Preface

About This Manual

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

The design examples in this manual were created with the VHSIC Hardware Description Language (VHDL); compiled with the Synopsys FPGA Compiler; and targeted for XC4000 devices. However, most of the design methodologies also apply to Verilog, other synthesis tools, and other Xilinx FPGAs. Xilinx equally endorses both Verilog and VHDL. VHDL may be more difficult to learn than Verilog and usually requires more explanation. You can download the VHDL design examples and the Verilog versions of many of the designs either from the Xilinx Internet Site or the Xilinx Technical Bulletin Board.

Before using this manual, you should be familiar with the operations that are common to all Xilinx’s software tools: how to bring up the system, select a tool for use, specify operations, and manage design data. These topics are covered in the Development System Reference Guide.

Refer to the Synopsys (XSI)—FPGAs Interface/Tutorial Guide and the Synopsys (XSI)—EPLDs Interface/Tutorial Guide for information on implementing XC3000, XC3100, and XC7000 devices and for more information on implementing XC4000 device features.

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, design methodology notes, and SOLV-IT for additional information.

How to Use This Manual

Chapter 6, “Understanding High-Density Design Flow”, describes the steps in the high-density design flow and includes information from the five previous chapters in the manual. You may want to review Chapter 6 first to get an overview of the design flow and then refer to the appropriate chapters for detailed information on a specific step in the flow.

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,” discusses HDL coding hints and design examples to help you develop an efficient coding style.
- Chapter 3, “HDL Coding for FPGAs,” provides design examples to help you incorporate FPGA system features into your HDL designs.
- Chapter 4, “Floorplanning Your Design,” describes basic operation of the Xilinx Floorplanner and provides HDL design examples that facilitate floorplanning.
- Chapter 5, “Building Design Hierarchy,” describes how to partition your designs to improve synthesis results and reduce routing congestion.
- Chapter 6, “Understanding High-Density Design Flow,” discusses the design flow for high-density HDL designs.
• Appendix A, “Accelerate FPGA Macros with One-Hot Approach,” reprints an article describing one-hot encoding in detail.

• Appendix B, “Top Design Scripts,” includes the three script files used to compile the Top design described in the “Building Design Hierarchy” chapter of this manual.

• Appendix C, “Tactical Software and Design Examples,” lists the tactical software and design examples that are described in this manual.
Conventions

The following conventions are used in this manual’s syntactical statements.

Courier font regular 
System messages or program files appear in regular Courier font.

Courier font bold
Literal commands that you must enter in syntax statements are in bold Courier font.

italic font
Variables that you replace in syntax statements are in italic font.

[ ]
Square brackets denote optional items or parameters.

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

.
A vertical ellipsis indicates material that has been omitted.

...
A horizontal ellipsis indicates that the preceding can be repeated one or more times.

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

\[
\]
This symbol denotes a carriage return.
Selecting VHDL Coding Styles ..................................................... 2-3
Selecting a Capitalization Style .................................................. 2-4
Using Xilinx Naming Conventions ............................................. 2-4
Naming Identifiers, Types, and Packages ................................. 2-5
Using Labels ........................................................................... 2-5
Using Variables for Constants ................................................ 2-6
Using Named and Positional Association ............................... 2-6
Managing Your Design ........................................................... 2-7
Creating Readable Code ........................................................ 2-7
Indenting Your Code .......................................................... 2-7
Using Empty Lines ............................................................. 2-7
Using Spaces ..................................................................... 2-8
Breaking Long Lines of Code ................................................ 2-8
Adding Comments .................................................................. 2-8
Using Std_logic Data Type ..................................................... 2-8
Declaring Ports ...................................................................... 2-9
Minimizing the Use of Ports Declared as Buffers .................... 2-9
Comparing Signals and Variables............................................ 2-10
Using Schematic Design Hints with HDL Designs .............. 2-12
Barrel Shifter Design ............................................................... 2-12
Implementing Latches and Registers ...................................... 2-16
Resource Sharing .................................................................. 2-20
Gate Reduction ....................................................................... 2-25
Preset Pin or Clear Pin ........................................................... 2-27
Using Clock Enable Pin ........................................................ 2-30
Using If Statements ................................................................. 2-31
Using Case Statements .......................................................... 2-32
Using Nested_If Statements ................................................... 2-33
Comparing If Statement and Case Statement ......................... 2-37

Chapter 3 HDL Coding for FPGAs

Using Global Low-skew Clock Buffers ................................ 3-2
Inserting Clock Buffers ......................................................... 3-4
Instantiating Internal Global Clock Buffers ......................... 3-4
Using Dedicated Global Set/Reset Resource ......................... 3-4
Startup State ......................................................................... 3-5
Preset vs. Clear ...................................................................... 3-5
Increasing Performance with the GSR Net ....................... 3-6
Design Example without Dedicated GSR Resource .... 3-6
Design Example with Dedicated GSR Resource .............. 3-8
Contents

Design Example with Dedicated GSR Resource and Additional Preset Signal ........................................... 3-11
Encoding State Machines ................................................................. 3-13
Using Binary Encoding ................................................................. 3-14
Using Enumerated Type Encoding ................................................. 3-16
Using One-Hot Encoding .............................................................. 3-17
Summary of Encoding Styles....................................................... 3-18
Comparing Synthesis Results for Encoding Styles ....................... 3-19
Initializing the State Machine ...................................................... 3-20
Using Dedicated I/O Decoders ....................................................... 3-21
Instantiating X-BLOX Modules .................................................... 3-25
Using X-BLOXGen...................................................................... 3-26
Syntax ...................................................................................... 3-27
Options ..................................................................................... 3-28
Output Files .............................................................................. 3-28
X-BLOXGen Example ............................................................... 3-28
Using RPMs .............................................................................. 3-32
Instantiating an RPM ................................................................. 3-34
Implementing Memory ................................................................. 3-36
Implementing XC4000 RAMs ....................................................... 3-36
Implementing XC4000 ROMs ....................................................... 3-36
Using MemGen........................................................................ 3-38
Implementing Boundary Scan (JTAG 1149.1) ............................ 3-40
Instantiating the Boundary Scan Symbol .................................... 3-40
Implementing Logic with IOBs ..................................................... 3-42
XC4000/A/D IOBs ....................................................................... 3-43
Inputs ...................................................................................... 3-43
Outputs .................................................................................... 3-43
XC4000/D Slew Rate ................................................................. 3-43
XC4000A Slew Rate ................................................................. 3-44
XC4000H IOBs .......................................................................... 3-44
Inputs ...................................................................................... 3-44
Outputs .................................................................................... 3-44
XC4000H Slew Rate ................................................................. 3-45
Instantiating Bidirectional I/O ..................................................... 3-45
Moving Registers into the IOB ..................................................... 3-46
Using Unbonded IOBs (XC4000/A/D Only) ............................... 3-48
Implementing Multiplexers with Tristate Buffers ......................... 3-50
Setting Timing Constraints .......................................................... 3-53
Using the Synthesis Tool ............................................................ 3-53
Using PPR Command Line Options ........................................... 3-54
Chapter 4 Floorplanning Your Design

Using the Floorplanner ......................................................... 4-2
Creating a MAP File................................................................. 4-2
Using XMake ..................................................................... 4-2
Using PPR .............................................................. 4-2
Using Prep for Floorplanner Command................................. 4-2
Overview of Floorplanner Windows ........................................ 4-3
Task Window ..................................................................... 4-3
Design Window .................................................................. 4-3
Floorplan Window .............................................................. 4-4
Deciding What Elements to Floorplan ..................................... 4-5
Running the Floorplanner and Opening a File ........................ 4-6
Using the Command Line.................................................. 4-6
Using the Floorplanner Task Window ............................... 4-6
Setting Boundaries in the Floorplan Window ........................ 4-8
Floorplanning RPMs, RAMs, and ROMs......................... 4-9
RPM and RAM/ROM Example............................................. 4-10
Floorplanning Tristate Buffers ........................................ 4-13
BUFT Example ................................................................ 4-14
Floorplanning BUFT Example ........................................ 4-16
Comparing Hierarchical and Flat Designs .......................... 4-20
Method 1: Compiling Flat without X-BLOX .................... 4-24
Method 2: Compiling Flat with X-BLOX ............................ 4-26
Method 3: Compiling with Hierarchy and without X-BLOX .... 4-26
Method 4: Compiling with Hierarchy and X-BLOX .............. 4-27
Floorplanning to Reduce Routing Congestion .................. 4-28
Positioning and Aligning Buses ........................................ 4-29
Aligning Structures Along Buses ..................................... 4-29
Floorplanning RAMs to Reduce Routing Congestion .......... 4-34
Chapter 5  Building Design Hierarchy

Using the Synthesis Tool .............................................................  5-2
Modifying Design Hierarchy for PPR ...........................................  5-3
Top Design Example....................................................................  5-4
  Compiling Top Design as One Flat Module ............................  5-7
  Compiling Top Design Using Original Hierarchy .................  5-8
  Floorplanning RPMs ..........................................................  5-8
  Meeting Speed Requirements ...........................................  5-10
  Compiling Top Design After Modifying the Hierarchy ..........  5-11
  Evaluating A New Hierarchy ...........................................  5-15
  Defining and Compiling the New Hierarchy ..........................  5-15
  Setting Boundaries and Floorplanning the Modules ..........  5-19
  Floorplanning Structured Cells ..........................................  5-21
  Placing and Routing the Top Design .....................................  5-22
  Adding Probe Points to Debug a Design .............................  5-23
  Comparing Top Design Methodologies ..................................  5-24
    Flat Design ....................................................................  5-24
    Original Design Hierarchy..................................................  5-24
    Modified Hierarchy ..........................................................  5-24

Chapter 6  Understanding High-Density Design Flow

Step 1: Estimating Your Design Size ...........................................  6-4
  Determining Device Utilization ...........................................  6-5
Step 2: Evaluating Your Design for Coding Style and System
  Features ........................................................................  6-7
Step 3: Modifying Your Design Hierarchy .................................  6-8
  Estimating Area Utilization .............................................  6-9
  Creating a New Hierarchy ..................................................  6-11
Step 4: Synthesizing and Optimizing Your Design ......................  6-11
Step 5: Translating Your Design and Adding Group TimeSpecs.  6-12
  Translating Your Design ..................................................  6-12
  Adding Timing Specifications ...........................................  6-13
    Using the Synthesis Tool .............................................  6-13
    Using PPR Command Line Options .................................  6-13
    Using A Constraints File .............................................  6-13
Step 6: Building Your Design Hierarchy ......................................  6-15
Step 7: Floorplanning Your Design ...........................................  6-15
  Creating a MAP File .......................................................  6-15
  Using XMake ....................................................................  6-16
  Using PPR .......................................................................  6-16
Using Prep for Floorplanner Option ................................  6-16
Floorplanning Design Components .................................  6-16
Writing a Constraints File .............................................  6-17
Step 8: Placing and Routing Your Design .........................  6-17
Using PPR Options .......................................................  6-18
Determining If PPR Can Route Your Design .....................  6-19
Step 9: Evaluating the Results ........................................  6-19
Evaluating Module Placement with the Floorplanner ...........  6-20
Modifying Design Placement ........................................  6-25
Using Guided Design .....................................................  6-26
Using Iterative Guided Design ........................................  6-27
Using Incremental Guided Design ....................................  6-27
Using XDE .................................................................  6-27
Effectively Using Guided Design .....................................  6-28
Understanding Guided Design for XC4000 Designs ..........  6-28
Adding a New Module to Your Design .............................  6-28
Making a Design Change to a Module ...............................  6-29

Appendix A Accelerate FPGA Macros with One-Hot Approach
Appendix B Top Design Scripts
  VHDL Script Files ....................................................... B-1
  Verilog Script Files .................................................... B-6
Appendix C Tactical Software and Design Examples
  Tactical Software ........................................................ C-1
  Design Examples ........................................................ C-2
Chapter 1

Getting Started

Hardware Description Languages (HDLs) are used to describe the behavior and structure of system and circuit designs. This chapter provides a general overview of designing FPGAs with HDLs. It also includes design hints for the novice HDL user and for the experienced user who is designing FPGAs for the first time. 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. Understanding FPGA architecture allows you to create HDL code that effectively uses FPGA system features.

Understanding HDL Design Flow for FPGAs

Application Specific Integrated Circuit (ASIC) designs or sections of these designs that are targeted for FPGAs are often created with HDLs. However, the design flow for processing ASIC HDL code is slightly different from the flow used to process HDL code written specifically for FPGAs.

Figure 1-1 shows the design flow for an FPGA design. This design flow includes the following steps.

1. Creating your FPGA design with an HDL.
2. Performing a Register Transfer Level (RTL) simulation of your design.
3. Synthesizing your design.
5. Performing a functional simulation of your design.
6. Floorplanning your design. This step is optional.
7. Placing and routing (implementing) your design.
8. Performing a timing simulation of your design.

Figure 1-1 HDL Flow Diagram for a New Design

The design flow for ASICs differs depending on the quality of the existing code. You must analyze the ASIC design to determine if the code meets speed and area requirements for FPGAs. Additionally, you should structure the design hierarchy for FPGA implementation.

Entering Your Design

When coding in HDL, you should create efficient code that utilizes FPGA system features and is structured into hierarchical blocks. These topics are described in detail in this manual.
Verifying Your Design

You can behaviorally simulate your HDL designs to test system and device functionality before synthesis. After simulation, your design is synthesized and optimized for the target device. The hierarchical HDL code is then written as XNF files. After placement and routing, the design is simulated with the actual gate and wire delays.

Xilinx recommends that you perform an RTL or functional simulation of your design before floorplanning the cells (CLBs, IOBs, BUFTs) into the FPGA. If you find functional errors during a simulation performed after floorplanning, you must correct your code, resynthesize your design, and repeat the floorplanning process. The Xilinx Floorplanner builds a constraints file that includes the cell names in your design. If the cell names change, as they might if you resynthesize your design, the names in the constraints file are no longer correct.

Floorplanning Your Design

Floorplanning is an optional step in the design flow. You can improve device density and increase the speed of critical paths by floorplanning parts or all of your design with the Xilinx Floorplanner. You can generate a constraints file that is read by PPR. Refer to the “Floorplanning Your Design” chapter in this manual for more information on floorplanning.

Placing and Routing Your Design

After floorplanning, run PPR to place and route your design. PPR reads the constraints file generated by the Floorplanner and places logic that is not floorplanned. After your design is placed and routed, perform a timing simulation. You can back-annotate timing information to the Synopsys timing analysis tool.

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 gate level allows you to make any necessary changes early in the design process.

- **Automatic Conversion of HDL Code to Gates**
  
  You can automatically convert your hardware description to a design implemented with gates. This step decreases design time by eliminating the traditional gate-level bottleneck. This automatic conversion to gates also reduces the number of errors that may be introduced during a manual translation of a hardware description to a schematic design. Additionally, you can apply the techniques used by the synthesis tool during the optimization of your design to the original HDL code, resulting in greater efficiency.

- **Type Checking**
  
  HDLs provide type checking. For example, you cannot connect a 3- or 5-bit wide signal to a component that requires a 4-bit wide signal type. Additionally, if the range of a bus is 1 to 15, you cannot assign the bus a value of 0. Using incorrect types is a major source of errors in HDL descriptions. Type checking eliminates these errors in the description before a design is generated.

- **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 language syntax, the synthesis tool, the architecture of the target device, and the implementation tools. This section gives you some design hints to help you create FPGAs with HDLs.

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. In addition, the various synthesis tools use different subsets of the VHDL language. The examples provided in this manual are written in VHDL. 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 can greatly increase the number of gates.

Design hierarchy is important in the implementation of an FPGA and also during incremental or interactive changes. You should partition large designs (greater than 5,000 gates) into modules. The size and content of the modules influence synthesis results and design implementation. How to create effective design hierarchy is described in this manual.

Specifying Speed Requirements

To meet timing requirements, you should understand how to set timing constraints in both the synthesis and placement/routing tools. You should also know how to manually place critical paths and structured modules with the Xilinx Floorplanner. See the Floorplanner Reference/User Guide for more information.

Installing Design Examples and Tactical Software

The information in this section supplements the information in the Xilinx Synopsys Interface Version 3.3 Release Document. Read and follow the instructions in the Release Documents for Xilinx Synopsys Interface V3.3 or V5.1.

Three tactical software programs are required for the HDL examples in this manual. The three programs are X-BLOXGen, MakeTNM, and AddTNM. These programs are not included in the Xilinx Synopsys Interface or the XACTstep Development System.

AddTNM and MakeTNM were created with Perl 4.0. To run these programs, you must have either Perl 4.0 or 5.0.
This manual includes numerous HDL design examples. These designs were created with VHDL, however, Xilinx equally endorses both Verilog and VHDL. VHDL may be more difficult to learn than Verilog and usually requires more explanation. You can obtain Verilog versions of many of the design examples either from the Xilinx Internet Site or the Xilinx Technical Bulletin Board, as described below.

Note: See “Appendix C” for a complete listing of tactical software and design examples.

Software Requirements

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

Table 1-1 Software Versions

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Synopsys Interface (XSI)</td>
<td>3.2.0 or later</td>
</tr>
<tr>
<td>XACTstep</td>
<td>5.1.0 or later</td>
</tr>
<tr>
<td>XACTstep Foundry*</td>
<td>7.0 or later</td>
</tr>
<tr>
<td>Synopsys FPGA Compiler</td>
<td>3.2 or later</td>
</tr>
<tr>
<td>Xilinx Floorplanner</td>
<td>Contact Xilinx sales representative for copy of Floorplanner.</td>
</tr>
<tr>
<td>XC4025 die files</td>
<td>Contact Xilinx sales representative.</td>
</tr>
</tbody>
</table>

*XACTstep Foundry v7 does not support the Xilinx Floorplanner.

Note: The design examples in this manual were compiled with Synopsys V3.3a and XACTstep V5.2.0 (pre-release), however, all programs, scripts, and design examples are compatible with the versions in Table 1-1.
**SPARC and HP-PA Requirements**

The system requirements for the SPARC and HP-PA are identical to those described in Xilinx Synopsys Interface Release Document V3.3 or V5.1. Refer to this release documentation for more information.

**Disk Space Requirements**

Before you install the programs and files, verify that your system meets the requirements listed in the tables below for the various options. The disk space requirements listed are an approximation and may not exactly match the actual numbers.

**Xilinx Internet Site**

To download the programs and files from the Xilinx Internet Site, you need to meet the disk requirements listed in Table 1-2.

**Table 1-2 Internet Files**

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
<th>Compressed File</th>
<th>Directory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSI_files</td>
<td>• Tactical Code</td>
<td>83 K</td>
<td>344 K</td>
</tr>
<tr>
<td></td>
<td>• XNF files for RPMs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Default Synopsys setup file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XSI_vhdl</td>
<td>VHDL Examples <em>with</em> SIM, SYN and MRA files in Work directory</td>
<td>5.1 MB</td>
<td>13 MB</td>
</tr>
<tr>
<td>XSI_vhdl_no_work</td>
<td>VHDL Examples <em>without</em> SIM, SYN and MRA files in Work directory</td>
<td>3.3 MB</td>
<td>10.2 MB</td>
</tr>
<tr>
<td>XSI_verilog</td>
<td>Verilog Examples</td>
<td>3 MB</td>
<td>9.2 MB</td>
</tr>
</tbody>
</table>

The XSI_files directory contains tactical software, XNF files for RPMs, and a default XC4000 FPGA Compiler setup file.
The XSI_vhdl_no_work directory is smaller than the XSI_vhdl directory because it does not contain the contents of the Work directory. The Work directory contains the analyzed files, SYN, MRA, and SIM for each VHDL design file. You can create these files by analyzing the VHDL design files or by running the design script files. Copy one of these directories only; it is not necessary to copy both. If you want to decrease the download time, copy the XSI_VHDL_no_work directory.

**Xilinx Technical Bulletin Board**

To download the programs and files from the Xilinx Technical Bulletin Board (XTBB), you need to meet the disk space requirements listed in Table 1-3. Due to file size restrictions on the XTBB, the VHDL and Verilog directories listed in Table 1-3 do not contain the entire set of files that are available via the Internet. However, you can generate the complete set of files by running the design scripts and invoking the Xilinx tools.

**Table 1-3  XTBB Files**

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
<th>Compressed File</th>
<th>Directory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>tactical.uu</td>
<td>• Tactical Code</td>
<td>115 K</td>
<td>344 K</td>
</tr>
<tr>
<td></td>
<td>• XNF files for RPMs</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Default Synopsys setup file</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vhdl_ex.uu</td>
<td>design.vhd, design.log, design.rpt, design.prp, design.lca, design.timing, design.fpga, design.sxnf, design.db, design.map, design.script</td>
<td>2.6 MB</td>
<td>5.1 MB</td>
</tr>
<tr>
<td>ver_ex.uu</td>
<td>design.v, design.log, design.rpt, design.prp, design.lca, design.timing, design.fpga, design.sxnf, design.db, design.map, design.script</td>
<td>2.5 MB</td>
<td>4.7 MB</td>
</tr>
</tbody>
</table>

**Retrieving Tactical Software and Design Examples**

You can retrieve the tactical software and the HDL design examples from the Xilinx Internet Site or the XTBB. 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 XSI DS-401 and the Synopsys tools. Do not install the files into the XSI DS-401 or XACTstep DS-502 directories since the files may be overwritten when the next version of XSI or XACTstep software is installed.

From Xilinx Internet FTP Site

You can retrieve the programs and files from the Xilinx Internet FTP (File Transfer Protocol) Site. To access the Xilinx Internet FTP Site, you must have FTP available on your machine. For UNIX users, FTP is a UNIX utility. You can obtain the PC version of the FTP utility through third-party vendors.

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 Internet FTP Site, use the following procedure:

1. Go to the directory on your local machine where you want to download the files:

   \texttt{cd directory}

2. Invoke the FTP utility:

   UNIX users, type: \texttt{ftp}
   
   PC users: contact your system administrator for assistance

3. Connect to the Xilinx Internet machine, www.xilinx.com:

   \texttt{ftp> open xilinx.www.com}

4. Log into a guest account. This account gives you download privileges.

   \texttt{Name (machine:user-name) : ftp}
   
   Guest login ok, send your complete e-mail address as the password.
   
   Password: \texttt{your_email_address}
5. Go to the pub/XSI_HDL directory:
   
   ftp> cd pub/XSI_HDL

6. Retrieve the appropriate design files as follows:
   
   ftp> get design_files.tar.Z

7. Extract the files as described in the “Extracting the Files” section below.

From Xilinx Technical Bulletin Board

The Xilinx Technical Bulletin Board (XTBB) is a 24-hour electronic bulletin board available to all registered XACTstep customers. XTBB includes application notes, utility programs, bug fixes, and updated data files such as package and speed files. If you have full XTBB privileges, you can read and retrieve files on the bulletin board, including the design examples in this manual. You can also upload files and leave messages for Xilinx personnel or other XTBB users. Refer to the 1994 version of *The Xilinx Programmable Logic Data Book* for a complete description of the XTBB, including how to locate and download files.

To retrieve the programs and files from the XTBB:

1. Go to the directory on your local machine where you want to download the files:
   
   cd directory

2. Access the XTBB.

3. Locate the files in the application area of the XTBB. The directory names are listed in Table 1-3.

4. Retrieve the uuencoded files.

5. Extract the files as described below.

Extracting the Files

You must install the retrieved files on the same system as XSI DS-401 and the Synopsys tools. However, do not install the files into the XSI DS-401 or XACTstep DS-502 directories since the files may be overwritten when the next version of XSI or XACTstep software is installed.
To extract the files, use the following procedure.

**Note:** The first step only applies to files retrieved from the XTBB.

1. Undecode the files:
   ```
   uudecode design.uu
   ```
2. Uncompress the files:
   ```
   uncompress design.tar.Z
   ```
3. Extract the files:
   ```
   tar xvf design.tar
   ```
4. Copy the tactical programs, AddTNM, MakeTNM, and X-BLOXGen, into a directory in your search path or into your working directory.

## Directory Tree Structure

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

```
XSI_files/
  xbloxgen
  addtnm
  maketnm
  fc4k.synopsys_db.setup
rpm_xnf/
  acc16/
  acc4/
  acc8/
  add16/
  add4/
  add8/
  adsu16/
  adsu4/
  adsu8/
  cc16ce/
  cc16cle/
  cc16cled/
  cc16re/
  cc8ce/
  cc8cle/
  cc8cled/
  cc8re/
  compc16/
  compmc8/
```
XSI_verilog/
   alarm/
   align_str/
   barrel/
   bidi_reg/
   bnd_scan/
   bufts/
   clock_enable/
   clr_pre/
   d_latch/
   d_register/
   ff_example/
   gate_clock/
   gate_reduce/
   gsr/
   io_decoder/
   mux_vs_3state/
   res_sharing/
   rom16x4/
   rom_memgen/
   rpm_example/
   rpm_ram/
   state_machine/
   top_hier/
   unbonded_io/
   xbloxgen_ex/

XSI_vhdl/ or XSI_vhdl_no_wk/
   alarm/
   align_str/
   barrel/
   bidi_reg/
   bnd_scan/
   bufts/
   case_vs_if/
   clock_enable/
   clr_pre/
   d_latch/
   d_register/
   ff_example/
   gate_clock/
   gate_reduce/
   gsr/
   io_decoder/
   mux_vs_3state/
   nested_if/
   res_sharing/
   rom16x4/
   rom_memgen/
   rpm_example/
Synopsys Startup File and Library Setup

Follow the procedures in the “Getting Started” chapter of the Synopsys (XSI)—FPGAs Interface/Tutorial Guide for instructions on setting up the Synopsys start-up file for XC4000 designs using the Synopsys FPGA Compiler.

Technical Support

This manual and associated files come with free technical and product information telephone support (toll-free in the U.S. and Canada). You can also fax or email your questions to Xilinx.

- United States and Canada
  - Technical Support Hotline 1-800-255-7778
  - Technical Support FAX (24 hours/7 days) 1-408-879-4442
  - Technical Support Bulletin Board (24 hours/7 days) 1-408-559-9327
  - Internet E-mail Address (24 hours/7 days) hotline@xilinx.com

- International
  - Technical Support Hotline 1-408-879-5199
  - Technical Support FAX (24 hours/7 days) 1-408-879-4442
  - Technical Support Bulletin Board (24 hours/7 days) 1-408-559-9327
  - Internet E-mail Address (24 hours/7 days) hotline@xilinx.com

Important Issues

This section includes important issues that are not covered in the remaining chapters of this manual.
Instantiating XNF Files in Verilog Designs

To instantiate a module that is not in one of the target libraries, such as an XNF file created using MemGen or X-BLOXGen, you must create an additional module-endmodule statement for the XNF file in your Verilog code. The XNF file for the module must exist in the working directory (or in the search path for XNFMerge).

For example, use the following procedure to instantiate the acc4.xnf file in the rpm_example.v design. Acc4.xnf was created with X-BLOXGen.

