Xilinx Synopsys Interface FPGA User Guide

Introduction

Getting Started

FPGA Compiler Tutorial

Design Compiler Tutorial

Using the FPGA Compiler

Using the Design Compiler

Simulating Your FPGA Design

Files, Programs, and Libraries

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

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.

Contents

Chapter 1 Introduction to the Xilinx Synopsys Interface

What Is XSI? 1-1
Design Compiler Versus FPGA Compiler 1-1
Xilinx Documentation Set 1-2
  XSI Documentation 1-2
  XACT Documentation 1-3

Chapter 2 Getting Started

Software Configuration 2-1
Verifying Software Installation 2-1
Modifying the Default Synopsys Startup File 2-3
  Using the FPGA Compiler 2-4
    Generic FPGA Compiler Startup File Contents 2-4
    Modifying the Search Paths 2-6
    Modifying the DesignWare Library Search Path 2-7
    Using Synlibs with the FPGA Compiler 2-7
  Using the Design Compiler for XC4000 Designs 2-8
    Generic Design Compiler Startup File Contents 2-9
    Modifying the Search Path 2-11
    Modifying the DesignWare Library Search Path 2-11
    Using Synlibs with the Design Compiler 2-11
  Using the Design Compiler for XC3000 Designs 2-12
    Generic Design Compiler Startup File Contents 2-13
    Modifying the Search Path 2-14
    Using Synlibs for XC3000 Devices 2-15

Chapter 3 FPGA Compiler Tutorial for XC4000 Designs

Before You Begin 3-1
  Required Files 3-1
  Exiting the Tutorial 3-2
Design Flow 3-3
Count8 Design Description 3-4
Invoking the Design Analyzer 3-6
Reading the Design File 3-8

Analyzing the Design File 3-8
Creating the Design File 3-10
Inserting I/O Buffers 3-13
Defining Input Ports as Pads 3-14
Defining the Output Port as a Pad 3-16
Using the Insert Pads Command 3-17
Estimating Pre-Layout Timing 3-18
Selecting the Operating Condition 3-18
Setting the Wire-Load Models.. 3-19
Optimizing for Speed... 3-19
Compiling the Design 3-21
Evaluating the Results 3-22
Viewing the Estimated Area Results...................................... 3-24
Viewing the Estimated Timing Results................................... 3-26
Saving the Area and Timing Results to a File........................ 3-27
Saving the Design 3-28
Writing the DB File 3-29
Replacing CLBs and IOBs with Gates 3-29
Setting the Design Part Type 3-29
Removing BLKNM Attributes 3-29
Saving the Design File as an SXNF File 3-30
Exiting the Design Analyzer 3-31
Executing the Commands from a Script File 3-31
Placing and Routing Your Design Using XMake 3-34
If XSI Is on Same Network as XACT Software 3-35
If XSI Is on Different Network Than XACT Software............... 3-35
Running Syn2XNF 3-35
Running XMake 3-36
Examining XMake Output Files.. 3-37
Reviewing the XMake OUT File.. 3-37
Checking for Warnings and Errors in the PRP File............ 3-37
Checking the RPT File.. 3-38
Comparing Actual Versus Estimated Area Results 3-44
Using XDelay 3-45
Invoking XDelay 3-46
Comparing Actual Versus Estimated Timing Results 3-46
Verifying Your Design Using XChecker 3-47

Chapter 4 Design Compiler Tutorial for XC3000A Designs

Before You Begin 4-1
Required Files... 4-1

- Exiting the Tutorial 4-2
Design Flow 4-3
Count8 Design Description 4-4
Invoking the Design Analyzer 4-5
Reading the Design File 4-7
- Analyzing the Design File 4-7
- Creating the Design File 4-9
Inserting I/O Buffers 4-12
- Defining Input Ports as Pads 4-13
- Defining the Output Port as a Pad 4-15
- Using the Insert Pads Command 4-16
Estimating Pre-Layout Timing 4-16
- Selecting the Operating Condition 4-17
- Setting the Wire-Load Models 4-17
Optimizing for Speed 4-17
Compiling the Design 4-19
Evaluating the Results 4-20
- Viewing the Estimated Area Results 4-22
- Viewing the Estimated Timing Results 4-25
- Saving the Report Results to a File 4-25
Saving the Design 4-28
- Writing the DB File 4-28
- Setting the Design Part Type 4-28
- Saving the Design File as an SEDIF File 4-28
Exiting the Design Analyzer 4-29
Executing the Commands from a Script File 4-30
Placing and Routing Your Design Using XMake 4-33
- If XSI Is on Same Network as XACT Software 4-34
- If XSI Is on Different Network Than XACT Software 4-34
  - Running Syn2XNF 4-34
  - Running XMake 4-35
- Examining XMake Output Files 4-35
  - Reviewing the XMake OUT File 4-36
  - Checking for Warnings and Errors in the PRP File 4-36
  - Checking the RPT File 4-37
  - Comparing Actual Versus Estimated Area Results 4-42
Using XDelay 4-43
- Invoking XDelay 4-44
- Comparing Actual Versus Estimated Timing Results 4-45
Verifying Your Design Using XChecker 4-45

