 4          --
-- 16x1 synchronous RAMs               --
-- HDL Synthesis Design Guide for FPGAs --
-- May 1997                            --
-----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;


entity ram_primitive is
```

```
   port ( DATA_IN, ADDR  : in STD_LOGIC_VECTOR(3 downto 0);
          WE, CLOCK      : in STD_LOGIC;
          DATA_OUT       : out STD_LOGIC_VECTOR(3 downto 0));

end ram_primitive;


architecture STRUCTURAL_RAM of ram_primitive is

   component RAM16X1S
       port (D, A3, A2, A1, A0, WE, WCLK : in STD_LOGIC;
             O : out STD_LOGIC);
   end component;

begin

   RAM0 :  RAM16X1S port map (O => DATA_OUT(0), D => DATA_IN(0),
                              A3 => ADDR(3), A2 => ADDR(2),
                              A1 => ADDR(1), A0 => ADDR(0),
                              WE => WE, WCLK => CLOCK);

   RAM1 :  RAM16X1S port map (O => DATA_OUT(1), D => DATA_IN(1),
                              A3 => ADDR(3), A2 => ADDR(2),
                              A1 => ADDR(1), A0 => ADDR(0),
                              WE => WE, WCLK => CLOCK);

   RAM2 :  RAM16X1S port map (O => DATA_OUT(2), D => DATA_IN(2),
                              A3 => ADDR(3), A2 => ADDR(2),
                              A1 => ADDR(1), A0 => ADDR(0),
                              WE => WE, WCLK => CLOCK);

   RAM3 :  RAM16X1S port map (O => DATA_OUT(3), D => DATA_IN(3),
                              A3 => ADDR(3), A2 => ADDR(2),
                              A1 => ADDR(1), A0 => ADDR(0),
                              WE => WE, WCLK => CLOCK);

end STRUCTURAL_RAM;
```

Verilog - Instantiating RAM

```
      ////////////////////////////////////////
     // RAM_PRIMITIVE.V                     //
    // Example of instantiating 4          //
   // 16x1 Synchronous RAMs               //
  // HDL Synthesis Design Guide for FPGAs //
```

```
 // August 1997                          //
//////////////////////////////////////////


module ram_primitive (DATA_IN, ADDR, WE, CLOCK, DATA_OUT);

input  [3:0] DATA_IN, ADDR;
input        WE, CLOCK;
output [3:0] DATA_OUT;

RAM16X1S RAM0 (.O(DATA_OUT[0]), .D(DATA_IN[0]), .A3(ADDR[3]),
               .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
               .WE(WE), .WCLK(CLOCK));

RAM16X1S RAM1 (.O(DATA_OUT[1]), .D(DATA_IN[1]), .A3(ADDR[3]),
               .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
               .WE(WE), .WCLK(CLOCK));

RAM16X1S RAM2 (.O(DATA_OUT[2]), .D(DATA_IN[2]), .A3(ADDR[3]),
               .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
               .WE(WE), .WCLK(CLOCK));

RAM16X1S RAM3 (.O(DATA_OUT[3]), .D(DATA_IN[3]), .A3(ADDR[3]),
               .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
               .WE(WE), .WCLK(CLOCK));

endmodule
```

Using LogiBLOX to Implement Memory

Use LogiBLOX to create a memory module larger than 32X1 (16X1 for Dual Port). Implementing memory with LogiBLOX is similar to implementing any module with LogiBLOX except for defining the Memory initialization file. Use the following steps to create a memory module.

Note: Refer to the “Using LogiBLOX in HDL Designs” section for more information on using LogiBLOX.

1. Before using LogiBLOX, verify the following.
   - Xilinx software is correctly installed
   - Environment variables are set correctly
   - Your display environment variable is set to your machine’s display

2. To run LogiBLOX, enter the following command.

   `lbgui`

   The LogiBLOX Setup Window appears after the LogiBLOX module generator is loaded. This window allows you to name and customize the module you want to create.

3. Select the Vendor tab in the Setup Window. Select Synopsys in the Vendor Name field to specify the correct bus notation for connecting your module.

   Select the Project Directory tab. Enter the directory location of your Synopsys project in the LogiBLOX Project Directory field.

   Select the Device Family tab. Select the target device for your design in the Device Family field.

   Select the Options tab and select the applicable options for your design.

   Select OK.

4. Enter a name in the Module Name field in the Module Selector Window.

   Select the Memories module type from the Module Type field to specify that you are creating a memory module.

   Select a width (any value from 1 to 64 bits) for the memory from the Data Bus Width field.

   In the Details field, select the type of memory you are creating (ROM, RAM, SYNC_RAM, or DP_RAM).

   Enter a value in the Memory Depth field for your memory module.

Note: Xilinx recommends (this is not a requirement) that you select a memory depth value that is a multiple of 16 since this is the memory size of one lookup table.

5. If you want the memory module initialized to all zeros on power up, you do not need to create a memory file (Mem File). However, if you want the contents of the memory initialized to a value other than zero, you must create and edit a memory file. Enter a memory file name in the Mem File field and click on the Edit button. Continue with the following steps.

a) A memory template file in a text editor is displayed. This file does not contain valid data, and must be edited before you can use it. The data values specified in the memory file Data Section define the contents of the memory. Data values are specified sequentially, beginning with the lowest address in the memory, as defined.

b) Specify the address of a data value. The default radix of the data values is 16. If more than one radix definition is listed in the memory file header section, the last definition is the radix used in the Data Section.

The following definition defines a 16-word memory with the contents 6, 4, 5, 5, 2, 7, 5, 3, 5, 5, 5, 5, 5, 5, 5, 5, starting at address 0. Note that the contents of locations 2, 3, 6, and 8 through 15 are defined via the default definition. Two starting addresses, 4 and 7, are given.

```
depth 16
default 5
data 6,4,
4: 2, 7,
7: 3
```

c) After you have finished specifying the data for the memory module, save the file and exit the editor.

6. Click the OK button. Selecting OK initiates the generation of a component instantiation declaration, a behavioral model, and an implementation netlist.

7. Copy the HDL module declaration / instantiation into your HDL design. The template file created by LogiBLOX is *module_name*.vhi for VHDL and *module_name*.vei for Verilog, and is saved in the project directory as specified in the LogiBLOX setup.

8. Complete the signal connections of the instantiated LogiBLOX memory module to the rest of your HDL design, and complete initial design coding.

9. Perform a behavioral simulation on your design. For more information on behavioral simulation, refer to the “Simulating Your Design” chapter.

10. Create a Synopsys implementation script. Add a Set_dont_touch attribute to the instantiated LogiBLOX memory module, and compile your design.

Note: Logiblox_instance_name is the name of the instantiated module in your design. Logiblox_name is the LogiBLOX component that corresponds to the .ngc file created by LogiBLOX.

`set_dont_touch` *logiblox_instance_name*

`compile`

Also, if you have a Verilog design, use the Remove Design command before writing the .sxnf netlist.

Note: If you do not use the Remove Design command, Synopsys may write an empty .sxnf file. If this occurs, the Xilinx software will trim this module/component and all connected logic.

`remove_design` *logiblox_name*

`write –format xnf –hierarchy –output` *design.sxnf*

11. Compile your design and create a .sxnf file. You can safely ignore the following warning messages.

`Warning: Can't find the design in the library` *WORK*. `(LBR-1)`

`Warning: Unable to resolve reference` *LogiBLOX_name* `in` *design_name*`. (LINK-5)`

`Warning: Design` *design_name* `has` *1* `unresolved references. For more detailed information, use the "link" command. (UID-341)`

12. Implement your design with the Xilinx tools. Verify that the .ngc file created by LogiBLOX is in the same project directory as the Synopsys netlist.

You may get the following warnings during the NGDBuild and mapping steps. These messages are issued if the Xilinx software can not locate the corresponding .ngc file created by LogiBLOX.

`Warning: basnu - logical block` *LogiBLOX_instance_name* `of type` *LogiBLOX_name* `is unexpanded. Logical Design DRC complete with 1 warning(s).`

If you get this message, you will get the following message during mapping.

`ERROR:basnu - logical block` *LogiBLOX_instance_name* `of type` *LogiBLOX_name* `is unexpanded. Errors detected in general drc.`

If you get these messages, first verify that the .ngc file created by LogiBLOX is in the project directory. If the file is there, verify that the module is properly instantiated in the code.

13. To simulate your post-layout design, convert your design to a timing netlist and use the back-annotation flow applicable to Synopsys.

Note: For more information on simulation, refer to the "Simulating Your Design" chapter.

Implementing Boundary Scan (JTAG 1149.1)

Note: Refer to the *Development System Reference Guide* for a detailed description of the XC4000/XC5200 boundary scan capabilities.

XC4000, Spartan, and XC5200 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. To access the boundary scan logic after configuration in HDL designs, you must instantiate the boundary scan symbol, BSCAN, and the boundary scan I/O pins, TDI, TMS, TCK, and TDO.

The XC5200 BSCAN symbol contains three additional pins: RESET, UPDATE, and SHIFT, which are not available for XC4000 and Spartan. These pins represent the decoding of the corresponding state of the boundary scan internal state machine. If this function is not used, you can leave these pins unconnected in your HDL design.

Note: Do not use the FPGA Compiler boundary scan commands such as set_jtag_implementation, set_jtag_instruction, and set_jtag_port with FPGA devices.

Instantiating the Boundary Scan Symbol

To incorporate the boundary scan capability in a configured FPGA using Synopsys 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.

You *must* assign a Synopsys Set Don't Touch attribute to the net connected to the TDO pad before you use the Insert Pads 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.

VHDL - Boundary Scan

```
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;
```

Verilog - Boundary Scan

```
      ////////////////////////////////////////////////
     // 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), .CE(CE_P),
               .RST(RESET_P), .DATA(DATA_P), .COUT(COUT_P));

endmodule
```

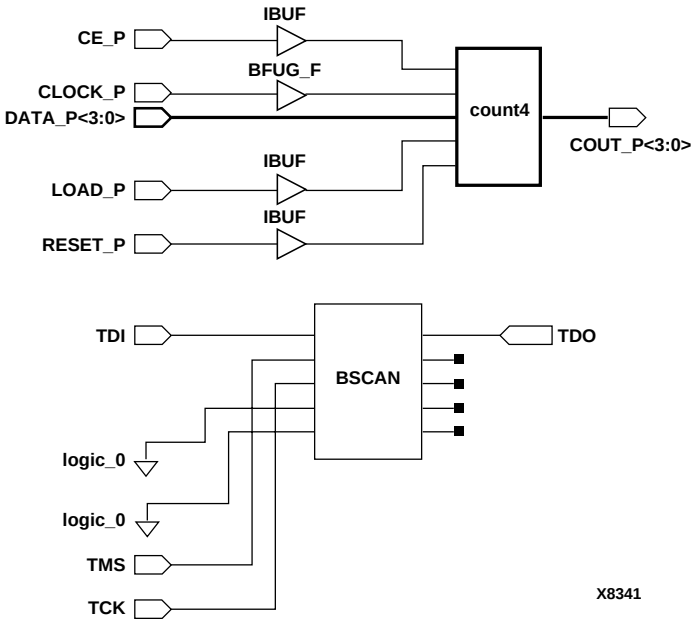

Figure 4-7 Bnd_scan Schematic

Implementing Logic with IOBs

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 XC4000 and Spartan devices have different IOB functions. The following sections provide a general description of the IOB function in XC4000E/EX/XLA/XL/XV and Spartan devices. A description of how to manually implement additional I/O features is also provided.

XC4000E/EX/XLA/XL/XV and Spartan IOBs

You can configure XC4000E/EX/XLA/XL/XV and Spartan 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 the Design Analyzer. The Set Port Is Pad command in conjunction with the Insert Pads command allows Synopsys to create the appropriate buffer structure according to the direction of the specified port in the HDL code. Attributes can also be added to these commands to further control pull-up, pull-down, and clock buffer insertion, as well as slew-rate control.

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.

To avoid external hold-time requirements, IOB input flip-flops and latches have 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.

If an IOB or register is instantiated in your HDL code, do not use the Set Port Is Pad command on that port. Doing so may automatically infer a buffer on that port and create an invalid double-buffer structure.

Note: Registers that connect to an input or output pad and require a Direct Clear or Preset pin are not implemented by the FPGA or Design Compiler in the IOB.

Outputs

The output signal that drives the programmable tristate output buffer can be a registered 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 XC4000 and Spartan output buffers can sink 12 mA. Two adjacent outputs can be inter-connected externally to sink up to 24mA.

Note: The FPGA Compiler and Design Compiler can optimize flip-flops attached to output pads into the IOB. However, these compilers cannot optimize flip-flops into an IOB configured as a bidirectional pad.

Slew Rate

Add the following to your Synopsys script to control slew rate; add after the Set Port Is Pad command, and before the Insert Pads command.

```
set_pad_type -slewrate HIGH {output_ports}
set_pad_type -slewrate LOW {output_ports}
```

Note: Synopsys High slew control corresponds to Xilinx Slow slew rate. Synopsys Low slew control corresponds to Xilinx Fast slew rate. If a slew rate is not specified, the default is High or Slow.

You can also specify slew rate by instantiating the appropriate OBUF primitive. To specify a fast slew rate, instantiate a fast output primitive (such as, OBUF_F and OBUFT_F). The default output buffer has a slow slew rate (such as, OBUF_S and OBUFT_S), and you do not need to instantiate it in your HDL code. If you do instantiate an I/O buffer or register, make sure that the Set Port Is Pad is not performed on this port to prevent creation of a double I/O buffer.

Pull-ups and Pull-downs

XC4000 and Spartan devices have programmable pull-up and pull-down resistors available in the I/O 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.

To specify these internal pull-up or pull-down I/O resistors, add the following to your Synopsys script rather than instantiating them in your HDL code. Add the following to your script after executing the Set Port Is Pad command and before executing the Insert Pads command.

```
set_pad_type -pullup {port_names}
set_pad_type -pulldown {port_names}
```

XC4000EX/XLA/XL/XV Output Multiplexer/2-Input Function Generator

A function added to XC4000EX / XLA / XL / XV 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 and / 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 additional 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, (OAND2, OOR2, OXOR2, etc.) 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.

- VHDL - Output Multiplexer

```
----------------------------------------
-- 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;
```

- Verilog - Output Multiplexer

```
      ////////////////////////////////////////
     // 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
```

XC5200 IOBs

XC5200 IOBs consist of an input buffer and an output buffer that can be configured as an input, output, or bi-directional I/O. The structure of the XC5200 is similar to the XC4000 IOB except the XC5200 does not contain a register/latch. The XC5200 IOB has a programmable pull-up or pull-down resistor, and two slew rate control modes (Fast and Slow) to minimize bus transients. The input buffer can be globally configured to TTL or CMOS levels, and the output buffer can sink or source 8.0 mA.

I/O buffer structures (as with the XC4000 IOBs) can be inferred from a Synopsys script with the Set Port Is Pad command in conjunction with the Insert Pads command. Controlling pull-up and pull-down insertion and slew rate control are performed as previously described for the XC4000 IOB.

The XC5200 IOB also contains a delay element so that an input signal that is directly registered or latched can have a guaranteed zero hold time at the expense of a longer setup time. You can disable this (equivalent to NODELAY in XC4000) by instantiating an IBUF_F buffer for that input port. This only needs to be done for ports that connect directly to the D input of a register in which a hold time can be tolerated.

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
- Create the I/O using LogiBLOX

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 tri-stated 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 tri-stated output with a tri-state 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, the FPGA Compiler 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 tri-state buffer. The active low READ_WRITE signal enables the tri-state gate.

- VHDL - Inferring a Bi-directional Pin

```
------------------------------------------------------
--  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)
    begin

        if (READ_WRITE = '1') then
            LATCH_OUT <= DATA;
        end if;

    end process;

    process(READ_WRITE)
        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;
```

- Verilog - Inferring a Bi-directional Pin

```
      ////////////////////////////////////////////////////////////////
     // BIDIR_INFER.V                                                //
    // 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.         //
 // August 1997                                                   //
////////////////////////////////////////////////////////////////

module bidir_infer (DATA, READ_WRITE);

    input       READ_WRITE ;
    inout [1:0] DATA ;

    reg   [1:0] LATCH_OUT ;

    always @ (READ_WRITE)
        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 the 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 Bi-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 command on the instantiated ports to prevent the I/O buffers from being inferred by Synopsys. This precaution also prevents the creation of an illegal structure.