1. Create an additional module-endmodule statement for acc4.xnf in rpm_example.v as follows:

   ```verilog
   module rpm_example(B_IN3, B_IN2, B_IN1, B_IN0, D_IN3, D_IN2, D_IN1, D_IN0, CI_IN, L_IN, ADD_IN, CE_OUT, CLK_IN, Q_OUT3, Q_OUT2, Q_OUT1, Q_OUT0, CO_OUT, OFL_OUT);
   input B_IN3, B_IN2, B_IN1, B_IN0, D_IN3, D_IN2, D_IN1, D_IN0, CI_IN, L_IN, ADD_IN, CE_OUT, CLK_IN;
   output Q_OUT3, Q_OUT2, Q_OUT1, Q_OUT0, CO_OUT, OFL_OUT;
   acc4 U1 (.B3(B_IN3), .B2(B_IN2), .B1(B_IN1), .B0(B_IN0), .D3(D_IN3), .D2(D_IN2), .D1(D_IN1), .D0(D_IN0), .CI(CI_IN), .L(L_IN), .ADD(ADD_IN), .CE(CE_OUT), .C(CLK_IN), .Q3(Q_OUT3), .Q2(Q_OUT2), .Q1(Q_OUT1), .Q0(Q_OUT0), .CO(CO_OUT), .OFL(OFL_OUT));
   endmodule
   
   module acc4 (B3, B2, B1, B0, D3, D2, D1, D0, CI, L, ADD, CE, C, Q3, Q2, Q1, Q0, CO, OFL);
   input B3, B2, B1, B0, D3, D2, D1, D0, CI, L, ADD, CE, C;
   output Q3, Q2, Q1, Q0, CO, OFL;
   endmodule
   ```

2. In your Synopsys script file, immediately before you write the SXNF file, enter this command:

   ```bash
   remove_design acc4
   ```

3. Run XMake on the top level file:

   ```bash
   xmake rpm_example
   ```
Block Names are Not Written by Default in Synopsys FPGA Compiler V3.3b

In Synopsys FPGA Compiler V3.3b, block names for CLBs mapped with registers are not written to the SXNF file by default. If your script file contains the following statement, you can remove it.

```
set_attribute find(design, "*") "xnfout_use_blknames" \\
    -type boolean FALSE
```

The design examples in this manual do not contain this statement in the example script file. However, the example scripts in the Xilinx Synopsys Interface V3.2 and V3.3 include this statement. If you are using Synopsys V3.2 or earlier, you should set this attribute to FALSE and include the statement in your script file. If you are using Synopsys V3.3b or later, you can remove this statement because the attribute version is set to FALSE by default.

Creating MAP Files

Do not use the outfile= option when creating a MAP file.
Chapter 2

HDL Coding Hints

HDLs contain many complex constructs that can be difficult to understand at first. Additionally, the methods and examples included in HDL manuals do not always apply to designing FPGAs. If you currently use HDLs to design ASICs, your established coding style may increase the number of gates in FPGA designs. ASICs have more gates and routing resources than FPGAs, therefore, a design that fits an ASIC device may be unroutable in an FPGA.

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, but they should help you develop an efficient coding style.

The following topics are included in this chapter:

- Comparing Synthesis and Simulation Results
- Selecting VHDL Coding Styles
- Using Schematic Design Hints with HDL Designs
Comparing Synthesis and Simulation Results

VHDL is a hardware description and simulation language and was not originally intended as an 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 the VHDL language. VHDL 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.

Omit the ...After XX ns Statement

Do not use the ...After XX ns statement in your code. An example of this statement is:

\[(Q <= 0 \text{ after } XX \text{ ns})\]

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 either If-Else statements or Case statements to create state machines. Synthesis tools that support both types of 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 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 influences design performance. For example, the following two statements are not equivalent:

\[
\text{ADD} \leq A_1 + A_2 + A_3 + A_4;
\]

\[
\text{ADD} \leq (A_1 + A_2) + (A_3 + A_4);
\]

The first statement cascades three adders in series. The second statement creates two adders in parallel: \(A_1 + A_2\) and \(A_3 + A_4\). 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).

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 such as the following:

\[
\text{variable SUM:INTEGER:=0;}
\]

Selecting VHDL Coding Styles

Because VHDL designs are often created by design teams, Xilinx recommends that you agree on a coding style 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 VHDL 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. In Xilinx FPGA designs, entity names must be in lowercase letters because the XACTstep Development System does not recognize names in uppercase letters. Based on this restriction, you may want to specify that VHDL reserved words are in lowercase letters and other keywords are in uppercase letters.

The following capitalization style is used for the examples in this manual.

- Use lowercase letters for entity names and VHDL reserved words
- Use uppercase letters for the following:
  - Keywords that are not entity names and VHDL reserved words
  - Variable, signal, instance, and module names
  - Labels
  - Libraries, packages, and data types
- For the names of standard or vendor packages, follow the style used by the vendor or use uppercase letters as shown for IEEE in the following example:

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

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 1024 characters long

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

• Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), clock buffers, tristate buffers (BUFTs), oscillators, package pin names, 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 P2 for component names

• Do not use pad names such as PAD1 for component names

**Note:** See the “Floorplanning Your Design” chapter for additional naming conventions that are important when using the Floorplanner.

### Naming Identifiers, Types, and Packages

You can use long (1024 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:

```vhdl
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 example:

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

You can use optional labels on flow control constructs to make the code structure more obvious, as shown in Figure 2-1. However, you should note that these labels are not translated to gate or register names in your implemented design.
Using Variables for Constants

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 code example in Figure 2-2, you should specify the seven as a constant and refer to it by name in your code.

```vhdl
-- D_REGISTER.VHD
-- Xilinx HDL Synthesis Design Guide for FPGA
-- June 1995
-- Changing Latch into a D-Register

entity d_register is
    port (CLK, DATA in BIT;
        Q out BIT);
end d_register;

architecture BEHAV of d_register is
begin
    REG: process [CLK, DATA]
    begin
        if (CLK'event and CLK='1') then
            Q <= DATA;
        end if;
    end process; --End REG
end BEHAV;
```

Figure 2-1 Labeling Flow Control Constructs

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. Do not combine positional and named association in the same statement as illustrated in the following line of code:

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

Figure 2-2 Defining Constants
The correct coding style is:

```vhdl
CLK_1: BUFGS port map
    (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 example in Figure 2-3 illustrates the indentation
style used in this manual.

```vhdl
architecture BEHAV of dlatch
begin
    LATCH_P: process (A,B)
    begin
        if CONDITION then
            STATEMENT;
        else CONDITION
            STATEMENT;
        end if;
    end process;--End LATCH_P
end BEHAV;
```

**Figure 2-3 Indenting Your Code**

**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. The following conventions are used for the examples in this manual.

- You can omit or use spaces between signal names as shown in the following lines of code:

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

- Use a space after colons as shown in the following lines of code:

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

Breaking Long Lines of Code

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

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

Adding Comments

Add comments to your code to improve readability, debugging, and maintenance.

Using Std_logic Data Type

**Note:** This section is an edited excerpt from a document in the Synopsys SOLV-IT! knowledge base. For more information on SOLV-IT, send e-mail to solvit@synopsys.com with the word help in the message body.

It is important to select the correct logic type for your VHDL designs. If you use the Synopsys compiler, the Std_logic (IEEE 1164) type is recommended for synthesis. This type is effective for hardware descriptions because it has nine different values. Additionally, the Std_logic type is 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.

**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. An example of using the Std_logic package for port declarations is shown in Figure 2-4.

```
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;
```

*Figure 2-4 Using Std_logic Package for Port Declaration*

**Minimizing the Use of Ports Declared as Buffers**

Declare a buffer when a signal is used internally and as an output port. In the example in Figure 2-5, signal C is used internally and it is used 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
        wait until CLK'event and CLK='1';
        C <= UNSIGNED(A) + UNSIGNED(B) + UNSIGNED(C);
    end process;
end BEHAVIORAL;
```

*Figure 2-5 Example of Buffer Output Signal*
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. To reduce the amount of coding in hierarchical designs, you may want to insert a dummy signal and declare port C as an output, as shown in Figure 2-6.

```vhdl
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);
begn
    C <= C_INT;
    process begin
      wait until CLK'event and CLK='1';
      C_INT <= UNSIGNED(A) + UNSIGNED(B) + UNSIGNED(C_INT);
    end process;
end BEHAVIORAL;
```

**Figure 2-6 Replacing Buffer Ports with a Dummy Signal**

### Comparing Signals and Variables

**Note:** This section is an edited excerpt from a document in the Synopsys SOLV-IT! knowledge base. For more information on SOLV-IT, send e-mail to solvit@synopsys.com with the word help in the message body.

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, they can mask glitches that may impact how your design functions. Because of this potential masking problem, Xilinx recommends that you use signals for hardware descriptions, however, variables allow quick simulation. Figure 2-7 shows a synthesized design that uses signals and Figure 2-8 shows a synthesized design that uses variables. These examples are shown implemented with gates in Figure 2-9 and Figure 2-10.
**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.

```vhdl
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.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 2-7 Using Signals**

```vhdl
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 2-8 Using Variables**
Using Schematic Design Hints with HDL Designs

This section describes how you can 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) in the 1994 version of The Xilinx Programmable Logic Data Book. 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 devices; however, it can also be used for XC4000 devices. This section includes two VHDL versions of the barrel shifter design.

Note: In the 16-bit barrel shifter example in the XAPP 026.001 application note, the select lines are registered to take advantage of the CLB DIN pin. You do not need to register the select lines in your HDL design because most synthesis tools do not use the DIN pin.

Figure 2-11 is a VHDL design of a 16-bit barrel shifter. The barrel shifter is 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 devices), a 20-input function requires at least five logic blocks. Therefore, the minimum design size is 80 (16 x 5) logic blocks.

```
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
    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(0);
            when "0011" =>
                B_P(12 downto 0) <= A_P(15 downto 3);
                B_P(15 downto 11) <= 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 others =>
                B_P <= (others => '0');
        end case;
    end process;
end RTL;
```

HDL Synthesis for FPGAs 2-13
The modified VHDL design in Figure 2-12 uses two levels of multiplexers and is twice as fast as the design in Figure 2-11. This design is 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.

Figure 2-11 16-bit Barrel Shifter

The modified VHDL design in Figure 2-12 uses two levels of multiplexers and is twice as fast as the design in Figure 2-11. This design is 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.
HDL Coding Hints

-- XAPP 26 (1994 Data Book p. 8-152)
-- 18-bit barrelshifter (shift right)
-- 2 November 94

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_LOGICVECTOR (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(14 downto 0) <= A,P(15 downto 1);
    when "10" => -- Shift by 2
      C(13 downto 0) <= A,P(15 downto 2);
    when "11" => -- Shift by 3
      C(12 downto 0) <= A,P(15 downto 3);
    when others =>
      C <= A,P;
    end case;
  end process; --End FIRST_LVL

  SECOND_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 0);
      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);
    end case;
  end process; --End SECOND_LVL

end RTL;
When these two designs are implemented in an XC4005-5 device using the Synopsys FPGA compiler, there is a 54% improvement in the gate count (91 occupied CLBs reduced to 36 occupied CLBs) in the design in Figure 2-12 as compared to the design in Figure 2-11. Additionally, there is a 25% improvement in speed from 54.2 ns (4 CLB block levels) to 46.4 ns (3 CLB block levels).

**Implementing Latches and Registers**

HDL compilers infer latches from incomplete specifications of conditional expressions. Latch primitives are not available in CLBs, however, the IOBs contain input latches. Latches described in RTL HDL are implemented with gates in the CLB function generators. For example, the D latch shown in Figure 2-13 is implemented with one function generator. The D latch implemented with gates is shown in Figure 2-14.
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.

**Figure 2-13 D Latch Inference**

---

```vhdl
-- D_LATCH.VHD
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- June 1995

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

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

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

**Figure 2-14 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.

**Figure 2-13 D Latch Inference**

---

```vhdl
-- D_LATCH.VHD
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- June 1995

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

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

architecture BEHAV of dLatch is
begin
  LATCH: process (GATE, DATA)
  begin
    if (GATE = '1') then
      Q <= DATA;
    end if;
  end process; -- end LATCH
end BEHAV;
```
When you run the Partition, Place, and Route (PPR) program on the example in Figure 2-12, the following warning message appears.

** Warning: [tspec: COMBINATIONAL_LOOPS]
This design has 1 purely combinational loop. Such loops should be avoided. If at all possible, please modify the design to eliminate all unclocked feedback paths.

To eliminate this warning message, use D registers instead of latches. For example, in the code example in Figure 2-13, to convert the D latch to a D register, use an Else statement, a Wait Untill statement, or modify the code to resemble the code in Figure 2-15.

In the example in Figure 2-15, you can use a Wait Until statement instead of an If statement, however, use an If statement when possible because it gives you more control over the inferred register’s capabilities. For more information on latches and registers, refer to the Synopsys VHDL compiler documentation.

-- D_REGISTER.VHD
-- Xilinx HDL Synthesis Design Guide for FPGA
-- June 1995

-- Changing Latch into a D-Register
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsignad.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
REG: process (CLK, DATA)
begin
  if (CLK'event and CLK='1') then
    Q <= DATA;
  end if;
end process; --End REG
end BEHAV;

Figure 2-15 Converting a D Latch to a D Register

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 XC4000 devices, you can implement a D latch by instantiating a RAM 16x1 primitive, as illustrated in Figure 2-16.

![D Latch Implemented by Instantiating a RAM](image)

**Figure 2-16 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.

Table 2-1 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

<table>
<thead>
<tr>
<th></th>
<th>D Latch</th>
<th>XC4000 RAM 16x1 Primitive</th>
<th>D Flip-Flop</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages/Disadvantages</strong></td>
<td>RTL HDL that infers D latch implemented with gates. Combinatorial feed-back loop results in hold-time requirement.</td>
<td>Structural HDL. Instantiated RAM 16x1 primitive. No hold time or combinatorial loop.</td>
<td>Requires change to the RTL HDL to convert D latches to D flip-flops. No hold time or combinatorial loop.</td>
</tr>
<tr>
<td><strong>Area</strong></td>
<td>1 Function Generator</td>
<td>1 Function Generator</td>
<td>1 Register</td>
</tr>
<tr>
<td><strong>Speed</strong></td>
<td>1 Logic Level; combinatorial feed-back loop.</td>
<td>1 Logic Level; no combinatorial loop.</td>
<td>1 Logic Level; no combinatorial loop.</td>
</tr>
</tbody>
</table>

1Area is the number of function generators and registers required. Each CLB has two function generators and two registers in XC4000 devices.

2Speed is the number of CLB logic levels required.

### 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 functions. The Xilinx DesignWare functions use X-BLOX modules that take advantage of the carry logic in XC4000 CLBs. XC4000 carry logic and its dedicated routing increase the speed of arithmetic functions that are larger than 4-bits. To increase speed, use the Xilinx X-BLOX DesignWare 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 Xilinx DesignWare 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 Figure 2-17. The HDL example in Figure 2-17 is shown implemented with gates in Figure 2-18.
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 (? downto 0);
         Z1: out STD_LOGIC_VECTOR (? 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;

Figure 2-17 Resource Sharing
If you disable resource sharing with the \texttt{Hdl\_resource\_allocation = none} command or if you code the design with the adders in separate processes, the design is implemented using two X-BLOX modules as shown in Figure 2-19.
Figure 2-19 Implementation of No Resource Sharing
Table 2-2 provides a comparison of the number of CLBs used and the delay for the design in Figure 2-17 with and without resource sharing. The last column in Table 2-2 provides CLB and delay information for the same design with resource sharing and without X-BLOX modules.

**Table 2-2 Resource Sharing/No Resource Sharing Comparison**

<table>
<thead>
<tr>
<th></th>
<th>Resource Sharing with Xilinx DesignWare</th>
<th>No Resource Sharing with Xilinx DesignWare</th>
<th>Resource Sharing without Xilinx DesignWare</th>
</tr>
</thead>
<tbody>
<tr>
<td>F/G Functions</td>
<td>24</td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td>Fast Carry Logic CLBs</td>
<td>5</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>Longest Delay</td>
<td>53.2 ns</td>
<td>46.0 ns</td>
<td>92.6 ns</td>
</tr>
<tr>
<td>Advantages/Disadvantages</td>
<td>Potential for area reduction</td>
<td>Potential for decreased critical path delay</td>
<td>No carry logic increases CLB count; longer path delays</td>
</tr>
</tbody>
</table>

**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 X-BLOX DesignWare library because the underlying logic of the components is not available when the design is compiled. The component logic is created later when the X-BLOX program is run.

In the design shown in Figure 2-20, two instances of the xblox_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.
Figure 2-20 Gate Reduction

Table 2-3 provides a comparison of the number of CLBs used and the delay for the design in Figure 2-20 using the Synopsys DesignWare library and the X-BLOX DesignWare 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/X-Blox DesignWare

<table>
<thead>
<tr>
<th>DesignWare Library</th>
<th>Synopsys FPGA Report</th>
<th>PPR Report</th>
<th>XDelay (CLB Levels)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synopsys DesignWare Library</td>
<td>48 CLBs</td>
<td>61 Occupied</td>
<td>C2S\textsuperscript{1}: 33.7 (4)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P2S\textsuperscript{2}: 53.0 (6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C2P\textsuperscript{3}: 16.4 (0)</td>
</tr>
<tr>
<td>X-BLOX DesignWare Library</td>
<td>71 CLBs</td>
<td>82 Occupied</td>
<td>C2S\textsuperscript{1}: 48.9 (8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P2S\textsuperscript{2}: 32.1 (3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C2P\textsuperscript{3}: 15.8 (0)</td>
</tr>
</tbody>
</table>

\textsuperscript{1}ClockToSetup
\textsuperscript{2}PadToSetup
\textsuperscript{3}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 flip-flops have a dedicated clock enable pin and either a clear (asynchronous reset) pin or a preset (asynchronous set) pin. All non-register functions and latches are implemented with combinatorial logic in the function generators.

You can configure XC4000 CLB registers to have either a preset pin or a clear pin. You cannot configure the CLB for both pins. You must modify any process that requires both pins to use only one pin or you must use two registers to implement the process. An XC4000 CLB is shown in Figure 2-21.
Figure 2-21 XC4000 Configurable Logic Block

The HDL design in Figure 2-22 shows how to describe a register with a clock enable and either a preset or a clear.
HDL Coding Hints

-- Example of Implementing Registers
-- 10/10/94 HTInux

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
      D.Q_OUT <= "00000000";
    elsif (CLOCK'event and CLOCK='1') then
      D.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;

Figure 2-22 Register Inference

HDL Synthesis for FPGAs 2-29
Using Clock Enable Pin

Use the CLB clock enable pin instead of gated clocks in your designs. Figure 2-23 illustrates a design that uses a gated clock and Figure 2-24 shows the design implemented with gates. Figure 2-25 shows how you can modify this design to use the clock enable pin of the CLB and Figure 2-26 shows this design implemented with gates.

```vhdl
-- GATE_CLOCK.VHD
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- June 1995

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

derive gate_clock is
    port (IN1,IN2,DATA,CLK,LOAD: in STD_LOGIC;
          OUT: 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='1' and GATECLK='0') then
            if (LOAD='1') then
                OUT <= DATA;
            end if;
        end if;
    end process; --End GATE_PR
end BEHAVIORAL;
```

Figure 2-23 Gated Clock

![Figure 2-23 Gated Clock](image)

Figure 2-24 Implementation of Gated Clock

![Figure 2-24 Implementation of Gated Clock](image)
The VHDL syntax for If statements is as follows:

```vhdl
if condition then
    sequence_of_statements;
elsif condition then
    sequence_of_statements;
else
    sequence_of_statements;
end if;
```

---

**Figure 2-25 Gated Clock Modified to Use Clock Enable Pin**

![Gated Clock Diagram]

**Figure 2-26 Implementation of Clock Enable**

**Using If Statements**

The VHDL syntax for If statements is as follows:

```vhdl
if condition then
    sequence_of_statements;
elsif condition then
    sequence_of_statements;
else
    sequence_of_statements;
end if;
```
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.

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

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

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

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;

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 clause are executed when the given expression does not match
any of the choices. The Others clause is optional, however, if you do not use it, you must include all possible values for expression. Also, each When 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 Figure 2-27. This description is shown implemented in gates in Figure 2-28. The same example is shown in Figure 2-29, however, the Case construct is used with the Nested_If to more effectively describe the same function. The Case construct reduces the delay by approximately 10 ns (using an XC4005-5 part). The implementation of this description is shown in Figure 2-30.
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
        CLR: 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) <= "02";
      elsif (ADDR_A = "01") then
        DEC_Q(5 downto 4) <= unsigned(ADDR_B) + '1';
        DEC_Q(3 downto 2) <= unsigned(ADDR_B) + '1';
        if (ADDR_C = '1') then
          DEC_Q(5 downto 4) <= unsigned(ADDR_D) + '1';
          DEC_Q(3 downto 2) <= ADDR_A;
          if (ADDR_C = '0') then
            DEC_Q <= "000000";
          end if;
        end if;
      else
        DEC_Q(5 downto 4) <= ADDR_D;
      end if;
    end if;
  else
    end if;
end process;

Figure 2-27 Inefficient Use of Nested_If Statement
Figure 2-28 Implementation of Nested_If
HDL Synthesis for FPGAs

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 "00010111" =>
          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 "00011000" =>
          DEC_Q(5 downto 4) <= unsigned(ADDR_B) + '1';
          DEC_Q(3 downto 2) <= unsigned(ADDR_C) + '1';
          DEC_Q(1 downto 0) <= unsigned(ADDR_D) + '1';
        when "00011111" =>
          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 "00100000" =>
          DEC_Q(5 downto 4) <= ADDR_B;
          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;

Figure 2-29 Nested-If Example Modified to Use 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 code example in Figure 2-31 uses an If construct in a 4-to-1 multiplexer design. Figure 2-32 shows the implementation of this design. The code example in Figure 2-33 uses a Case construct for the
same multiplexer. Figure 2-34 shows the implementation of this design. 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.

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

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

architecture BEHAV of ifmux 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;
    else MUX_OUT <= '0';
  end if;
  end process; -- END IF_PRO
end BEHAV;
```

Figure 2-31 4-to-1 Multiplexer Design with If Construct
Figure 2-32 If_Ex Implementation

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;
         MDU_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' => MDU_OUT <= A;
      when '01' => MDU_OUT <= B;
      when '10' => MDU_OUT <= C;
      when '11' => MDU_OUT <= D;
      when others => MDU_OUT <= '0';
    end case;
  end process; --End CASE_PRO
end BEHAV;

Figure 2-33 4-to-1 Multiplexer Design with Case Construct
Figure 2-34 Case_Ex Implementation
Chapter 3

HDL Coding for FPGAs

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, XC4000 FPGAs contain system-oriented features such as the following.

- Global low-skew clock or signal distribution network
- Wide edge decoders
- On-chip RAM and ROM
- IEEE 1149.1 - compatible boundary scan logic support
- Flexible I/O with Adjustable Slew-rate Control and Pull-up/Pull-down Resistors
- 12-mA sink current per output and 24-mA sink per output pair
- Dedicated high-speed carry-propagation circuit

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

This chapter also provides information on implementing the following in your designs:

- State machines
- X-BLOX modules
- Relationally Placed Modules (RPMs)
- XACT-Performance Timing Constraints
Using Global Low-skew Clock Buffers

XC4000 devices have four primary (BUFGP) and four secondary (BUFGS) global clock buffers that share four global routing lines, as shown in Figure 3-1.

Figure 3-1 Global Buffer Routing Resources

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 Figure 3-2. 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 Figure 3-2, 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 3-2 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**

Most synthesis tools can automatically insert global 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 Design 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 generic BUFG clock buffer to ports tagged with a clock attribute. At a later stage in the place and route process, the XNFPrep program replaces the BUFG with the appropriate clock buffer.

   **Note:** Refer to the *Synopsys (XSI)—FPGAs Interface/Tutorial Guide* for more information on inserting I/O buffers and clock buffers.

**Instantiating Internal Global Clock Buffers**

If a high-fanout signal is sourced internally, you must instantiate the BUFGS in your HDL code in order to use the dedicated routing resource. It is easier to change the name of the signal that drives the buffer rather than the lines that are driven by this signal.

**Using Dedicated Global Set/Reset Resource**

XC4000 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.

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 resources to connect to these pins. If your design has a Preset or Clear signal that affects every flip-flop in your design, use the GSR net to increase design performance and reduce routing congestion. After performing an RTL simulation of your design, remove the Preset or Clear signal from the synthesized design and connect the Clear signal to the GSR pin of the STARTUP block.

**Startup State**

The GSR pin on the STARTUP 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 XNF 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 XC3000 devices, all flip-flops and latches are reset to zero after configuration.

**Preset vs. Clear**

XC4000 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.
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 (XSI)—FPGAs Interface/Tutorial Guide for information on changing the initial state of registers that do not use the Preset or Clear pins.

### Increasing Performance with the GSR Net

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

To ensure that your HDL simulation results at the RTL level match the synthesis results, modify your code so that every flip-flop and latch is preset or cleared when the GSR signal is asserted. The Synthesis tool cannot infer the GSR net from HDL code. To utilize the GSR net, you must instantiate the STARTUP block, as shown in Figure 3-5.

### Design Example without Dedicated GSR Resource

In the No_GSR design shown in Figure 3-3, the signal RESET initializes all the registers in the design. This design includes 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. Figure 3-4 shows the No_GSR design implemented with gates.
-- NO_GSR Example
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- The signal RESET initializes all registers
-- May 1995

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;
      UP_CNT : out STD_LOGIC_VECTOR (3 downto 0);
      DN_CNT : 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;

UP_CNT <= UP_CNT;
DN_CNT <= DN_CNT;
end SIMPLE;

Figure 3-3 Design without Dedicated GSR Resource
To reduce routing congestion and improve the overall performance of the reset net in the No_GSR design, use the dedicated GSR net instead of the general purpose routing. Instantiate the STARTUP block in your design and use the GSR pin on the block to access the global reset net. The modified design (Use_GSR) is shown in Figure 3-5. The Use_GSR design implemented with gates is shown in Figure 3-6.

On assertion of the GSR net, flip-flops return to a clear (or Low) state by default. You can override this default by using the flip-flop’s preset pin or by adding the INIT=S attribute to the flip-flop (described below).

The Use_GSR design explicitly states 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 one of the following two methods to prevent this reset to zeros.

- Remove the comment characters from the last few lines of code in the Use_GSR design. These lines of code correctly describe the behavior of the design (in response to the assertion of reset). However, when you synthesize the design, the Preset pins on the flip-flops that form the down-counter are used and the Clear pins on the flip-flops that form the up-counter are used. Using these pins defeats the purpose of using the GSR net.
• Attach the INIT=S attribute to the down-counter flip-flops as follows:

```python
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 (XSI)—FPGAs Interface/Tutorial Guide* for more information on assigning attributes.

Xilinx recommends removing the comment characters from the last few lines of the Use_GSR code when you perform an RTL simulation and attaching the INIT=S attribute to the relevant flip-flops when you synthesize the design.

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

```python
dont_touch cell_name
```

The Xilinx X-BLOX architecture optimizer automatically uses the GSR net if each flip-flop and IOB latch in your design uses a common signal to drive the Preset or Clear pins.
-- USE_GSR Example
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- The signal RESET is connected to the GSR pin of
-- the STARTUP block
-- May 1995

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

entity use_gsr is
port ( CLOCK: in STD_LOGIC;
      RESET: in STD_LOGIC;
      UP_CNT: out STD_LOGIC_VECTOR (3 downto 0);
      DN_CNT: out STD_LOGIC_VECTOR (3 downto 0));
end use_gsr;

architecture SIMPLE of use_gsr is

component STARTUP
port ( GSR: in STD_LOGIC);
end component;

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

begin
    UI: STARTUP port map(gsr=>reset);
    UP_COUNTER: process(clock)
    begin
        if (CLOCK'event and CLOCK = '1') then
            UP_CNT <= UP_CNT + 1;
        end if;
    end process;
    DN_COUNTER: process(clock)
    begin
        if (CLOCK'event and CLOCK = '1') then
            DN_CNT <= DN_CNT - 1;
        end if;
    end process;
    UP_CNT <= UP_CNT;
    DN_CNT <= DN_CNT;

    -- RESET_COUNTERS: process (RESET)
    -- begin
    -- if RESET = '1' then
    --     UP_CNT <= "0000";
    --     DN_CNT <= "1111";
    -- end if;
    -- end process;

end SIMPLE;

Figure 3-5 Design with Dedicated GSR Resource
The Use_GSR design is modified to allow a reset of the down-counter to all ones by either asserting the global reset net or by asserting an additional preset signal. A port designated “preset” is added to the design. This new port only effects the down-counter. The new design, Use_GSR_PRE, is shown in Figure 3-7. Figure 3-8 shows this design implemented with gates.
Figure 3-7 Design with Dedicated GSR Resource and Additional Preset Signal
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 Figure 3-10; refer to this figure when reviewing Figure 3-11 and Figure 3-12.