Chapter 5 Using the FPGA Compiler

Before You Begin 5-2
FPGA Compiler Design Flow 5-2
XSI on Same Platform as XACT Software 5-2
XSI on Different Platform than XACT Software 5-3
Setting the Wire-Load Model 5-5
Wire-Load Models for Xilinx FPGAs 5-5
Changing the Wire-Load Model 5-6
How Wire-Load Models Are Determined 5-7
Operating Conditions 5-7
Configuring IOBs 5-7
XC4000/A/D IOBs 5-8
Inputs 5-8
Outputs 5-9
XC4000/D Slew Rate 5-9
XC4000A Slew Rate 5-10
XC4000H IOBs 5-11
Inputs 5-12
Outputs 5-13
XC4000H Slew Rate 5-14
Assigning and Prohibiting Pad Locations 5-15
Implementing 3-State Registered Output 5-15
Not Directly Driving the 3-State Signal 5-15
Directly Driving the 3-State Signal 5-17
Inserting Bidirectional I/Os 5-19
Instantiating a Registered Bidirectional I/O 5-20
Compiling Bidirectional I/O 5-21
Using Unbonded IOBs (XC4000/A Only) 5-25
Adding Pull-Up and Pull-Down Resistors 5-25
Removing the Default Input Delay 5-26
Initializing the IOB Flip-Flop to Preset 5-26
Inserting Clock Buffers 5-26
Controlling Clock Buffer Insertion 5-27
Determining the Number of Clock Buffers 5-30
Preventing the Insertion of Clock Buffers 5-30
Using Memory 5-31
XC4000 RAMs 5-31
XC4000 ROMs 5-32
Using MemGen 5-34
Performing Boundary Scan 5-37

Using the Global Set/Reset Net 5-38
Startup State 5-38
Preset Versus Direct Clear 5-39
Changing States 5-40
Increasing Performance with the GSR Net 5-40
Using the X-BLOX DesignWare Library 5-47
HDL Operators Using X-BLOX Modules 5-47
Improving the Timing of X-BLOX Modules 5-48
Creating Timing Specifications 5-50
Setting Timing Constraints 5-50
Create Specifications for Input Ports and Clock Net 5-50
Create Specifications for Input and Output Ports 5-51
Create Tighter Constraints on Output Ports 5-51
Create Tighter Constraints on Input Ports 5-51
Prevent Specifications on Indicated Paths 5-52
Create Clocks on All Input Ports 5-52
Controlling How Timing Specifications Are Written 5-52
Control the Number of Constraints Written 5-52
Create Default Timing Constraints 5-53
Compiling the Design 5-53
Optimizing Logic Across Hierarchical Boundaries 5-54
Flattening the Design 5-54
Compiling the Design with Hierarchy 5-56
Compiling the Design Without Hierarchy 5-56
Creating Unique Names for Multiple Instances 5-57
Compiling a Design That Contains Feedthroughs 5-57
Compiling a Design with Instantiated I/O Cells 5-57
Compiling XC4000 Designs 5-57
Compiling XC4000H Designs 5-60
Creating the Area Report 5-64
Evaluating Timing Delays 5-65
Generating Reports for Debugging 5-66
Generating a Configuration Report 5-66
Generating a Hierarchical Schematic 5-69
Creating a Level for Each CLB and IOB 5-70
Creating a Level for Each Function Generator 5-70
Writing and Saving the Design 5-71
Saving the DB File 5-71
Replacing CLBs and IOBs with Gates 5-72
Invoking the Replace FPGA Command 5-72
If Your Design Contains Hierarchy 5-72