- VHDL - Instantiation of a Bi-directional Pin

```
-----------------------------------
--  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;
```

- Verilog - Instantiation of a Bi-directional Pin

```
 ////////////////////////////////////////
// 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
```

Using LogiBLOX to Create Bi-directional I/O

You can use LogiBLOX to create I/O structures in an FPGA. LogiBLOX gives you the same control as instantiating I/O primitives, and is usually less verbose. LogiBLOX is especially useful for bused I/O ports.

Note: Refer to the "Using LogiBLOX in HDL Designs" section section, for details on creating, instantiating, and compiling LogiBLOX modules.

Do not use the Set Port Is Pad command on LogiBLOX-created ports. Also, when designing with Verilog, you must issue a Remove Design command before writing out the .sxnf files from Synopsys.

The following VHDL and Verilog examples show how to instantiate bi-directional I/O created with LogiBLOX. These examples produce the same results as the examples in the "Instantiating Bi-directional I/O" section.

- VHDL - Using LogiBLOX to Create a Bi-directional Port

```
--------------------------------------
--  BIDIR_LOGIBLOX.VHD              --
--  Example of using LogiBLOX       --
--  to create a Bi-directional port  --
--  August 1997                     --
--------------------------------------
------------------------------------------------------
-- LogiBLOX BIDI Module "bidir_io_from_lb"
-- Created by LogiBLOX version M1.3.7
--    on Mon Sep  8 13:14:02 1997
-- Attributes
--    MODTYPE = BIDI
--    BUS_WIDTH = 2
--    IN_TYPE = LATCH
--    OUT_TYPE = TRI
------------------------------------------------------
```

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity bidir_logiblox is

    port (DATA :       inout STD_LOGIC_VECTOR(1 downto 0);
          READ_WRITE : in    STD_LOGIC);

end bidir_logiblox;

architecture XILINX of bidir_logiblox is

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

    ------------------------------------------------
    -- Component Declaration
    ------------------------------------------------
    component bidir_io_from_lb
        PORT( O:     IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
              OE:    IN    STD_LOGIC;
              IGATE: IN    STD_LOGIC;
              IQ:    OUT   STD_LOGIC_VECTOR(1 DOWNTO 0);
              P:     INOUT STD_LOGIC_VECTOR(1 DOWNTO 0));
    end component;

    begin

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

    ------------------------------------------------
    -- Component Instantiation
    ------------------------------------------------
    BIDIR_BUSSED_PORT : bidir_io_from_lb
        port map (O => DATA_OUT, OE => READ_WRITE,
                  IGATE => READ_WRITE, IQ => LATCH_OUT, P => DATA);

end XILINX;
```

- Verilog - Using LogiBLOX to Create a Bi-directional Port

```
 ////////////////////////////////////////
// BIDIR_LOGIBLOX.V                     //
```

```
   // This is an example of using LogiBLOX   //
  // to create a bi-directional port.       //
 // August 1997                            //
//////////////////////////////////////////

//-----------------------------------------------------
// LogiBLOX BIDI Module "bidir_io_from_lb"
// Created by LogiBLOX version M1.3.7
//    on Mon Sep  8 17:10:15 1997
// Attributes
//    MODTYPE = BIDI
//    BUS_WIDTH = 2
//    IN_TYPE = LATCH
//    OUT_TYPE = TRI
//-----------------------------------------------------

module bidir_logiblox (DATA, READ_WRITE);

    input       READ_WRITE ;
    inout [1:0] DATA ;

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


    assign DATA_OUT[0] = LATCH_OUT[0] & LATCH_OUT[1];
    assign DATA_OUT[1] = LATCH_OUT[0] | LATCH_OUT[1];

    // LogiBLOX instantiation

    bidir_io_from_lb  BIDIR_BUSSED_PORT
        ( .O(DATA_OUT), .OE(READ_WRITE), .P(DATA),
          .IQ(LATCH_OUT), .IGATE(READ_WRITE));

endmodule


module bidir_io_from_lb (O, OE, P, IQ, IGATE);
    input  [1:0] O;
    input        OE;
    input        IGATE;
    inout  [1:0] P;
    output [1:0] IQ;
endmodule
```

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 a Synopsys script prior to writing out the netlist file or from a User Constraints File (UCF), as follows.

- Assigning Pad Location with a Synopsys Script

  `set_attribute` *port_name* **`"pad location" -type string`** *pad_location*

- Assigning Pad Location with a UCF

  `NET` *port_name* **`LOC`** = *pad_location* **`;`**

Use one or the other method, but not both. 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

Note: XC5200 devices do not have input and output flip-flops.

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.

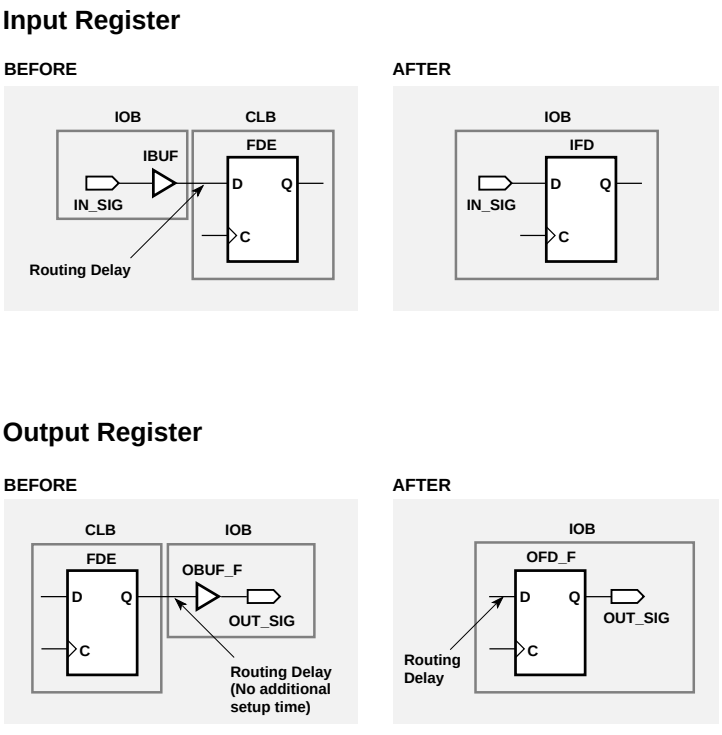

Figure 4-8 Moving Registers into the IOB

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. The Synopsys Compiler automatically moves 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

XC4000EX / XL / XV this applies to non-I / O latches, as well as flip-flops.

2. IOBs must have input or output flip-flops. XC5200 devices do not have IOB flip-flops.
3. The flip-flop does not use an asynchronous set or reset signal.
4. 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.
5. In XC4000 or Spartan 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 XC4000EX / XL / XV output MUX is used in the same IOB.
6. In XC4000EX / XV devices, if a constant 0 or 1 is driven on the IOPAD, a flip-flop / latch with a CE is not added to the input side of the IOB.

Using Unbonded IOBs (XC4000E/EX/XLA/XL/XV and Spartan 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. You can implement shift registers with these unbonded IOBs. The VHDL and Verilog examples in this section show how to instantiate unbonded IOB flip-flops in a 4-bit shift register in an XC4000 device.

Note: The Synopsys compilers cannot infer unbonded primitives. Refer to the *Synopsys (XSI) Interface/Tutorial Guide* for a list of library primitives that can be used for instantiations.

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

```
------------------------------------------------
-- 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;
```

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

```
        ////////////////////////////////////////////////
       // UNBONDED.V                                   //
      // XC4000 family has unbonded IOBs which have    //
     // storage elements that can be used to build    //
    // functions lie 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
```

Implementing Multiplexers with Tristate Buffers

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 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.

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 BUFTs” figure.

VHDL - Mux Implemented with Gates

```
-- MUX_GATE.VHD
-- 5-to-1 Mux Implemented in Gates
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- 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;
```

Verilog - Mux Implemented with Gates

```
/* MUX_GATE.V
 * Synopsys HDL Synthesis Design Guide for FPGAs
 * 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
```

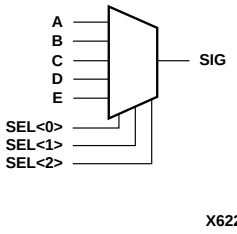

Figure 4-9 5-to-1 MUX Implemented with Gates

VHDL - Mux Implemented with BUFTs

```
-- MUX_TBUF.VHD
-- 5-to-1 Mux Implemented in 3-State Buffers
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- 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;
```

Verilog - Mux Implemented with BUFTs