Note: The angle bracket 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 Figure 3-9 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 Figure 3-10. This design example shows 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. This design uses three flip-flops to implement seven states.

Figure 3-9 State Machine Bubble Diagram
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 := "001 010 011 100 101 110 111";
  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 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;
      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 <= '0';
        CONTIG <= '1';
        SINGLE <= '0';
        if (A or D) then
          NS <= S4;
        else
          NS <= S3;
        end if;
      when others =>
        NS <= NS;
    end case;
  end process: -- End COMB_PROC

end binary;
The recommended encoding style for state machines depends on which synthesis tool you are using. If you use the Synopsys synthesis tool, you can explicitly declare state vectors or you can allow the 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 Figure 3-11. 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 Figure 3-10 for the complete Case statement portion of the code.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.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 BEHAV of enum is

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

begin
  SVNC_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 SVNC_PROC
  COMB_PROC: process (CS, A, B, C, D, E)
  begin
    case CS is
      when S1 =>
      MULTI <= '0';
      CONTIG <= '0';
      SINGLE <= '0';
      ..
    end case;
  end process: --End COMB_PROC

end enum;
```

**Figure 3-11 Enumerated Type Encoded State Machine**

### Using One-Hot Encoding

The example in Figure 3-12 shows 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. This example uses one flip-flop for each of the seven states.

**Note:** Refer to Figure 3-10 for the complete Case statement portion of the code. See “Appendix A” of this manual for a detailed description of one-hot encoding and its applications.
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.

```vhdl
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, CONFIG : 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: STRING;
  attribute ENUM_ENCODING: STRING;
  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

  COMP_PROC: process (CS, A, B, C, D, E)
  begin
    case CS is
      when S1 =>
        MULTI <= '0';
        CONFIG <= '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;
      ...
    end case;
  end process;
end;
```

Figure 3-12 One-hot Encoded State Machine
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

Table 3-1 summarizes the synthesis results from the different methods used to encode the state machine in Figure 3-10, Figure 3-11, and Figure 3-12. The results are for an XC4005PC84-5 device.
Table 3-1  Comparing State Machine Encoding Styles (XC4005-5)

<table>
<thead>
<tr>
<th></th>
<th>One-Hot</th>
<th>Binary</th>
<th>Enum (One-hot)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occupied CLBs</td>
<td>15</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>CLB Flip-flops</td>
<td>7</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>PadToSetup</td>
<td>33.8 ns (2)</td>
<td>42.8 ns (4)</td>
<td>27.6 ns (2)</td>
</tr>
<tr>
<td>ClockToPad</td>
<td>23.9 ns (1)</td>
<td>24.6 ns (1)</td>
<td>22.1 ns (1)</td>
</tr>
<tr>
<td>ClockToSetup</td>
<td>19.9 ns (1)</td>
<td>29.3 ns (3)</td>
<td>15.8 ns (1)</td>
</tr>
</tbody>
</table>

*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.

**Note:** XDelay was used to obtain the timing results in Table 3-1.

**Initializing the State Machine**

When you use one-hot encoding, assign an INIT=S attribute to the initial state register to ensure that the FPGA is initialized to a Set state. If you use the FPGA Compiler, use the following command to specify your design’s start-up state.

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

**Note:** The “\” character in this command represents a continuation marker.

Alternatively, you can add the following lines of code to your design to specify the initial state.

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

**HDL Synthesis for FPGAs**

3-20  

*Xilinx Development System*
In the example shown in Figure 3-10, the signal RESET 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 the XC4000 device has four wide decoder circuits at each edge (two in XC4000A devices). 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 edge decoders in HDL, you must instantiate edge decoder primitives. The primitive names you can use vary with the synthesis tool you are using. If you use 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 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 example in Figure 3-13 shows you 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. Figure 3-14 shows the schematic block diagram representation of this I/O decoder design.
HDL Synthesis for FPGAs

-- Edge Decoder
-- A X4000 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
-- outputs (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 (2 downto 0);
      CLK: In std_logic;
      QUT: out std_logic_vector (3 downto 0));
end io_decoder;

Architecture STRUCTURE of io_decoder is

Component decode1_to
  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 decode
  Port (A1, 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, CKEN, 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: DECODE10 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_INV(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_INV(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_INV(0), DECODE(2));
C3: DECODE1_INT port map (CLB_INV(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_INV(0), DECODE(3));
D3: DECODE1_INT port map (CLB_INV(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;

HDL Synthesis for FPGAs
HDL Synthesis for FPGAs

-----“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 3-13 Using Dedicated I/O Decoders
Figure 3-14 Schematic Block Representation of I/O Decoder

Note: In Figure 3-14, the pull-up resistors are inside the Decoder blocks.

Instantiating X-BLOX Modules

This section describes how to instantiate X-BLOX modules in your HDL code. Most synthesis tools can infer arithmetic X-BLOX modules from VHDL or Verilog arithmetic operators (+, -, <, <=, >, >=, +1, and -1). These X-BLOX modules use the XC4000 dedicated carry logic to improve the area and speed of designs. For bus widths greater than four, X-BLOX modules are generally faster unless multiple instances of the same function are compiled together.
Note: Refer to the “Resource Sharing” and “Gate Reduction” sections in the “HDL Coding Hints” chapter for more information.

The synthesis tool automatically infers the ADD_SUB, INC_DEC, and COMPARE modules. Xilinx X-BLOX modules include additional functions that cannot be automatically inferred by the synthesis tool. You must explicitly instantiate these X-BLOX functions.

Using X-BLOXGen

X-BLOXGen is a program that allows you to instantiate X-BLOX modules in your HDL code. You can use X-BLOXGen to instantiate the X-BLOX modules listed in Table 3-2.

Table 3-2  X-BLOX Modules Instantiated with X-BLOXGen

<table>
<thead>
<tr>
<th>X-BLOX Modules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCUM</td>
<td>Universal accumulator</td>
</tr>
<tr>
<td>ADD_SUB</td>
<td>Adder and subtracter</td>
</tr>
<tr>
<td>COMPARE</td>
<td>Compares the magnitude and equality of two values</td>
</tr>
<tr>
<td>COUNTER</td>
<td>Universal counter</td>
</tr>
<tr>
<td>DATA_REG</td>
<td>Universal register</td>
</tr>
<tr>
<td>DECODE</td>
<td>Translates data from any encoding to one-hot encoding</td>
</tr>
<tr>
<td>INC_DEC</td>
<td>Increments and decrements by a constant</td>
</tr>
<tr>
<td>SHIFT</td>
<td>Register that loads and shifts data in parallel or</td>
</tr>
<tr>
<td></td>
<td>serially; also shifts data out</td>
</tr>
<tr>
<td>CLK_DIV</td>
<td>Clock divider</td>
</tr>
</tbody>
</table>

X-BLOXGen prompts you for the name of the X-BLOX module and for the size and values of the attributes for that module. X-BLOXGen does not check the validity of the information that you enter in response to the screen prompts. Any data input errors are reported by X-BLOX in the X-BLOX log file (design.blx).

Copy the component declaration and template of the instantiation into your HDL code. Complete the instantiation by assigning the
signals that are connected to the X-BLOX module. Figure 3-15 shows the design flow for X-BLOXGen.

**Note:** Refer to the X-BLOX Reference/User Guide for information on X-BLOX modules and attributes.

![Figure 3-15 X-BLOXGen Flow Diagram](image)

**Syntax**

To use X-BLOXGen, enter the following on the command line.

```
xbloxgen  X-BLOX module name -options
```

Specify the X-BLOX module you want to generate in lower case, such as accum, add_sub, compare, counter, compare, data_reg, decode, inc_dec, shift, or clk_div. If you do not specify a module name, you are prompted for a name.
**Options**

If you do not specify an option, you are prompted for one.

- **vhdl**

  Creates an XNF file and an output file that contains the VHDL component declaration and template for the component instantiation. Copy the declaration and template into your HDL code. The default data types are Std_logic and Std_logic_vector.

- **verilog**

  Creates an XNF file and an output file that contains the Verilog component declaration and template for the component instantiation. Copy the declaration and template into your HDL code.

**Output Files**

X-BLOXGen creates the output files shown in Table 3-3.

**Table 3-3  X-BLOXGen Output Files**

<table>
<thead>
<tr>
<th>Output File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>design.xnf</td>
<td>XNF file for the instantiated X-BLOX module. XMake merges this file into your top-level design.</td>
</tr>
<tr>
<td>design.xbg</td>
<td>Log file.</td>
</tr>
<tr>
<td>design.blx</td>
<td>Report file.</td>
</tr>
<tr>
<td>design.vhd</td>
<td>VHDL template file for the X-BLOX module.</td>
</tr>
<tr>
<td>design.v</td>
<td>Verilog template file for the X-BLOX module.</td>
</tr>
</tbody>
</table>

**X-BLOXGen Example**

To instantiate a 16-bit accumulator X-BLOX module using X-BLOXGen, follow the steps in this section.

**Note:** The example in this section is for a VHDL design.

1. Enter the following command:

   `xbloxygen accum -vhdl`

   The following information is displayed on your screen.
XBLOXGEN Version 1.12

Looking for XBLOX...XBLOX found.

Looking for data files:
  partlist.xct found.
  4002p100.pkg found.
  4003p100.pkg found.
  4004p160.pkg found.
  4005p208.pkg found.
  4006p208.pkg found.
  4008p208.pkg found.
  4001p208.pkg found.
  4013p240.pkg found.
  4025g299.pkg found.

The following questions will define the characteristics of the ACCUM to be generated.

2. Respond to the screen prompts appropriately as follows:

   **Note:** The bold text indicates your response to the screen prompts.

Enter the name of the ACCUM: **alu**

Enter the number of bits of the ACCUM (or 'q' to quit): **16**

The available arithmetic bus encodings are:

1) **UBIN**
2) **TWO_COMP**

Enter the number of the desired encoding (or 'q' to quit): **1**

The available ACCUM arithmetic operations are:

1) **ADD** (default)
2) **SUB**
3) **ADD/SUB**

Enter the number of the desired operation (return for default) (or 'q' to quit): **3**

Will the c.in input be needed? Enter 'y' or 'n' (or 'q' to quit): **y**

Will the ACCUM be loadable? Enter 'y' or 'n' (or 'q' to quit): **n**

Will the clk_en input be needed? Enter 'y' or 'n' (or 'q' to quit): **y**
Will an asynchronous control be needed?
Enter 'y' or 'n' (or 'q' to quit): y

Enter the asynchronous value (return for default 0)
or 'q' to quit): 2#1001111111101011#

Will a synchronous control be needed?
Enter 'y' or 'n' (or 'q' to quit): n

Will the q_out parallel output be needed?
Enter 'y' or 'n' (or 'q' to quit): y

Will the c_out output be needed?
Enter 'y' or 'n' (or 'q' to quit): y

Will the ovfl output be needed?
Enter 'y' or 'n' (or 'q' to quit): n

The available ACCUM implementation styles are:
1) ALIGNED (default)
2) UNALIGNED
3) RIPPLE

Enter the number of the desired style (return for default)
or 'q' to quit): 1

Should an RLOC be generated?
Enter 'y' or 'n' (or 'q' to quit): y

The following is a list of device types that have been found in your system:
4002
4003
4004
4005
4006
4008
4010
4013

What is the SMALLEST device the ACCUM 'alu' will be used in?
Enter device number (or 'q' to quit): 4005

Note: Specify the smallest XC4000 device in which the accumulator will be used. If you select a device that is smaller than the specified device, XNFPrep may fail.

Generating ACCUM with the following characteristics:

smallest device: 4005
size: 16-bit
encoding: UBIN
HDL Coding for FPGAs

style: ALIGNED
create rpm: TRUE
async_val: 2#1001111111101011#
async_ctrl: yes
sync_ctrl: no
operation: ADD_SUB
b input used: yes
clk: yes
carry-in: yes
load: no
clk_ena: yes
q_out: yes
carry-out: yes
overflow: no

X-BLOXGen builds the accumulator by running X-BLOX. The files alu.xnf, alu.vhd, alu.xbg, and alu.blx are generated as follows:

Running X-BLOX to create 'alu.xnf'.
The file 'alu.xnf' has been successfully created.

Generating VHDL example in file 'alu.vhd'.
-- The following code is a VHDL example of how to instantiate
-- the ACCUM 'alu' you have created.
--
-- The component declaration:
--
component alu
  port (  
    ASYNC_CTRL: in STD_LOGIC;
    ADD_SUB: in STD_LOGIC;
    B: in STD_LOGIC_VECTOR (15 downto 0);
    CLOCK: in STD_LOGIC;
    C_IN: in STD_LOGIC;
    CLK_EN: in STD_LOGIC;
    Q_OUT: out STD_LOGIC_VECTOR (15 downto 0);
    C_OUT: out STD_LOGIC
  );
end component;
--
-- The component instantiation:
--
  U0: alu port map(
    ASYNC_CTRL=>ASYNC_CTRL_SIG,
    ADD_SUB=>ADD_SUB_SIG,
    B=>B_BUS,
    CLOCK=>CLOCK_SIG,
    C_IN=>C_IN_SIG,
    }
HDL Synthesis for FPGAs

```vhdl
CLK_EN=>CLK_EN_SIG,
Q_OUT=>Q_OUT_BUS,
C_OUT=>C_OUT_SIG
);

Saving log file 'xbloxgen.log' to file 'alu.xbg'.

Would you like to create another component?
Enter 'y' or 'n' (or 'q' to quit): n

Cleaning................DONE
Goodbye

3. Cut and paste the VHDL code from the alu.vhd file into your top-level design. Replace the names async_ctrl_sig, add_sub_sig, b_bus, clock_sig, c_in_sig, clk_en_sig, q_out_bus, and c_out_sig with the actual signal names.

4. Synthesize your design.

5. Run XMake on the output from the synthesis tool.
   XMake automatically merges the XNF file for the X-BLOX module into your top-level design.

Using RPMs

The Xilinx Libraries include Relationally Placed Modules (RPMs). These modules are XC4000 functions that use the XC4000 carry logic. Additionally, RPMs are soft macros that contain logic symbols with Relative Location (RLOC) parameters. Use RLOC parameters to define the spatial relationship between logic symbols. PPR maintains these spatial relationships as it searches for the best absolute placement of the logic in the device. RLOCs do not define the absolute placement of the logic in the device. Optionally, you can define absolute placement with an RLOC_ORIGIN parameter on an RPM.

Note: RPMs replace all Xilinx-supplied hard macros. Do not use pre-Unified Libraries hard macros in new designs.

In addition to the RPMs listed in the Libraries Guide, you can create your own RPMs with schematic entry tools as follows:

1. Place logic on your schematic and assign RLOC parameters where applicable.
2. Translate your RPMs to an XNF file with a schematic-to-XNF translator.

3. Copy the XNF file for the RPM to your working directory.

4. Instantiate the RPM in your HDL design as described in the next section.

The XNF files for the RPMs in the *Libraries Guide* are provided with this manual or they can be obtained from the Xilinx hotline or bulletin board. Table 3-4 lists the RPMs in the *Libraries Guide*.

**Note:** Refer to your schematic entry tool documentation for more information on creating RPMs.

**Table 3-4 RPMs in the *Libraries Guide***

<table>
<thead>
<tr>
<th>RPM</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc16</td>
<td>16-bit Loadable Cascadable Accumulator with Carry-in, Carry-out, and Synchronous Reset</td>
</tr>
<tr>
<td>acc4</td>
<td>16-bit Loadable Cascadable Accumulator with Carry-in, Carry-out, and Synchronous Reset</td>
</tr>
<tr>
<td>acc8</td>
<td>8-bit Loadable Cascadable Accumulator with Carry-in, Carry-out, and Synchronous Reset</td>
</tr>
<tr>
<td>add16</td>
<td>16-bit Cascadable Full Adder with Carry-in, Carry-out, and Overflow</td>
</tr>
<tr>
<td>add4</td>
<td>4-bit Cascadable Full Adder with Carry-in, Carry-out, and Overflow</td>
</tr>
<tr>
<td>add8</td>
<td>8-bit Cascadable Full Adder with Carry-in, Carry-out, and Overflow</td>
</tr>
<tr>
<td>adsu16</td>
<td>16-bit Cascadable Adder/Subtracter with Carry-in, Carry-out, and Overflow</td>
</tr>
<tr>
<td>adsu4</td>
<td>4-bit Cascadable Adder/Subtracter with Carry-in, Carry-out, and Overflow</td>
</tr>
<tr>
<td>adsu8</td>
<td>8-bit Cascadable Adder/Subtracter with Carry-in, Carry-out, and Overflow</td>
</tr>
<tr>
<td>cc16ce</td>
<td>16-bit Cascadable Binary Counter with Clock Enable and Clear</td>
</tr>
</tbody>
</table>
Instantiating an RPM

This section describes a procedure for instantiating an RPM from the Libraries Guide in your HDL design. The RPM in the example provided is an ACC4 (4-bit loadable cascadable accumulator with carry-in, carry-out, and synchronous reset). This example is targeted for Synopsys compilers. To instantiate the ACC4 RPM, follow the steps listed here.

1. Go to the acc4 directory.

   If you created the RPM with a schematic entry tool, generate an XNF file with the schematic-to-XNF translator. Save the XNF file in the same directory as the HDL code that will contain the RPM.

2. Instantiate the RPM in your code as shown in Figure 3-16.

   You must list the individual signals for each bus to prevent warnings and errors from XNFMerge and XNFPrep. For example, in Figure 3-16, B_IN bus is declared as signals B_IN3, B_IN2, B_IN1, and B_IN0.
Note: Refer to the Libraries Guide for the pin names of the RPM you want to instantiate.

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

declaration

entity rpm_example is
  port (IN1, IN2, IN3, IN4, IO, IO1: in STD_LOGIC;
        CE, L: in STD_LOGIC;
        QO, QO1, QO2, QO3: out STD_LOGIC);
end entity;

architecture STRUCTURE of rpm_example is
begin

  component acc4
    port (B3, B2, B1, B0, D3, D2, D1, DO : in STD_LOGIC;
          Cl, L, AO, CE, C: in STD_LOGIC;
          Q1, Q2, Q3, Q4: out STD_LOGIC;
          CO, OF1: out STD_LOGIC);
  end component;

  port map (B3=>B_IN3, B2=>B_IN2, B1=>B_IN1, B0=>B_IN0,
            D3=>D_IN3, D2=>D_IN2, D1=>D_IN1, D0=>D_IN0,
            Cl=>CI_IN, L=>L_IN,
            AO=>AO1_IN, CE=>CE_OUT, C=>CLK_IN,
            Q1=>Q_OUT3, Q2=>Q_OUT2, Q3=>Q_OUT1, Q4=>Q_OUT0,
            CO=>CO_OUT, OF1=>OF1_OUT);

  end STRUCTURE;

Figure 3-16 Instantiating an RPM in VHDL

3. Before you compile the design, set the Don't Touch attribute on the instantiated RPM using the Synopsys tools.

4. Compile your design using either the FPGA Compiler or the Design Compiler.
   The compilers do not synthesize or optimize the RPM; the RPM is categorized as a "black box".

5. Synthesize your design and save it as an SXNF file.

6. Run XMake on the SXNF file. XMake merges in the RPM XNF files (acc4.xnf, adsu4.xnf, and m2_1.xnf) in your working directory.
Implementing Memory

You can use on-chip RAM for status registers, index registers, counter storage, distributed shift registers, LIFO stacks, and FIFO buffers.

XC4000 devices can efficiently implement RAM and ROM using CLB function generators. XC4000 libraries include 16 x 1 (deep x wide) and 32 x 1 RAM and ROM primitives that you can instantiate in your code.

You can also implement memory using the MemGen program, which is included in the XACTstep Development System. Use MemGen to create RAMs and ROMs that are between 1 to 32 bits wide and 2 to 256 bits deep. See the “Using MemGen” section below for more information.

**Note:** Refer to the Development System Reference Guide for detailed information on MemGen.

Implementing XC4000 RAMs

**Note:** Do not use RTL descriptions of RAMs in your VHDL code because compiling creates combinatorial loops.

You can implement RAMs in your HDL code as follows:

- Instantiate 16 x 1 and 32 x 1 RAM primitives
- Use MemGen to implement any other RAM size

Implementing XC4000 ROMs

You can implement ROMs in your HDL code as follows:

- Use RTL descriptions of ROMs
- Instantiate 16 x 1 and 32 x 1 ROM primitives
- Use MemGen to implement any other ROM size

An RTL description of a ROM is shown in Figure 3-17.
Figure 3-17 RTL Description of 16 x 4 ROM

The synthesis tool creates ROMs from random logic gates that are implemented using function generators. Alternatively, you can implement ROMs using MemGen as shown in Figure 3-18 and Figure 3-19.

To instantiate the 16 x 1 and 32 x 1 ROM primitives in your HDL design, use the Set Attribute command to define the ROM value as follows:

```vhdl
set_attribute "instance_name" xnf_init "rom_value"
```

```vhdl
type string
```
Note: Refer to the appropriate Xilinx interface documentation for more information on defining the ROM value.

Instantiating ROMs or RAMs does not allow you to functionally simulate your design or easily migrate between FPGA families; however, instantiation is the most efficient way to implement memory in XC4000 devices.

Using MemGen

Follow these steps to use MemGen to instantiate a ROM in your HDL code:

1. Create a memory description file filename.mem. Figure 3-18 shows an example of a memory description file with the name promdata.mem.

Note: Refer to the Development System Reference Guide for more information on MemGen.

```
; Memory file for PROM symbol called 'PROM_DATA'
; ---------------------------------------------------
TYPE = ROM
WIDTH = 4
DEPTH = 16
DEFAULT = 0 ; -- default value here
DATA = 20H0000H,
       20H0001H,
       20H0010H,
       20H0100H,
       20H1000H,
       20H1010H,
       20H1100H,
       20H1101H,
       20H1110H,
       20H1111H ; -- END of ROM data
```

Figure 3-18 Memory Description File (Promdata.mem)

2. Run MemGen on the filename.mem file to create an XNF file:

   memgen  promdata.mem
3. Instantiate the memory module in your HDL design, as shown in the rom_memgen design in Figure 3-19.

The address lines must be named A0 – A3 and the output data lines must be named O0 – O3. When the rom_memgen design is compiled in the FPGA Compiler, the following warning occurs:

Warning: Unable to resolve reference ‘promdata’ in ‘ROM_INT’ (LINK-5)

You can safely ignore this message.

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

entity rom_memgen is
  port (ADDR: in STD_LOGIC_VECTOR (3 downto 0);
        DATA: out STD_LOGIC_VECTOR (3 downto 0));
end rom_memgen;

architecture BEHAV of rom_memgen is

component promdata
  port (A0,A1,A2,A3: In STD_LOGIC;
        DATA: out STD_LOGIC);
end component;