Removing the Synopsys Mapping 5-73
Removing FMAP and HMAP Symbols 5-73
Removing BLKNM Attributes 5-73
Setting the Design Part Type 5-74
Saving the SXNF File 5-74
Translating SXNF Files to XNF Files Using Syn2XNF 5-74
Syntax 5-75
Input Files 5-75
Output Files 5-75
Options 5-76
–dir 5-76
–force 5-76
–help 5-76
–l 5-76
–out 5-77
–parttype 5-77
Using the XACT Development System 5-77
If XSI Is on Same Platform as XACT Software 5-78
If XSI Is on Different Platform Than XACT Software 5-78

Chapter 6 Using the Design Compiler

Before You Begin 6-1
Design Compiler Design Flow 6-1
If XSI Is on Same Platform as XACT Software 6-2
If XSI Is on Different Platform Than XACT Software 6-2
Setting the Wire-Load Model 6-3
Wire-Load Models for Xilinx FPGAs 6-4
Changing the Wire-Load Model 6-5
How Wire-Load Models Are Determined 6-6
Operating Conditions 6-6
Configuring the IOBs 6-6
XC4000/A/D IOBs 6-7
Inputs 6-7
Outputs 6-8
XC4000/D Slew Rate 6-8
XC4000A Slew Rate 6-9
XC4000H IOBs 6-10
Inputs 6-10
Outputs 6-11
XC4000H Slew Rate 6-12
XC3000/A/L and XC3100/A IOBs 6-13

Inputs ... 6-14
Outputs ... 6-14
XC3000/A/L and XC3100/A Slew Rate ... 6-14
Assigning and Prohibiting Pad Locations ... 6-15
Implementing 3-State Output ... 6-15
Not Directly Driving the 3-State Signal ... 6-15
Directly Driving the 3-State Signal ... 6-17
Inserting Bidirectional I/Os ... 6-19
Instantiating a Registered Bidirectional I/O ... 6-19
Compiling Bidirectional I/O ... 6-21
Using Unbonded IOBs ... 6-25
Adding Pull-Up and Pull-Down Resistors ... 6-25
Removing the Default Input Delay (XC4000 Only) ... 6-26
Initializing the IOB Flip-Flop to Preset (XC4000 Only) ... 6-26
Inserting Clock Buffers ... 6-26
XC4000/A/D/H Clock Buffers ... 6-27
XC3000/A/L and XC3100/A Clock Buffers ... 6-27
Controlling Clock Buffer Insertion ... 6-28
Determining the Number of Clock Buffers ... 6-31
Preventing the Insertion of Clock Buffers ... 6-31
Using Memory ... 6-32
XC4000 RAMs ... 6-32
ROMs ... 6-33
Using MemGen ... 6-35
Performing Boundary Scan for XC4000 Devices ... 6-38
Using the Global Set/Reset Net ... 6-39
XC4000 Devices ... 6-39
Startup State ... 6-39
Preset Versus Direct Clear ... 6-40
Changing States ... 6-41
Increasing Performance with the GSR Net ... 6-42
XC3000 and XC3100 Devices ... 6-48
Using the X-BLOX DesignWare Library ... 6-54
HDL Operators Using X-BLOX Modules ... 6-54
Improving the Timing of X-BLOX Modules ... 6-55
Compiling the Design ... 6-57
Compiling a Design That Contains Feedthroughs ... 6-58
Compiling XC3000 and XC4000 Designs ... 6-58
Compiling a XC4000H Design ... 6-61
Creating the Area Report ... 6-65
Evaluating Timing Delays ... 6-66

Writing and Saving the Design 6-67
Saving the DB File 6-67
Setting the Design Part Type 6-68
Saving the SEDIF File 6-68
Translating SEDIF Files to XNF Files Using Syn2XNF 6-68
Syntax 6-69
Input Files 6-69
Output Files 6-70
Options 6-70
–dir 6-70
–force 6-70
–help 6-70
–l 6-70
–map 6-71
–out 6-71
–parttype 6-71
–sub 6-71
Using the XACT Development System 6-72
If XSI Is on Same Platform as XACT Software 6-72
If XSI Is on Different Platform Than XACT Software 6-72

Chapter 7 Simulating Your FPGA Design

Recommended FPGA Simulation Strategy 7-1
Editing the VSS Setup File 7-2
Check Your Source File 7-3
Controlling Initial States of Registers 7-3
Simulating Global Set/Reset 7-3
Preparing for Timing Simulation 7-4
Preparing for Functional Simulation 7-5
Creating a Test Bench File 7-6
Initializing Registers 7-6
Configuration Declaration 7-8
Functional Simulation 7-9
Design Implementation 7-12
Preparing the Timing Model 7-14
Timing Simulation 7-14