```
/* MUX_TBUF.V
 * Synopsys HDL Synthesis Design Guide for FPGAs
 * 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
```

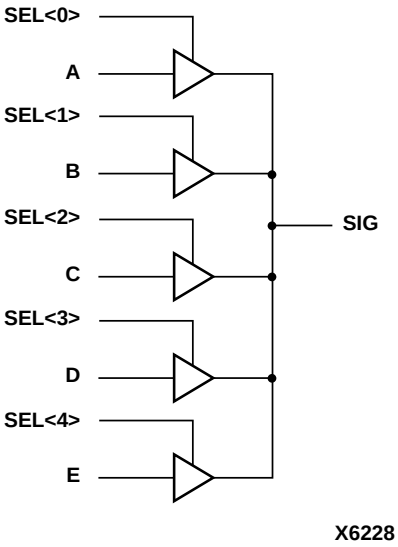

Figure 4-10 5-to-1 MUX Implemented with BUFTs

A comparison of timing and area for a 5-to-1 multiplexer built with gates and tristate buffers in an XC4005EPC84-2 device is provided in the following table. When the multiplexer is implemented with tristate buffers, no CLBs are used and the delay is smaller.

Table 4-5 Timing/Area for 5-to-1 MUX (XC4005EPC84-2)

Timing/Area	Using BUFTs	Using Gates
Timing	15.31 ns (1 block level)	17.56 ns (2 block levels)
Area	0 CLBs, 5 BUFTs	3 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 since the pipeline is created at no cost in terms of device resources. Because data is now on a multi-cycle path, special considerations must be used for the rest of your design to account for the added path latency. You must also be careful when defining timing specifications for these paths.

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.

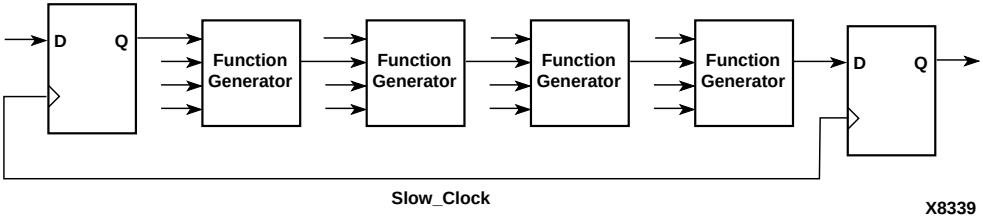

Figure 4-11 Before Pipelining

After Pipelining

This is an example of the same data path in the previous example after pipelining. Since 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.

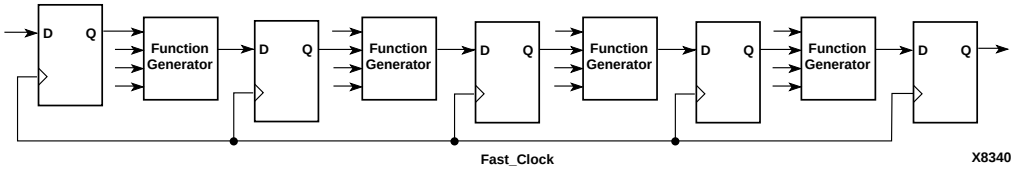

Figure 4-12 After Pipelining

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 design 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 Synopsys 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.

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 the *Synopsys Interface Guide* for more information on optimizing logic across hierarchical boundaries and compiling hierarchical designs.

Chapter 5

Simulating Your Design

This chapter describes simulation methods for verifying the functional timing of your designs. It includes the following sections.

- "Introduction"
- "Functional Simulation"
- "Timing Simulation"
- "Using VHDL / Verilog Libraries and Models"
- "Simulating Global Signals"
- "Adapting Schematic Global Signal Methodology for VHDL"
- "Setting VHDL Global Set / Reset Emulation in Functional Simulation"
- "Using Oscillators (VHDL)"
- "Compiling Verilog Libraries"
- "Setting Verilog Global Set / Reset"
- "Setting Verilog Global Tristate (XC4000, Spartan, and XC5200 Outputs Only)"

Introduction

Xilinx supports functional and timing simulation of HDL designs at the following three points in the HDL design flow as shown in the following figure.

- Register Transfer Level (RTL) simulation which may include the following.
    - Instantiated UniSim library components

  - LogiBLOX modules
  - LogiCore models
- Post-synthesis functional simulation with one of the following.
  - Gate-level UniSim library components

  or
  - Gate-level pre-route SimPrim library components
- Post-implementation back-annotated timing simulation with the following.
  - SimPrim library components
  - Standard Delay Format (SDF) file

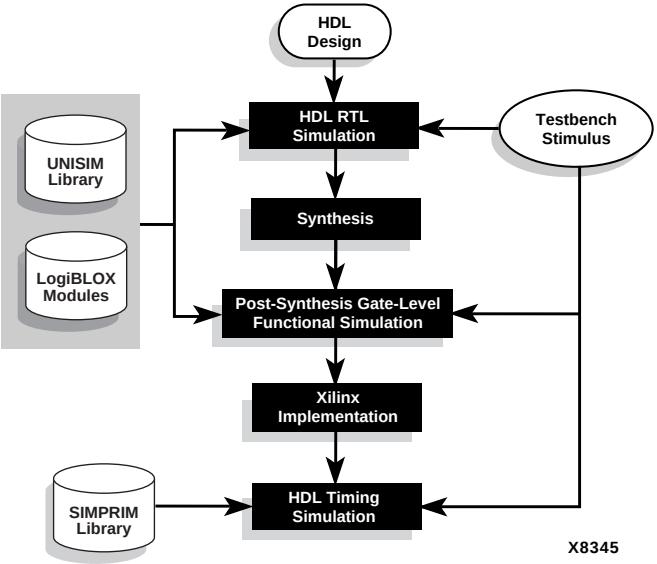

Figure 5-1 Three Simulation Points for HDL Designs

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 5-1 Five Simulation Points in HDL Design Flow

Simulation		UniSim	LogiBLOX Models	SimPrim	SDF
1.	RTL	X	X		
2.	Post-Synthesis	X	X		
3.	Functional Post-NGDBuild (Optional)			X	
4.	Functional Post-MAP (Optional)			X	X
5.	Post-Route Timing			X	X

These simulation points are described in detail in the "Functional Simulation" section and the "Timing Simulation" section. The libraries required to support the simulation flows are described in detail in the "Using VHDL/Verilog Libraries and Models" section. The new flows and libraries now support closer functional equivalence of initialization behavior between functional and timing simulations. This is due to the addition of new methodologies and library cells to simulate GSR/GTS behavior.

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, your various design descriptions are not functionally equivalent and your simulation results will not match. In addition to the behavioral representation for GSR, you need to add a Xilinx implementation directive. This directive is used to specify to the place and route tools to use the special purpose GSR net that is pre-routed on the chip, and not to use the local asynchronous set/reset pins. 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.

Even if GSR behavior is not described, the actual chip initializes during configuration, and the post-route netlist will have this net that must be driven during simulation. The "Simulating Global Signals" section includes the methodology to describe this behavior, as well as the GTS behavior for output buffers.

Xilinx VHDL simulation supports the VITAL standard. This standard allows you to simulate with any VITAL-compliant simulator, including MTI/Mentor® ModelSim, and Synopsys VSS.

Built-in Verilog support allows you to simulate with the Cadence Verilog-XL and other compatible simulators. Xilinx HDL simulation supports all current Xilinx FPGA and CPLD devices. Refer to the "Using VHDL/Verilog Libraries and Models" section for the list of supported VHDL and Verilog standards.

Functional Simulation

Functional simulation of HDL designs includes support for FPGA and CPLD architecture features, such as global set/reset, global tristate enable, on-chip oscillators, RAMs, and ROMs.

You can perform functional simulation at the following points in the design flow.

- RTL simulation, including instantiated UniSim library components, LogiBLOX modules, and LogiCore models
- Post-synthesis gate-level VHDL or Verilog netlist
- Post-NGDBuild gate-level VHDL or Verilog netlist from implementation after NGDBuild using the SimPrim library
- Post-map partial-timing functional simulation with netlist and SDF file from implementation after mapping, and before place and route, using the SimPrim library

Register Transfer Level (RTL)

The first simulation point is the RTL-level simulation that allows you to verify or simulate a description at the system or chip level. At this

level, the system or chip is described with high-level RTL language constructs. VHDL and Verilog simulators exercise the design to check its functionality before it is implemented in gates. A test bench is created to model the environment of the system or chip. At this level, the Unified Simulation Library (UniSim) is used to instantiate cells. You can also instantiate LogiBLOX 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.

Post-Synthesis (Pre-NGDBuild) Gate-Level Simulation

Note: Post-synthesis, pre-NGDBuild simulation with FPGA Compiler Verilog-based designs is not possible at this time.

After the RTL simulation is error-free, the system or chip is synthesized to gates. The test bench is used again to simulate the synthesized result and check its consistency with the original design description.

Gate-level simulation in the Xilinx design flow includes any simulation performed after any of the synthesis, map, or place and route stages. The post-synthesis pre-NGDBuild gate-level simulation is a functional simulation with unit delay timing. The gates are expressed in terms of UniSim components.

This gate-level functional simulation allows you to directly verify your design after it has been generated by the synthesis tool. If there are differences in the behavior of the original RTL description and the synthesized design, this may indicate a problem with the synthesis tool. Although RTL-level and gate-level simulation may differ even when the synthesis is correct. This is due to the change in the level of abstraction. In general, overall functionality should be the same, but timing differences may occur.

Post-synthesis simulation is synthesis vendor-dependent, and the synthesis tool must write VHDL or Verilog netlists in terms of UniSim library components. Check with your synthesis vendor for this feature. The library usage guidelines for RTL simulation also apply to post-synthesis pre-NGDBuild gate-level functional simulation. LogiBLOX models remain as behavioral blocks and can be simulated in the same design as structural UniSim components. You may have to insert library statements into the HDL code.

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, NGDBuild translates the EDIF or XNF output of synthesis to SimPrim library components. Like post-synthesis, 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, GR (global reset), PRLD (preload), and GTS nets that must be initialized, just as for post-map partial timing simulation.

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, two important considerations are specifying different simulation libraries for pre- and post-implementation simulation, and the different gate-level cells in pre- and post-implementation netlists.

You can pause the implementation tools after reading in your source files; select either a VHDL or Verilog netlist as an output file; and write the file.

Post-Map Partial Timing (CLB and IOB Block Delays)

The third type of post-synthesis functional simulation is post-map partial timing simulation. This gate-level description is also expressed in terms of SimPrim components.

The SDF file that is created contains timing numbers for the CLB and IOB mapped blocks. At this point, your design is not routed and does not contain net delays, except for pre-laid out macros, such as cores. Like the post-NGDBuild and full-timing simulation, there are GSR, GR, PRLD, and GTS nets that must be initialized. If you want partial timing delays, you can use the SDF file (this is optional).

You can pause the implementation tools after the Map program stops, and write one of the HDL formats.

Creating a Simulation Test Bench

When you create a test bench for functional simulation, refer to the following table for coding style examples. You should refer to your synthesis vendor's documentation before using these styles because many of the styles cannot be synthesized.

Table 5-2 Common Coding Examples for Verilog/VHDL

Description	Verilog	VHDL
Delay or wait 20 ns	`'timescale 1 ns/ 100 ps` `#20`	`wait for 20 ns;`

Creation of a free running clock	`Initial`
	`begin`
	`clock = 0;`
	`#25 forever #25 clock = ~clock;`
	`end`
		`Loop`
		`wait for 25 ns;`
		`clock <=not (clock);`
		`end loop;`

Print "Text." to screen	`$display("Text");`	`report "Text."`
Print value of signal to screen whenever the value changes	`$monitor(%r",$real-time,"%b", clock,"%b"my_signal);`	
Apply a binary value 1010 to an input bus X.	`X = 4'b1010;`	`X <="1010";`

Creation of a for loop 0 to 10	`for(x=0; x < 10; x=x+1)`
	`begin`
	actions
	`end`
		`for x in 0 to 9 loop`
		actions
		`end loop;`

Write "X = *value*" to an output file	`$dumpfile ("file-name.dmp");`
	`$dumpvars (X);`
		`variable TEMP;`
		`write (TEMP, "X =");`
		`write (TEMP, X);`
		`writeline (filename, TEMP);`

Wait until X is logic one	`wait (X == 1'b1);`	`wait (X == '1');`

Table 5-2 Common Coding Examples for Verilog/VHDL

Description	Verilog	VHDL
Wait until X transitions to a logic one	`@(posedge X);`	`wait on X;`

If-Else construct	`always @ (X)`
	`begin`
	`  if (X = 1)`
	`  Y = 1'b0;`
	`  else`
	`  Y = 1'b1;`
	`end`
		`process (X)`
		`  if (X = '1') then`
		`  Y = '0';`
		`  else`
		`  Y = '1';`
		`  end if;`
		`end process;`

Case construct	`always @(X or A)`
	`  case (X)`
	`  2'b00 : Y = 1'b0;`
	`  2'b01 : Y = 1'b1;`
	`  default : Y = A;`
	`  endcase`
		`process (X,A)`
		`begin`
		`  case X is`
		`  when "00" => Y ='0';`
		`  when "01" => Y = '1';`
		`  when others => Y = A;`
		`  end case;`
		`end process;`

Example instantiation of an OFD	`OFD U1`
	`(.Q(D_OUT),.D(D_IN),`
	`  .C(CLOCK));`
		`U1:  OFD port map`
		`(Q => D_OUT), D => D_IN,`
		`  C => CLOCK);`

XSI Pre-implementation Simulation

For XSI designs, simulation is not limited to pure RTL VHDL. Pre- and post-synthesis functional gate-level simulations are also possible. For every pre-synthesis simulation flow, there is a corresponding post-synthesis flow. Make sure you meet the following requirements before simulating your design.

- You must have a .synopsys_vss.setup file that points to the compiled UniSim libraries. Refer to the sample template.synopsys_vss.setup file in $XILINX/synopsys/examples.
- If LogiBLOX components are instantiated, use the VHDL simulation model for the instantiated LogiBLOX components created by the LogiBLOX tool for pre- and post-synthesis functional simulation.

- If GSR or GTS is under your control, you must instantiate a STARTBUF, TOC, ROC, or ROCBUF.
- Before synthesizing or performing a post-synthesis simulation, remove any library UniSim or use UniSim.xxxx.xxxx reference in your VHDL code. Alternatively, you can insert pragma commands to ignore these references, as follows.

  Before the library statements, use the following.

  ```
  synopsys translate_off
  ```

  After the library statements, use the following.

  ```
  synopsys translate_on
  ```

XSI Post-implementation Functional Simulation

The XSI post-implementation simulation flows allow you to check the functional behavior of your design after implementation. There are three post-implementation simulation flows: post-NGDBuild, post-Map, and post-PAR. Each one is described in this section.

All three of these flows must have a .synopsys_vss.setup file that points to the compiled SimPrim XSI libraries. Refer to the template.synopsys_vss.setup file located in $XILNX/synopsys/examples. If the GSR or GTS is under your control via a top-level port, the GSR and GTS nets must be controlled by you at the start of post-implementation simulation.

Note: Post-PAR simulation is the only flow that provides back-annotated timing.

Post-NGDBuild Simulation

Since post-ngdbuild simulation is SimPrim-based, your .synopsys_vss.setup file does not have to point to the UniSim, XDW, or LogiBLOX libraries. However, this file must point to the location of the compiled SimPrim libraries. All logic in your design has been reduced to the Xilinx common simulation primitive components.

Post-Map Simulation

Like post-NGDBuild simulation, post-Map simulation is SimPrim-based, and your .synopsys_vss.setup file does not have to point to the UniSim, XDW, or LogiBLOX libraries. This file must point to the

location of the compiled SimPrim libraries. All logic in your design has been reduced to the XILINX common simulation primitive components. Post-Map simulation is useful for detecting if trimming has changed the functionality of your design.

Post-PAR Simulation (timing simulation)

Post-PAR simulation is back-annotated timing simulation. Your .synopsys_vss.setup file must point to the compiled SimPrim libraries. If the GSR or GTS is under your control via top-level ports, the GSR and GTS nets must be controlled by you in the test bench to simulate their behavior.

Timing Simulation

Timing simulation is important in verifying the operation of your circuit after the worst-case placed and routed delays are calculated for your design. In many cases, you can use the same test bench that you used for functional simulation to perform a more accurate simulation with less effort. 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.

Post-Route Full Timing (Block and Net Delays)

After your design is routed using PAR, it can be simulated with the actual block and net timing delays with the same test bench used in the behavioral simulation.

The back-annotation process (NGDAnno) produces a netlist of SimPrims annotated with an SDF file with the appropriate block and net delay data from the place and route process. This netlist has GSR, GR, PRLD, and GTS nets that must be initialized. For more information, refer to the “Simulating Global Signals” section.

Using VHDL/Verilog Libraries and Models

The five simulation points listed previously require the UniSim, SimPrim, XDW (Xilinx DesignWare), and LogiBLOX 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 or Logi-BLOX components. To support these instantiations, Xilinx provides a functional UniSim library and a behavioral LogiBLOX library.

The second point, post-synthesis (pre-NGDBuild) gate-level simulation uses the UniSim and XDW libraries. 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 5-3 Simulation Phase Library Information

Simulation Point	Compilation Order of Library Required
RTL	UniSim LogiBLOX
Post-Synthesis	UniSim (Device Dependent) XDW
Post-NGDBuild	SimPrim
Post-MAP	SimPrim
Post-Route	SimPrim

Adhering to Industry Standards

The standards in the following table are supported by the Xilinx simulation flow.

Table 5-4 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 UniSim and SimPrim libraries adhere to IEEE-STDs. The VHDL library uses the VITAL IEEE-STD-1076.4 standard, and the Verilog library uses the IEEE-STD-1364 standard. By following these stan-

dards, Xilinx provides support for customers who use HDL tools from various vendors.

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.

VITAL libraries include some overhead 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; however, you or your system administrator can turn them off. You must edit and recompile the SimPrim components file after setting the generics.

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 5-5 Simulation Library Source Files

Library	Location of Source Files		Required Libraries	
	Verilog	VITAL VHDL	Verilog	VITAL VHDL
UniSim 4K Family, Spartan (use UNI4000E for Spartan)	$XILINX/verilog/src/unisims	$XILINX/vhdl/src/unisims	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)
UniSim 52K Family	$XILINX/verilog/src/uni5200	$XILINX/vhdl/src/unisims	Not required for Verilog-XL; see vendor documentation for other simulators	Required; typical compilation order: unisim_VCOMP52K.vhd unisim_VITAL.vhd unisim_VITAL52K.vhd unisim_VCFG52K.vhd
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

Installing and Configuring XSI VHDL Libraries

To install and configure the XSI VHDL libraries, perform the following steps.

1. Install the Xilinx and the Synopsys software. Refer to the current *Alliance Install and Release Document* for more information.
2. Set up your workstation environment to use the Xilinx and Synopsys software.
3. The XSI software is compiled for Synopsys 3.4b. If you are using a later version, you must recompile the XDW and simulation libraries. To compile the XDW and simulation libraries, perform the following two steps.
4. If necessary, recompile the XDW libraries by first setting up your Xilinx and Synopsys environments, and then running the install_dw.dc script in each of the following directories.

   Run this script as follows.

   ```
   dc_shell -f install_dw.dc
   ```

   $XILINX/synopsys/libraries/dw/src/xc4000e
   $XILINX/synopsys/libraries/dw/src/xc4000ex
   $XILINX/synopsys/libraries/dw/src/xc4000l
   $XILINX/synopsys/libraries/dw/src/xc4000xl
   $XILINX/synopsys/libraries/dw/src/xc4000xv
   $XILINX/synopsys/libraries/dw/src/xc5200
   $XILINX/synopsys/libraries/dw/src/virtex
   $XILINX/synopsys/libraries/dw/src/spartan
5. If necessary, recompile the simulation libraries by first setting up your Xilinx and Synopsys environments, and then running the analyze.csh script in each of the following directories.

   $XILINX/synopsys/libraries/sim/src/logiblox
   $XILINX/synopsys/libraries/sim/src/simprims
   $XILINX/synopsys/libraries/sim/src/xdw

Note: The LogiBLOX and SimPrim libraries are identical to those in $XILINX/vhdl/src.

Note: The analyze.csh script compiles the Xilinx simulation libraries with the vhdlan –c command, which uses a C compiler. The XSI simulation libraries use a C compiler to speed up the VHDL simulation flow for VSS. The behavior of vhdlan –c varies with different versions of the Synopsys software. If vhdlan –c fails, use vhdlan –i instead. Refer to the Synopsys documentation for the correct version of the C compiler, and for more information on vhdlan.

6. To compile the UniSim libraries, go to the $XILINX/vhdl/src/unisims directory.
7. In this directory, create a .synopsys_vss.setup file with the following lines.