begin
  promdata port map (A0=>ADDR(0),A1=>ADDR(1),A2=>ADDR(2),A3=>ADDR(3),
                     0=>DATA(0),1=>DATA(1),2=>DATA(2),3=>DATA(3));
end BEHAV;
```

Figure 3-19 Instantiating a 16 x 4 ROM

4. Save the design as an SXNF file, for example, rom_memgen.sxnf.

5. Translate the SXNF output file to an XNF file using the appropriate translation program. For example, if you are using Synopsys tools, run the Syn2XNF program.

Note: The Syn2XNF translator automatically merges in the XNF file for the memory, for example, promdata.xnf. Refer to the Synopsys (XSI)—FPGAs Interface/Tutorial Guide for more information on Syn2XNF.
Implementing Boundary Scan (JTAG 1149.1)

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

XC4000 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.

**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 XC4000 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 example in Figure 3-20 shows how to instantiate the boundary scan symbol, BSCAN, into your HDL code. In this example, the four TAP pins are declared as ports. The schematic for this design is shown in Figure 3-21.

You **must** assign a Synopsys 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.
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;
        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 structure of bnd_scan is
  component BSCAN
    port (TDI, TMS, TCK: in STD_LOGIC;
          TDO: out STD_LOGIC);
  end component;
  component TDI
    port (I: in STD_LOGIC);
  end component;
  component TMS
    port (I: in STD_LOGIC);
  end component;
  component TCK
    port (I: in 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;
begin
  U1: BSCAN port map (TDO => TDI_P,
                     TDI => TDI_P,
                     TMS => TMS_P,
                     TCK => TCK_P);
  U2: TDI port map (I => TDI_P);
  U3: TCK port map (I => TCK_P);
  U4: TMS port map (I => TMS_P);
  U5: TDO port map (O => TDO_P);
  U6: count4 port map (LOAD => LOAD_P,
                        CE => CE_P,
                        CLOCK => CLOCK_P,
                        RST => RESET_P,
                        DATA => DATA_P,
                        COUT => COUT_P);
end structure;

Figure 3-20 Boundary Scan Design (Bnd_scan)
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 family devices have different IOB functions. The following sections provide a general description of the IOB function in XC4000/A/D/H devices. A description of how to manually implement additional I/O features is also provided.

Note: For specific information on implementing IOB functions, refer to the appropriate Xilinx interface document for the Synthesis tool you are using to process your designs.
**XC4000/A/D IOBs**

You can configure XC4000/A/D IOBs as input, output, or bidirectional signals. You can also specify pull-up or pull-down resistors, independent of the pin usage.

**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.

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.

**Note:** Registers that connect to an input or output pad and require a Clock Enable, 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. (PPR automatically optimizes any inverters into the IOB.) The XC4000 output buffers can sink 12 mA. The XC4000A output buffers can sink 24 mA.

**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.

**XC4000/D Slew Rate**

XC4000/D output buffers have a default slow slew rate that alleviates many ground bounce problems. Optionally, these output buffers can have a fast slew rate that reduces the output delay. The slow slew rate increases the transition time and reduces the noise...
level. The fast slew rate decreases the transition time and increases the noise level.

**XC4000A Slew Rate**

XC4000A devices have output slew rate control options for each output drive. These options are fast, medium fast, medium slow, and slow. Slew control can alleviate ground bounce problems when multiple outputs switch simultaneously. It can also reduce or eliminate cross-talk and transmission-line effects on printed circuit boards.

**XC4000H IOBs**

XC4000H FPGAs are designed for I/O-intensive applications. Compared to the XC4000, the XC4000H has almost twice as many IOBs and I/O pins. The XC4000H allows you to select either CMOS- or TTL-level output and input thresholds (selectable per pin). The output from this device sinks 24 mA and provides improved tristate and slew-rate control.

**Inputs**

*Note: XC4000H devices do not have input flip-flops.*

To individually configure the inputs with TTL or CMOS thresholds, you must set the threshold level for each input. If you do not specify the threshold, the Synopsys tools assign a random input threshold for each input. Set the input threshold after compiling your design to prevent the optimization of the registers into the IOBs.

*Note: Refer to the *Synopsys (XSI)—FPGAs Interface/Tutorial Guide* for information on setting the input threshold values.*

**Outputs**

*Note: XC4000H devices do not have output flip-flops.*

To individually configure the outputs as either TTL or CMOS compatible, select TTL-level outputs for systems that use TTL-level input thresholds and select CMOS for systems that use CMOS input thresholds. If you use Synopsys tools, you must set the threshold level for each output. If you do not specify the threshold, the tools assign a random output threshold for each output. Set the output
threshold after compiling your design. Also, to prevent the tools from placing flip-flops in the IOBs, insert pads after compiling your design.

**XC4000H Slew Rate**

XC4000H devices have a capacitive and a resistive slew rate and the outputs sink 24 mA. You can configure each output for either of the two slew rate options.

A resistive load has a pull-down transistor that is driven hard, resulting in an almost constant on-resistance of about 10 ohms. A resistive load provides the fastest High-to-Low transition and the ability to sink 24 mA with a voltage drop of 500 mV. You may get excessive ground bounce when too many outputs switch High-to-Low simultaneously.

When you configure the output for a capacitive load (or soft edge), the High-to-Low transition starts as described in the previous paragraph, but the drive to the pull-down transistor is reduced as soon as the output voltage reaches a value close to 1V. A capacitive load provides higher resistance in the pull-down transistor, slowing down of the falling edge, and decreased ground bounce.

*Note:* Refer to the 1994 version of *The Xilinx Programmable Logic Data Book* for more information.

**Instantiating Bidirectional I/O**

This section includes an HDL example that shows you how to instantiate bidirectional I/Os using the FPGA Compiler or Design Compiler. The I/O cell names depend on which synthesis tool you are using.

The VHDL design, bidi_reg.vhd, shown in Figure 3-22 is a top-level design that instantiates the reg4.vhd core design. In this example, two clock buffers, CLOCK1 and CLOCK2, automatically infer a BUFG buffer. The reset and load signals, RST and LOADA, automatically infer an IBUF when you run the Set_port_is_pad, Set_pad_type, and Insert_pads commands. However, the FPGA Compiler cannot automatically infer tristate IOB flip-flop (OFDT_F in Figure 3-22) cells in bidirectional I/Os. Therefore, these cells and the IBUF are instantiated in the top-level design.
HDL Synthesis for FPGAs

-- Xilinx HDL Synthesis Design Guide for FPGAs
-- June 1995

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

entity bidi_reg is
  port (S1: in std_logic_vector (3 downto 0);
        LOADA, CLOCK1, CLOCK2, RST: in std_logic);
end bidi_reg;

architecture BEHAV of bidi_reg is
  component reg4
    port (IN: in std_logic_vector (3 downto 0);
          LOAD: in std_logic);
    end component;

  component ORBT_F
    port (O: out std_logic;
          B: in std_logic;
          C: in std_logic;
          T: in std_logic);
    end component;

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

  signal INA, OUTA: std_logic_vector (3 downto 0);

begin
  U0: reg4 port map (IN<>INA, LOAD<>LOADA, CLOCK<>CLOCK1, RST<>RST, OUT<>OUTA);
  U0: ORBT_F port map (O<>SIGA(0), B<>OUTA(0), C<>CLOCK2, T<>LOADA);
  U1: ORBT_F port map (O<>SIGA(1), B<>OUTA(1), C<>CLOCK2, T<>LOADA);
  U2: ORBT_F port map (O<>SIGA(2), B<>OUTA(2), C<>CLOCK2, T<>LOADA);
  U3: ORBT_F port map (O<>SIGA(3), B<>OUTA(3), C<>CLOCK2, T<>LOADA);
  U4: IBUF port map (O<>INA(0), I<>SIGA(0));
  U5: IBUF port map (O<>INA(1), I<>SIGA(1));
  U6: IBUF port map (O<>INA(2), I<>SIGA(2));
  U7: IBUF port map (O<>INA(3), I<>SIGA(3));
end BEHAV;

Figure 3-22 Instantiating Bidirectional I/O

Moving Registers into the IOB

Note: XC4000H 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 clock enable, reset direct, or set direct function can be moved into IOBs. Moving registers or latches into IOBs reduces 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 Figure 3-23.

**Input Register**

![Input Register Diagram](image1)

**Output Register**

![Output Register Diagram](image2)

**Figure 3-23 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 Figure 3-23.

The Design Compiler automatically moves registers into IOBs if the Preset, Clear, and Clock Enable pins are not used. You can also utilize the IOB registers and latches in your HDL code as follows:

- Run X-BLOX on the output XNF file from Syn2XNF. X-BLOX moves registers without a clock enable, reset direct, or set direct pin connected to an I/O into IOBs.

**Note:** X-BLOX may not merge OUTFFs into the IOB if there is internal feedback.
or

- You can instantiate primitives in your HDL code. Refer to the Synopsys (XSI)—FPGAs Interface/Tutorial Guide for more information.

**Using Unbonded IOBs (XC4000/A/D 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 HDL example in Figure 3-24 shows 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)—FPGAs Interface/Tutorial Guide for a list of library primitives that can be used for instantiations.
-- Unbonded I/Os
-- XC4000 I/OA has unbonded I/OB which have storage elements
-- that can be used to build shift registers.
-- Below is a 4-bit Shift Register using Unbonded I/O Flip Flops

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 STRUCTURE 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.Q : STD_LOGIC;

begin
U.Q <= A and B;
Q.OUT <= U.Q(0);

U3: OFD_U port map (Q => U.Q(0),
D => U.D,
C => U.CLK);
U2: IFDI_U port map (Q => U.Q(2),
D => U.Q(3),
C => U.CLK);
U1: OFDI_U port map (Q => U.Q(1),
D => U.Q(3),
C => U.CLK);
U0: IFD_U port map (Q => U.Q(0),
D => U.Q(1),
C => U.CLK);

end STRUCTURE;

Figure 3-24 4-bit Shift Register Using Unbonded I/O
Implementing Multiplexers with Tristate Buffers

A 4-to-1 multiplexer is efficiently implemented in a single XC4000 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 multiplexers larger than 4-to-1.

Multiplexers (larger than 4-to-1) 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 examples. A VHDL design of a 5-to-1 multiplexer built with gates is shown in Figure 3-25. Typically, the gate version of this multiplexer has binary encoded selector inputs and requires three select inputs (SEL<2:0>). The schematic representation of this design is shown in Figure 3-26.

The VHDL design shown in Figure 3-27 is 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 this design is shown in Figure 3-28.
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;

Figure 3-25 Implementing 5-to-1 MUX with Gates

Figure 3-26 5-to-1 MUX Implemented with Gates
-- MUX_BUF.VHD
-- 5-to-1 MUX Implemented in 3-State Buffers
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- May 1995

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

entity muxbuf is
  port (SEL : in STD_LOGIC_VECTOR (4 downto 0);
       A, B, C, D, E : in STD_LOGIC;
       SIG : out STD_LOGIC);
end muxbuf;

architecture RTL of muxbuf 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;

Figure 3-27 Implementing 5-to-1 MUX with BUFTs

Figure 3-28 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 XC4005APC84-5 device is provided in Table 3-5. When the multiplexer is implemented with tristate buffers, no CLBs are used and the delay is smaller.

<table>
<thead>
<tr>
<th>Timing/Area</th>
<th>Using BUFTs</th>
<th>Using Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing</td>
<td>30.3 ns (1 block level)</td>
<td>31.1 ns (2 block levels)</td>
</tr>
<tr>
<td>Area</td>
<td>0 CLBs, 5 BUFTs</td>
<td>2 CLBs</td>
</tr>
</tbody>
</table>

Table 3-5 Timing/Area for 5-to-1 MUX (XC4005APC84-5)

Setting Timing Constraints

The XACT-Performance tool is part of the XACTstep Development System. XACT-Performance allows you to specify precise timing requirements for your design. You can also specify the maximum allowable delay on any given set of paths in your design. To specify a set of paths, you must identify a group of start and end points. The start and end points can be flip-flops, I/O pads, IOB latches, or RAMs. You can control worse-case timing on the set of paths by specifying a single delay requirement for all paths in the set.

You can specify XACT-Performance timing constraints in the following ways:

- Set timing constraints in the synthesis tool (FPGA Compiler only). The synthesis tool passes the constraints to the XNF file.
- Specify default timing constraints on the PPR command line.
- Specify timing constraints for groups of logic in a constraints file.

Timing constraints specified on the command line have precedence over constraints specified in a constraints file. Timing constraints specified in a constraints file have precedence over constraints in the XNF file created by the synthesis tool.

Using the Synthesis Tool

Note: To set timing constraints in the synthesis tool, you must be using the FPGA Compiler.
Use the following Synopsys commands to create timing specifications for your designs.

create_clock
g.set_input_delay
g.set_output_delay
g.set_max_delay
g.max_period

These commands allow you to specify optimization goals and provide timing information for the Synopsys tools. These constraints are passed to PPR through XACT-Performance specifications that are written to the netlist.

XACT-Performance timing specifications generated by Synopsys tools may create a large number of constraints. Too many constraints can increase PPR run time. To decrease the number of constraints, use the following command from the FPGA Compiler:

xnfout_constraints_per_endpoint=5

You can also decrease the number of constraints by specifying timing constraints for groups of logic in a constraints file (described below).

**Note:** Refer to the Synopsys documentation and the *Synopsys (XSI)—FPGAs Interface/Tutorial Guide* for more information on setting XACT-Performance constraints from the synthesis tool.

**Using PPR Command Line Options**

You can specify default timing constraints on the PPR command line. These command line options do not give you as much control as the synthesis tool or the constraints file. For maximum flexibility, use a constraints file to specify timing constraints; however, for a simple one clock design, you can set basic timing requirements on the PPR command line. The default clock-to-setup, clock-to-pad, pad-to-setup, and pad-to-pad constraints can be set using the PPR options: Dc2s, Dc2p, Dp2s, and Dp2p.

**Note:** Refer to the *Development System Reference Guide* for more PPR command line options.
Using A Constraints File

XACT-Performance timing specifications generated by Synopsys tools may create a large number of constraints. To decrease the number of constraints, specify timing constraints for groups of logic in a constraints file.

Create timing requirements by specifying a set of paths and the maximum allowable delay on these paths. Specify the set of paths by grouping start and end points in one of the following ways.

- Refer to a predefined group by specifying one of the corresponding keywords — FFS, PADS, LATCHES, or RAMS.
- Create your own groups within a predefined group by tagging symbols with TNM (pronounced tee-name) attributes.
- Create groups that are combinations of existing groups using TIMEGRPs.
- Create groups by pattern matching on signal names.

Note: Although you can use end point specifications (using groups) in the same design with existing path-type specifications, Xilinx does not recommend combining the two methods.

A TNM (timing name) is a flag that you can use to group specific pads, latches, RAMS, or flip-flops. Symbols tagged with the same TNM identifier are considered a group. When schematic entry tools are used to create designs, TNMs are added to the schematic. Because synthesis tools (such as the FPGA Compiler) do not contain primitives and symbols for adding timing groups to HDL code, you cannot add TNMs to your HDL designs.

Note: Refer to the XACT-Performance chapter in the Development System Reference Guide for more information on using TNMs.

Using TIMESPEC and TIMEGRP Commands

Note: TIMESPEC and TIMEGRP primitives and flags cannot be used in HDL code. Use the AddTNM and MakeTNM programs, which are described in a subsequent section of this chapter.

To specify timing requirements for groups of logic, use the following procedure.

1. Define the groups in a TNM control file.
2. Specify the requirements with the TIMESPEC or TIMEGRP command in a PPR constraints file.

3. Use the TIMESPEC command to serve as a placeholder for the TS attribute timing specifications as follows:

   TIMESPEC = "<timespec-parameter>";

   The spaces in the command syntax are optional; the double quotes around the parameter text are required. TS attributes must be defined in a TIMESPEC command. These attributes begin with the letters “TS” and end with a unique identifier that can consist of letters, numbers, or the underscore character (_).

4. Use the TIMEGRP command to create groups that are combinations of existing groups as follows:

   TIMEGRP = "<timegrp-parameter>";

   The spaces in the command syntax are optional; the double quotes around the parameter text are required.

   **Note:** Refer to the XACT-Performance chapter in the Development System Reference Guide for more information on these parameters.

### Using TIMESPEC and TIMEGRP Constraints File Statements

This section includes examples of TIMESPEC and TIMEGRP constraints file statements.

Use the following statement to group pads with names that begin with “updata” into a group called “updata_io”. The asterisk (*) denotes a wildcard.

   TIMEGRP = "updata_io=PADS (updata<*>)";

Use the following statement to specify a delay of 50 ns for all paths from the group “LATCHES” to the group “updata_io”. Since the predefined group “LATCHES” contains all the latches in the design, this statement specifies that all paths from all latches to all pads defined by the group “updata_io” are constrained to 50 ns.

   TIMESPEC = “TS04=FROM:LATCHES:TO:updata_io=50ns”;
Additional examples of TIMESPEC and TIMEGRP statements are as follows:

\[ \text{TIMEGRP} = \text{"CLK40\_POS=RISING:CLK40"}; \]
\[ \text{TIMEGRP} = \text{"CLK40\_NEG=FALLING:CLK40"}; \]
\[ \text{TIMESPEC} = \text{"TS02=FROM:CLK40\_POS:TO:CLK40\_POS=20MHz"}; \]
\[ \text{TIMESPEC} = \text{"TS03=FROM:CLK40\_POS:TO:CLK40\_NEG=40MHz"}; \]

**Using MakeTNM and AddTNM**

The MakeTNM and AddTNM programs allow you to use TNMs. Use these programs to create a control file for defining the timing groups and to add the appropriate primitives and flags to your design file.

MakeTNM and AddTNM were created with Perl V4.0 (programs are compatible with Perl V5.0). You must have Perl loaded on your system to run these programs. Perl is usually located at `/usr/local/bin` on your system. If Perl is not at this location, you must modify the first line of each program to point to the Perl executable. Alternatively, you can invoke Perl directly by executing the programs with the keyword “perl”, as shown in the following example.

```
perl addtnm arguments
```

**Adding TNMs**

Figure 3-29 shows the design flow for adding TNMs using MakeTNM and AddTNM.
To add TNMs, use the following procedure.

1. To prevent the FPGA Compiler from writing timing specifications to the SXNF file, enter the following command at the DC shell or Design Analyzer prompt:

   \texttt{xnfout\_constraints\_per\_endpoint = 0}

2. Create an XFF (flattened XNF file) file by running Syn2XNF on the output from the compiler.

3. Run MakeTNM on the XFF file to create a template control file, \texttt{design.tt}, by entering one of the following commands:

   \texttt{maketnm design.xff}

   or

   \texttt{perl maketnm design.xff}
The template control file lists all flip-flops, RAMs, I/O pads, and latches in your design by instance name. X-BLOX modules that expand to include flip-flops are also listed; they are prefaced by the FFS keyword. Each line of the file contains one symbol. If the symbols have similar names, a wildcard character is used. A template control file is shown in Figure 3-30.
HDL Synthesis for FPGAs

H

flip-flops

H

FFS $\overline{q}11/\overline{q}0$
FFS $\overline{q}11/q1$
FFS $\overline{q}11/q2$
FFS $\overline{q}11/q3$
FFS $\overline{q}11/q4$
FFS $\overline{q}11/q5$
FFS $\overline{q}11/q6$
FFS $\overline{q}11/q7$
FFS $\overline{q}11/0/\overline{q}0$
FFS $\overline{q}11/0/q1$
FFS $\overline{q}11/0/q2$
FFS $\overline{q}11/0/q3$
FFS $\overline{q}11/0/q4$
FFS $\overline{q}11/0/q5$
FFS $\overline{q}11/0/q6$
FFS $\overline{q}11/0/q7$
FFS $\overline{q}11/0/q8$
FFS $\overline{q}11/0/q9$
FFS $\overline{q}11/0/q10$
FFS $\overline{q}11/0/q11$
FFS $\overline{q}11/0/q12$
FFS $\overline{q}11/0/q13$
FFS $\overline{q}11/0/q14$
FFS $\overline{q}11/0/q15$
FFS $\overline{q}11/0/q16$
FFS $\overline{q}11/0/q17$
FFS $\overline{q}11/0/q18$
FFS $\overline{q}11/0/q19$
FFS $\overline{q}11/0/q20$
FFS $\overline{q}11/0/q21$
FFS $\overline{q}11/0/q22$
FFS $\overline{q}11/0/q23$
FFS $\overline{q}11/0/q24$
FFS $\overline{q}11/0/q25$
FFS $\overline{q}11/0/q26$
FFS $\overline{q}11/0/q27$
FFS $\overline{q}11/0/q28$
FFS $\overline{q}11/0/q29$
FFS $\overline{q}11/0/q30$
FFS $\overline{q}11/0/q31$
FFS $\overline{q}11/0/q32$
FFS $\overline{q}11/0/q33$
FFS $\overline{q}11/0/q34$
FFS $\overline{q}11/0/q35$
FFS $\overline{q}11/0/q36$
FFS $\overline{q}11/0/q37$

3-60

Xilinx Development System
HDL Coding for FPGAs

RAM $1I23$

# I/O pads

PADS $1I173$
PADS $1I177$
PADS $1I194$
PADS $1I196$
PADS A.SHIFT_OUT_PAD0
PADS A.SHIFT_OUT_PAD1
PADS A.SHIFT_OUT_PAD2
PADS A.SHIFT_OUT_PAD3
PADS A.SHIFT_OUT_PAD4
PADS A.SHIFT_OUT_PAD5
PADS A.SHIFT_OUT_PAD6
PADS A.SHIFT_OUT_PAD7
PADS CLOCK_A_PAD0
PADS CLOCK_B_PAD0
PADS IN_A_PAD0
PADS IN_A_PAD1
PADS IN_A_PAD2
PADS IN_A_PAD3
PADS IN_A_PAD4
PADS IN_A_PAD5
PADS IN_A_PAD6
PADS IN_A_PAD7
PADS IN_B_PAD0
PADS IN_B_PAD1
PADS IN_B_PAD10
PADS IN_B_PAD11
PADS IN_B_PAD12
PADS IN_B_PAD13
PADS IN_B_PAD14
PADS IN_B_PAD15
PADS IN_B_PAD2
PADS IN_B_PAD3
PADS IN_B_PAD4
PADS IN_B_PAD5
PADS IN_B_PAD6
PADS IN_B_PAD7
PADS IN_B_PAD8
PADS IN_B_PAD9
PADS IND_PAD0
PADS IND_PAD1
PADS IND_PAD2
PADS IND_PAD3
PADS IND_PAD4
PADS IND_PAD5
PADS IND_PAD6
PADS IND_PAD7
PADS IN_A0R0
PADS IN_A0R1
PADS IN_A0R2
PADS IN_A0R3
4. Modify the design.tt file to create a TNM file that includes timing groups.

To create the TNM file, add the desired group names to the appropriate instances. You can use wildcard characters in instance names to simplify the file. Use the asterisk ("*") to represent an arbitrary string and a question mark to represent a single character. Statements without group names are ignored by AddTNM.

5. After modifying the template file, change the design.tt file name to design.tnm.

This file is the input file to AddTNM. A TNM file created from the template control file in Figure 3-30 is shown in Figure 3-31. The instances that were not tagged with a TNM have been deleted. You can retain the untagged instances in the file, however, they are ignored by AddTNM.
6. Run AddTNM on `design.tnm` as follows:

```bash
addtnm design.tnm
```

or

```bash
perl addtnm design.tnm
```

AddTNM adds the timing group information from `design.tnm` to the flattened XFF file. AddTNM includes a debug mode that allows you to examine the group name assigned to an instance. To run AddTNM with this verbose output, use the following command:

```bash
addtnm -d design
```

or

```bash
perl addtnm -d design
```

The output from AddTNM is written to a file with a `.txff` extension to prevent overwriting the source XFF file.

---

**Figure 3-31 TNM File**

```plaintext
#include <std_logic_1164.h>

entity shift is
  port (data_in : in std_logic_vector(17 downto 0);
         clock : in std_logic;
         data_out : out std_logic_vector(17 downto 0));
end shift;

architecture Behavioral of shift is
  signal data_reg : std_logic_vector(17 downto 0);
begin
  data_reg <= data_reg << 1 when clock'event and clock = '1' else data_in;
  data_out <= data_reg;
end Behavioral;
```

---

HDL Coding for FPGAs

HDL Synthesis for FPGAs 3-63
7. Immediately after running AddTNM, change the name of design.txff to design.xff.

8. After the timing groups are created, use the TIMESPEC and TIMEGRP commands to specify timing constraints.
   
   Place these commands in a PPR constraints file with a .cst extension. The constraints file contains the actual timing constraints using the group names defined in the TNM file. The constraints file is read by XNFPrep and PPR. Figure 3-32 shows a constraints file.

   **Note:** Refer to the Development System Reference Guide for PPR constraints file syntax.

   ```
   TIMESPEC="TSS01\=FROM\=a:TO\=b;acc_a=25\ ns"
   TIMESPEC="TSS02\=FROM\=RAMS\=TO\=count\_b=10\ ns"
   TIMESPEC="TSS03\=FROM\=sh\_a\=TO\=a\_pads=13\ns"
   TIMESPEC="TSS04\=FROM\=count\_b\=TO\=b\_pads=40\ns"
   ```

   **Figure 3-32 Constraints File**

9. Run XNFPrep on the design.xff file to create an XTF or XTG (if design has X-BLOX modules) file:

   ```
   xnfprep design.xff
   ```

   XNFPrep reads the constraints file and the XFF file with the TNMs included and writes an XTF (or XTG) file with timing information.

10. Run PPR on the XTF (or XTG) file to create a Logic Cell Array (LCA) file. The timing specifications are include in a PPR constraints file. Run PPR as follows:

   ```
   ppr design.xtf cstfile=design.cst
   ```

   **Note:** The constraints file can have any name.

11. Look at the report file created by PPR.

   This file contains information on the XACT-Performance specifications and indicates if the timing constraints were met.

12. Run XDelay to obtain accurate timing information. Figure 3-33 shows a section of an XDelay report file.
Creating A TNM Control File Without Using MakeTNM

If you know the names of the registers, pads, RAMs, or latches that you want to group, you can create your own TNM file, instead of generating one with MakeTNM.

The TNM control file should have the same prefix as the XFF file (output from XNFMerge) with a .tnm extension. The syntax for creating your own TNM file is as follows:

```plaintext
endpoint-type instance-name : group [ group2 ...]
```

An example of a TNM control file is shown in Figure 3-34.

---

**Figure 3-33 XDelay Report File**
# example design.tnm file

FFS REG_3A/* : reg_3a_group
FFS REG_3?/Q0 : bit0_group
FFS REG_3C/Q7: 3c7_group other_group
PADS ENA : enable_input
SIG CLOCK40 : mainclk

**Figure 3-34 TNM Control File**

Endpoint-type is either FFS, RAMS, or PADS. IOB latches and X-BLOX modules are grouped together with the FFS type by AddTNM.

Use the instance-name variable to specify the names of the symbols that you want to group. You can use the wildcards “?” (single character) and “*” (multiple character) to specify multiple symbols. The symbols must be the same type specified by the endpoint-type variable.

Use the group variable to specify the name of the group for the selected instance(s). You can use this name as endpoints in the From-To TIMESPEC statements or as a building block in TIMEGRP statements (both can be specified via the constraints file). You can assign more than one name to an instance and you should separate group names by spaces or tabs.

Follow these rules when creating your own control file.

- Use a pound (#) character for comments
- Do not include comments on the same line as a statement
- Do not use a semicolon at the end of a statement
- Text in the file is not case-sensitive

**Adding TNMs to Signals**

AddTNM allows you to place TNMs on signals. Because synthesis tools usually generate random signal names, the ability to add TNMs to signals is not always advantageous. However, in some cases you may want to add a TNM to a device input signal. For example, if your design has an external clock signal, you can place a TNM on this signal to allow AddTNM to group the flip-flops that this signal
controls. Adding a TNM is an easy way to group flip-flops that have a common clock or enable signal.

**Note:** Refer to the XACT-Performance chapter in the *Development System Reference Guide* for more information on adding TNMs.

Add a TNM to a signal by adding the following line to your TNM control file:

```
SIG signal-name  :  group [ group2 ... ]
```

Specify the external signal you want to add the TNM to with the `signal-name` variable. You can use “*” and “?” as wildcards. Use the `group` variable to specify the name of the group for the selected signal.
Chapter 4

Floorplanning Your Design

Xilinx gives you two design implementation options. The XACTstep Foundry v7 software provides automatic implementation of your designs and does not support the Xilinx Floorplanner. Alternatively, if you want to control part of the implementation process, use the XACTstep v5.2 software with the Floorplanner.

For high-density devices, Xilinx recommends that you floorplan specific parts of your design to improve PPR performance. Due to the complexity and size of larger designs, PPR is limited in its ability to recognize structure. Your design may not route or meet timing constraints without structured placement. Based on your knowledge of a design’s structure, you can create a floorplan that significantly improves the placement of the design. Generally, you should floorplan the parts of your designs that are timing critical or heavily congested.

This chapter provides examples of HDL designs that facilitate floorplanning. This chapter does not provide a complete description of the Xilinx Floorplanner tool. Refer to the Floorplanner Reference/User Guide, High-Density Design Guide application note, and the Floorplanner online tutorial for complete information on using the Floorplanner.

The following topics are included in this chapter:

- Creating MAP files
- Using the Floorplanner tool
- Floorplanning RPMs, RAMs/ROMs, and BUFTs
- Floorplanning hierarchical and flat designs
- Floorplanning to reduce routing congestion
Using the Floorplanner

This section describes creating a MAP file, using the Floorplanner, and selecting design elements for floorplanning.

Creating a MAP File

The Floorplanner requires a MAP file as input. Before creating a MAP file, you must synthesize your design and save it as an SXNF or an EDIF file. Use one of the following procedures to create a MAP file.

Using XMake

Specify a MAP file as the target file with XMake as follows:

```
xmake [options] design design.map
```

Using PPR

1. Follow the XC4000 design flow (includes Syn2XNF, XNFPrep, X-BLOX) to create an XTF file.
2. Create a MAP file at the command line with the following PPR options:

```
ppr design.xtf map_fgs=true place=false route=false report=false lca=false run_pic2map=true
```

Using Prep for Floorplanner Command

Use the Prep for Floorplanner option in the XACT Design Manager (XDM) to create a MAP file.
1. Invoke XDM.
2. Choose the XMake command from the Translate menu.
3. Select the -X option (Use XNF files only) from the XMake pop-up menu and press Done.
4. Select the appropriate XNF file from the pop-up menu.
5. Select Prep for Floorplanner from the pop-up menu.

XMake generates a MAP file.
Overview of Floorplanner Windows

The Floorplanner has three windows; Task, Design, and Floorplan. Each window is described below.

Task Window

The Task window is shown in Figure 4-1. This window appears at the top of your screen and is the initial Floorplanner screen. The File and Help menus allow you to open a file for the Floorplanner, exit the application, and open the online help.

Design Window

The Design window and Floorplan window are shown in Figure 4-2.
The Floorplanner generates a hierarchical representation of the design from the MAP file input. The Design window displays a fully expandable and annotated hierarchy. The design hierarchy represents the mapped design hierarchy that is created by the synthesis tool from your HDL design. A section of the Design window is shown in Figure 4-3.

The various colors in the hierarchy display distinguish the macros, which are annotated with the instance name from the MAP file. The black hierarchy structure lines indicate the hierarchical level of each macro. Each macro has a gray box with a minus sign, “–”, or a plus sign, “+”. The “–” indicates that the macro is expanded. The “+” indicates that the macro is collapsed. The icon next to the gray box represents the type of macro. For example, RAMs are represented by overlapping squares, as shown in Figure 4-3.

![Figure 4-3 Section of the Design Window](image)

**Floorplan Window**

The upper left corner of the Floorplan window is shown in Figure 4-4. The Floorplan window displays the die for a selected part type, such as XC4005PC84. This window is a scrollable, scalable view of a resource map of the device that is specified in the design. You place the selected logic from the Design window into this window.
Deciding What Elements to Floorplan

To obtain optimal design performance, floorplan the following structured items.

- Large objects such as RPMs, registers, counters, and RAMs
- Buses (place all BUFTs and bus elements)
- BUFTs with I/O or RPM inputs
- Multiple BUFTs (except VCC or GND) with identical source pin inputs

You can floorplan elements other than those listed, but constraining too many elements, especially those without any specific structure, can decrease design performance.

Note: Generally, you do not have to floorplan state machines because they are efficiently placed and routed by PPR.
Xilinx recommends that you create hierarchical designs because the various design modules are hierarchically displayed in the Floorplanner, making it easier to identify each module. If your design is not hierarchical, the logic is displayed as a large, flat group that is difficult to floorplan. For information on building hierarchical designs, see the “Comparing Hierarchical and Flat Designs” section in this chapter and the “Building Design Hierarchy” chapter.

You should label all symbols, nets, processes, procedures, functions, and blocks in your HDL code. However, because synthesis tools optimize the logic in your code, not all component names are preserved in the MAP file. Component names that are preserved include those for registers, I/Os, and instantiated cells. Labeling design components makes it easier to floorplan your design because you can identify the individual components in the Floorplanner.

Running the Floorplanner and Opening a File

To run the Floorplanner and open a file, use one of the following procedures.

Using the Command Line

1. At the command line, type:

   `fplan design.map &`

   Use the “&” character to run the Floorplanner in the background.

2. The Floorplanner reads the MAP file, loads the correct device (part type), opens the Design window, and opens the Floorplan window with the appropriate FPGA die.

Using the Floorplanner Task Window

1. At the command line, type:

   `fplan &`

   Use the “&” character to run the Floorplanner in the background. The Task window appears at the top of your screen as shown in Figure 4-5.
2. Select File ➝ Open.

The File Open dialog box appears, as shown in Figure 4-6.

3. Select the MAP file in the Files field and double-click to open it.

Figure 4-5 Task Window

Figure 4-6 File Open Dialog Box
4. The Floorplanner reads the MAP file, loads the correct device (part type), opens the Design window, and opens the Floorplan window with the appropriate FPGA die.

**Setting Boundaries in the Floorplan Window**

Use the following procedure to define an area in the Floorplan window in which to place selected logic. See Figure 4-7 for an example of setting boundaries in the Floorplan window.

1. From the design hierarchy, select the logic that you want to place in the Floorplan window.

2. Move the pointer to the Floorplan window and click on the Allocate Area toolbar button.

   After clicking on the Allocate Area toolbar button, the pointer changes to a large plus sign (+).

3. Move the pointer to the Floorplan window, then press and hold the left mouse button, dragging out a rectangular area.

4. Place the selected logic into the newly created boundary by releasing the left mouse button.

5. Select the Check Floorplan command from the process menu.

   If you have allocated an area in the floorplan large enough to accommodate the logic, a dialog box appears indicating that the floorplan passes all basic placement checks.

   If you have not defined an area large enough, the Check Floorplan Warnings dialog box appears and indicates that more logic resources are needed in the boundary to accommodate the selected logic.
Floorplanning Your Design

Figure 4-7 Floorplanning Modules into Areas

Floorplanning RPMs, RAMs, and ROMs

**Note:** RPMs replace all Xilinx-supplied hard macros. Do not use pre-Unified Libraries hard macros in new designs.

X-BLOX modules are usually inferred by the synthesis tool. Arithmetic X-BLOX modules use RPMs to take advantage of fast-carry logic. In addition to arithmetic functions, other X-BLOX modules can be instantiated in your HDL code. You can instantiate 16 x 1 and 32 x 1 RAMs/ROMs from the Xilinx Synopsys Interface (XSI) primitive libraries. You can also implement any other
RAM/ROM size using the MemGen program, which is included in the XACTstep Development System. You can also behaviorally describe ROMs in your code.

**Warning:** Do not behaviorally describe RAMs in your HDL code because compiling creates combinatorial loops.

For more information on RAMs/ROMs, refer to the Synopsys (XSI)—FPGAs Interface/Tutorial Guide. To obtain optimal design performance, use the Floorplanner to place all RPMs, RAMs, and ROMs.

The RPM icon appears in the Design window as three adjacent squares in an “L” shape, as shown in Figure 4-8.

![Figure 4-8 RPM Icon](image)

**Figure 4-8 RPM Icon**

RAMs are grouped according to hierarchy and are represented in the Design window by three overlapping squares, as shown in Figure 4-9. The Floorplanner usually places the related RAMs/ROMs together in one group. To improve the design timing, place the RAMs/ROMs according to width and depth, depending on the number of address lines, data lines, and the desired module shape.

![Figure 4-9 RAM, ROM, or non-RPM Counter Icon](image)

**Figure 4-9 RAM, ROM, or non-RPM Counter Icon**

**RPM and RAM/ROM Example**

**Note:** Before completing the following steps, make sure you have retrieved the necessary design files from the Xilinx Internet Site or the Xilinx Technical Bulletin Board as described in the “Getting Started” chapter of this manual.

To floorplan an RPM, perform the following steps.

1. Run the Floorplanner and select **File → Open** as described in the “Using the Floorplanner” section.
2. Go to the rpm_ram directory.

3. Select the rpm_ram.map file in the Files field and double-click on this file to open it.

   The Floorplanner reads the MAP file, loads the correct device, opens the Design window, and opens the Floorplan window with the correct FPGA die. The design is displayed in the Design window, as shown in Figure 4-10.

4. Click on the Expand button (“+” sign) for the RAM block pointed to by the arrow in Figure 4-10.

   The next level of hierarchy is displayed, as shown in Figure 4-11.

5. Click on the Collapse button (“–” sign) to collapse the RAM block and return to the level of hierarchy shown in Figure 4-10.

6. To floorplan the RAM block, click on the RAM block icon in the Design window.

   The RAM icon changes to a ghost image that moves with the mouse.
7. Move the cursor to the Floorplan window.
8. Place the RAM by clicking in the area shown in Figure 4-12.

![Figure 4-12 Floorplanned RAM](image)

9. To floorplan the RPM block, click on the RPM block icon pointed to by the arrow in Figure 4-10.
   The RPM icon changes to a ghost image that moves with the mouse.
10. Move the cursor to the Floorplan window.
11. Place the RPM by clicking in the area shown in Figure 4-13.
Floorplanning Tristate Buffers

Designs with large multiplexers and bidirectional buses can be difficult to route. You can implement these multiplexers and buses with internal tristate buffers (BUFTs) to improve the routability of the design and conserve CLB resources. BUFTs are aligned with each CLB and the IOBs on the left and right edges of the chip, as shown in a section of the Floorplan window in Figure 4-14.
BUFT Example

The BUFT design (buft_ex) in Figure 4-15 is a simple behavioral description of an internal tristate bus. This design has two external buses, DATAIN0 and DATAIN1, that are multiplexed to the DATAOUT bus. The ADD3_STATE process places DATAIN0 on the bus when the SEL signal is low. The ADD3_STATE2 process places DATAIN1 on the bus when SEL is high. The tristate bus is then registered and placed on the DATAOUT output.
Floorplanning Your Design

-- BUFT_EX.VHD
-- Xilinx HDL Synthesis Design Guide for FPGAs
-- This is an example that show how to infer tri-state buffers.
-- June 1995

library IEEE;
use IEEE.std_logic_1164.all;

entity buft_ex is
  port (DATAIN0 : in STD_LOGIC_VECTOR(3 downto 0);
        DATAIN1 : in STD_LOGIC_VECTOR(3 downto 0);
        SEL : in STD_LOGIC;
        CLK : in STD_LOGIC;
        DATAOUT : out STD_LOGIC_VECTOR(3 downto 0));
end buft_ex;

architecture BUFT_BEHAV of buft_ex is
  signal mainline : STD_LOGIC_VECTOR(3 downto 0);
begin
  ADD_3STATE: process (SEL, DATAIN0)
    begin
      if (SEL = '0') then
        mainline <= DATAIN0;
      else
        mainline <= "ZZZ";
      end if;
    end process ADD_3STATE;

  ADD_3STATE2: process (SEL, DATAIN1)
    begin
      if (SEL = '1') then
        mainline <= DATAIN1;
      else
        mainline <= "ZZZ";
      end if;
    end process ADD_3STATE2;

  ADD_REG: process
    begin
      wait until CLK'event and CLK = '1';
      DATAOUT <= mainline;
    end process ADD_REG;
end BUFT_BEHAV;

Figure 4-15 VHDL Inference of Tristate Buffers
Floorplanning BUFT Example

Note: Before you perform the following steps, make sure you have retrieved the necessary design files from the Xilinx Internet Site or the Xilinx Technical Bulletin Board as described in the “Getting Started” chapter of this manual.

To floorplan the tristate buffers, follow these steps.

1. Run the Floorplanner and Select File → Open as described in the “Using the Floorplanner” section.
2. Go to the bufts directory.
3. Find the buft.map file in the Files field and double-click on this file to open it.

The Floorplanner reads the MAP file, loads the correct device (part type), opens the Design window with a hierarchical design, and opens the Floorplan window with the correct FPGA die. A section of the Design window is shown in Figure 4-16.

The BUFT design contains 1 FG, 13 IOBs, 8 BUFTs, and 1 BUFG. The 8 BUFTs are located in the “U” macro.

Note: Because the Synopsys compiler generates symbol names, these names do not always correspond to the names in your design and may change with each run of the synthesis tool. However, you can use the net names to help you correctly floorplan the BUFTs because the Floorplanner net names correspond to the names in your HDL code.

Figure 4-16 Section of the Design Window
4. Click on the Expand button ("+") sign to expand the U block and display the next level of hierarchy.

BUFT symbols and names (U75 to U82) are displayed, as shown in Figure 4-17.

**Note:** The BUFT and BUFGS labels may not exactly match the labels shown in Figure 4-17.

![Figure 4-17 Expanded Stack of BUFTs](image)

5. To view the device resources in the Floorplan window, click on the Resources button in the Toolbar or select **View** → **Resources**.

6. If you select **View** → **Resources**, the Resources dialog box is displayed, as shown in Figure 4-18.

To display the device resources, select each field in the Resource Graphics area of the dialog box. Click OK to display the available device resources in the Floorplan window.
HDL Synthesis for FPGAs

7. Click on the U82 BUFT icon in the Design window.

   The BUFT icon changes to a ghost image that moves with the mouse.

8. Move the cursor to the Floorplan window.

9. Place U82 in the top tristate buffer resource in Row 1, Column 1 by clicking in that area.

10. Continue placing BUFTs U81, U79, and U77 in the same column directly beneath U82, as shown in Figure 4-19.

Note: Xilinx recommends that you order the bits with the MSB at the top and the LSB at the bottom because RPMs with carry logic follow this convention.

Figure 4-18 Resources Dialog Box
11. Place U75 in Row 1, Column 2.

12. Place the remaining BUFTs in the same column directly beneath U75, as shown in Figure 4-20.

BUFT outputs are driven onto horizontal longlines. There are special fast connections from the horizontal longlines to the IOB output pins on both vertical edges of the device. Therefore, locate I/Os that connect to horizontal longlines, such as bused I/Os, on the left or the right side near the fast connection. PPR may not be able to completely route your design if you randomly lock the pins at the beginning of the design entry process or lock I/Os that connect to bidirectional buses to the top or bottom of the device.
Comparing Hierarchical and Flat Designs

Hierarchical designs are easier to floorplan because the various design modules are hierarchically displayed in the Floorplanner. You can easily identify and place each module. If your design is not hierarchical, the logic is displayed as a large, flat group that is difficult to identify for floorplanning. Structured logic, such as RPMs and RAMs, are easy to identify in the Design window for floorplanning. This section of the manual compares floorplanning the same design (Alarm design) using the following design methodologies.

- Design is compiled as one flat module without X-BLOX DesignWare modules
- Design is compiled as one flat module using X-BLOX DesignWare modules
- Design is compiled using the design’s original hierarchy without X-BLOX DesignWare modules
Design is compiled using the design's original hierarchy with X-BLOX DesignWare modules.

The Alarm design example is a digital display alarm clock consisting of six blocks, as shown in Figure 4-21. There are three levels of hierarchy in this design, as shown in Figure 4-22.
The Alarm VHDL design is shown in Figure 4-23. The arrows point to the instantiation of the six blocks in the code. You should use distinct labels that are easy to recognize when instantiating blocks of logic in your code because these labels are used in the Floorplanner Design window to distinguish the levels of hierarchy. In the Alarm design, the labels U1-U6 are used for the six logic blocks.
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity alarm is
  port TOGGLE,SWITCH,SET,TIME,
       ALARM_HRS,MINS,CLK   : in STD_LOGIC;
       SPEAKER_OUT           : out STD_LOGIC;
       ALARM_DISP            : out STD_LOGIC;
       DISP1,DISP2           : out STD_LOGIC_VECTOR(13 downto 0));
end;

architecture BEHAVIOR of alarm is
  component time_block
    port SET,TIME,HRS,MINS,CLK   : in STD_LOGIC;
    connect    : buffer INTEGER range 1 to 12;
    connect0   : buffer INTEGER range 0 to 59;
    connect8   : buffer STD_LOGIC;
  end component;

  component alarm_block
    port ALARM,HRS,MINS,CLK   : in STD_LOGIC;
    connect9   : buffer INTEGER range 1 to 12;
    connect10  : buffer INTEGER range 0 to 59;
    connect11  : buffer STD_LOGIC;
  end component;

  component convertor_clk
    port CONNECT13  : in UNSIGNED(9 downto 0);
    DISP1,DISP2    : out STD_LOGIC_VECTOR(13 downto 0));
  end component;

  component comparator
    port ALARM_HRS,CLOCK_HRS : in INTEGER range 1 to 12;
    ALARM_MINS,CLOCK_MINS : in INTEGER range 0 to 59;
    ALARM_AMPM,CLOCK_AMPM : in STD_LOGIC;
    RINGER           : out STD_LOGIC;
  end component;

  component alarm_sn
    port COMPARE_IN,TOGGLE_ON : in STD_LOGIC;
    CLOCK           : in STD_LOGIC;
    RING            : out STD_LOGIC;
  end component;

  component aux
    port ALARM_HRS  : in INTEGER range 1 to 12;
    ALARM_MINS    : in INTEGER range 0 to 59;
    ALARM_AMPM    : in STD_LOGIC;
    TIME_HRS      : in INTEGER range 1 to 12;
    TIME_MINS     : in INTEGER range 0 to 59;
    TIME_AMPM     : in STD_LOGIC;
    ALARM_SET     : in STD_LOGIC;
    OUTBUS        : out UNSIGNED(10 downto 0));
  end component;
Method 1: Compiling Flat without X-BLOX

The Alarm design is initially compiled as one flat module without using X-BLOX DesignWare modules. The hierarchical representation of the design in the Design window is shown in Figure 4-24. The logic is separated into the three blocks U, U1 and U2. Most of the logic is contained in the U block. The U1 and U2 blocks contain only flip-flops. The U block is expanded by clicking on the Expand button ("+" sign) and the next level of hierarchy is displayed, as shown in Figure 4-25.

--- Top level nets that connect major modules

```vhdl
signal KONNECT7,KONNECT10 : INTEGER range 0 to 59;
signal KONNECT8,KONNECT9,KONNECT12 : STD_LOGIC;
signal KONNECT6,KONNECT10 : UNSIGNED(10 downto 0);

begin
  AMP_DISPLAY <= KONNECT13(0);
  U1: time_block port map( SET_TIME, HRS, MINS, CLK, KONNECT8, KONNECT7, KONNECT13);
  U2: alarm_block port map( ALARM, HRS, MINS, CLK, KONNECT5, KONNECT10, KONNECT11);
  U3: converter_ckt port map( KONNECT13_10_downto_1, DISP1, DISP2);
  U4: comparator port map( KONNECT7, KONNECT10, KONNECT11, KONNECT7, KONNECT11, KONNECT12);
  U4: alarm_sm1 port map( KONNECT12, TOGGLE_SWITCH, CLK, SPEAKER_OUT);
  U6:mux port map( KONNECT7, KONNECT10, KONNECT11, KONNECT7, KONNECT8, ALARM, KONNECT13);