```
TIMEBASE =NS
TIME_RES_FACTOR  =0.01

WORK  > DEFAULT
DEFAULT : .
UNISIM : ../../data

-- VHDL library to UNIX dir mappings --
SYNOPSYS      : $SYNOPSYS/packages/synopsys/lib
IEEE          : $SYNOPSYS/packages/IEEE/lib
```

8. If you are simulating an XC5200 design, use Procedure B, otherwise create a C shell script with Procedure A.

Procedure A

```
#!/bin/csh -f
vhdlan -i w UNISIM $XILINX/vhdl/src/unisims/unisim_VCOMP.vhd
vhdlan -i w UNISIM $XILINX/vhdl/src/unisims/unisim_VPKG.vhd
vhdlan -i w UNISIM $XILINX/vhdl/src/unisims/unisim_VITAL.vhd
```

Procedure B (XC5200)

```
#!/bin/csh -f
vhdlan -i w UNISIM $XILINX/vhdl/src/unisims/unisim_VCOMP52K.vhd
vhdlan -i w UNISIM $XILINX/vhdl/src/unisims/unisim_VPKG.vhd
vhdlan -i w UNISIM $XILINX/vhdl/src/unisims/unisim_VITAL.vhd
vhdlan -i w UNISIM $XILINX/vhdl/src/unisims/unisim_VITAL52K.vhd
vhdlan -i w UNISIM $XILINX/vhdl/src/unisims/unisim_VCFG52K.vhd
```

9. Run the script created in the previous step in the $XILINX/vhdl/src directory.

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 except in a few cases where functional differences occur.

Note: Synopsys FPGA Compiler users can also use a VHDL DesignWare library. For details, refer to the *Synopsys Interface Guide*.

For example, the decode8 in XC4000 devices has a pull-up, and in XC5200 devices, it does not. 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 each technology because Verilog does not have a configuration statement.

Schematic macros are not provided because most schematic vendors provide the lower-level netlist for importing into a synthesis tool. 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. You can also use a post-NGDBuild or post-Map netlist to simulate netlists with embedded schematic macros. This lower-level netlist for a schematic macro is also required for implementation. The VHDL models for Synopsys DesignWare components are in the Xilinx Synopsys Interface at $XILINX/synopsys/libraries/sim/src/xdw. Since Verilog versions of the DesignWare components are not currently available, use post-NGDBuild functional simulation instead.

- VHDL UniSim Library

  The VHDL version of the UniSim library is VITAL-compliant, and can be accelerated, however it is not back-annotated with an SDF file. The files are in $XILINX/vhdl/src/unisims. The source file should be compiled into a library named UNISIM. Refer to the “Installing and Configuring XSI VHDL Libraries” section in this chapter.

- Verilog UniSim Library

The Verilog version of the UniSim library may not have to be compiled, depending on the Verilog tool. Since there are a few cells with functional differences between Xilinx devices, a separate library is provided for each supported device. For example, decoders contain pull-ups in some devices and not in others. The libraries are in uppercase only. The libraries are located at $XILINX/verilog/src/*technology*, where *technology* is UNI3000, unisims, UNI5000, or UNI9000.

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.

Compiling the UniSim Library

The UniSim VHDL library (or Verilog library) can be compiled to any physical location. The VHDL source files are found in $XILINX/vhdl/src/unisims and are listed here in the order in which they must be compiled.

1. unisim_VCOMP.vhd (component declaration file)
2. unisim_VCOMP52K.vhd (substitutional component declaration file for XC5200 designs)
3. unisim_VPKG.vhd (package file)
4. unisim_VITAL.vhd (model file)
5. unisim_VITAL52K.vhd (additional model file for XC5200 designs)
6. unisim_VCFG4K.vhd (configuration file for XC4K edge decoders)
7. unisim_VCFG52K.vhd (configuration file for XC5200 internal decoders)

Note: To use both 4K and 52K, compile them into separate directories as a UniSim library. Change the mapping of the UNISIM logical name to the appropriate directory for each design.

The uppercase Verilog components are found in individual component files in the following directories.

1. $XILINX/verilog/src/uni3000 (Series 3K)

2. $XILINX/verilog/src/unisims (Series 4KE, 4KX, 4KL, 4KXV, Spartan, Virtex)
3. $XILINX/verilog/src/uni5200 (Series 5200)
4. $XILINX/verilog/src/uni9000 (Series 9500)

Instantiating UniSim Library Components

You can instantiate UniSim library components in your design and simulate them during RTL simulation. Your VHDL code must refer to the UniSim library compiled by you or by your system administrator. The VHDL simulation tool must map the logical library to the physical location of the compiled library. VHDL component declarations are provided in the library and do not need to be repeated in your code. Verilog must also map to the UniSim Verilog library.

Using UniSim Libraries with XSI VHDL Designs

You can use the UniSim libraries with your XSI VHDL designs as follows.

- You can simulate a Synopsys VHDL design pre- and post-synthesis (pre-NGDBuild) with VSS. This is possible even if your VHDL RTL code includes instantiated synthesis cells (such as flip-flops and OBUFTs) connected to GSR/GTS/GR.
- You can simulate the behavior of the GSR/GTS/GR even if it is not under your control.
- You can simulate your design during all phases of synthesis and implementation. The additional components and libraries that make this possible are UniSim, XDW, LogiBLOX, SimPrim, ROC, TOC, STARTBUF, ROCBUF, and TOCBUF.

Using the LogiBLOX Library

LogiBLOX is a module generator used for schematic-based design entry of modules such as adders, counters, and large memory blocks. In the HDL flow, you can use LogiBLOX to generate large blocks of memory for instantiation. Refer to the *LogiBLOX Guide* for more information.

In addition to the RTL code that results in synthesized logic, you can generate modules such as counters, adders, and large memory arrays with LogiBLOX. You can enter the desired parameters into LogiBLOX

and select a VHDL model as output. The VHDL model is at the behavioral level because it allows for quicker simulation times. LogiBLOX is primarily useful for building large memory arrays that cannot be inferred. VHDL models are provided for LogiBLOX modules from the schematic environment, or for large memory arrays. Most LogiBLOX modules contain registers and require the global set/rest (GSR) initialization. Since the modules do not contain output buffers going off-chip, the global tristate enable (GTS) initialization does not apply.

LogiBLOX models begin as behavioral in VHDL but are mapped to SimPrim structural models in the back-annotated netlist. The behavioral model is also used for any post-synthesis simulation because the module is processed as a "black box" during synthesis. It is important that the initialization behavior is consistent for the behavioral model used for RTL and post-synthesis simulation and for the structural model used after implementation. In addition, the initialization behavior must work with the method used for synthesized logic and cores.

Note: For Verilog, the LogiBLOX model is a structural netlist of SimPrim models. Do *not* synthesize this netlist; it is for functional simulation only.

Compiling the LogiBLOX Library

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

You can compile the LogiBLOX VHDL library (or Verilog) to any specified physical location. The VHDL source files are in $XILINX/vhdl/src/logiblox, and are listed below in the order in which they must be compiled.

1. mvlutil.vhd
2. mvlarith.vhd
3. logiblox.vhd

The Verilog source files are in $XILINX/verilog/src/logiblox.

Instantiating LogiBLOX Modules

LogiBLOX components are simulated with behavioral code. They are not intended to be synthesized, but they can be simulated. The synthesizer processes the components as a “black box”. Implementation uses the NGO file created by LogiBLOX. The source libraries for LogiBLOX packages are in $XILINX/vhdl/src/logiblox and $XILINX/verilog/src/logiblox. The actual models are output from the LogiBLOX tool. The package files must be compiled into a library named logiblox. The component model from the LogiBLOX GUI should be compiled into your working directory with your design.

Using the LogiCORE Library

In addition to synthesized or generated logic, you can use high-level pre-designed LogiCORE models. These models are high-level VHDL behavioral or RTL models that are mapped to SimPrim structural models in the back-annotated netlist. The behavioral model is used for any post-synthesis simulation because synthesis processes the core as a “black box”. As with LogiBLOX models, the initialization behavior must be consistent for the behavioral model used for RTL and post-synthesis simulation and for the structural model used after implementation. In addition, the initialization behavior must work with the method used for synthesized logic and LogiBLOX modules.

The UniSim VHDL and Verilog libraries can emulate the global set/reset and global tristate network in Xilinx FPGAs. VHDL uses special components for driving the local reset and tristate enable lines, and sending implementation directives to move the nets to the global signal resources. Verilog uses a macro definition.

The local signals emulate the fully routed global signals in a post-routed netlist. Both the VHDL and Verilog post-route netlists use the SimPrim Library and have global reset and output tristate enable signals fully routed; they are not emulated.

LogiBLOX and LogiCORE models are at a behavioral level and do not use library components for global signals. However the LogiBLOX model does require the packages that are compiled into the LogiBLOX library.

Simulating Global Signals

Xilinx PLDs have register (flip-flops and latches) set/reset circuitry that pulses at the end of the configuration mode and after power-up. This pulse is automatic and does not need to be programmed. All the flip-flops and latches in a PLD receive this pulse through a dedicated global set/reset (GSR), PRLD, or reset (GR) net. The registers either set or reset, depending on how the registers are defined.

In addition to the set/reset pulse, all output buffers are tristated during configuration mode and after power-up with the dedicated global output tristate enable (GTS) net. The global tri-state and reset net names are provided in the following table.

Table 5-6 Global Reset and Tristate Names for Xilinx Devices

Device Family	Global Reset Name	Global Tristate Name	Default Reset Polarity
XC3000	GR	Not Available	Low
XC4000	GSR	GTS	High
XC5000	GR	GTS	High
XC9500	PRLD	GTS	High
SPARTAN	GSR	GTS	High

These PRLD, GSR and GR nets require 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, PRLD, or GR net receives a reset-on-configuration pulse from the initialization controller, as shown in the following figure.

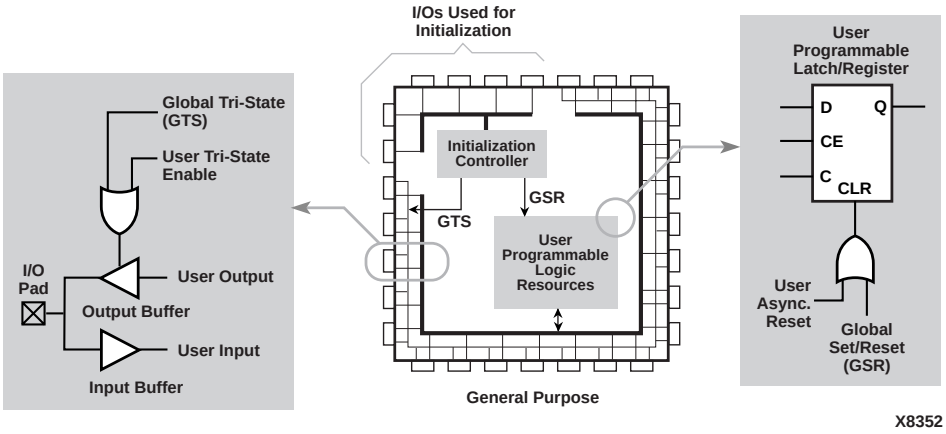

Figure 5-2 Built-in FPGA Initialization Circuitry

This pulse occurs during the configuration or power-up mode of the PLD. 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" figure also occurs during the configuration or power-up mode of the PLD. 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 PLD. All general-purpose outputs are affected whether they are regular, tristate, or bi-direct outputs during normal operation. This ensures that the outputs do not erroneously drive other devices as the PLD 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 TPOR, as described in *The Programmable Logic Data Book*. Simulation models use this pulse width parameter for determining HDL simulation for global reset and tri-state circuitry (initially developed for schematic design.)

Adapting Schematic Global Signal Methodology for VHDL

There are no global set/reset or output tristate enable pins on the simulation, synthesis, or implementation models of the register components in schematic-based designs. During synthesis, both the global and local reset and tristate-state enable signals are connected to the local pin. Schematic simulators can simulate global signals without a pin. The global signals are represented as internal signals in the schematic simulation model and the test vectors drive the internal global signals directly. If you want complete control of initialization, use registers with asynchronous set/reset to emulate the GSR, even if local set/reset is not required. Synchronous set/reset registers will initialize on their own at time zero. They can be synchronously set after that but cannot emulate GSR behavior after time zero. Some memory components without asynchronous clears will exhibit similar behavior.

In VHDL designs, you must declare as ports any signals that are stimulated or monitored from outside a module. 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.

You do not need to incorporate any ports into schematic designs for simulators to mimic the device's global reset (GSR) or global tristate (GTS) networks. Schematic simulators specify these signals on the register model as 'global' to indicate to the simulator that these signals are all connected. 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.

For the HDL synthesis flow, the global reset and output tristate enable signals are emulated with the local reset and tristate enable signals. Special implementation directives are put on the nets to move them to the special pre-routed nets for global signals.

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 initializations.

Global Signal Considerations (VHDL)

The following are important to 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 LogiCOREs.
- Intellectual property cores from the LogiCORE product line are represented as behavioral, RTL, or structural 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 5-7 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 Internal signal driving GSR
Case 3	Don't Care

Note: Reset-on-Configuration for PLDs is similar to Power-on-Reset for ASICs except it occurs at power-up and during configuration of the PLD.

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)

Complimentary 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, GR, or PRLD (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 GR, GSR, or PRLD 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.

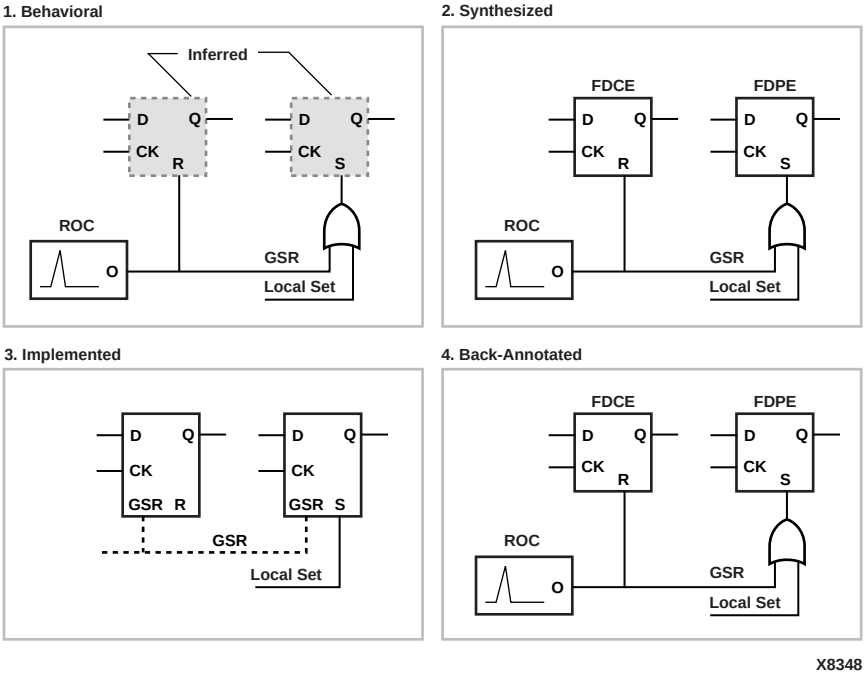

Figure 5-3 ROC Simulation and Implementation

- **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. 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. 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.

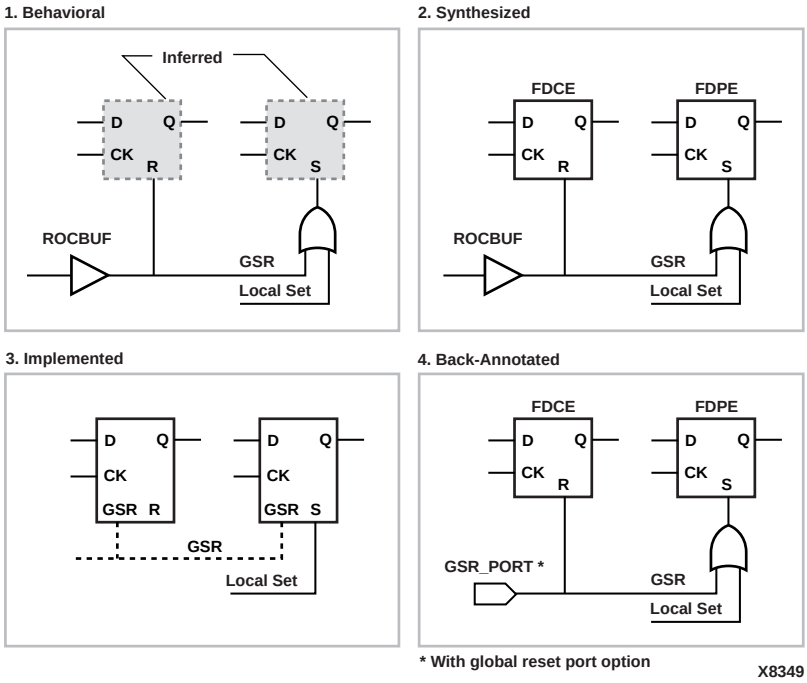

Figure 5-4 ROCBUF Simulation and Implementation

- **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 and 2B)

The STARTUP block is traditionally instantiated to identify the GR, PRLD, or GSR signals for implementation if the global reset on 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 Case 2A and 2B, 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 two 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 tri-stating 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 table below shows the correspondence of pins between STARTBUF and STARTUP.

Table 5-8 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
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;
```

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 5-9 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 Internal signal driving GTS
Case C	Don't Care

Case A is defined as follows.

- Tri-stating 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

to 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 GR, GSR, or PRLD 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 Xilinx online Data Book for the particular part and mode you have implemented. For example, a 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)

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.

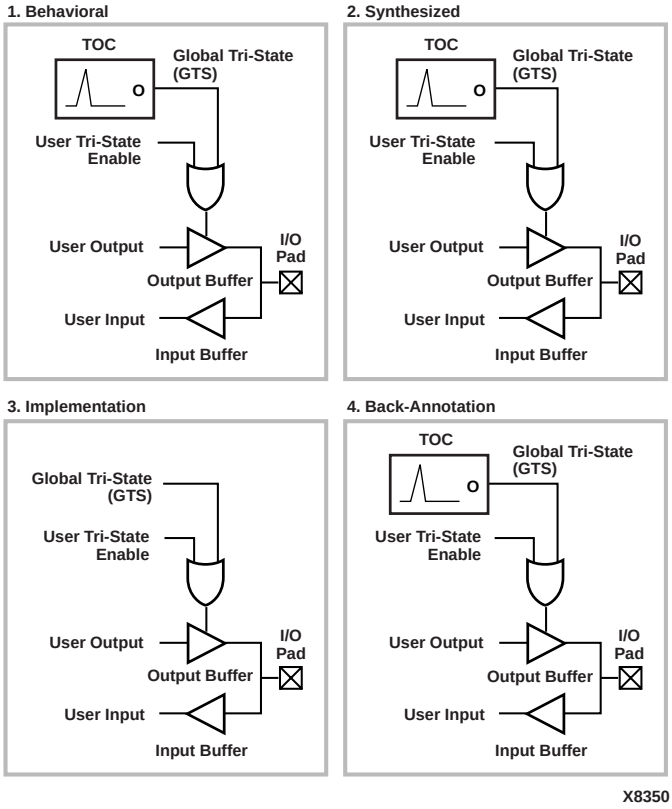

Figure 5-5 TOC Simulation and Implementation

- **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 B1)

For Case B1, 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 B1)

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') 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;
```

TOCBUF Model in Four Design Phases (Case B1)

The following figure shows the progression of the TOCBUF model and its interpretation in the four main design phases.

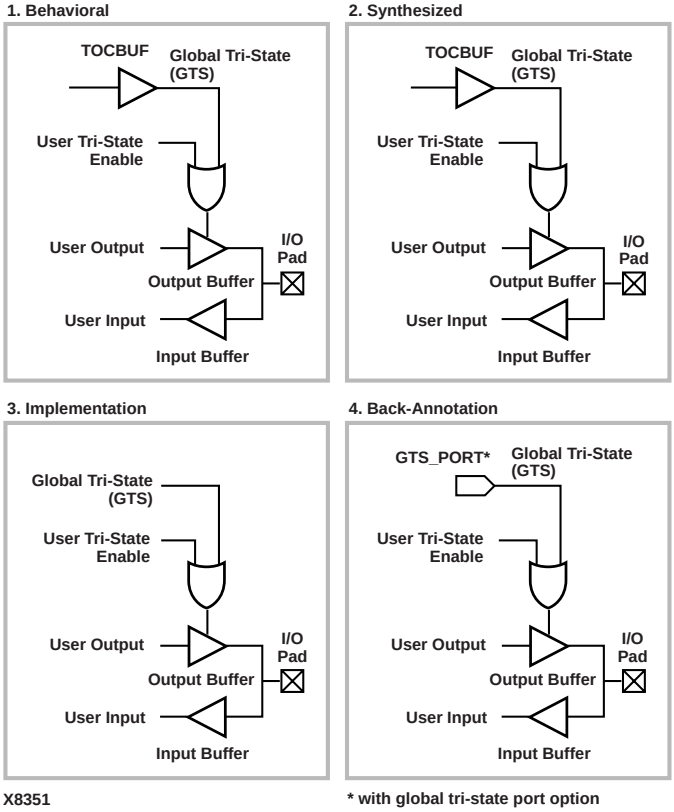

Figure 5-6 TOCBUF Simulation and Implementation

- **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 Oscillators (VHDL)

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 250 ns to 100ns. An example of this follows.

VHDL Oscillator Example

```
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

```
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.

Compiling Verilog Libraries

For some Verilog simulators, such as NC-Verilog and ModelSim, you may need to compile the Verilog libraries before you can use them for design simulations. A pre-compiled library methodology has the advantage of speeding up the simulation of your designs. You do not need to compile the libraries for Verilog-XL because it uses an interpretive compilation of the libraries. To simulate Xilinx designs, you need the following simulation libraries.

- **UniSim Library**—The UniSim library is used for behavioral (RTL) simulation with instantiated components in the netlist, and for post-synthesis (pre-M1) simulation. The Verilog library has separate libraries for each device family: uni3000, unisims (XC4000E/L/X, Spartan/XL, and Virtex), uni5200, uni9000.

- **LogiBLOX Library**—The LogiBLOX library is used for designs containing LogiBLOX components, during pre-synthesis (RTL), and post-synthesis simulation. Verilog uses SimPrim libraries.
- **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.

Compiling Libraries for ModelSim

For detailed instructions on compiling these simulation libraries, see the instructions in Xilinx Solution # 1923 which is available at http://www.xilinx.com/techdocs/1923.htm.

After compiling the libraries, notice that ModelSim creates a file called modelsim.ini. View this file and notice that the upper portion defines the locations of the compiled libraries. When doing a simulation, you must provide the modelsim.ini file either by copying the file directly to the directory where the HDL files are to be compiled and the simulation is to be run, or by setting the MODELSIM environment variable to the location of your master .ini file. You must set this variable since the ModelSim installation does not initially declare the path for you. For UNIX, type the following.

```
setenv MODELSIM /path/to/the/modelsim.ini
```

Setting Verilog Global Set/Reset

For Verilog simulation, all behaviorally described (inferred) and instantiated registers should have a common signal which asynchronously sets or resets the register. You must toggle the global set/reset signal (GSR for XC4000E/L/X, Spartan/XL, and Virtex designs, or GR for XC5200, XC3000A/L, or XC3100A/L designs). Toggling the global set/reset emulates the Power-On-Reset of the FPGA. If you do not do this, the flip-flops and latches in your simulation enter an unknown state.

The GSR signal in XC4000E/L/X, Spartan/XL, and Virtex devices, and the GR signal in XC5200 devices are active High. The GR signal in XC3000A/L and XC3100A/L devices are active Low.

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.

If you want to set the global set/reset pulse width so that it reflects the actual amount of time it takes for the chip to go through the reset process when power is supplied to it, refer to *The Programmable Logic Data Book* for the device you are simulating. The duration of the pulse is specified as T_{POR} (Power-On-Reset).

The general procedure for specifying global set/reset or global reset during a pre-NGDBuild Verilog UniSims simulation involves defining the global reset signals with 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 tri-state signals. However, Verilog allows a global signal to be modeled as a wire in a global module, and, thus, does not contain these cells.

Note: In the Xilinx software, the Verilog UniSims library is only used in RTL simulations of your designs. Simulation at other points in the flow use the Verilog SimPrims Libraries.

Defining GSR in a Test Bench

For pre-NGDBuild UniSims functional simulation, you must set the value of the appropriate Verilog global signals (glbl.GSR or glbl.GR) to the name of the GSR or GR net, qualified by the appropriate scope identifiers.

The scope identifiers are a combination of the test module scope and the design instance scope. The scope qualifiers are required because the scope information is needed when the glbl.GSR and glbl.GR wires are interpreted by the Verilog UniSims simulation models to emulate a global reset signal.

For post-NGDBuild and post-route timing simulation, the testfixture template (.tv file) produced by running NGD2VER with the –tf option contains most of the code previously described for defining and toggling GSR or GR.

Use the following steps to define the global set/reset signals in a testfixture for your design.

Note: In the following steps, *testfixture_name* refers to the test fixture module name and *instance_name* refers to the designated instance name for the instantiated design netlist within the test bench.

1. For Verilog simulation without a STARTUP block in design, Xilinx recommends naming the global set/reset net to *testfixture_name.instance_name*.GSR or *testfixture_name.instance_name*.GR (Verilog is case-sensitive), and the signal should be declared as a Verilog reg data-type.
2. For Verilog simulation with a STARTUP block in the design, the GSR/GR pin is connected to an external input port, and glbl.GSR/glbl.GR is defined within the STARTUP block to make the connection between the user logic and the global GSR/GR net embedded in the Unified models. For post-NGDBuild functional simulation, post-Map timing simulation, and post-route timing simulation, glbl.GSR/glbl.GR is defined in the Verilog netlist that is created by NGD2VER.

   The signal you toggle at the beginning of the simulation is the port or signal in your design that is used to control global set/reset. This is usually an external input port in the Verilog netlist, but it may also be a wire if global reset is controlled by logic internal to your design.
3. When invoking Verilog-XL, or ModelSim to run the simulation, compile the Verilog source files in any order since Verilog is compile order independent. However, Xilinx recommends that you specify the test fixture file before the Verilog netlist of your design, as in the following examples.

   - Cadence Verilog-XL

     For RTL simulation, enter the following.

     verilog –y $XILINX/verilog/src/unisims
     design.**stim** *design*.**v $XILINX/verilog/src/glbl.v**

     The path specified with the –y switch points the simulator to the UniSims models and is only necessary if Xilinx primitives are instantiated in your code. When targeting a device family other than the XC4000E/L/X, Spartan/XL, or Virtex families, change the *unisims* reference in the path to the targeted device family.

     For post-implementation simulation, enter the following.

verilog *design*.**stim** *time_sim*.**v $XILINX/verilog/src/glbl.v**

In this example, the same test fixture file is declared first followed by the simulation netlist created by the Xilinx tools. The name of the Xilinx simulation netlist may change depending on how the file was created. It is also assumed that the –ul switch was specified during NGD2VER to specify the location of the SimPrims libraries using the **'uselib** directive.

- MTI ModelSim

  For RTL simulation, enter the following.

  vlog *design*.**stim** *design*.**v $XILINX/verilog/src/glbl.v**

  vsim –L unisims *testfixture_name glbl*

  This example targets the XC4000E / L / X, Spartan / XL, or Virtex families and assumes the UniSims libraries are properly compiled and named *unisims*. For more information on the compilation of the ModelSim libraries, refer to http: / / www.xilinx.com / techdocs / 1923.htm

  For post-implementation simulation, enter the following.

  vlog *design*.**stim** *time_sim*.**v $XILINX/verilog/src/glbl.v**

  vsim –L simprims *testfixture_name glbl*

  This example is based on targeting the SimPrims libraries, which have been properly compiled and named *simprims*. Also, the name of the simulation netlist may change depending on how the file is created.

Note: 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

Designs without a STARTUP Block

If you do not have a STARTUP block in your design, you should add the following to the test fixture module.

- XC4000E / L / X, Spartan / XL, or Virtex devices.

```
reg GSR;

assign glbl.GSR = GSR;

assign testfixture_name.instance_name.GSR = GSR; //
Only for RTL modeling of GSR
```

- XC5200, XC3000A / L, and XC3100A / L devices.

```
reg GR;

assign glbl.GR = GR;

assign testfixture_name.instance_name.GR = GR; //
Only for RTL modeling of GR
```

For post-NGDBuild functional simulation, post-Map timing simulation, and post-route timing simulation, you must omit the assign statement for the global reset signal. This is because the net connections exist in the post-NGDBuild design, and retaining the assign definition causes a possible conflict with these connections.

Note: The terms “test bench” and “test fixture” are used synonymously throughout this manual.

Example 1: XC4000E/L/X, Spartan/XL, or Virtex RTL Functional Simulation (No STARTUP/ STARTUP_VIRTEX Block)

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.

You should reference the global set / reset net as GSR in XC4000E / L / X, Spartan / XL, or Virtex designs without a STARTUP / STARTUP_VIRTEX block. The Verilog module defining the global net must be referenced as glbl.GSR because this is how it is modeled in the Verilog UniSims library.

In the design code, declare GSR as a Verilog wire, however, it is not specified in the port list for the module. Describe 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

// FDCE instantiation
// GSR is modeled as a wire within a global module. So,
// CLR does not need to be connected to GSR and the flop
// will still be reset with GSR.

FDCE U0 (.Q (Q), .D (D), .C (CLK), .CE (1'b1), .CLR (1'b0));

endmodule
```

Since GSR is declared as a floating wire and is not in the port list, the synthesis tool optimizes the GSR signal out of the design. GSR is replaced later by the implementation software for all post-implementation simulation netlists.

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.

```
‘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
```

In this example, the active high GSR signal in the XC4000 family device is activated by driving it high. 100 ns later, it is deactivated by driving it low. (100 ns is an arbitrarily chosen value.)

You can use the same test fixture for simulating at other stages in the design flow if this methodology is used.

Example 2: XC5200 RTL Functional Simulation (No STARTUP Block)

For pre-NGDBuild functional simulation, the active High GR net in XC5200 devices should be simulated in the same manner as GSR for XC4000E / L / X, Spartan / XL, or Virtex.

In the design code, declare GR as a Verilog wire, however, it is not specified in the port list for the module. Describe GR to reset or set every inferred register or latch in your design. GR 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 GR;
reg [3:0] COUT;
```

```
always @(posedge GR or posedge CLK)
  begin
    if (GR == 1'b1)
      COUT = 4'h0;
    else
      COUT = COUT + 1'b1;
  end

// FDCE instantiation
// GR is modeled as a wire within a global module. So,
// CLR does not need to be connected to GR and the flop
// will still be reset with GR.