end;
```

Figure 4-23 Alarm Design Compiled Flat without X-BLOX
The U block contains 160 4-input function generators, 38 FGH function generators, 1 flip-flop, and 1 BUFGS. Since the function generator names do not provide hierarchical information, the logic cannot be identified and placed in the Floorplan window. Flattening large designs can reduce the number of CLBs or improve the design.
speed, however, these designs are difficult to floorplan. Method 1 should only be used for very simple designs.

**Method 2: Compiling Flat with X-BLOX**

Next, the Alarm design is compiled flat with X-BLOX DesignWare modules. This library contains arithmetic functions that are implemented with RPMs. The hierarchical representation of the design in the Design window is shown in Figure 4-26. Compared to the design created by Method 1, this design methodology produces a design that is easier to floorplan because the RPMs are structured logic that are easy to identify and place. However, since the original design hierarchy is not preserved, the remaining logic cannot be identified and placed in the Floorplan window. Method 2 should only be used for very simple designs.

![Figure 4-26 Alarm Design Compiled Flat with X-BLOX](image)

**Method 3: Compiling with Hierarchy and without X-BLOX**

The Alarm design is compiled with the original design hierarchy and without X-BLOX modules. The hierarchical representation of the design in the Design window is shown in Figure 4-27. Compared to Methods 1 and 2, this design methodology produces a design that can be floorplanned because the original design hierarchy is preserved in the Design window.
Method 4: Compiling with Hierarchy and X-BLOX

The Alarm design is compiled with the original design hierarchy and X-BLOX DesignWare modules. The hierarchical representation of the design in the Design window is shown in Figure 4-28. The dashed arrows point to the RPMs and the solid arrows point to the hierarchical blocks. This design methodology produces a design that is easier to floorplan than the three previous designs because it contains the original hierarchy as well as RPMs.
When creating your HDL designs, you should understand the architecture of the targeted device. You should know what device resources are available as well as how device limitations influence PPR results. This section describes how you can write your code to make floorplanning easier and improve PPR results. In addition, the examples in this section show how routing resources can become a limiting factor if you do not consider the device architecture when creating your designs.
Positioning and Aligning Buses

Position the major buses in your design first because they are usually the largest design components. Once you have positioned the buses, you can place the registers, counters, and other structured elements along the buses.

Aligning Structures Along Buses

During floorplanning you should consider the placement of resources within the FPGA. Good resource placement can reduce PPR processing time as well as improve the speed and routability of your design. The design example in Figure 4-29 consists of two 8-bit registers (labeled A[7:0] and B[7:0]), two 8-bit counters (labeled C[7:0] and D[7:0]), and an 8-bit bidirectional bus (labeled X[7:0]). The counters are multiplexed (using tristate buffers) to the data bus, which also connects to I/O pins.
HDL Synthesis for FPGAs

-- ALIGN_STR.vhd
-- Xilinx HDL Synthesis Guide for FPGAs
-- This example contains two 8-bit registers, two 8-bit counters, and a 8-bit
-- bidirectional bus. The counters are tri-states to the bidirectional bus.
-- align_str.vhd 5/24/95
--
library IEEE;
use IEEE.std_logic_1164.all;

entity align_str is
  port(X : in std_logic_vector(7 downto 0);
       CLK : in std_logic;
       CLR_C : in std_logic;
       CLR_D : in std_logic;
       EN_C : in std_logic;
       EN_D : in std_logic;
       MUX_SEL : in std_logic_vector(1 downto 0);
       SEL : in std_logic);
end entity;

architecture BEHAVIORAL of align_str is
  component count3
    port(CLK, CLEAR, ENABLE : in std_logic;
         COUNT : out std_logic_vector(7 downto 0));
  end component;

  signal A : std_logic_vector(7 downto 0);
  signal B : std_logic_vector(7 downto 0);
  signal C : std_logic_vector(7 downto 0);
  signal D : std_logic_vector(7 downto 0);
  signal K_I : std_logic_vector(7 downto 0);

  begin
    countC : count3 port map(CLK, CLR_C, EN_C, C);
    countB : count3 port map(CLK, CLR_D, EN_D, D);
    X <= X_I when (SEL = '0') else "ZZZZZZZZ";

    MUX_SEL_A : process (MUX_SEL, A)
    begin
      if (MUX_SEL = '00') then
        K_I <= A;
      else
        K_I <= "ZZZZZZZZ";
      end if;
    end process;

    MUX_SEL_B : process (MUX_SEL, B)
    begin
      if (MUX_SEL = '01') then
        K_I <= B;
      else
        K_I <= "ZZZZZZZZ";
      end if;
    end process;
end architecture;

Xilinx Development System
This design is placed, routed, and loaded into the Floorplanner. The placement results are displayed in the Floorplan window, as shown in Figure 4-30. The registers and RPMs are horizontally aligned across the device and share the same horizontal longlines. However, this placement does not conserve resources because the RPMs, registers, and I/Os are too widely dispersed in the chip.
Because this design is highly structured, you should floorplan the design first and then run PPR. The 8-bit counters and the X[7:0] I/O bus should be floorplanned, as shown in Figure 4-31.
After the design is floorplanned, PPR is run, and the design is loaded into the Floorplanner. The new placement is shown in Figure 4-32. PPR placed the logic that was not floorplanned around the counters and horizontally placed the bidirectional buses to align with the logic. The logic is contained in the upper left quadrant to conserve longline resources.

Figure 4-31 Floorplanned Constraints
Floorplanning RAMs to Reduce Routing Congestion

You can floorplan RAMs in vertical columns or in horizontal rows. For example, a 16 x 8 RAM can fit into four CLBs in a column since two 16 x 1 RAMs fit into one CLB. However, larger RAMs (over 32 words deep) require additional logic to form output multiplexers and write-enable strobes (decoders). For example, a 128 x 4 RAM requires four 32 x 4 RAM banks (DA[3:0]-DD[3:0]), as shown in Figure 4-33. In this case, each RAM bank consists of four 32 x 1 RAMs aligned vertically. Each RAM will occupy one CLB. The 4-to-1 output multiplexers (O[3:0]) can be efficiently implemented with CLBs since a 4-to-1 multiplexer fits in one CLB.

Deeper RAMs require larger output multiplexers. For example, a 256 x 4 RAM requires eight 32 x 4 RAM banks and four 8-to-1 multiplexers. Implementing these multiplexers with gates results in an inefficient use of CLBs and produces slower RAMs. Larger multiplexers are more efficiently implemented using BUFTs. It is important to organize the RAM banks so that the “n” bit appears in the same row for each bank. This allows efficient routing between the RAM CLBs, BUFTs, and horizontal longlines. Also, placing this
rectangular structure in any of the four quadrants of the device (to avoid straddling the device center lines) allows the half-longlines in the three other quadrants to be used for other signals.

![Figure 4-33 Rectangular Placement of RAM](image)

**Figure 4-33 Rectangular Placement of RAM**
Chapter 5

Building Design Hierarchy

Large HDL designs (more than 5,000 gates) for FPGAs are usually synthesized as either one flat module or as many small modules. Flat designs can be difficult to route if the logic is placed in one region of the device. Many small modules can increase the gate count, which can result in a design that does not fit the target device. Although these design methodologies are effective for implementing ASIC devices, usually they are not the most efficient strategies for implementing high-density FPGAs such as the XC4010, XC4013, and XC4025. Effective design partitioning provides the following benefits:

- Allows you to efficiently manage the design flow
- Reduces design time by allowing you to use existing design modules more than once
- Allows you to produce designs that are easy to understand

This chapter describes how you should partition your designs to improve synthesis results and reduce routing congestion. The same design is synthesized using three different design methodologies and a comparison of the results is provided.

Gate counts are not an accurate representation of FPGA device utilization because you cannot determine the number of gates mapped to one CLB. Therefore, the gate counts in this chapter are provided to give you only an estimate of CLB utilization. An approximate gate to CLB ratio is 800 gates per 100 CLBs.
Using the Synthesis Tool

By effectively partitioning your designs, you can significantly reduce compile time and improve synthesis results. This section provides recommendations for partitioning your designs.

- **Restrict Shared Resources to Same Hierarchy Level**
  Resources that can be shared should be on the same level of hierarchy. If these resources are not on the same level of hierarchy, the synthesis tool cannot determine if these resources should be shared.

- **Compile Multiple Instances Together**
  You may want to compile multiple occurrences of the same instance together to reduce the gate count. However, to increase design speed, do not compile a module in a critical path with other instances.

- **Restrict Related Combinatorial Logic to Same Hierarchy Level**
  Keep related combinatorial logic in the same hierarchical level to allow the synthesis tool to optimize an entire critical path in a single operation. Boolean optimization does not operate across hierarchical boundaries. Therefore, if a critical path is partitioned across boundaries, logic optimization is restricted. In addition, constraining modules is difficult if combinatorial logic is not restricted to the same level of hierarchy.

- **Separate Speed Critical Paths from Non-critical Paths**
  To achieve satisfactory synthesis results, locate design modules with different functions at different levels of the hierarchy. Design speed is the first priority of optimization algorithms. To achieve a design that efficiently utilizes device area, remove timing constraints from design modules.

- **Restrict Combinatorial Logic that Drives a Register to Same Hierarchy Level**
  To reduce the number of CLBs used, restrict combinatorial logic that drives a register to the same hierarchical block.
● Restrict Module Size

Restrict module size to 100 - 200 CLBs. This range varies based on your computer configuration; the time required to complete each optimization run; if the design is worked on by a design team; and the target FPGA routing resources.

Although smaller blocks give you more control, you may not always obtain the most efficient design. For XC4000 designs, do not exceed 5,000 gates for any module because designs with more than this number of gates may not route. For XC3000 and XC3100 devices, do not exceed 4,000 gates for each module. These gate counts are slightly smaller for I/O intensive designs, asynchronous designs, and designs that are difficult to route.

● 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.

● 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.

Modifying Design Hierarchy for PPR

Complex designs are easier to create with HDL than with traditional schematic entry methods. As you create larger designs, it is important that you add structure to your designs. Structured designs make the design process more manageable and are easier to route. For example, you can floorplan a large design that consists of medium-sized modules (approximately 5,000 gates each) into separate regions of the device. Adding structure to your design by
floorplanning allows PPR to divide the logic evenly throughout the device. Structure provides the following benefits:

- **Reduces Gate Count**

  Optimization reduces the number of gates by combining logic into function generators and combining similar functions. For example, if a design contains several small 4-bit incrementers, resource sharing occurs if these incrementers are in the same VHDL process. If they are not in the same process and are implemented in gates (using the Synopsys DesignWare Library), they are combined by the optimizer to further reduce the gate count.

- **Improves Routability**

  When you group modules and constrain them to a region of the device according to your design’s hierarchy and data flow, you are adding structural information to your design that PPR uses. Constraining design modules into different regions of the device evenly divides the logic and improves routability.

- **Reduces Routing Time**

  When you constrain modules to certain regions of the device, you reduce the routing time by specifying a smaller area for PPR to evaluate.

- **Reduces Time Required for Small Design Changes**

  You can easily modify the logic in a module without effecting other modules by using the PPR Guide option. Since designs usually require several changes, this is an important benefit of structured design.

- **Reduces Debugging Time**

  Debugging your design is easier because the various modules are isolated to specific regions of the device. In addition, module content and location are defined by you.

### Top Design Example

This section provides an example of a design that is implemented in a Xilinx XC4025pg299-5 device. This design is synthesized with the
Synopsys FPGA Compiler using three different design methodologies:

**Note:** The Top design VHDL code is not included in this manual, however, the script files used to run the Synopsys FPGA Compiler are provided in “Appendix B”.

- Design is compiled as one flat module.
- Design is compiled using the design’s original hierarchial structure.
- Design is compiled in several mid-size modules (Recommended).

A comparison of the results from the three different methods used to process the Top design is provided in Table 5-3 at the end of this chapter. This table also includes ClockToSetup and PadToSetup requirements for this design.

The original Top design hierarchy consists of four main blocks at the core level, as shown in Figure 5-1. The core level of this design contains two large modules, R0 and X0, and two small modules, UP0 and DD0. The two large modules contain approximately 30 sub-modules ranging in size from four CLBs to 591 CLBs.

![Figure 5-1 Original Hierarchy of Top Design](image)

Table 5-1 provides the resource statistics for the Top design with the original hierarchy.
Table 5-1  Top Design's Resource Statistics

<table>
<thead>
<tr>
<th>Module Name</th>
<th>RPMs</th>
<th>CLBs</th>
<th>Flip-flops</th>
<th>Tristate Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP</td>
<td>50</td>
<td>962</td>
<td>958</td>
<td>14</td>
</tr>
<tr>
<td>X0</td>
<td>16</td>
<td>342</td>
<td>335</td>
<td>0</td>
</tr>
<tr>
<td>R0</td>
<td>34</td>
<td>591</td>
<td>582</td>
<td>0</td>
</tr>
<tr>
<td>UP0</td>
<td>0</td>
<td>25</td>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td>DD0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>N1</td>
<td>8</td>
<td>130</td>
<td>174</td>
<td>0</td>
</tr>
<tr>
<td>N2</td>
<td>0</td>
<td>33</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>N3</td>
<td>3</td>
<td>80</td>
<td>89</td>
<td>0</td>
</tr>
<tr>
<td>N4</td>
<td>0</td>
<td>19</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>N5</td>
<td>4</td>
<td>63</td>
<td>55</td>
<td>0</td>
</tr>
<tr>
<td>N6</td>
<td>6</td>
<td>112</td>
<td>113</td>
<td>0</td>
</tr>
<tr>
<td>N7</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>N8</td>
<td>2</td>
<td>18</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>N9</td>
<td>7</td>
<td>53</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>N10</td>
<td>0</td>
<td>14</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>N11</td>
<td>2</td>
<td>26</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>N12</td>
<td>0</td>
<td>7</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>N13</td>
<td>2</td>
<td>24</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>M1</td>
<td>0</td>
<td>34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M2</td>
<td>4</td>
<td>21</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>M3</td>
<td>0</td>
<td>19</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>M4</td>
<td>0</td>
<td>41</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>M5</td>
<td>1</td>
<td>37</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>M6</td>
<td>0</td>
<td>14</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>M7</td>
<td>0</td>
<td>38</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>M8</td>
<td>0</td>
<td>23</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>M9</td>
<td>9</td>
<td>77</td>
<td>83</td>
<td>0</td>
</tr>
<tr>
<td>M10</td>
<td>2</td>
<td>27</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>M11</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Xilinx Development System
Compiling Top Design as One Flat Module

The Top design is compiled as one flat module using the Synopsys Compile -ungroup_all command on the core level. Although this design utilizes only 71% of the XC4025 device, it is unroutable because the logic is densely placed in one region of the device, as shown in Figure 5-2.

Figure 5-2 Ratsnest of Top Design Compiled Flat
Compiling Top Design Using Original Hierarchy

The Top design is compiled using the original hierarchy and the X-BLOX DesignWare library. This library contains arithmetic functions that are implemented with RPMs. RPMs contain RLOCs to align the logic within the RPM. Arithmetic functions that are 8-bits or larger and implemented with X-BLOX DesignWare modules are usually faster and easier to route. However, for large designs with several small arithmetic functions (smaller than 8-bits), use the Synopsys DesignWare libraries to take advantage of the synthesis tool's ability to reduce the gate count. This decrease in the number of gates occurs when arithmetic functions are compiled with modules that contain similar functions. Gate reduction does not occur with the X-BLOX DesignWare library because the underlying logic of the components is not available when the design is compiled. The component logic is created later when the X-BLOX program is run. If you use the Synopsys FPGA Compiler or Design Compiler, use the Synopsys DesignWare library instead of the X-BLOX DesignWare library for arithmetic functions.

Note: Refer to the “Resource Sharing” and “Gate Reduction” sections in the “HDL Coding Hints” chapter for more information.

Compiling the Top design with the original hierarchy increases the number of packed CLBs by 12% compared to compiling the design as one flat module. This increase in CLB utilization occurs because the design hierarchy prevents the synthesis tool from fully optimizing the design.

This design methodology allows PPR to place unconstrained cells at any location in the device making it difficult to debug critical paths. If any changes are made to the design, PPR must be run again. Also, the placement and routing information from the previous design iteration cannot be used to guide the modified design. A design change may result in an unroutable design that requires additional floorplanning.

Floorplanning RPMs

PPR is run on this design before and after floorplanning the RPMs. This design is unroutable when the RPMs are not floorplanned. Local routing congestion similar to that shown in Figure 5-2 occurs because the logic is placed in one region of the device. Floorplanning the
RPMs forces PPR to evenly place the logic in the device. The RPM floorplan is shown in Figure 5-3 and the placed and routed design is shown in Figure 5-4. Compare Figure 5-2 and Figure 5-4. Although the routing is not shown in Figure 5-4, it is apparent that the floorplanned design is evenly placed with less routing congestion.

**Figure 5-3 RPM Floorplan for Top Design Compiled Using the Original Hierarchy**
Meeting Speed Requirements

The Top design requires an 8 MHz internal clock speed. Using the original hierarchy with the X-BLOX DesignWare modules, the longest constrained ClockToSetup delay is 143.5 ns; this exceeds the 125 ns requirement. All constrained PadToClock delays meet the requirements shown in Table 5-3.
Compiling Top Design After Modifying the Hierarchy

To obtain fast, routable designs, Xilinx recommends that you divide large designs into medium-sized modules. The original Top design hierarchy consists of four main blocks at the core level.

The Synopsys FPGA Compiler’s Report_fpga command is used to determine the CLB utilization in the original Top design. The results are as follows:

- R0 block uses approximately 591 CLBs
- X0 block uses approximately 342 CLBs
- UP0 block uses approximately 25 CLBs
- DD0 block uses approximately four CLBs

Note: Table 5-1 also lists these numbers.

The Top design is modified to create a more efficient hierarchy in which the design modules use approximately 100 to 200 CLBs. For example, in the original design hierarchy, the R0 module uses 591 CLBs. This module is separated into four modules as shown in Figure 5-5. The X0 module uses 342 CLBs in the original design hierarchy. This module is separated into two modules as shown in Figure 5-6. The new design hierarchy is shown in Figure 5-7.

You may find it difficult to divide some designs into modules with the recommended number of CLBs. This can occur if the routing is not contained within the modules or if there are numerous interconnects between modules. If you do not modify your design’s hierarchy and the design does not route, divide the design into modules with a CLB count as close as possible to the ideal size.
Figure 5-5 R0 Module Divided into Four Sub-modules
Figure 5-6 X0 Module Divided into Two Sub-modules
The new hierarchy is based on the original module size and the interconnect between modules. An ideal hierarchy should result in fewer gates and less routing congestion between modules. Table 5-2 provides an estimate of the new module size based on the CLB numbers in Table 5-1. The actual number of CLBs used per module when the Top design is compiled with and without the X-BLOX DesignWare modules is also provided in Table 5-2. These numbers are usually smaller than the estimated number because further gate reduction occurs.
Evaluating A New Hierarchy

You can evaluate the effectiveness of a new hierarchy by verifying the following:

- The number of RPMs, CLBs, and flip-flops should decrease or remain the same.
- The design modules should use between 100-200 CLBs.
- The total number of modules should be approximately 5-8 for an XC4025 design (this does not include the two small modules, UP0 and DD0.)

The two smaller modules, UP0 and DD0, are not combined with other modules because these modules have a significant amount of interconnect with the X1, X2, and R1-R4 modules.

Defining and Compiling the New Hierarchy

The HDL code is not changed to modify the hierarchy. The Synopsys Group command is used to define the new hierarchy. Next, each module is compiled together using the Compile -ungroup_all

### Table 5-2  Estimated and Actual CLB Utilization

<table>
<thead>
<tr>
<th>New Module Name</th>
<th>Sub-Modules in Group</th>
<th>Estimate CLB Number</th>
<th>Actual CLB Number with X-BLOX</th>
<th>Actual CLB Number without X-BLOX</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>M1, M2, M3, M4, M5</td>
<td>152</td>
<td>102</td>
<td>90</td>
</tr>
<tr>
<td>X2</td>
<td>M6, M7, M8, M9, M10, M11</td>
<td>194</td>
<td>174</td>
<td>151</td>
</tr>
<tr>
<td>R1</td>
<td>N7, N8, N9, N10, N11, N12, N13</td>
<td>157</td>
<td>132</td>
<td>106</td>
</tr>
<tr>
<td>R2</td>
<td>N2, N3, N4, N5</td>
<td>195</td>
<td>153</td>
<td>144</td>
</tr>
<tr>
<td>R3</td>
<td>N6</td>
<td>130</td>
<td>120</td>
<td>103</td>
</tr>
<tr>
<td>R4</td>
<td>N1</td>
<td>112</td>
<td>109</td>
<td>102</td>
</tr>
<tr>
<td>UP0</td>
<td></td>
<td>25</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>DD0</td>
<td></td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Figure 5-8 shows the Synopsys script file that is used to process the core module using the FPGA Compiler. This script file defines the new hierarchical groups, compiles these groups, and creates the XNF file for the core level. The lowest level modules are compiled before this script is run and are saved as .db files (such as N1.db). The script (not shown) used to process the top level modules reads in the top level, reads in the core level, assigns the I/Os, and writes the design to the top.sxnf file.

By compiling larger groups of logic together, the gate count is reduced by 30 CLBs. An additional gate reduction of 85 CLBs is achieved when Synopsys DesignWare modules are used instead of RPMS with small bit widths (the RPMs in the Top design are 4 - 6 bits wide).
Building Design Hierarchy

/* Sample Script for Synopsys to Xilinx Using */
/* the FPGA Compiler */
/* May 1995 */
/* */

/* Read in the design */
/* Read in previously compiled hierarchical modules */
/* */

/* Module R0 */
read -format db N1.db
read -format db N2.db
read -format db N3.db
read -format db N4.db
read -format db N5.db
read -format db N6.db
read -format db N7.db
read -format db N8.db
read -format db N9.db
read -format db N10.db
read -format db N11.db
read -format db N12.db
read -format db N13.db
analyze -format vhdl R0.vhd
elaborate R0

/* Module X0 */
read -format db M1.db
read -format db M2.db
read -format db M3.db
read -format db M4.db
read -format db M5.db
read -format db M6.db
read -format db M7.db
read -format db M8.db
read -format db M9.db
read -format db M10.db
read -format db M11.db
analyze -format vhdl X0.vhd
elaborate X0

/* Modules UPO and DDO */
read -format db UPO.db
read -format db DDDO.db

/* Module CORE */
analyze -format vhdl CORE.vhd
elaborate CORE

/* Define the new hierarchical groups for X0 */
/* */
current_design = CORE
current_instance = R0
group {N7,N8,N9,N10,N11,N12,N13} -design_name R1 -cell_name R1
group {N2,N3,N4,N5} -design_name R2 -cell_name R2
group {N6} -design_name R3 -cell_name R3

HDL Synthesis for FPGAs 5-17
HDL Synthesis for FPGAs

```verilog
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Compile the new modules R1, R2, R3, R4 */
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++ */

current_design = R1
current_clock = "clk_1" -period 100 -waveform { "0", "50" } { "clk_1" }
current_design = R2
current_clock = "clk_1" -period 100 -waveform { "0", "50" } { "clk_1" }
current_design = R3
current_clock = "clk_1" -period 100 -waveform { "0", "50" } { "clk_1" }
current_design = R4
current_clock = "clk_1" -period 100 -waveform { "0", "50" } { "clk_1" }
```

```verilog
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Define the new hierarchical groups for X0 */
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
current_instance = X0

current_design = X1
```

```verilog
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Compile the new modules X0 */
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++ */

current_design = X1
```

```verilog
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Replace_fpga for the module UPU and UDU */
/* +++++++++++++++++++++++++++++++++++++++++++++++++++++++ */

current_design = UPU
```

Xilinx Development System
Figure 5-8 Script File for Compiling Core Modules

Setting Boundaries and Floorplanning the Modules

Note: Refer to the “Floorplanning Your Design” chapter in this manual and the Floorplanner Reference/User Guide for more information on floorplanning.

The modules are constrained to specific device areas in the Floorplanner. Boundaries are selected and the modules are placed as described in the “Setting Boundaries in the Floorplan Window” section in the “Floorplanning Your Design” chapter. The modules are floorplanned as shown in Figure 5-9 and Figure 5-10. Figure 5-9 illustrates where the various modules are located and Figure 5-10 shows the actual placement of the modules in the Floorplanner.

The area size must be large enough to accommodate a module as well as provide enough space for PPR to add feed-throughs to route the design. The height of an area must accommodate the tallest structure in the module. For example, in an XC4000 device, an 8-bit adder requires an area that is five CLBs high. The location of the areas is determined by the data flow.

In the Top design, the two smaller blocks, UP0 and DD0, have many interconnects to all the modules and, therefore, are not constrained to a specific area of the device. As shown in Figure 5-9, the cells in these modules “float” to allow PPR to calculate the best placement. Any unused CLBs are placed in the center of the device for these modules.
Figure 5-9 Overview of Floorplanned Modules
Floorplanning Structured Cells

After the design modules are floorplanned to specific device areas, any structured cells within the modules are floorplanned. Floorplanning structured elements improves design routability and timing. Structured cells include RPMs, registers, BUFTs, and memory. These cells are placed in the same area as the modules that contain them. After the design is floorplanned, a constraints file is written.
**Placing and Routing the Top Design**

PPR uses the constraints file to place and route the Top design. PPR is run on this design as follows:

```bash
ppr top placer_effort=4 router_effort=3 cstfile=top
```

**Note:** Refer to the “Understanding High-Density Design Flow” chapter for more PPR options.

Figure 5-11 shows the Top design after it has been placed by PPR. Floorplanning the RPMs and registers helps PPR evenly divide the logic in the device. Compare Figure 5-2 and Figure 5-11. Although the routing is not shown in Figure 5-11, it is apparent that the floorplanned design is evenly placed with less routing congestion.

The PPR runtime for the modified design hierarchy is approximately 6.5 hours less than the runtime for the original design hierarchy. The ClockToSetup time is reduced from 143.2 ns to 106.7 ns (with X-BLOX). The PadToSetup time slightly increases by 6 ns.
Adding Probe Points to Debug a Design

To debug a design, internal signals are routed to unused I/O. If the I/O pins are constrained, it can be difficult to make design changes. Use the XACT Design Editor’s (XDE) Defineprobe and Assignprobe commands to select an unused IOB for a probe point and to route an internal net to the selected point.
For the Top design, two probe points are defined for the original design and for the design after the hierarchy is modified. In the design with the original hierarchy, the probe points are unroutable. In the design with the new hierarchy, the probe points are easy to route because the unused logic in the center of the device contains unused interconnects that are used to route the probe points.

**Comparing Top Design Methodologies**

This section compares the three methodologies used to compile the Top design. A summary of the results is provided in Table 5-3.

**Flat Design**

When the Top design is compiled as one flat module, the fewest device resources are used, however, the design is densely packed and is unroutable.

**Original Design Hierarchy**

When the Top design is compiled using the original design hierarchy, the RPMs must be floorplanned to force PPR to evenly place the logic in the device. This design utilizes 72% (packed CLBs) of an XC4025 device. The longest constrained ClockToSetup delay is 143.2 ns and the longest constrained PadToClock delay is 100.1 ns. Any small changes to this design may make the design unroutable.

**Modified Hierarchy**

When the Top design is compiled after the original hierarchy is modified, floorplanning individual cells is not required because the new hierarchy assists PPR in placing the logic. This design uses 63% of an XC4025 device. The longest ClockToSetup delay is reduced to 106.7 ns. When a small change is made to this design, only the module in which the change is made is processed. PPR runtime is reduced because the PPR Guide option is used to retain the placement and routing of the unchanged modules. The new hierarchy makes it easy to modify the design and place any unused CLBs in the center of the device.
<table>
<thead>
<tr>
<th>Design Methodology</th>
<th>Occupied CLBs</th>
<th>Packed CLBs</th>
<th>RPMs</th>
<th>Flip-flops</th>
<th>Clock ToSetup</th>
<th>Pad ToSetup</th>
<th>PPR Run Time (CPU Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat Design (no X-BLOX)</td>
<td>737</td>
<td>619</td>
<td>0</td>
<td>958</td>
<td>n/a*</td>
<td>n/a*</td>
<td>n/a*</td>
</tr>
<tr>
<td>Original Design Hierarchy; no Floorplanning</td>
<td>1024</td>
<td>745</td>
<td>50</td>
<td>958</td>
<td>n/a*</td>
<td>n/a*</td>
<td>n/a*</td>
</tr>
<tr>
<td>Original Design Hierarchy; with Floorplanning</td>
<td>1024</td>
<td>745</td>
<td>50</td>
<td>958</td>
<td>143.2 ns</td>
<td>100.1 ns</td>
<td>01:13 02:05 12:53 16:14</td>
</tr>
<tr>
<td>Re-Group Design Hierarchy with X-BLOX</td>
<td>1015</td>
<td>715</td>
<td>46</td>
<td>958</td>
<td>106.7 ns</td>
<td>108.8 ns</td>
<td>01:05 01:34 08:07 10:49</td>
</tr>
<tr>
<td>Re-Group Design Hierarchy without X-BLOX</td>
<td>957</td>
<td>630</td>
<td>0</td>
<td>958</td>
<td>113.4 ns</td>
<td>107.6 ns</td>
<td>01:02 05:39 04:29 11:12</td>
</tr>
</tbody>
</table>

*The flat design did not route and was not floorplanned. No PPR runtime is provided.*
Chapter 6

Understanding High-Density Design Flow

This chapter describes the design flow for high-density HDL designs that you should follow when analyzing and modifying your designs to improve design performance. A summary of the steps in the flow is illustrated in Figure 6-1. If your design does not route or meet speed requirements, you can evaluate the design’s hierarchy, test various synthesis options, modify timing specifications, floorplan design elements, or select different PPR options.

The design example used in this chapter is the Top design described in the “Building Design Hierarchy” chapter. This design is implemented in an XC4025pg299-5 FPGA. The Top design VHDL code is not included in this manual, however, the script files used to run the Synopsys FPGA Compiler are provided in “Appendix B”. The Top design contains approximately 31 sub-modules that range in size from 7 - 113 CLBs. The detailed design flow, including files and programs, for implementing the Top design in an XC4025 FPGA is shown in Figure 6-2.

This chapter also includes information on using guided design with your high-density HDL designs.

Note: Most of the information in this chapter is described in detail in the previous chapters of this manual. When applicable, you are referred to the appropriate chapter for more information.
Figure 6-1 High-Density Design Flow
Understanding High-Density Design Flow

STEP 1: Estimate Design Size

Synopsys FPGA Compiler
compile -ungroup_all
report_fpga

top.snxf

Syn2XNF

top.xfl

XNFPrep

top.xfl/top.xtg

PPR estimate=true

top.rpt

STEP 2: Evaluate Design

top.vhd

STEP 3 & 4: Modify Hierarchy, Synthesize, & Optimize Design

Synopsys FPGA Compiler
compile
report_fpga

top.snxf

X-BLOX

top.rpt

STEP 5: Translate Design & Add TimeSpecs

Syn2XNF

top.xfl

AddTNM

top.bfl

mv top.bfl top.xfl

XNFPrep

top.xfl/top.xtg

X-BLOX

top.xg
Step 1: Estimating Your Design Size

Generally, the first step in implementing a high-density HDL design is determining if your design fits in the target device. Designs that are compiled as one flat module are usually the smallest designs. However, these designs can be difficult to route, debug, and modify.

To determine how difficult it is to route a flat design, run PPR on your design after compiling it as one flat module. If PPR can route your design, it should route easily after it is compiled with hierarchy.

To compile your design as one flat module, use the following command:

```
compile -ungroup_all
```
To obtain a report of device resource utilization after compiling, use the following command:

```
report_fpga
```

Figure 6-3 shows a sample report file generated by the Report FPGA command for the Top design.

```
Xilinx FPGA Design Statistics

FG Function Generators: 1269
H Function Generators: 267
Number of CLB cells: 659
Number of Hard Macros and Other Cells: 252
Number of CLBs in Other Cells: 155
Total Number of CLBs: 751
Number of Ports: 0
Number of Clock Pads: 0
Number of IOBs: 0
Number of Flip Flops: 921
Number of 3-State Buffers: 14
Total Number of Cells: 965
```

**Figure 6-3 Area Utilization Report**

**Note:** For more information on the Report FPGA command, refer to the *Synopsys (XSI)—FPGAs Interface/Tutorial Guide*.

**Determining Device Utilization**

To determine if your design fits the targeted device, perform the following steps.

**Note:** You can substitute your design name and part type in the following steps.

1. Translate your design as follows:

```
syn2xnf -p 4025pg299-5 top.sxnf
xnfprep top.xff
```
2. If your design contains X-BLOX modules, run X-BLOX to synthesize these modules:

**Note:** If you used the X-BLOXGen program to instantiate an X-BLOX module, you do not need to run X-BLOX because X-BLOXGen runs this program automatically.

```
xblox top.xtg
```

Run XNFPrep again on the output from X-BLOX (XG file):

```
xnfpreg top.xg
```

3. Run PPR:

```
ppr top.xtg estimate=true
```

The PPR screen output appears as shown in Figure 6-4. A preliminary estimate of device utilization for the Top design is listed. PPR may use additional CLBs as feedthroughs to help route the design.

```
ppr: Mapping function generators...

Preliminary estimate of device utilization for part 9409PCG29:
------------------------------------------------------------------------
12\% utilization of I/O pins. \( \{ 31 \text{ of } 256 \} \)
70\% utilization of CLB function generators. \( \{ 1431 \text{ of } 2048 \} \)
47\% utilization of CLB flip-flops. \( \{ 958 \text{ of } 2048 \} \)
------------------------------------------------------------------------
Consult the PPR report file for final device utilization statistics.
ppr: Making report file...
```

**Figure 6-4 PPR Screen Output**

4. Use the report file generated by PPR to determine if your design fits the targeted device. The utilization statistics for the Top design are shown in Figure 6-5.

If the percentage (% Used column) of packed CLBs, bonded I/O pins, or CLB flip-flops exceeds 100, your design does not fit the device. In this case, select a larger device or remove some logic.

If the percentage of packed CLBs, bonded I/O pins, or CLB flip-flops is below 50%, your design will most likely fit the device and meet timing requirements.
If the percentage of packed CLBs, bonded I/O pins, or CLB flip-flops is greater than 50% and less than 100%, successful routing depends on the type of logic implemented, the synthesis strategy, use of hierarchy, and coding styles.

Highly structured, pipelined, or synchronous designs are usually easy to route. For designs that contain many interconnects and are not completely structured, you may need extra CLBs to route the design or meet timing requirements. I/O-intensive designs can also be difficult to route. Generally, this method of determining if your design fits the targeted device is accurate if your designs are partly or completely structured and if they utilize 60 - 80% of the device.

**Step 2: Evaluating Your Design for Coding Style and System Features**

*Note:* Refer to the “HDL Coding Hints” and the “HDL Coding for FPGAs” chapters for more information on the topics included in this section.
The next step in the design flow is evaluating your design for poor coding styles or coding styles that are difficult to implement in an XC4000 FPGA.

After correcting any coding style problems, incorporate FPGA system features into your design to improve resource utilization and enhance the speed of critical paths. A few ways of incorporating FPGA system features are listed below.

- Use the global clock buffers and global set/reset net to reduce routing congestion and improve design performance.
- Place the four highest fanout signals on the BUFGS.
- Modify large multiplexers to use tristate buffers.
- Use one-hot encoding for large or complex state machines.
- Use I/O registers where possible.
- Use I/O decoders where possible.
- Use the STARTUP block.

**Step 3: Modifying Your Design Hierarchy**

**Note:** Refer to the “Building Design Hierarchy” chapter for more information on the topics included in this section.

Large HDL designs (more than 5,000 gates) for FPGAs are usually synthesized as either one flat module or as many small modules. Flat designs can be difficult to route because the logic is always placed in one region of the device, which can result in routing congestion. Many small modules can cause an increase in the number of gates, which can result in a design that does not fit the target device. Although these design methodologies are used to implement ASIC devices, they are usually not the most effective methodologies for implementing high-density FPGAs.

To efficiently use high-density FPGAs, structure your design hierarchy to guide the placement and routing of the device. Effective design hierarchy can reduce routing congestion and improve timing. In addition, hierarchical designs are easier to debug and modify. This section describes how to modify your design hierarchy to reduce routing congestion.
Estimating Area Utilization

For large designs, Xilinx recommends that you divide your design into mid-sized modules. To determine the most efficient way to group existing modules, you can estimate the area utilization for each module with the following procedure.

1. Separately synthesize each module in your design; do not use the ungroup command. Refer to the Synopsys (XSI)—FPGAs Interface/Tutorial Guide or Synopsys manuals for more information.

2. Run the following command on each module:

   `report_fpga`

   This command generates an area utilization report for each module.

3. Complete the worksheet in Table 6-1 using the values from the report file generated by the Report FPGA command. The resource statistics for the Top design from the “Building Design Hierarchy” chapter are shown in Table 6-2. Use this example as a guide for completing the worksheet in Table 6-1.

Table 6-1  Worksheet for Design Module Resource Statistics

<table>
<thead>
<tr>
<th>Module Name</th>
<th>RPMs</th>
<th>CLBs</th>
<th>Flip-flops</th>
<th>Tristate Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6-2  Top Design’s Resource Statistics

<table>
<thead>
<tr>
<th>Module Name</th>
<th>RPMs</th>
<th>CLBs</th>
<th>Flip-flops</th>
<th>Tristate Buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP</td>
<td>50</td>
<td>962</td>
<td>958</td>
<td>14</td>
</tr>
<tr>
<td>X0</td>
<td>16</td>
<td>342</td>
<td>335</td>
<td>0</td>
</tr>
<tr>
<td>R0</td>
<td>34</td>
<td>591</td>
<td>582</td>
<td>0</td>
</tr>
<tr>
<td>UP0</td>
<td>0</td>
<td>25</td>
<td>37</td>
<td>0</td>
</tr>
<tr>
<td>DD0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>N1</td>
<td>8</td>
<td>130</td>
<td>174</td>
<td>0</td>
</tr>
<tr>
<td>N2</td>
<td>0</td>
<td>33</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>N3</td>
<td>3</td>
<td>80</td>
<td>89</td>
<td>0</td>
</tr>
<tr>
<td>N4</td>
<td>0</td>
<td>19</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>N5</td>
<td>4</td>
<td>63</td>
<td>55</td>
<td>0</td>
</tr>
<tr>
<td>N6</td>
<td>6</td>
<td>112</td>
<td>113</td>
<td>0</td>
</tr>
<tr>
<td>N7</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>N8</td>
<td>2</td>
<td>18</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>N9</td>
<td>7</td>
<td>53</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>N10</td>
<td>0</td>
<td>14</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>N11</td>
<td>2</td>
<td>26</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>N12</td>
<td>0</td>
<td>7</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>N13</td>
<td>2</td>
<td>24</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>M1</td>
<td>0</td>
<td>34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M2</td>
<td>4</td>
<td>21</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>M3</td>
<td>0</td>
<td>19</td>
<td>34</td>
<td>0</td>
</tr>
<tr>
<td>M4</td>
<td>0</td>
<td>41</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>M5</td>
<td>1</td>
<td>37</td>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>M6</td>
<td>0</td>
<td>14</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>M7</td>
<td>0</td>
<td>38</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>M8</td>
<td>0</td>
<td>23</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>M9</td>
<td>9</td>
<td>77</td>
<td>83</td>
<td>0</td>
</tr>
<tr>
<td>M10</td>
<td>2</td>
<td>27</td>
<td>27</td>
<td>0</td>
</tr>
<tr>
<td>M11</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>
Creating a New Hierarchy

**Note:** See the “Top Design Example” section in the “Building Design Hierarchy” chapter for more information on modifying your design hierarchy.

The core level of the original Top design contains two large modules, R0 and X0, and two small modules, UP0 and DD0. The R0 block has 591 CLBs; the X0 block has 342 CLBs; the UP0 block has 25 CLBs; and the DD0 block has 4 CLBs. To create a more efficient hierarchical structure for the placement and routing tools, each design module should use approximately 100 to 200 CLBs. To obtain this ideal size for the modules in your design, create a new hierarchy as follows:

1. Use the Synopsys Design Analyzer to view the interconnect between the modules in your design.
2. Separate large modules into modules that have approximately 100 - 200 CLBs.

Step 4: Synthesizing and Optimizing Your Design

**Note:** Refer to the “Building Design Hierarchy” chapter for more information on the topics covered in this section.

Next, perform the following steps to synthesize and optimize your design.

1. Use the Synopsys Group command to define the new hierarchy. For the Top design, the following groups are defined.

   **Note:** The “\” character represents a continuation marker.

   group {M1,M2,M3,M4,M5} -design_name X1 -cell_name X1
   group {M6,M7,M8,M9,M10,M11} -design_name X2 \ -cell_name X2
   group {N7,N8,N9,N10,N11,N12,N13} -design_name R1 \ -cell_name R1
   group {N2,N3,N4,N5} -design_name R2 -cell_name R2
   group {N6} -design_name R3 -cell_name R3
   group {N1} -design_name R4 -cell_name R4
2. Compile the modules together as follows:

```
compile -ungroup_all
```

The Synopsys script file for compiling the Top design’s core modules is included in the “Building Design Hierarchy” chapter. This file defines the new hierarchical groups, compiles these groups, and creates the XNF file for the core level. The lowest level modules are compiled before this script is run and are saved as .db files. The Top level modules are processed by a separate script. This script reads in the Top level, reads in the core level, assigns the I/Os, and writes the design to the top.sxnf file.

**Note:** The Top design VHDL code is not included in this manual, however, the script files used to run the Synopsys FPGA Compiler are provided in “Appendix B”.

---

**Step 5: Translating Your Design and Adding Group TimeSpecs**

**Note:** Refer to the “HDL Coding for FPGAs” chapter for more information on the topics in this section.

This section describes how to translate the SXNF file created in the previous step to an XNF file as well as how to add timing specifications to your design.

**Translating Your Design**

To translate your design to an XNF file, follow the applicable instructions in your synthesis tool documentation. If you are using the Synopsys FPGA Compiler, perform the following step to translate your design.

Enter the following command:

```
syn2xnf -p 4025pg299-5 top.sxnf
```

An XFF file is generated.

**Note:** For more information on Syn2XNF, refer to the *Synopsys (XSI)—FPGAs Interface/Tutorial Guide*. 
Adding Timing Specifications

You can specify XACT-Performance timing constraints in the following ways.

- Set timing constraints in the synthesis tool (FPGA Compiler only). The synthesis tool passes the constraints to the XNF file.
- Specify default timing constraints using PPR command line options.
- Specify timing constraints for groups of logic in a constraints file.

Using the Synthesis Tool

Path timing specifications (added by the Synopsys XNF Writer) generate a TimeSpec line in the XNF file for every constrained endpoint, which results in a very large XNF file.

**Note:** Not all Synopsys timing specification commands are translated to XNF. For increased accuracy, use the following Synopsys command before writing the XNF file, and use one of the methods described below.

```
xnfout_constraints_per_endpoint=0
```

Using PPR Command Line Options

If your design contains one clock or multiple clocks with the same timing requirements, Xilinx recommends that you specify default timing constraints using PPR command line options. You can set the default clock-to-setup, clock-to-pad, pad-to-setup, and pad-to-pad constraints with the PPR options: Dc2s, Dc2p, Dp2s, and Dp2p.

**Note:** For more information on PPR, refer to the “PPR” chapter in the *Development System Reference Guide*.

Using A Constraints File

The Xilinx tools allow you to specify timing constraints for groups of logic in a constraints file. You can specify a set of paths and the maximum allowable delay on these paths. You can refer to a predefined group by specifying one of the corresponding keywords — FFS, PADS, LATCHES, or RAMS.
To specify timing constraints for designs that are more complex, use the MakeTNM and AddTNM programs after running Syn2XNF.

**Note:** Refer to the “HDL Coding for FPGAs” chapter for more information on MakeTNM and AddTNM.

1. Use MakeTNM to create a TNM file:

   ```
   maketnm top.xff
   ```

2. MakeTNM creates a template `top.tnm` file called `top.tt`. Edit this file to reflect the desired time groups and save the file as `top.tnm`.

3. Use AddTNM to add timing group information from the TNM file to the XFF file:

   ```
   addtnm top.tnm
   ```

4. After the timing groups are created, use the TIMESPEC and TIMEGRP commands to specify timing constraints. Place these commands in a PPR constraints file with a .cst extension. The constraints file is read by XNFPrep and PPR.

5. Run XNFPrep to check for design errors:

   ```
   xnfprep top.xff
   ```

   XNFPrep reads the constraints file and the XFF file with the TNMs included and writes an XTF (or XTG) file with timing information. If the `top.xff` file already contains timing specifications (such as those generated automatically by Synopsys), they can be ignored with the following command.

   ```
   ignore_timespec=top
   ```

6. If your design contains X-BLOX modules, run X-BLOX to synthesize these modules:

   **Note:** If you used the X-BLOXGen program to instantiate an X-BLOX module, you do not need to run X-BLOX because X-BLOXGen runs this program automatically.

   ```
   xblox top.xtg
   ```

   Run XNFPrep again on the output from X-BLOX (XG file):

   ```
   xnfprep top.xg
   ```
Step 6: Building Your Design Hierarchy

**Note:** Refer to the “Floorplanning Your Design” and “Building Design Hierarchy” chapters for more information on the topics included in this section.

Next, constrain your design modules to specific device areas in the Floorplanner. Define boundaries in the Floorplan window and place the selected modules within the specified boundaries. The area size must be large enough to accommodate a module as well as provide enough space for PPR to add feed-throughs to route the design. The height of an area must accommodate the tallest structure in the module.

Step 7: Floorplanning Your Design

**Note:** Refer to the “Floorplanning Your Design” chapter in this manual and the Floorplanner Reference/User Guide for more information on the topics included in this section.

Xilinx gives you two design implementation options. The XACTstep Foundry v7 software provides automatic implementation of your designs and does not support the Xilinx Floorplanner. Alternatively, if you want to control part of the implementation process, use the XACTstep v5.2 software with the Floorplanner.

For high-density devices, Xilinx recommends that you floorplan specific parts of your design to improve PPR performance. Due to the complexity and size of larger designs, PPR is limited in its ability to recognize structure. Your design may not route or meet timing constraints without structured placement. Based on your knowledge of a design’s structure, you can create a floorplan that significantly improves the placement of the design. Generally, you should floorplan the parts of your designs that are timing critical or heavily congested.

Creating a MAP File

The Floorplanner requires a MAP file as input. Before creating a MAP file, you must synthesize your design and save it as an SXNF or an EDIF file. Use one of the following procedures to create a MAP file.
Using XMake

Specify a MAP file as the target file with XMake as follows:

```
xmake [options] top.sxnf top.map
```

Using PPR

1. Follow the XC4000 design flow (includes Syn2XNF, XNFPrep, X-BLOX) to create an XTF file.
2. Create a MAP file at the command line with the following PPR options:

```
ppr top.xtf map_fgs=true place=false route=false report=false lca=false run_pic2map=true
```

Using Prep for Floorplanner Option

Use the Prep for Floorplanner option in the XACT Design Manager (XDM) to create a MAP file as follows:

1. Invoke XDM.
2. Choose the XMake command from the Translate menu.
3. Select the appropriate SXNF file from the pop-up menu.
4. Select Prep for Floorplanner from the pop-up menu.

XMake generates a MAP file.

Floorplanning Design Components

Floorplan the following structured items in your design.

- Large objects such as RPMs, registers, counters, and RAMs
- Buses (place all BUFTs and bus elements)
- BUFTs with I/O or RPM inputs
- Multiple BUFTs (except VCC or GND) with identical source pin inputs

You can floorplan elements other than those listed, but constraining too many elements, especially those without any specific structure, can decrease design performance.
Writing a Constraints File

Use the Write Constraints command in the Floorplanner to create a constraints file, such as top.cst. This file is read by PPR to place and route your design.

Step 8: Placing and Routing Your Design

Note: Refer to the “Building Design Hierarchy” chapter and the Floorplanner Reference/User Guide for more information on the topics in this section.

After floorplanning, run PPR to place and route your design. An example PPR command with various options set is as follows:

```
ppr top.xtf placer_effort=4 router_effort=3
```

Alternatively, you can run PPR from the Process menu in the Floorplanner.

For large devices, such as the XC4025, Xilinx recommends that you set placer_effort to 4 or 5 and router_effort to 3. However, if your design includes floorplanned RPMs, set placer_effort to 2.

Optionally, you can ignore the mapping generated by Synopsys by specifying ignore_maps=true. You may want to run PPR with this option set both ways and evaluate the results. Generally, use the PPR options that generate the fewest CLBs.

You can observe PPR’s progress on your screen as it processes your design, as shown in Figure 6-6. If you want to terminate PPR, press control-c. You are prompted to either save the LCA and report (RPT) file or to quit without saving. If PPR is having problems routing your design, save the LCA design file. Load the LCA file into the Floorplanner and evaluate routing congestion using the Ratsnest or Congestion commands in the View menu.
Using PPR Options

The following is a list of PPR command line options that you may want to use when processing your designs.

- **Ignore_maps=true**
  Use this option to ignore the mapping of combinatorial logic into function generators that is generated by the Synopsys FPGA Compiler. This option may result in a more efficient mapping of the logic.

- **Placer_effort=4**
  Do not use this option if you floorplanned any RPMs. Use the default value of 2 if you have specified many placement constraints.

- **Router_effort=3**
  Use Router_effort=3 for designs that are large or difficult to route. The default value is 2.

- **Ignore_rlocs=true**
  Use this option to override any RLOCs in the XTF file.

- **Outfile=new_name**
  Use this option to redirect PPR output to a file with a name that is different from the input XTF file name. This option is useful for multiple runs of PPR.
Determining If PPR Can Route Your Design

To determine if PPR can route your design, observe your computer screen while PPR is running. The routing percentage should quickly increase from 0 to 98-99%. If the routing percentage stops at a number below 95% and does not progress after a few passes, it is unlikely that your design will route. The PPR screen output for an unroutable design is shown in Figure 6-7. PPR screen output for a design that can be routed is shown in Figure 6-8. If the PPR screen output indicates that your design cannot be routed, stop PPR by pressing control-c and save the LCA file for evaluation.

Figure 6-7 PPR Screen Output for an Unroutable Design

Figure 6-8 PPR Screen Output for a Routable Design

Step 9: Evaluating the Results

PPR generates a report file that lists unrouted nets and summarizes whether or not the timing specifications were met.
If your design routed, use XDelay to further evaluate the path timing. To simulate your design, you can use any HDL or gate simulator that supports the targeted Xilinx device.

If your design does not route or does not meet the timing specifications, evaluate the floorplan as described in the next section.

**Evaluating Module Placement with the Floorplanner**

*Note:* Refer to the “Floorplanning Your Design” chapter in this manual and the *Floorplanner Reference/User Guide* for more information on the topics included in this section.

This section describes how you can evaluate your design in the Floorplanner. An efficiently placed design has the following characteristics.

- Logic is evenly distributed throughout the device
- Most of the interconnects are within the modules
- A few interconnects join adjacent modules
- A very small number of interconnects join non-adjacent modules
- High-fanout clocks, clock enables, and reset nets are routed using the global clock buffer routing resources or the global set/reset routing resources

An inefficient placement of the Top design is shown in Figure 6-9.
After running PPR and determining that your design cannot be routed, perform the following steps.

1. Load your design into the Floorplanner.

   The floorplan of the Top design is shown in Figure 6-10. Observe that the logic is not evenly distributed in the device. Logic is densely packed in the upper right-hand corner, which is the location of the R1 and R4 modules. The device areas allocated for these modules are too small.
To evaluate the module interconnections, use the Find Nets command in the Edit menu in the Floorplan window.

The Find Nets dialog box and the Ratsnest dialog box appear. The Ratsnest dialog box is shown in Figure 6-11.
3. In the Available Nets field, select the nets that correspond to one of your design modules.

For example, in Figure 6-11, the nets associated with the R4 (CORE0/RECEIVER0/RECEIVER4) module are selected.

4. Select Add in the Ratsnest dialog box to add the selected nets to the Displayed Nets list.

5. Select the Close command to close the dialog boxes.

6. Figure 6-12 shows the R4 module interconnects displayed in the Floorplan window.
7. Repeat the previous steps to display the ratsnest for other modules in your design.

Figure 6-13 shows the X1 module interconnects displayed in the Floorplan window. Both modules (R4 and X1) have numerous interconnects to the UP0 module. The ratsnest display for modules X2, R1, R2, and R3 also have numerous interconnects to UP0. These interconnections indicate that UP0 is poorly placed.
Modifying Design Placement

When constraining design modules to specific device areas, the following recommendations can help you produce a design that PPR can route. Alternatively, you can use the Congestion command in the Floorplanner View menu to evaluate routing congestion.
Make sure that the device area is large enough to accommodate a specific module and provides enough space for PPR to add feedthroughs to route the design.

Allow modules that have many interconnects to other modules to “float” to allow PPR to calculate the best placement. For example, in the Top design, the two smaller blocks, UP0 and DD0, have many interconnects to all the modules and, therefore, are not constrained to a specific area of the device. Any unused CLBs are placed in the center of the device for these modules.

An efficient placement of the Top design is shown in Figure 6-14.

![Figure 6-14 Efficient Placement of Top Design](image)

**Using Guided Design**

The term guided design refers to the process in which a previously implemented design — also known as a guide file — is used to guide mapping, placement, and routing. Guided design allows you to modify or add logic to a design while preserving the layout and performance from a previous run of PPR. You can reduce the number of timing changes between iterations of PPR as well as decrease PPR.
run time with guided design. The three ways of performing guided design are iterative design, incremental design, and using XDE.

**Note:** For more information on PPR and guided design, refer to the “PPR” chapter in the Development System Reference Guide.

### Understanding High-Density Design Flow

**Using Iterative Guided Design**

If you need to make logic changes in your design after it has been verified for timing, use iterative guided design to minimize the impact of the changes on the new layout. Iterative design simplifies the mapping, placement, and routing process, as well as verifies the design timing.

In iterative design, your original design is specified as the guide file. PPR copies as much of your guide file’s mapping, placement, and routing as possible. PPR implements logic that has not changed by copying the LCA resources in your guide file, ensuring identical timing. For logic that is changed, PPR uses the standard mapping, placement, and routing process.

### Using Incremental Guided Design

You can implement and verify your design in stages using incremental guided design as follows:

1. Run PPR on a single functional block.
2. Verify the timing internal to the block.
3. Add a second functional block to your design.
4. Run PPR.

   PPR maps, places, and routes your design using the results from the initial PPR run as a guide file.

5. Verify the timing of the new logic.
6. Repeat this process to build and verify your design piece by piece.

### Using XDE

Guided design is also useful when you manually place and route critical paths using the EditLCA program in XDE. Using the edited design as a guide design allows you to specify exactly how your
critical paths are routed while still allowing PPR to place and route less timing-critical logic. Generally, critical paths are difficult to manually route in HDL designs because these designs usually have random net and block names. If you can determine the critical paths, you can use XDE to manually route these paths.

**Effectively Using Guided Design**

Guided design uses signal names to match logic between the guide file and the input design netlist. For this reason, do not re-synthesize design modules that have not been modified. Although you have not made any changes to the HDL code for these modules, optimization changes the signal names. Guided design requires minimal signal name changes.

Xilinx recommends that you use the guide option only with synthesized designs that have been hierarchically grouped and floorplanned, as described in the “Building Design Hierarchy” chapter. You can make small changes to a design module and run PPR again. Modules that have not been changed are guided by the previous run of PPR.

Do not use guided design if you have made extensive design changes or have made changes at a high level in your design’s hierarchy. These type of changes generally result in a large number of signal name changes. In addition, you should synthesize your design again after extensive design changes.

**Understanding Guided Design for XC4000 Designs**

For XC4000 designs, the guided design process is controlled by PPR. The guide file is an LCA file generated by a previous run of PPR.

**Adding a New Module to Your Design**

The Top design is used to illustrate how you can add a new module to your HDL design.

1. Modify your HDL code by adding a new sub-module to your design.

2. Compile only the new module and the top level module that connects the new module to the other modules in the design. Do not repeat the compilation process for any other sub-modules.
3. Modify the PPR constraints file to specify the device region for the new sub-module.

4. Run PPR as follows:

   `ppr top.xtf lock_routing=none guide=top.lca router_effort=3 placer_effort=4 outfile= top_2`

   The Lock Routing option is set to none to allow PPR to unroute the guide file routing on a signal and reroute that signal to improve the timing. It is important to note that PPR does not automatically discard the guide routing when the Lock Routing option is set to none. PPR starts with the guided routing, but if it cannot route a signal because a path is blocked by guided routing, the Lock Routing option specifies whether the guide routing can be rerouted.

Making a Design Change to a Module

The following steps use the Top design to illustrate how you can make a small change to a design module.

1. The X1 module is modified in the HDL code.

2. The modified module as well as any modules further up in the design hierarchy that use the X1 module are re-synthesized.

3. The specified device area is verified to be large enough to accommodate the modified module. If extensive changes are made to X1, XDE is used to unroute X1 from the guide design. This allows PPR to process the entire X1 module and not just the modified section.

4. PPR is run using an earlier LCA file or the edited LCA file as a guide.
Appendix A

Accelerate FPGA Macros with One-Hot Approach

S


tate 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 field-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.

A state machine implemented with a highly encoded state sequence will...
Inverters are required at the D input and the Q output of the state flip-flop to ensure it powers on in the proper state. Combinatorial logic dictates 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 path out of the state is not valid.

Limiting Fan-In

A good state-machine approach for FPGAs limits the amount of fan-in to 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.

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 XC3000 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 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 4 flip-flops. At first glance, OHE may seem counter-intuitive. For designers accustomed to using PLDs, more flip-flops typically indicate 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 place and route 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...
Accelerate FPGA Macros with One-Hot Approach

OHE state machines are simple, leading 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 (Fig. 2). 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 "arrows" 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 Config. For this example, State 1, which must be asserted at power-on, has a doubly-inverted flip-flop structure (shaded region of Fig. 2).

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 XC4020-100000-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 175 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 combinational logic to decode the encoded state values.

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.

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-flops 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 Fig. 2).

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 (Fig. 2, again). 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 (Fig. 1, again).

The first path is valid whenever \((A \oplus B \oplus C)\) is true, which leads into State 2. The second path is valid whenever \((A \oplus B \oplus C)\) is true, leading into State 4. To build the default logic, State 1 is ANDed with the inverse of all of the conditional paths leaving State 1.

6. S-R FLIP-FLOPS OFFER ANOTHER APPROACH TO DECODING THE CARRY OUTPUT. They can also store logic blocks, especially when an output is asserted for a long sequence of contiguous states.

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-6 (Fig. 2, again).

State 4 is the most complex state in the state-machine example. However, the logic for its next-state control follows the same basic method as described earlier. To begin, with State 4 in its 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, a four-input OR-gate feeds the flip-flop (OR 1 in Fig. 2).

The first conditional path comes from State 3. Following the methods established earlier, an AND of State 3 and the conditional logic, which is \((A \oplus B \oplus C)\) with State 3, must be implemented (AND-2 and OR-3 in Fig. 3). The next conditional path is from State 2, which requires an AND of State 2 and variable D (AND-4 in Fig. 3). 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, \((A \oplus B \oplus C)\) (AND-3 and AND-4 in Fig. 3).

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, \((A \oplus B \oplus C)\), is true. Consequently, State 4 must be ANDed with the inverse of the product,\((A \oplus B \oplus C)\). 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-5 and shares the output of AND-6.

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 \((A \oplus B \oplus C)\) 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 (Fig. 4, top). State 3, like States 1 and 4, has a default state, and combines the A, D, State 3, and State-3 feedback to control the flip-flop’s D input (Fig. 4, bottom).

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 transitioning to State 1 (Fig. 5).

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 Config—are each fall into one of three primary output types:

1. Outputs asserted during one state, which is the simplest case. The output signals 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 Config. 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
Accelerate FPGA Macros with One-Hot Approach

STATE MACHINE DESIGN

such example is Multi, which is asserted during State 3 and State 4. One-Hot 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, so 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. These 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 macrocell 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 (Fig. 6). 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 asserted. Simple decoding of the active states is generally best for non-contiguous states. If the output 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 impossible for the state machine to be in only one state at a time. The logic may 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 LCA). 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, gives 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 example meets 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).³


HDL Synthesis for FPGAs
Top Design Scripts

This appendix includes the three script files that are used to compile the Top design described in this manual. Script files for compiling the Top design created with VHDL as well as with Verilog are included. The script files for the VHDL design use the elaborate and analyze commands, while the Verilog script files use the read command.

VHDL Script Files

This section includes the three script files for the Top design created with VHDL.

```
/* Sample Script for Synopsys to Xilinx Using the FPGA Compiler */
/* *-----------------------------------*/
/* Read in the design */
/* *-----------------------------------*/
/* Read in previously compiled hierarchical modules */
/* Module RO */
read -format db M1.db
read -format db M2.db
read -format db M3.db
read -format db N1.db
read -format db N2.db
read -format db N3.db
read -format db N4.db
read -format db N5.db
read -format db N6.db
read -format db N7.db
read -format db N8.db
read -format db N9.db
read -format db M10.db
read -format db M11.db
read -format db M12.db
read -format db M13.db
analyze -format vhdl RO.vhd
evaluate RO

/* Module X0 */
read -format db M1.db
read -format db M2.db
read -format db M3.db
```
HDL Synthesis for FPGAs

B-2 Xilinx Development System
// Define the new hierarchical groups for X0

// ++++++++++++++++++++++++++++++++++++ * /
current_design = CORE

current_instance = X0

% group: M1,M2,M3,M4,M5 -cell: X1
% group: M6,M7,M8,M9,M10,M11 -cell: X2

// ++++++++++++++++++++++++++++++++++++ * /

// Compile the new modules X0

// ++++++++++++++++++++++++++++++++++++ * /

current_design X1
create_clock -name "clk_2" -period 100 -waveform { "0" "50" }
report_fdga > x1.fdga
replace_fdga

current_design X2
create_clock -name "clk_2" -period 100 -waveform { "0" "50" }
report_fdga > x2.fdga
replace_fdga

// ++++++++++++++++++++++++++++++++++++ * /

// Replace_fdga for the module UPO and DDO

// ++++++++++++++++++++++++++++++++++++ * /

current_design UPO
replace_fdga

current_design DDO
replace_fdga

// ++++++++++++++++++++++++++++++++++++ * /

// Write out the db and the XNF file for the CORE level

// ++++++++++++++++++++++++++++++++++++ * /

current_design = CORE
write -format db -hierarchy -output CORE.db
write -format xnf -hierarchy -output core.xnf
exit

Figure B-1 Core Script (VHDL)
```vhdl
/* Example of compiling one of the lower levels */
read_design
/
analyze -format vhdl m1.vhd
elaborate m1
/
create_clock -name "clk_1" -period 100 -waveform {"0" "50"} {"clk_1"}
set_wire_load "40M3-S_avg"
compile
/
write -format db -output m1.db
/*exit */
```

Figure B-2 M1 Script (VHDL)
/* Sample Script for Synopsys to Xilinx Using */
/* the FPGA Compiler */
/* May 1995 */
/*****************************************************************************/
/* Read in the design */
/*****************************************************************************/

read -format db CORE.db
analyze -format vhdl top.vhd
elaborate top

/* Set the current design to the top level */
current_design top
dont_touch core0

/* I/O's where instantiated in this design */
dont_touch { u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, \u11, u12, u13, u14, u15, u16, u17, u18, \u19, u20, u21, u22, u23, u24, u25, u26, \u27, u28, u29, u30, u31, u32, u33 }

create_clock -name 'clk_1' -period 100 -waveform { "0" "50" } {'clk_1'}
create_clock -name 'clk_2' -period 100 -waveform { "0" "50" } {'clk_2'}

/*****************************************************************************/
/* Compile the design */
/*****************************************************************************/
compile -boundary_optimization

/*****************************************************************************/
/* Save the design */
/*****************************************************************************/
write -format db -hierarchy -output top.db
report_fpga > report.fpga
report_timing -worst 5

replace_fpga
write -format db -hierarchy -output top_after_rf.db
set_attribute find(design, "") "xfout_use_blknames" -type boolean FALSE
set_attribute find(design, "") "xfout_constraints_pur_einpoint" = 0

/* SAVE .conf */
write -format xnf -hierarchy -output top.xnf
exit /*

Figure B-3 Top Script (VHDL)
Verilog Script Files

This section includes the three Verilog script files for the Top design.

/* Sample Script for Synopsys to Xilinx Using Xilinx Compiler */
/* May 1995 */
/*===================================================================*/
/* Read in the design */
/* Read in previously compiled hierarchical modules */
/* Module RO */
read -format db N1.db
read -format db N2.db
read -format db N3.db
read -format db N4.db
read -format db N5.db
read -format db N6.db
read -format db N7.db
read -format db N8.db
read -format db N9.db
read -format db N10.db
read -format db N11.db
read -format db N12.db
read -format db N13.db

read -format verilog RO.v
/* Module XO */
read -format db M1.db
read -format db M2.db
read -format db M3.db
read -format db M4.db
read -format db M5.db
read -format db M6.db
read -format db M7.db
read -format db M8.db
read -format db M9.db
read -format db M10.db
read -format db M11.db

read -format verilog XO.v
/* Modules UPD and DDD */
read -format db UPD.db
read -format db DDD.db
/* Module CORE */
read -format verilog CORE.v
/* Define the new hierarchical groups for XO */
current_design = CORE
current_instance = RO
Top Design Scripts

HDL Synthesis for FPGAs

```verbatim
/* ******************************************** */
/* Compile the new modules R1, R2, R3, R4 */
/* ******************************************** */

current_design = R1
create_clock -name "clk_1" -period 100 -waveform { "0" "50" } { "clk_1" }
compile -ungroup_all
report_fpga > R1.fpga
replace_fpga

current_design = R2
create_clock -name "clk_1" -period 100 -waveform { "0" "50" } { "clk_1" }
compile -ungroup_all
report_fpga > R2.fpga
replace_fpga

current_design = R3
create_clock -name "clk_1" -period 100 -waveform { "0" "50" } { "clk_1" }
compile -ungroup_all
report_fpga > R3.fpga
replace_fpga

current_design = R4
create_clock -name "clk_1" -period 100 -waveform { "0" "50" } { "clk_1" }
compile -ungroup_all
report_fpga > R4.fpga
replace_fpga