FDCE U0 (.Q (Q), .D (D), .C (CLK), .CE (1'b1), .CLR (1'b0));

endmodule
```

Since GR is declared as a floating wire and is not in the port list, the synthesis tool optimizes the GR signal out of the design. GR is replaced later by the implementation software for all post-implementation simulation netlists.

In the test fixture file, set GR to test.uut.GR (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 GR is made in the testfixture via an assign statement.

```
`timescale 1 ns / 1 ps

module test;
reg GR;

assign glbl.GR = GR;

assign test.uut.GR = GR;

.
.
.
```

```
initial begin
GR = 1; // if you wish to reset/set the device;
#100 GR = 0;     // deactivate GR

end
```

In this example, the active high GR signal in the XC5200 family device is activated by driving it high. 100 ns later, it is deactivated by driving it low. (100 ns is an arbitrarily chosen value.).

You can use the same test fixture for simulating at other stages in the design flow if this methodology is used.

Example 3: XC3000A/L, or XC3100A/L RTL Functional Simulation (No STARTUP Block)

For pre-NGDBuild functional simulation, asserting global reset in XC3000A/L or XC3100A/L designs is almost identical to the procedure for asserting global reset in XC5200 designs, except that GR is active Low.

Note: The STARTUP block is not supported on XC3000A/L or XC3100A/L devices.

In the design code, declare GR as a Verilog wire, however, it is not specified in the port list for the module. Describe GR to reset or set every inferred register or latch in your design. GR 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 GR;

reg [3:0] COUT;


always @(negedge GR or posedge CLK)

  begin

    if (GR == 1'b0)
```

```
      COUT = 4'h0;
    else
      COUT = COUT + 1'b1;
  end

// FDCE instantiation
// GR is modeled as a wire within a global module. So,
// CLR does not need to be connected to GR and the flop
// will still be reset with GR.

FDCE U0 (.Q (Q), .D (D), .C (CLK), .CE (1'b1), .CLR (1'b0));

endmodule
```

Since GR is declared as a floating wire and is not in the port list, the synthesis tool optimizes the GR signal out of the design. Although this is correct in the hardware, it is actually an implicit connection, and not listed in the netlist (XNF or EDIF). GR is replaced later by the implementation software for all post-implementation simulation netlists.

In the test fixture file, set GR to test.uut.GR (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 GR is made in the testfixture via an assign statement.

```
`timescale 1 ns / 1 ps

module test;
reg GR;

assign glbl.GR = GR;

assign test.uut.GR = GR;

.
.
.
initial begin
```

```
GR = 0; // if you wish to reset/set the device;
#100 GR = 1;     // deactivate GR

end
```

In this example, the active Low GR signal in the XC3000A / L and XC3100A / L family device is activated by driving it high. 100 ns later, it is deactivated by driving it low. (100 ns is an arbitrarily chosen value.).

The Global Reset (GR) signal in the XC3000A / L and XC3100A / L architecture is modeled differently in functional simulation netlists and SimPrims library-based netlists generated by NGD2VER. In the Verilog Unified Library, GR is modeled as a wire within a global module, while in a SimPrims-based netlist, it is always modeled as an external port. As a result, you cannot use the same test bench file for both Unified library simulation and SimPrims-based simulation.

Designs with a STARTUP Block

If you do have a STARTUP block in your design, the signal you toggle is the external input port that controls the global reset pin of the STARTUP block. You should add the following to the test fixture module for RTL modeling of the global reset pin.

Note: The terms “test bench” and “test fixture” are used synonymously throughout this manual.

- XC4000E / L / X, Spartan / XL, and Virtex devices.

  `reg` *port_connected_to_GSR_pin*`;`

- XC5200 devices.

  `reg` *port_connected_to_GR_pin*`;`

For post-NGDBuild functional simulation, post-map timing simulation, and post-route timing simulation, you must omit the assign statement for the global reset signal. This is because the net connections exist in the post-NGDBuild design, and retaining the assign definition causes a possible conflict with these connections.

By default for XC4000E / L / X, XC5200, Spartan / XL, and Virtex devices, the GSR / GR pin is active High. To change the polarity of these signals in your Verilog code, instantiate or infer an inverter to the net that sources the GSR / GR pin of the STARTUP block.

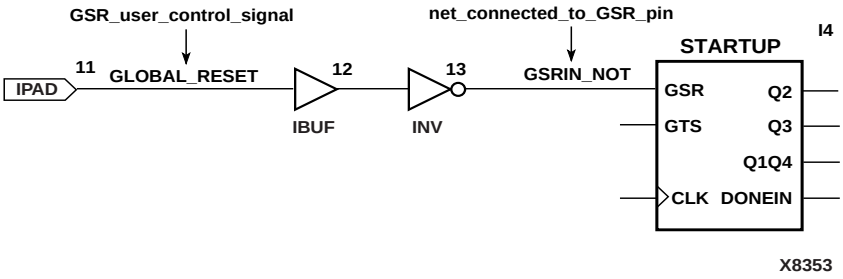

Figure 5-7 Inverted GSR

The inversion is absorbed inside the STARTUP block; a function generator is not used to generate the inverter.

In the following Verilog code, GSR is listed as a top-level port.

```
module my_counter (MYGSR, CLK, D, Q, COUT);
input MYGSR, CLK, D;
output Q;
output [3:0] COUT;

reg [3:0] COUT;

wire INV_GSR;
assign INV_GSR = !MYGSR; // Inverted GSR

// Modeling inverted GSR with RTL code
always @(posedge INV_GSR or posedge CLK)
  begin
    if (INV_GSR == 1'b1)
      COUT = 4'h0;
    else
      COUT = COUT + 1'b1;
  end
```

```
// FDCE instantiation
// 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 (INV_GSR), .GTS (1'b0), .CLK (1'b0));

endmodule
```

Example 1: XC4000E/L/X and Spartan/XL Simulation with STARTUP, or Virtex with STARTUP_VIRTEX

In the following figure, MYGSR is an external user signal that controls GSR.

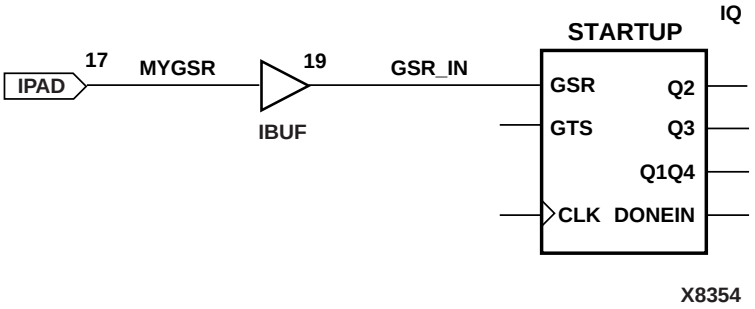

Figure 5-8 Verilog User-Controlled GSR

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 netlist (XNF and EDIF).

```
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

// FDCE instantiation
// 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 in XC4000E/L/X and Spartan/XL designs, or the STARTUP_VIRTEX in Virtex.

The global set/reset control signal should be toggled High, then Low in an initial block.

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

```
module test;
reg GSR;

.
.
.
initial begin
GSR = 1; // if you wish to reset/set the device;
#100 GSR = 0;     // deactivate GSR

end
```

In addition, a Verilog global signal called glbl.GSR is defined within the STARTUP / STARTUP_VIRTEX block to make the connection between the user logic and the global GSR net embedded in the Unified models. For post-NGDBuild functional simulation, post-Map timing simulation, and post-route timing simulation, glbl.GSR is defined in the Verilog netlist that is created by NGD2VER.

Example 2: XC5200 Simulation with STARTUP

For XC5200 designs with a STARTUP block, you should simulate the net controlling GR in the same manner as for the XC4000E / L / X, Spartan / XL, and Virtex.

Substitute MYGR for MYGSR in Example 1 to obtain the testfixture fragment for simulating GR in a Verilog RTL or post-synthesis simulation.

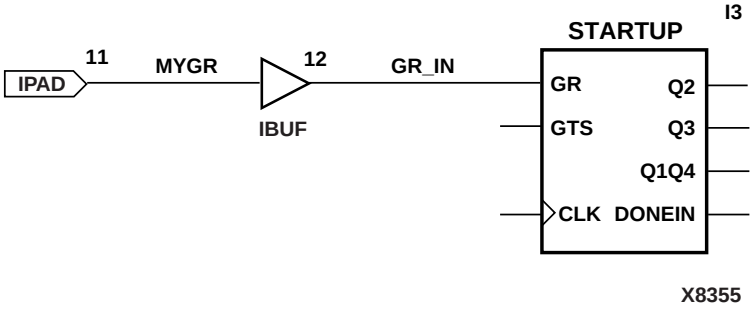

Figure 5-9 Verilog User-Controlled Inverted GR

In addition, a Verilog global signal called glbl.GR is defined within the STARTUP block to make the connection between the user logic and the global GR net embedded in the Unified models. For post-

NGDBuild functional simulation, post-map timing simulation, and post-route timing simulation, glbl.GR is defined in the Verilog netlist that is created by NGD2VER.

Example 3: XC3000A/L and XC3100A/L Designs

STARTUP is not supported or required in XC3000A / L and XC3100A / L designs. Follow the procedure for XC3000A / L and XC3100A / L designs without STARTUP blocks.

Setting Verilog Global Tristate (XC4000, Spartan, and XC5200 Outputs Only)

XC4000E / L / X, Spartan / XL, Virtex, and XC5200 devices also have a global control signal (GTS) that tristates all output pins. This allows you to isolate the actual device part during board level testing. You can also tristate the FPGA device outputs during board level simulation to assist in debugging simulation. In most cases, GTS is deactivated so that the outputs are active.

Although the STARTUP / STARTUP_VIRTEX component also gives you the option of controlling the global tristate net from an external pin, it is usually used for controlling global reset. In this case, you can leave the GTS pin unconnected in the design entry phase, and it will float to its inactive state level. The global tristate net, GTS, is implemented in designs even if a STARTUP / STARTUP_VIRTEX block is not instantiated. You can deactivate GTS by driving it low in your test fixture file, or by connecting the GTS pin to GND in your input design

Defining GTS in a Test Bench

For pre-NGDBuild UniSim functional simulation, you must set the value of the appropriate Verilog global signal, glbl.GTS, to the name of the GTS net, qualified by the appropriate scope identifiers.

The scope identifiers are a combination of the test module scope and the design instance scope. The scope qualifiers are required because the scope information is needed when the glbl.GTS wire is interpreted by the Verilog UniSim simulation models to emulate a global tri-state signal.

For post-NGDBuild and post-route timing simulation, the testfixutre template (.tv file) produced by running NGD2VER with the –tf option contains most of the code previously described for defining and toggling GTS.

The general procedure for specifying GTS is similar to that used for specifying the global set / reset signals, GSR and GR. You define the global tristate signal with Verilog global module, glbl.GTS. If you do not want to specify GTS for simulation, you do not need to change anything in your design or testfixture.

The GTS signal in XC4000E / L / X, Spartan / XL, Virtex, and XC5200 devices is active High. This global module is not used in timing simulation when there is a STARTUP / STARTUP_VIRTEX block in your design and the GTS pin is connected.

Designs without a STARTUP Block

If you do not have a STARTUP block in your design, you should add the following to the test fixture module.

```
reg GTS;

assign glbl.GTS = GTS;

assign testfixture_name.instance_name.GTS = GTS;

// Only for RTL simulation modeling of GTS
```

For post-NGDBuild functional simulation, post-map timing simulation, and post-route timing simulation, you must omit the assign statement for the global tri-state signal. This is because the net connections exist in the post-NGDBuild design, and retaining the assign definition causes a possible conflict with these connections.

Note: The terms “test bench” and “test fixture” are used synonymously throughout this manual.

XC4000E/L/X, Spartan/XL, Virtex and XC5200 RTL Functional Simulation (No STARTUP Block)

You can drive the GTS signal in a test fixture file at the beginning of a pre-NGDBuild RTL or post-synthesis functional simulation. The global tristate net is named GTS in XC4000E / L / X, Spartan / XL, Virtex, and XC5200 designs. The Verilog module defining the global

tri-state net must be referenced as glbl.GTS because this is how it is modeled in the Verilog UniSim library.

Designs with a STARTUP Block

If you do have a STARTUP block in your design, the signal you toggle at the beginning of simulation is the port or signal in your design that is used to control global tristate. This is usually an external input port in the Verilog netlist, but can be a wire if global tristate is controlled by internal logic in your design. A Verilog global signal called glbl.GTS is defined within the STARTUP block to make the connection between the user logic and the global GTS net embedded in the Unified models

Example 1: XC4000E/L/X, Spartan/XL, Virtex, and XC5200 Simulation (With STARTUP/ STARTUP_VIRTEX, GTS Pin Connected)

In the following figure, MYGTS is an external user signal that controls GTS.

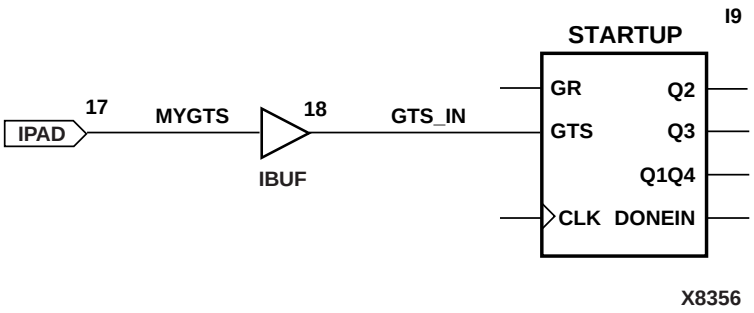

Figure 5-10 Verilog User-Controlled Inverted GTS

The following is an example of controlling the global tri-state 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 in XC4000E / L / X and Spartan / XL design, or the STARTUP_VIRTEX in Virtex. The global GTS model in the UniSim simulation models for output buffers (OBUF, OBUFT, and so on).

The global tri-state control signal should be toggled High, then Low in an initial block.

```
module test;
reg MYGTS;

.
.
.
initial begin
MYGTS = 1; // if you wish to tristate the device;
#100 MYGTS = 0;     // deactivate GTS

end
```

Example 2: XC4000E/L/X, Spartan/XL, Virtex, and XC5200 Simulation (With STARTUP/ STARTUP_VIRTEX, 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.

```
module test;
reg GTS;

assign glbl.GTS = GTS;

.
.
.
initial begin
GTS = 1; // if you wish to tristate the device;
#100 GTS = 0;     // deactivate GTS

end
```

Note: For post-route timing simulation, you can use the same test bench.

Appendix A

Accelerate FPGA Macros with One-Hot Approach

By Steven K. Knapp

Xilinx Inc.
2100 Logic Dr.
San Jose, CA 95124

Reprinted with permission from *Electronic Design*, September 13, 1990.

State machines—one of the most commonly implemented functions with programmable logic—are employed in various digital applications, particularly controllers. However, the limited number of flip-flops and the wide combinatorial logic of a PAL device favors state machines that are based on a highly encoded state sequence. For example, each state within a 16-state machine would be encoded using four flip-flops as the binary values between 0000 and 1111.

A more flexible scheme—called one-hot encoding (OHE)—employs one flip-flop per state for building state machines. Although it can be used with PAL-type programmable-logic devices (PLDs), OHE is better suited for use with the fan-in limited and flip-flop-rich architectures of the higher-gate-count filed-programmable gate arrays (FPGAs), such as offered by Xilinx, Actel, and others. This is because OHE requires a larger number of flip-flops. It offers a simple and easy-to-use method of generating performance-optimized state-machine designs because there are few levels of logic between flip-flops.

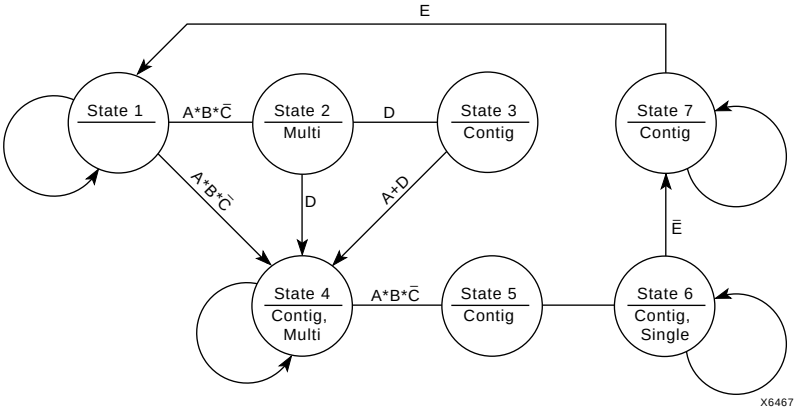

Figure A-1 A Typical State Machine Bubble

(In reference to the figure above) A Typical State Machine Bubble diagram shows the operation of a seven-state state machine that reacts to inputs A through E as well as previous-state conditions.

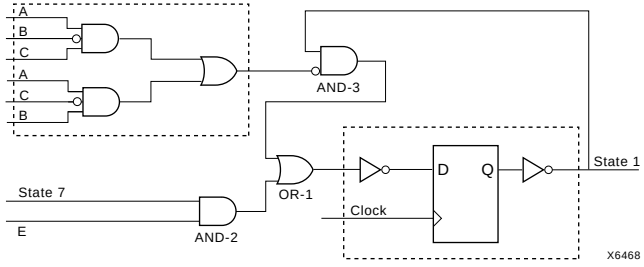

Figure A-2 Inverters

(In reference to the figure above) Inverters are required at the D input and the Q output of the state flip-flop to ensure that it powers on in the proper state. Combinatorial logic decodes the operations based on the input conditions and the state feedback signals. The flip-flop will remain in State 1 as long as the conditional paths out of the states are not valid.

A state machine implemented with a highly encoded state sequence will generally have many, wide-input logic functions to interpret the inputs and decode the states. Furthermore, incorporating a highly encoded state machine in an FPGA requires several levels of logic

between clock edges because multiple logic blocks will be needed for decoding the states. A better way to implement state machines in FPGAs is to match the state-machine architecture to the device architecture.

Limiting Fan-In

A good state-machine approach for FPGAs limits the amount of fan-in into one logic block. While the one-hot method is best for most FPGA applications, binary encoding is still more efficient in certain cases, such as for small state machines. It's up to the designer to evaluate all approaches before settling on one for a particular application.

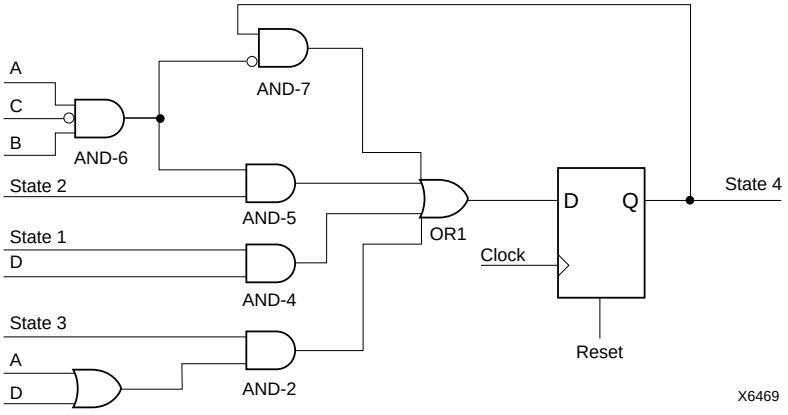

Figure A-3 The Seven States

(In reference to the figure above) Of the seven states, the state-transition logic required for State 4 is the most complex, requiring inputs from three other state outputs as well as four of the five condition signals (A - D).

FPGAs are high-density programmable chips that contain a large array of user-configurable logic blocks surrounded by user-programmable interconnects. Generally, the logic blocks in an FPGA have a limited number of inputs. The logic block in the Xilinx XC-3000 series, for instance, can implement any function of five or less inputs. In contrast, a PAL macrocell is fed by each input to the chip and all of the flip-flops. This difference in logic structure between PALs and FPGAs is important for functions with many inputs: where a PAL could implement a many-input logic function in one level of logic, an

FPGA might require multiple logic layers due to the limited number of inputs.

The OHE scheme is named so because only one state flip-flop is asserted, or “hot”, at a time. Using the one-hot encoding method for FPGAs was originally conceived by High-Gate Design—a Saratoga, Calif.-based consulting firm specializing in FPGA designs.

The OHE state machine's basic structure is simple—first assign an individual flip-flop to each state, and then permit only one state to be active at any time. A state machine with 16 states would require 16 flip-flops using the OHE approach; a highly encoded state machine would need just four flip-flops. At first glance, OHE may seem counter-intuitive. For designers accustomed to using PLDs, more flip-flops typically indicates either using a larger PLD or even multiple devices.

In an FPGA, however, OHE yields a state machine that generally requires fewer resources and has higher performance than a binary-encoded implementation. OHE has definite advantages for FPGA designs because it exploits the strengths of the FPGA architecture. It usually requires two or less levels of logic between clock edges than binary encoding. That translates into faster operation. Logic circuits are also simplified because OHE removes much of the state-decoding logic—a one-hot-encoded state machine is already fully decoded.

OHE requires only one input to decode a state, making the next-state logic simple and well-suited to the limited fan-in architecture of FPGAs. In addition, the resulting collection of flip-flops is similar to a shift-register-like structure, which can be placed and routed efficiently inside an FPGA device. The speed of an OHE state machine remains fairly constant even as the number of states grows. In contrast, a highly encoded state machine's performance drops as the states grow because of the wider and deeper decoding logic that's required.

To build the next-state logic for OHE state machine is simple, lending itself to a “cookbook” approach. At first glance, designers familiar with PAL-type devices may be concerned by the number of potential illegal states due to the sparse state encoding. This issue, to be discussed later, can be solved easily.

A typical, simple state machine might contain seven distinct states that can be described with the commonly used circle-and-arc bubble diagrams, see the “A Typical State Machine Bubble” figure. The label

above the line in each “bubble” is the state`s name. The labels below the line are the outputs asserted while the state is active. In the example, there are seven states labeled State 1-7. The “arcs” that feed back into the same state are the default paths. These will be true only if no other conditional paths are true.

Each conditional path is labeled with the appropriate logical condition that must exist before moving to the next state. All of the logic inputs are labeled as variables A through E. The outputs from the state machine are called Single, Multi, and Contig. For this example, State 1, which must be asserted at power-on, has a double-inverted flip-flop structure (shaded region of the “Inverters” figure)

The state machine in the example was built twice, once using OHE and again with the highly encoded approach employed in most PAL designs. A Xilinx XC3020-100 2000-gate FPGA was the target for both implementations. Though the OHE circuit required slightly more logic than the highly-encoded state machine, the one-hot state machine operated 17% faster (see the table). Intuitively, the one-hot method might seem to employ many more logic blocks than the highly encoded approach. But the highly encoded state machine needs more combinatorial logic to decode the encoded state values.

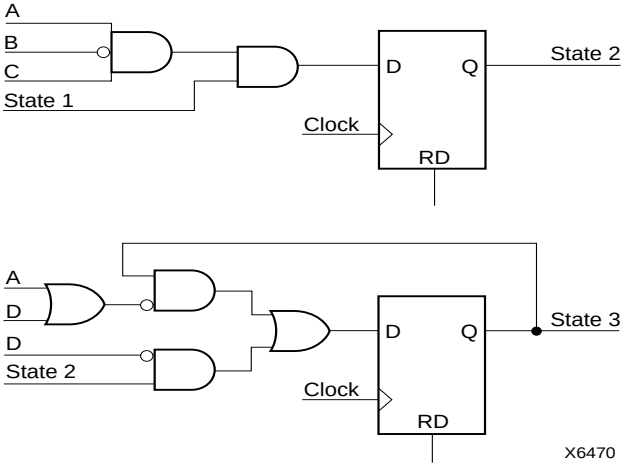

Figure A-4 Only a Few Gates

(In reference to the figure above) Only a few gates are required by States 2 and 3 to form simple state-transition logic decoding. Just

two gates are needed by State 2 (top), while four simple gates are used by State 3 (bottom).

The OHE approach produces a state machine with a shift-register structure that almost always outperforms a highly encoded state machine in FPGAs. The one-state design had only two layers of logic between flip-flops, while the highly encoded design had three. For other applications, the results can be far more dramatic. In many cases, the one-hot method yields a state machine with one layer of logic between clock edges. With one layer of logic, a one-hot state machine can operate at 50 to 60 MHz.

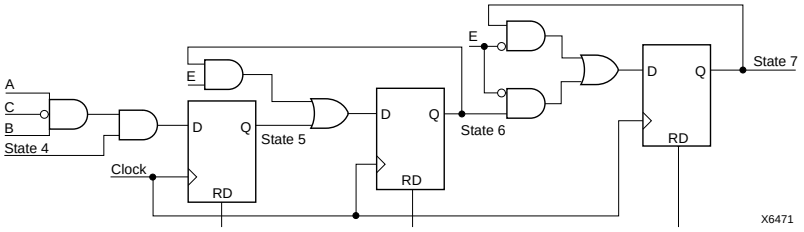

Figure A-5 Looking Nearly the Same

(In reference to the figure above) Looking nearly the same as a simple shift register, the logic for States 5, 6, and 7 is very simple. This is because the OHE scheme eliminates almost all decoding logic that precedes each flip-flop.

The initial or power-on condition in a state machine must be examined carefully. At power-on, a state machine should always enter an initial, known state. For the Xilinx FPGA family, all flip-flops are reset at power-on automatically. To assert an initial state at power-on, the output from the initial-state flip-flop is inverted. To maintain logical consistency, the input to flip-flop also is inverted.

All other states use a standard, D-type flip-flop with an asynchronous reset input. The purpose of the asynchronous reset input will be discussed later when illegal states are covered.

Once the start-up conditions are set up, the next-state transition logic can be configured. To do that, first examine an individual state. Then count the number of conditional paths leading into the state and add an extra path if the default condition is to remain in the same state. Second, build an OR-gate with the number of inputs equal to the number of conditional paths that were determined in the first step.

Third, for each input of the OR-gate, build an AND-gate of the previous state and its conditional logic. Finally, if the default should remain in the same state, build an AND-gate of the present state and the inverse of all possible conditional paths leaving the present state.

To determine the number of conditional paths feeding State 1, examine the state diagram—State 1 has one path from State 7 whenever the variable E is true. Another path is the default condition, which stays in State 1. As a result, there are two conditional paths feeding State 1. Next, build a 2-input OR-gate—one input for the conditional path from State 7, the other for the default path to stay in State 1 (shown as OR-1 in the “Inverters” figure).

The next step is to build the conditional logic feeding the OR-gate. Each input into the OR-gate is the logical AND of the previous state and its conditional logic feeding into State 1. State 7, for example, feeds State 1 whenever E is true and is implemented using the gate called AND-2, in the “Inverters” figure. The second input into the OR-gate is the default transition that's to remain in State 1. In other words, if the current state is State 1, and no conditional paths leaving State 1 are valid, then the state machine should remain in State 1. Note in the state diagram that two conditional paths are leaving State 1, in the “A Typical State Machine Bubble” figure.

The first path is valid whenever (A*B*C) is true, which leads into State 2. The second path is valid whenever (A*B*C) is true, leading into State 4. To build the default logic, State 1 is ANDed with the inverse of all the conditional paths leaving State 1. The logic to perform this function is implemented in the gate labeled AND-3 and the logic elements that feed into the inverting input of AND-3, in the “Inverters” figure.

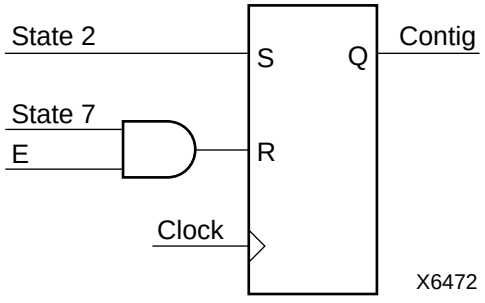

Figure A-6 S-R Flip-Flops

(In reference to the figure above) S-R Flip-Flops offer another approach to decoding the Contig output. They can also save logic blocks, especially when an output is asserted for a long sequence of contiguous states.

State 4 is the most complex state in the state-machine example. However, creating the logic for its next-state control follows the same basic method as described earlier. To begin with, State 4 isn't the initial state, so it uses a normal D-type flip-flop without the inverters. It does, however, have an asynchronous reset input, three paths into the state, and a default condition that stays in State 4. Therefore, four-input OR-gate feeds the flip-flop (OR-1 in the "The Seven States" figure).

The first conditional path comes from State 3. Following the methods established earlier, an AND of State 3 and the conditional logic, which is A ORed with D, must be implemented (AND-2 and OR-3 in the "The Seven States" figure). The next conditional path is from State 2, which requires an AND of State 2 and variable D (AND-4 in the "The Seven States" figure). Lastly, the final conditional path leading into State 4 is from State 1. Again, the State-1 output must be ANDed with its conditional path logic—the logical product, A*B*C (AND-5 and AND-6 in the "The Seven States" figure).

Now, all that must be done is to build the logic that remains in State 4 when none of the conditional paths away from State 4 are true. The path leading away from State 4 is valid whenever the product, A*B*C, is true. Consequently, State 4 must be ANDed with the inverse of the product, A*B*C. In other words, "keep loading the flip-flop with a high until a valid transfer to the next state occurs." The default path logic uses AND-7 and shares the output of AND-6.

One-State vs. Binary Encoding Methods

Method	Number of Logic Blocks	Worst-case performance
One-hot	7.5	40 Mhz
Binary encoding	7.0	34 Mhz

Configuring the logic to handle the remaining states is very simple. State 2, for example, has only one conditional path, which comes from State 1 whenever the product A*B*C is true. However, the state machine will immediately branch in one of two ways from State 2,

depending on the value of D. There's no default logic to remain in State 2, the "Only a Few Gates" figure. State 3, like States 1 and 4 has a default state, and combines the A, D, State 2, and State 3 feedback to control the flop-flop's D input in the "Only a Few Gates" figure.

State 5 feeds State 6 unconditionally. Note that the state machine waits until variable E is low in State 6 before proceeding to State 7. Again, while in State 7, the state machine waits for variable E to return to true before moving to State 1 in the "Looking Nearly the Same" figure.

Output Definitions

After defining all of the state transition logic, the next step is to define the output logic. The three output signals—Single, Multi, and Contig—each fall into one of three primary output types:

1. Outputs asserted during one state, which is the simplest case. The output signal Single, asserted only during State 6, is an example.
2. Outputs asserted during multiple contiguous states. This appears simple at first glance, but a few techniques exist that reduce logic complexity. One example is Contig. It's asserted from State 3 to State 7, even though there's a branch at State 2.