/* Define the new hierarchical groups for XU */
/* ******************************************** */

current_design = CORE
current_instance = XU

group {M1,M2,M3,M4,M5} -design_name X1 -cell_name X1
group {M6,M7,M8,M9,M10,M11} -design_name X2 -cell_name X2

/* Compile the new modules X0 */
/* ******************************************** */

current_design X1
create_clock -name "clk_2" -period 100 -waveform { "0" "50" } { "clk_2" }
compile -ungroup_all
report_fpga > X1.fpga
replace_fpga

current_design X2
create_clock -name "clk_2" -period 100 -waveform { "0" "50" } { "clk_2" }
compile -ungroup_all
report_fpga > X2.fpga
replace_fpga
```
HDL Synthesis for FPGAs

Figure B-4 Core Script (Verilog)

```verilog
/* ---------------------------------------------------------------*/
/* Replace_fpga for the module UDO and D00 */
/* ---------------------------------------------------------------*/

current_design UDO
replace_fpga

current_design D00
replace_fpga

/* Write out the db and the XNF file for the */
/* CORE level */
/* ---------------------------------------------------------------*/

current_design = CORE
write -format db -hierarchy -output CORE.db
write -format xnf -hierarchy -output core.xnf
exit
```

Figure B-5 M1 Script (Verilog)

```verilog
/*------------------------------*/
/* Example of compiling one of the lower */
/* levels */
/*------------------------------*/

read -format verilog m1.v

/*------------------------------*/
/* Compile the design */
/*------------------------------*/

create_clock -name "clk_1" -period 100 -waveform { "0" "50" } { "0"}
set_vleq_load "4013-5_avg"
compile

/*------------------------------*/
/* Save the design */
/*------------------------------*/

write -format db -output m1.db
exit
```

Xilinx Development System
/* Sample Script for Synopsys to Xilinx Using */
/* the FPGA Compiler */
/* -------------------------------------------------------------------------*/

/* Read in the design */
/* ***************************************************************************/
read -format db CORE.db
read -format verilog top.v

/* Set the current design to the top level */
current_design top
dont_touch CORE.db

/* I/O's where instantiated in this design */
dont_touch {u1, u2, u3, u4, u5, u6, u7, u8, u9, u10, \u11, u12, u13, u14, u15, u16, u17, u18, \u19, u20, \u21, u22, \u23, u24, \u25, u26, \u27, u28, u29, u30, u31, u32, u33 }

create_clock -name "clk_1" -period 100 -waveform { "0" "50" } { "clk_1" }
create_clock -name "clk_2" -period 100 -waveform { "0" "50" } { "clk_2" }

/* Compile the design */
/* -------------------------------------------------------------------------*/
compile -boundary_optimization

/* Save the design */
/* -------------------------------------------------------------------------*/
write -format db -hierarchy -output top.db
report_fpga > report.fpga
report_timing -worst 5
replace_fpga
write -format db -hierarchy -output top_after_rf.db
xnfout_constraints_per_endpoint = 0

/* SAVE .smf */
write -format xnf -hierarchy -output top.smf

/*exit */

Figure B-6 Top Script (Verilog)
Appendix C

Tactical Software and Design Examples

This appendix lists the tactical software and design examples that are described in this manual. Refer to the “Getting Started” chapter for information on retrieving and installing the files listed in this appendix.

Tactical Software

The three programs in Table C-1 are provided to help you utilize the new design methodologies described in this manual.

Table C-1 Tactical Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Description</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-BLOXGen</td>
<td>Allows you to instantiate X-BLOX modules in your HDL code</td>
<td>Chapters 3 and 6</td>
</tr>
<tr>
<td>MakeTNM</td>
<td>Allows you to create a template control file for defining timing groups</td>
<td>Chapters 3 and 6</td>
</tr>
<tr>
<td>AddTNM</td>
<td>Allows you to add timing group information to XFF file</td>
<td>Chapters 3 and 6</td>
</tr>
</tbody>
</table>

These programs are not included in the Xilinx Synopsys Interface or the XACTstep Development System.

AddTNM and MakeTNM were created with Perl 4.0. To run these programs, you must have either Perl 4.0 or 5.0.
Design Examples

The tables in this section include the design example directory names, the design examples in each directory, and whether the design or the chapter section applies to Verilog.

Table C-2 Chapter 1 Files

<table>
<thead>
<tr>
<th>Chapter Section</th>
<th>Design Example Directory</th>
<th>Design Example</th>
<th>Applies to Verilog?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding HDL Design Flow for FPGAs</td>
<td>None</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>Advantages of Using HDLs to Design FPGAs</td>
<td>None</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>Designing FPGAs with HDLs</td>
<td>None</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>- Using VHDL</td>
<td>None</td>
<td>None</td>
<td>No</td>
</tr>
</tbody>
</table>

Table C-3 Chapter 2 Files

<table>
<thead>
<tr>
<th>Chapter Section</th>
<th>Design Example Directory</th>
<th>Design Example</th>
<th>Applies to Verilog?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparing Synthesis and Simulation Results</td>
<td>None</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Selecting VHDL Coding Styles</td>
<td>None</td>
<td>alu</td>
<td>No</td>
</tr>
<tr>
<td>- Comparing Signals and Variables</td>
<td>sig_vs_var</td>
<td>xor_sig</td>
<td>No</td>
</tr>
<tr>
<td>- Implementing Latches and Registers</td>
<td>d_latch</td>
<td>d_latch</td>
<td>Yes</td>
</tr>
<tr>
<td>- Resource Sharing</td>
<td>d_register</td>
<td>d_register</td>
<td>Yes</td>
</tr>
<tr>
<td>- Gate Reduction</td>
<td>res_sharing</td>
<td>res_sharing</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>gate_reduce</td>
<td>gate_reduce</td>
<td>No</td>
</tr>
</tbody>
</table>
There is no noticeable difference between CASE vs. IF for Verilog as described for VHDL.

**Table C-4  Chapter 3 Files**

<table>
<thead>
<tr>
<th>Chapter Section</th>
<th>Design Example Directory</th>
<th>Design Example Directory</th>
<th>Applies to Verilog?</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Preset or Clear Pin</td>
<td>ff_example</td>
<td>ff_example</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>gate_clock</td>
<td>gate_clock</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>clock_enable</td>
<td>clock_enable</td>
<td>Yes</td>
</tr>
<tr>
<td>- Using If Statements</td>
<td>None</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>- Using Case Statements</td>
<td>None</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>- Using Nested_IF Statements’</td>
<td>nested_if</td>
<td>nested_if</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>if_case</td>
<td>if_ex</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>case_ex</td>
<td>case_ex</td>
<td>No</td>
</tr>
<tr>
<td>- Comparing If Statement and Case Statement’</td>
<td>case_vs_if</td>
<td>if_ex</td>
<td>No</td>
</tr>
</tbody>
</table>

* There is no noticeable difference between CASE vs. IF for Verilog as described for VHDL.
### Table C-5 Chapter 4 Files

<table>
<thead>
<tr>
<th>Chapter Section</th>
<th>Design Example Directory</th>
<th>Design Example</th>
<th>Applies to Verilog?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementing Boundary Scan</td>
<td>bnd_scan</td>
<td>bnd_scan</td>
<td>Yes</td>
</tr>
<tr>
<td>Implementing Logic with IOBs</td>
<td>bidi_reg unbonded_io</td>
<td>bidi_reg unbonded_io</td>
<td>Yes</td>
</tr>
<tr>
<td>Implementing Multiplexers with Tristate Buffers</td>
<td>mux_vs_3state</td>
<td>mux_gate mux_tbuf</td>
<td>Yes</td>
</tr>
<tr>
<td>Setting Timing Constraints</td>
<td>None</td>
<td>None</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table C-6 Chapter 5 Files

<table>
<thead>
<tr>
<th>Chapter Section</th>
<th>Design Example Directory</th>
<th>Design Example</th>
<th>Applies to Verilog?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the Floorplanner</td>
<td>None</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>Floorplanning RPMs, RAMs, and ROMs</td>
<td>rpm_ram</td>
<td>rpm_ram</td>
<td>Yes</td>
</tr>
<tr>
<td>Floorplanning Tristate Buffers</td>
<td>bufts</td>
<td>buft_ex</td>
<td>Yes</td>
</tr>
<tr>
<td>Comparing Hierarchical and Flat Designs</td>
<td>alarm</td>
<td>alarm (and sub-modules)</td>
<td>Yes</td>
</tr>
<tr>
<td>Floorplanning to Reduce Routing Congestion</td>
<td>align_str</td>
<td>align_str</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The entire chapter applies to Verilog and there are no design examples.
Table C-7  Chapter 6 Files

<table>
<thead>
<tr>
<th>Chapter Section</th>
<th>Design Example Directory</th>
<th>Design Example</th>
<th>Applies to Verilog?</th>
</tr>
</thead>
<tbody>
<tr>
<td>The entire chapter applies to Verilog and there are no design examples.</td>
<td>None</td>
<td>None</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Index

A
ACCUM module, 3-26
ADD_SUB module, 3-26
AddTNM, 3-55
  definition, 3-57, 3-66
  flow diagram, 3-58
  running, 3-63, 6-14
after xx ns statement, 2-2
Alarm design also see floorplanning, 4-21
arithmetic functions
  gate reduction, 2-25
  ordering and grouping, 2-3
  resource sharing, 2-21
  X-BLOX DesignWare, 5-8
ASIC
  comparing to FPGA, 1-5, 2-1
assignprobe command, 5-23
asynchronous reset pin, 2-27
asynchronous set pin, 2-27
B
back-annotation, 1-3
barrel shifter design, 2-12
behavioral simulation, 1-3
bidirectional I/O, 3-45
  design example, 3-46
binary encoded state machine, 3-14
boolean optimization, 5-2
boundary scan, 3-40
  design example, 3-41
  instantiating in HDL, 3-40
BSCAN, 3-40
BUFG, 3-4
BUFGP, 3-2
BUFGS, 3-2, 3-4
BUFT see tristate buffer
C
carry logic, 2-21, 3-25, 3-32, 4-9
case statement, 2-2
  comparing to if statement, 2-37
design example, 2-39
  syntax, 2-32
  when to use, 2-32
CLB
  initializing, 3-4
  XC4000, 2-28
clear pin, 2-27, 3-5, 3-8
CLK_DIV module, 3-26
clock enable pin, 2-27, 2-30
CMOS thresholds, 3-44
combinatorial feedback loop, 2-17
combinational logic, 5-2
comments in code, 2-8
COMPARE module, 3-26
compile ungroup_all command, 5-15, 6-4, 6-12
conditional expression, 2-16
congestion command, 6-17
constants, 2-6
constraints file, 1-3
  example, 3-64
  floorplanning, 1-3, 5-21
  PPR, 3-56
  using, 3-55, 6-13
  writing, 6-17
COUNTER module, 3-26
create_clock command, 3-54
critical paths, 5-2
cstfile option, 3-64

D
D latch, 2-17
D register, 2-18
design, 2-6
DATA_REG module, 3-26
debugging a design, 5-23
DECODE module, 3-26
decoders, 3-21
defineprobe command, 5-23
Design compiler
also see FPGA compiler
arithmetic functions, 5-8
design examples
installing, 1-6
design partitioning, 5-2
DesignWare, 5-8, 5-16
also see X-BLOX
gate reduction, 2-25
resource sharing, 2-20
directory tree structure, 1-12
disk space requirements, 1-8
don’t touch attribute, 3-9, 3-35, 3-40
downloading files, 1-8, 1-9

E
EditLCA, 6-27
else statement, 2-18
enumerated type encoded state machine, 3-16
extracting downloaded files, 1-11

F
Field Programmable Gate Array see FPGA
file transfer protocol, 1-10
find nets command, 6-22
Finite State Machine, 3-16, 3-20
changing encoding style, 3-19
extraction commands, 3-16
flip-flop, 2-19
floorplanning, 1-3, 4-1
constraints file, 5-21
deciding what to floorplan, 4-5
design hierarchy, 5-3
Floorplanner, 1-6, 4-1, 4-2, 6-15
Alarm design, 4-21
aligning structure, 4-30
aligning tristate buffers, 4-20
check floorplan command, 4-8
collapse button, 4-11
congestion command, 6-17, 6-25
creating a MAP file, 4-2, 6-15
design hierarchy with X-BLOX, 4-27
design hierarchy without X-BLOX, 4-26
Design window, 4-3
device resources, 4-14, 4-17
evaluating designs, 6-20
expand button, 4-11
file open dialog box, 4-7
find nets command, 6-22
flat design with X-BLOX, 4-26
flat design without X-BLOX, 4-24
flat designs, 4-20
Floorplan window, 4-4
fplan command, 4-6
hierarchy, 4-4, 4-6, 4-20
labeling elements, 4-6
positioning and aligning buses, 4-29
PPR results, 4-28
prep for floorplanning command, 4-2
process menu, 6-17
RAM, 4-34
RAM example, 4-10
Index

ratsnest command, 6-17
ratsnest dialog box, 6-23
ratsnest option, 4-32
resources dialog box, 4-17
routing congestion, 4-28
RPM example, 4-10
RPM, RAM, ROM, 4-9
running, 4-6
setting boundaries, 4-8, 5-19
software requirements, 1-7
Task window, 4-3, 4-6
translate menu, 4-2
tristate buffer example, 4-13, 4-16
windows, 4-3
HDL design flow, 1-2
high-density designs, 4-1
MAP file
  creating, 1-16
  RPM, 5-8
  state machines, 4-5
  structured cells, 5-21
  Top design, 5-19
FPGA
  comparing to ASIC, 1-5, 2-1
  creating with HDLs, 3-1
  global clock buffer, 3-2
  system features, 1-6, 3-1
FPGA compiler, 1-5, 5-8
  CLB block names, 1-16
  software requirements, 1-7
fplan command, 4-6
FSM see Finite State Machine
functional simulation, 1-3, 1-4
  comparing to synthesis, 2-2
  HDL design flow, 1-1
G
gate count, 5-1, 5-4, 5-8
  gate reduction
  CLB count, 2-27
  definition, 2-25
delay, 2-27
design example, 2-26
gated clocks, 2-30
global buffer routing resources, 3-2
global clock buffer, 3-2
  inserting, 3-4
  instantiating, 3-4
global set/reset, 3-4
  no_gsr design, 3-7
  STARTUP block, 3-5
  use_gsr design, 3-10
  use_gsr_pre design, 3-12
group command, 5-15, 6-11
GSR see global set/reset
guided design, 6-26
  incremental, 6-27
  iterative, 6-27
H
hardware description language see HDL
HDL
  also see VHDL
  behavioral simulation, 1-3
coding for FPGAs, 3-1
coding hints, 2-1
  converting to gates, 1-4
definition, 1-1
design entry, 1-2
designing FPGAs, 1-3, 1-5
FPGA design flow, 1-1, 1-2
FPGA system features, 3-1
  boundary scan, 3-40
  global clock buffer, 3-2
  global set/reset, 3-4
  I/O decoders, 3-21
  implementing logic with IOBs, 3-42
  on-chip RAM, 3-36

HDL Synthesis for FPGAs
HDL Synthesis for FPGAs

guided design, 6-28
high-density design flow, 6-1
implementing latches, 2-16
implementing registers, 2-16
schematic entry design hints, 2-12
state machine
  binary encoded, 3-15
  enumerated type encoded, 3-17
  one-hot encoded, 3-18
type checking, 1-4
hdlin_check_no_latch command, 2-18
hierarchy in designs, 1-6, 5-1, 5-15
also see Top design
  defining, 5-15
  improving routability, 5-4
module size, 5-3, 5-11
PPR, 5-3
reducing debugging time, 5-4
reducing gate count, 5-4
reducing routing time, 5-4
small design changes, 5-4
Top design, 5-4
high-density design flow, 6-1, 6-4
high-density designs
  adding group TimeSpecs, 6-12
  building design hierarchy, 6-15
coding style, 6-7
estimating area utilization, 6-9
estimating design size, 6-4
evaluating results, 6-19
floorplanning design, 6-15
modifying hierarchy, 6-8
optimizing, 6-11
placing and routing, 6-17
synthesizing, 6-11
system features, 6-7
translating, 6-12
hold-time requirement, 2-17, 3-43
HP-PA requirements, 1-8
I/O decoder, 3-21
design example, 3-22
if statement, 2-18, 2-19
  comparing to case statement, 2-37
design example, 2-38
  registers, 2-19
  syntax, 2-31
  when to use, 2-32
if-case statement
design example, 2-36
if-else statement, 2-2, 2-31
ignore_maps option, 6-17, 6-18
ignore_rlocs option, 6-18
INC_DEC module, 3-26
incremental design, 6-27
indenting HDL code, 2-7
INIT=S attribute, 3-5, 3-8, 3-9, 3-20
initialization statement, 2-3
insert_pads command, 3-4
installation
design examples, 1-6
directory tree structure, 1-12
disk space requirements, 1-8
downloading files, 1-8, 1-9
extracting downloaded files, 1-11
file transfer protocol, 1-10
HP-PA requirements, 1-8
internet site, 1-7, 1-8, 1-10
known issues, 1-14
Perl, 1-6
software requirements
  Floorplanner, 1-7
  Foundry, 1-7
  Synopsys FPGA compiler, 1-7
  XACTstep, 1-7
  XC4025 die files, 1-7
  XSI, 1-7
SPARC requirements, 1-8
Synopsys startup file, 1-14
Index

tactical software, 1-6
AddTNM, 1-6
MakeTNM, 1-6
X-BLOXGen, 1-6
technical bulletin board, 1-7, 1-9, 1-11
technical support, 1-14
XSI release notes, 1-6, 1-8
XTBB, 1-7
internet site, 1-8
IOB
  implementing logic, 3-42
  initializing, 3-4
  moving registers, 3-46, 3-47
  unbonded, 3-48
  unbonded design example, 3-49
iterative design, 6-27

J
JTAG 1149.1, 3-40

K
known issues, 1-14

L
latch
  combinatorial feedback loop, 2-17
  comparing speed and area, 2-20
  converting to register, 2-18
  D flip-flop, 2-19
  D latch, 2-16
  D latch design, 2-17
  D latch implemented with gates, 2-17
  hdlin_check_no_latch command, 2-18
  implementing in HDL, 2-16
  inference, 2-32
  latch count, 2-18
  PPR, 2-18
  RAM primitives, 2-19
LCA file, 6-17
lock_routing option, 6-29

M
macro
hard, 3-32, 4-9
soft, 3-32
MakeTNM, 3-55, 3-57, 6-14
flow diagram, 3-58
MAP file, 1-16, 6-15
  creating, 4-2
  outfile option, 1-16
max_period command, 3-54
MemGen, 3-36, 3-38, 4-10
memory description file, 3-38
module-endmodule statement, 1-15
multiplexer
  comparing gates and tristate buffer, 3-53
  implementing with gates, 3-51
  implementing with tristate buffer, 3-50, 3-52
  resource sharing, 2-21

N
named association, 2-6
naming conventions, 2-4, 2-5
nested if statement
  design example, 2-34
  when to use, 2-33
no_gsr design, 3-7
NODELAY attribute, 3-43

O
one-hot encoded state machine, 3-17
outfile option, 1-16, 6-18

P
parallel logic, 2-37
Partition, Place, and Route see PPR
partitioning designs, 5-2
Perl, 1-6, 3-57, 3-63
placer_effort option, 6-17, 6-18
port declarations, 2-9
positional association, 2-6
PPR, 1-3, 2-18
  constraints file, 3-56, 3-64, 6-29
  creating a MAP file, 4-2

HDL Synthesis for FPGAs
cstfile option, 3-64
design hierarchy, 5-3
floorplanning RPMs, 5-8
guided design, 6-26
HDL design flow, 1-2
improving performance, 4-1
LCA file, 6-17
lock_routing option, 6-29
options, 6-13, 6-18
report file, 6-6, 6-7, 6-17, 6-19
routing designs, 6-19
screen output, 6-6, 6-18, 6-19
terminating, 6-17
timing constraints, 3-64
Top design, 5-8, 5-22
prep for floorplanner command, 4-2, 6-16
preset pin, 2-27, 3-5, 3-8
preset signal, 3-11
priority-encoded logic, 2-33, 2-37

R
RAM
behavioral description, 4-10
floorplanning, 4-34
RAM/ROM
implementing in HDL, 3-36
ratsnest command, 6-17
ratsnest dialog box, 6-23
ratsnest option, 4-32
register
clear pin, 2-27
converting latch to register, 2-18
D register, 2-18
D register design, 2-18
if statement, 2-19
implementing in HDL, 2-16, 2-29
preset pin, 2-27
register transfer level see RTL
relationally placed module see RPM
relative location parameter see RLOC
remove_design command, 1-15
report_fpga command, 5-11, 6-9
report file, 6-5
resource sharing
CLB count, 2-25
definition, 2-20
delay, 2-25
design example, 2-21
disabling, 2-23
hdl_resource_allocation command, 2-23
hierarchy, 5-2
RLOC, 3-32, 5-8
RLOC_ORIGIN parameter, 3-32
ROM
instantiating in HDL, 3-37, 3-39
router_effort option, 6-17, 6-18
RPM, 3-32, 4-9, 5-8
creating, 3-32
design example, 3-35
floorplanning, 4-26, 4-27, 5-8
instantiating, 3-34
library, 3-33
RTL description of ROM, 3-37
RTL simulation, 3-5, 3-6, 3-9
before floorplanning, 1-3
definition, 1-4
FPGA design flow, 1-1

S
schematic entry design hints, 2-12
set_attribute command, 3-9, 3-20, 3-37
set_input_delay command, 3-54
set_max_delay command, 3-54
set_output_delay command, 3-54
set_pad_type command, 3-4
set_port_is_pad command, 3-4
set_resource_allocation command, 2-27
set_resource_implementation command, 2-27
SHIFT module, 3-26
signals, 2-10
software requirements, 1-7
SOLV_IT knowledge base, 2-8, 2-10
SPARC requirements, 1-8
STARTUP block, 3-5, 3-9
state machine, 3-13
  binary encoded, 3-14
    design example, 3-15
  bubble diagram, 3-14
  comparing encoding styles, 3-20
  encoding style summary, 3-18
  enumerated type encoded, 3-16
    design example, 3-17
  enumeration type, 3-18
  initializing, 3-20
  one-hot encoded, 3-17
    design example, 3-18
std_logic data type, 2-8
structured cells
  floorplanning, 5-21
  SXNF file, 6-12, 6-15
  Syn2XNF, 3-39, 3-47, 3-58, 6-5, 6-12, 6-14
Synopsys
  Design analyzer, 6-11
  Design compiler, 5-8
  DesignWare, 2-20, 2-25, 5-8, 5-16
  FPGA compiler, 5-8
    CLB block names, 1-16
      software requirements, 1-7
  startup file, 1-14
  timing analysis tool, 1-3
synthesis
  comparing to simulation, 2-2
  synthesis tools, 1-5
    design hierarchy, 5-2
T
tactical software
  installing, 1-6
TCK pin, 3-40
TDI pin, 3-40
TDO pin, 3-40
technical support, 1-14
template control file, 3-58

example, 3-60
TIMEGRP command, 3-55, 3-56, 3-64, 6-14
TIMEGRP primitive, 3-55
TIMESPEC command, 3-55, 3-56, 3-64, 6-14
TIMESPEC primitive, 3-55
timing
  also see XACT-Performance
    constraints, 1-6, 3-53, 3-55, 3-64, 6-13
      using constraints file, 3-55
      using PPR command line options, 3-54
      using synthesis tool, 3-53
  requirements, 1-6, 3-55
timing simulation, 1-3
HDL design flow, 1-2
TMS pin, 3-40
TNM
  adding to signals, 3-66
  AddTNM, 3-57
  attribute, 3-55
  MakeTNM, 3-57
TNM file, 3-55, 3-62, 3-64, 6-14
  creating your own, 3-65
  example, 3-63, 3-66
Top design, 5-4
  CLB utilization, 5-15
  comparing methodologies, 5-24
  debugging, 5-23
  flat, 5-7
    ratsnest, 5-7
  floorplanned modules, 5-20, 5-21
  floorplanning, 5-19, 6-22
  modified hierarchy, 5-11, 5-14, 6-11
  original hierarchy, 5-5, 5-8, 5-10
  placement, 5-23
    efficient, 6-26
    inefficient, 6-21
  PPR, 5-22
resource statistics, 5-6, 6-10
RPM floorplan, 5-9
script file, 5-19
speed requirements, 5-10
tristate buffer
comparing to gates, 3-53
design example, 4-15
implementing multiplexer, 3-50, 3-52
TS attribute, 3-56
TTL thresholds, 3-44
type checking, 1-4

U
unbonded IOBs, 3-48
design example, 3-49
ungroup_all command, 2-25, 5-7
use_gsr design, 3-8
use_gsr_pre design, 3-11

V
variables, 2-6, 2-10
Verilog, 1-7
instantiating XNF files, 1-15
module-endmodule statement, 1-15

VHDL
after xx ns statement, 2-2
also see HDL
arithmetic functions, 2-3
capitalization style, 2-4
case statement, 2-2
coding styles, 2-3
comments in code, 2-8
comparing signals and variables, 2-10
constants, 2-6
declaring ports, 2-9
definition, 1-5
decoding code, 2-7
if-else statement, 2-2
initialization statement, 2-3
labeling in code, 2-5
managing designs, 2-7
named association, 2-6
naming identifiers, 2-5
naming packages, 2-5
naming types, 2-5
position association, 2-6
simulation, 2-2
std_logic, 2-8
synthesis, 2-2
using empty lines, 2-7
using spaces, 2-8
variables, 2-6
wait for xx ns statement, 2-2
Xilinx naming conventions, 2-4
VHSIC Hardware Description Language
see VHDL

W
wait for xx ns statement, 2-2
wait until statement, 2-18
write constraints command, 6-17

X
XACT Design Editor, 5-23, 6-27
XACT-Performance, 3-53, 3-64, 6-13
also see timing
decreasing number of constraints, 3-54, 3-55
specifying timing constraints, 3-53
XACTstep
Foundry, 4-1, 6-15
software requirements, 1-7
X-BLOX, 4-9, 6-14
DesignWare, 2-21
definition, 5-8
instantiating modules, 3-25
log file, 3-26
X-BLOXGen, 3-26, 6-14
design example, 3-28
flow diagram, 3-27
options, 3-28
output files, 3-28
syntax, 3-27
XC3000, 5-3
XC3100, 5-3
Index

XC4000, 5-3
  CLB, 2-28
  global clock buffer, 3-2
  IOB, 3-43
  slew rate, 3-43, 3-45
XC4010, 5-1
XC4013, 5-1
XC4025, 5-1
XC4025 die files, 1-7
XDE, 5-23, 6-27
XDelay, 3-64
  report file, 3-65
Xilinx internet site, 1-7, 1-10
Xilinx netlist format see XNF
Xilinx Synopsys Interface see XSI
Xilinx technical bulletin board, 1-7, 1-9, 1-11
XMake
  creating a MAP file, 4-2
XNF
  definition, 1-1
XNF files
  instantiating in Verilog designs, 1-15
xnfout_constraints_per_endpoint
  command, 3-54, 3-58, 6-13
XNFPrep, 6-14
  timing constraints, 3-64
xor_sig design, 2-11
xor_var design, 2-12
XSI
  release notes, 1-6, 1-8
  software requirements, 1-7
XTBB, 1-7, 1-9, 1-11