3. Outputs asserted during multiple, non-contiguous states. The best solution is usually brute-force decoding of the active states. One such example is Multi, which is asserted during State 2 and State 4.

OHE makes defining outputs easy. In many cases, the state flip-flop is the output. For example, the Single output also is the flip-flop output for State 6; no additional logic is required. The Contig output is asserted throughout States 3 through 7. Though the paths between these states may vary, the state machine will always traverse from State 2 to a point where Contig is active in either State 3 or State 4.

There are many ways to implement the output logic for the Contig output. The easiest method is to decode States 3, 4, 5, 6, and 7 with a 5-input OR gate. Any time the state machine is in one of these states, Contig will be active. Simple decoding works best for this state machine example. Decoding five states won't exceed the input capability of the FPGA logic block.

Additional Logic

However, when an output must be asserted over a longer sequence of states (six or more), additional layers of decoding logic would be required. Those additional logic layers reduce the state machine's performance.

Employing S-R flip-flops gives designers another option when decoding outputs over multiple, contiguous states.Though the basic FPGA architecture may not have physical S-R flip-flops, most macro-cell libraries contain one built from logic and D-type flip-flops. Using S-R flip-flops is especially valuable when an output is active for six or more contiguous states.

The S-R flip-flop is set when entering the contiguous states, and reset when leaving. It usually requires extra logic to look at the state just prior to the beginning and ending state. This approach is handy when an output covers multiple, non-contiguous states, assuming there are enough logic savings to justify its use.

In the example, States 3 through 7 can be considered contiguous. Contig is set after leaving State 2 for either States 3 or 4, and is reset after leaving State 7 for State 1. There are no conditional jumps to states where Contig isn‘t asserted as it traverses from State 3 or 4 to State 7. Otherwise, these states would not be contiguous for the Contig output.

The Contig output logic, built from an S-R flip-flop, will be set with State 2 and reset when leaving State 7 in the “S-R Flip-Flops” figure. As an added benefit, the Contig output is synchronized to the master clock. Obvious logic reduction techniques shouldn't be overlooked either. For example, the Contig output is active in all states except for States 1 and 2. Decoding the states where Contig isn't true, and then asserting the inverse, is another way to specify Contig.

The Multi output is asserted during multiple, non-contiguous states - exclusively during States 2 and 4. Though States 2 and 4 are contiguous in some cases, the state machine may traverse from State 2 to State 4 via State 3, where the Multi output is unasserted. Simple decoding of the active states is generally best for non-contiguous states. If the output is active is active during multiple, non-contiguous states over long sequences, the S-R flip-flop approach described earlier may be useful.

One common issue in state-machine construction deals with preventing illegal states from corrupting system operation. Illegal states exist in areas where the state machine's functionality is undefined or invalid. For state machines implemented in PAL devices, the state-machine compiler software usually generates logic to prevent or to recover from illegal conditions.

In the OHE approach, an illegal condition will occur whenever two or more states are active simultaneously. By definition, the one-hot method makes it possible for the state machine to be in only one state at a time. The logic must either prevent multiple, simultaneous states or avoid the situation entirely.

Synchronizing all of the state-machine inputs to the master clock signal is one way to prevent illegal states. "Strange" transitions won't occur when an asynchronous input changes too closely to a clock edge. Though extra synchronization would be costly in PAL devices, the flip-flop-rich architecture of an FPGA is ideal.

Even off-chip inputs can be synchronized in the available input flip-flops. And internal signals can be synchronized using the logic block's flip-flops (in the case of the Xilinx LCAs). The extra synchronization logic is free, especially in the Xilinx FPGA family where every block has an optional flip-flop in the logic path.

Resetting State Bits

Resetting the state machine to a legal state, either periodically or when an illegal state is detected, give designers yet another choice. The Reset Direct (RD) inputs to the flip-flops are useful in this case. Because only one state bit should be set at any time, the output of a state can reset other state bits. For example, State 4 can reset State 3.

If the state machine did fall into an illegal condition, eventually State 4 would be asserted, clearing State 3. However, State 4 can't be used to reset State 5, otherwise the state machine won't operate correctly. To be specific, it will never transfer to State 5; it will always be held reset by State 4. Likewise, State 3 can reset State 2, State 5 can reset State 4, etc.—as long as one state doesn't reset a state that it feeds.

This technique guarantees a periodic, valid condition for the state machine with little additional overhead. Notice, however, that State 1 is never reset. If State 1 were "reset", it would force the output of State 1 high, causing two states to be active simultaneously (which, by definition, is illegal).

Index

A

active_low_gsr design, 4-24
after xx ns statement, 2-2
arithmetic functions
    gate reduction, 2-38
    ordering and grouping, 2-3
    resource sharing, 2-32
ASIC
    comparing to FPGA, 1-3, 2-1
asynchronous reset pin, 2-41
asynchronous set pin, 2-41

B

barrel shifter design, 2-17
bi-directional I/O, 4-70
    inferring, 4-70
    instantiating, 4-72
    using LogiBLOX, 4-75
binary encoded state machine, 4-27
boundary scan, 4-61
    instantiating in HDL, 4-62
BSCAN, 4-61
BUFGP, 4-3
BUFGS, 4-3
BUFT *see* tristate buffer

C

capitalization style in code, 2-4
case statement, 2-3
    comparing to if statement, 2-58
    design example, 2-61
    syntax, 2-50
    when to use, 2-51
CLB
    XC4000, 2-42
clear pin, 2-41, 4-9
clock buffers
    inserting, 4-6
    instantiating, 4-2, 4-7
clock enable pin, 2-41, 2-45
combinatorial feedback loop, 2-28
comments in code, 2-12
compile run script, 3-7
compiling large designs, 3-7
compiling your design, 3-6, 3-7
conditional expression, 2-26
constants, 2-8
constraint precedence, 3-13
cost-based clean-up option, 3-26
creating readable code, 2-10

D

D register, 2-28
    design, 2-7
decoders, 4-41
delay-based clean-up option, 3-26
design compiling, 3-6
design entry, 3-5
design examples
    installing, 1-5
design flow
    description, 3-1

diagram, 3-2
using the command line, 3-4
using the Design Manager, 3-3
design hierarchy, 3-5, 4-89, 4-90
design performance, 3-21
DesignWare
gate reduction, 2-38
resource sharing, 2-32
device downloading, 3-28
directory tree structure, 1-11
disk space requirements, 1-6
don't touch attribute, 4-18
downloading files, 1-7, 1-9
downloading to the device, 3-28

E

else statement, 2-28
entering your design, 3-5
enumerated type encoded state machine, 4-33
extracting downloaded files, 1-10

F

Field Programmable Gate Array *see* FPGA
file transfer protocol, 1-9
Finite State Machine, 4-33, 4-40
changing encoding style, 4-39
extraction commands, 4-33
flip-flop, 2-30
formatting styles, 2-4
FPGA
comparing to ASIC, 1-3, 2-1
creating with HDLs, 4-1
global clock buffer, 4-2
system features, 1-4, 4-1
FPGA compiler, 1-4
from
to style timing constraint, 3-10
FSM *see* Finite State Machine
functional simulation, 1-2, 3-5, 5-2, 5-4
comparing to synthesis, 2-1

G

gate reduction
CLB count, 2-40
definition, 2-38
delay, 2-40
design examples, 2-39
gated clocks, 2-45
global clock buffer, 4-2
global longlines, 4-5
global set/reset, 4-9, 5-26, 5-51
increasing performance, 4-11
STARTUP block, 4-9
test bench, 5-52
global signals, 5-21, 5-24
GSR *see* global set/reset
GSRIN, 4-9
GTS, 5-38, 5-67
guide option, 3-26

H

hardware description language *see* HDL
HDL
also see Verilog
also see VHDL
coding for FPGAs, 4-1
coding hints, 2-1
converting to gates, 1-2
definition, 1-1
designing FPGAs, 1-2, 1-3
FPGA system features, 4-1
boundary scan, 4-61
global clock buffer, 4-2
global set/reset, 4-9
I/O decoders, 4-41
implementing logic with IOBs, 4-65
on-chip RAM, 4-52
implementing registers, 2-26
schematic entry design hints, 2-17
hdl_resource_allocation command, 2-36

hdlin_check_no_latch command, 2-29
hierarchy in designs, 1-4
high-density design flow, 3-1
hold-time requirement, 2-28, 4-66

I

I/O decoder, 4-41
if statement, 2-30
  comparing to case statement, 2-58
  design example, 2-59
  registers, 2-30
  syntax, 2-49
  when to use, 2-49
if-case statement
  design example, 2-54
if-else statement, 2-3, 2-49
ignore timing paths, 3-11
indenting HDL code, 2-10
INIT=S attribute, 4-11, 4-17, 4-41
initialization statement, 2-4
Insert Pads command, 4-2
installation
  design examples, 1-5
  directory tree structure, 1-11
  disk space requirements, 1-6
  downloading files, 1-7, 1-9
  extracting downloaded files, 1-10
  file transfer protocol, 1-9
  internet site, 1-7, 1-9
  memory requirements, 1-6
  software requirements
    Synopsys FPGA compiler, 1-5
    Xilinx Development System, 1-5
    XSI, 1-5
  Synopsys startup file, 1-15
  tactical software, 1-5
  technical support, 1-15
internet site, 1-7
IOB
  implementing logic, 4-65
  moving registers, 4-78, 4-79
  unbonded, 4-80

J

JTAG 1149.1, 4-61

L

labeling in code, 2-6
latch
  combinatorial feedback loop, 2-28
  comparing speed and area, 2-31
  converting to register, 2-28
  D flip-flop, 2-30
  D latch implemented with gates, 2-28
  hdlin_check_no_latch command, 2-29
  implementing in HDL, 2-26
  inference, 2-50
  latch count, 2-29
  RAM primitives, 2-30
libraries, 5-10
LogiBLOX
  bi-directional I/O, 4-75
  implementing memory, 4-57
  instantiating modules, 4-46
  libraries, 5-10, 5-18
LogiCORE
  library, 5-20

M

mapping your design
  using design manager, 3-17
  using the command line, 3-19
maxskew, 3-12
memory
  implementing in HDL, 4-52
  requirements, 1-6
Modelsim simulator, 5-51
multi-pass place and route option, 3-24
multiplexer
  comparing gates and tristate buffer, 4-87

implementing with gates, 4-85
implementing with tristate buffer, 4-83
resource sharing, 2-32

N

named association, 2-9
naming conventions, 2-5, 2-6
nested if statement, 2-51
no_gsr design, 4-12
NODELAY attribute, 4-66

O

offset constraint, 3-11
OMUX, 4-68
one-hot encoded state machine, 4-36
oscillators, 5-47
output multiplexer, 4-68

P

pad location, 4-78
parallel logic, 2-58
period constraint, 3-10
pipelining, 4-88
placing and routing your design, 3-22
port declarations, 2-13
positional association, 2-9
post-route full timing simulation, 5-10
post-synthesis simulation, 5-5
-pr option, 4-79
preset pin, 2-41, 4-9, 4-11
priority-encoded logic, 2-51, 2-58
PROM file, 3-28
pull-downs, 4-67
pull-ups, 4-67

R

RAMs, 4-55
re-entrant routing option, 3-24
register
clear pin, 2-42
converting latch to register, 2-28
D register, 2-28
if statement, 2-30
implementing in HDL, 2-26
inference, 2-43
moving into IOB, 4-78
preset pin, 2-42
report_fpga command, 3-14
report_timing command, 3-15
reset on configuration, 5-27
reset on configuration buffer, 5-32
resource sharing
CLB count, 2-38
definition, 2-31
delay, 2-38
design examples, 2-32
disabling, 2-36
hdl_resource_allocation command, 2-36
ROC, 5-27
ROCBUF, 5-32
ROMs, 4-52
RTL simulation, 4-11, 5-1, 5-4
definition, 1-2

S

schematic entry design hints, 2-17
set don't touch attribute, 4-62
set_attribute command, 4-17, 4-41
set_resource_allocation command, 2-41
set_resource_implementation command, 2-41
signal skew, 3-12
signals, 2-15
SimPrim libraries, 5-10
simulating your design, 5-1
simulation
creating a test bench, 5-7
functional, 5-4
global signals, 5-21
industry standards, 5-11
library source files, 5-13
post-map, 5-6

post-NGDBuild, 5-6
post-synthesis, 5-5
timing, 5-10
simulation diagram, 5-2
slew rate, 4-67
software requirements, 1-5
Spartan
IOB, 4-65
STARTBUF, 4-9, 4-18, 5-35
STARTUP block, 4-9, 4-18
startup state, 4-10
state machine, 4-26
binary encoded, 4-27
bubble diagram, 4-28
encoding style summary, 4-39
enumerated type encoded, 4-33
design example, 4-33, 4-36
enumeration type, 4-39
initializing, 4-40
limiting fan-in, A-3
one-hot encoded, 4-36, A-1
one-state vs. binary encoded, A-8
resetting state bits, A-11
seven states, A-3
std_logic data type, 2-13
SXNF file, 3-7
Synopsys
creating compile run script, 3-7
DesignWare, 2-32, 2-38
setup file, 3-6
startup file, 1-15
synthesis
comparing to simulation, 2-1
synthesis tools, 1-4

T

tactical software
installing, 1-5
TCK pin, 4-61
TDI pin, 4-61
TDO pin, 4-61
technical support, 1-15
test bench, 5-7
TIG, 3-11
TIMEGRPs, 3-9
timing
constraint precedence, 3-13
constraint priority, 3-12
constraints, 1-5, 3-8, 3-10
requirements, 1-5
simulation, 3-26, 5-2, 5-10
simulation netlist
Command Line, 3-27
Design Manager, 3-27
TMS pin, 4-61
TNMs, 3-8
TOC, 5-39
TOCBUF, 5-44
TPSYNC keyword, 3-9
tristate buffer
comparing to gates, 4-87
implementing multiplexer, 4-83
tristate enable, 5-38
tristate on configuration, 5-39
tristate on configuration buffer, 5-44
turns engine option, 3-24

U

UCF, 4-78
unbonded IOBs, 4-80
ungroup_all command, 2-38
UniSim libraries, 5-10, 5-15
use_gsr design, 4-18
user constraints file, 4-78

V

variables, 2-15
Verilog
capitalization style, 2-4
constants for opcode, 2-9
definition, 1-3
global set/reset, 5-51

GTS, 5-67
libraries, 5-50
parameters for constants, 2-8
register inference, 2-44
VHDL
after xx ns statement, 2-2
also see HDL
arithmetic functions, 2-3
capitalization style, 2-5
case statement, 2-3
coding styles, 2-4
constants, 2-8
constants for opcode, 2-8
definition, 1-3
if-else statement, 2-3
initialization statement, 2-4
naming identifiers, 2-6
naming packages, 2-6
naming types, 2-6
register inference, 2-43
simulation, 2-1
std_logic data type, 2-13
synthesis, 2-1
variables for constants, 2-8
wait for xx ns statement, 2-2
Xilinx naming conventions, 2-5
XSI libraries, 5-13
VHSIC Hardware Description Language *see* VHDL

W

wait for xx ns statement, 2-2

X

XC4000
CLB, 2-42
IOB, 4-65
XC5200
IOB, 4-69
XDW libraries, 5-10
Xilinx Development System
software requirements, 1-5
Xilinx internet site, 1-9
Xilinx Synopsys Interface *see* XSI
xor_sig design, 2-15
xor_var design, 2-17
XSI
installing VHDL libraries, 5-13
post-implementation simulation, 5-9
pre-implementation simulation, 5-8
software requirements, 1-5