Chapter 8 Files, Programs, and Libraries

Directory Structure for XSI 8-1
File Descriptions 8-4
Program Descriptions 8-5

Library Descriptions 8-6
Supported Part Types and Speed Grades............................. 8-9
xprim_family–s.db and xprim_parttype–s.db 8-9
xio_4kparttype–s.db 8-11
xfpga_family–s.db... 8-11
xdc_family–s.db... 8-12
Unsupported Part Types and Speed Grades......................... 8-12

Appendix A XC3000/A/L and XC3100/A Primitives

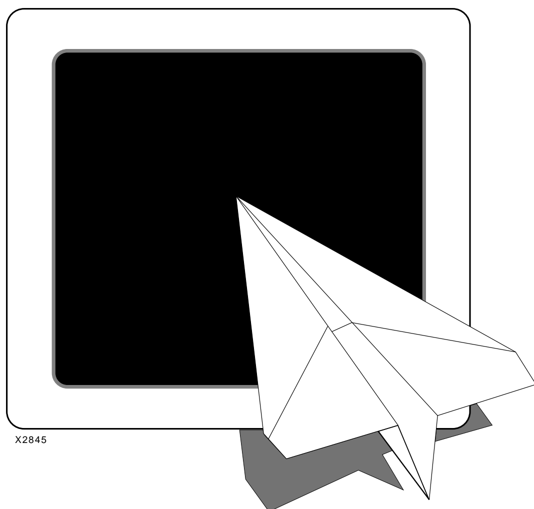
XC3000 Primitives A-2
Basic Gates A-3
Flip-Flops and Latches A-5
Clocks A-5
Oscillators A-6
I/O Primitives A-7
Special Functions A-9

Appendix B XC4000/A/D/H Primitives and Hard Macros

XC4000 Primitives B-3
Basic Gates B-3
Flip-Flops and Latches B-6
Clocks B-8
I/O Primitives B-8
Special Functions B-16
X-BLOX Modules B-19
XC4000 Hard Macros B-20

Appendix C Selection Guide

XC3000/A/L and XC3100/A Primitives.. C-1
XC4000/A/D/H Primitives.. C-5



Introduction

Xilinx Synopsys Interface FPGA User Guide

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Chapter 1

Introduction to the Xilinx Synopsys Interface

This chapter introduces XSI, discusses the two compiler options, FPGA Compiler and Design Compiler, and describes the XSI documentation set.

What Is XSI?

The XSI design tool kit allows you to implement Xilinx Field Programmable Gate Array (FPGA) designs using either the Synopsys FPGA Compiler or Design Compiler synthesis tool. These Synopsys High-Level Design Automation (HLDA) tools create and optimize circuit designs from hardware descriptions written in VHSIC Hardware Description Language (VHDL) or Verilog HDL.

Library support for the XC4000 family also includes a DesignWare™ library that maps adder/subtracter, comparator, and incrementer/decrementer functions to appropriate X-BLOX™modules. X-BLOX implements these functions using features in the XC4000 family such as fast carry logic. X-BLOX is included in all standard software packages.

Before starting a Xilinx design with Synopsys, read the next chapter, "Getting Started."

Design Compiler Versus FPGA Compiler

XSI contains libraries for the XC3000/A/L, XC3100/A, and XC4000/A/D/H families. You can use either the FPGA Compiler or Design Compiler to synthesize a design for all Xilinx devices.

The Design Compiler (V3.1*x* or later) provides the following features.

- Optimizes flip-flops and latches in the input/output block (IOB)
- Optimizes 3-state buffers in the IOB
- Encodes one-hot state machines
- Uses the configurable logic block (CLB) Clock Enable pin automatically

In addition to the features provided in the Design Compiler, the FPGA Compiler delivers more efficient results and more accurate timing and area reporting as follows.

- Optimizes logic to the XC4000 family CLB and IOB architectures
- Reports area and timing by device architecture, for example, CLB, IOB, and 3-state buffer
- Passes timing constraints to the XACT-Performance™ utility
- Reads XNF (Xilinx Netlist Format) reader files for design reuse and back-annotation of post-route results

Note: This manual assumes that you are using the FPGA Compiler synthesis tools for XC4000 devices. If you do not have the FPGA Compiler, XSI provides XC4000 libraries that you can use with the Design Compiler. You can use the FPGA Compiler for XC3000 and XC3100 devices; however, the libraries for these devices use the Design Compiler synthesis features.

Xilinx Documentation Set

The Xilinx documentation set consists of a series of books that help you use the XACT® Development System with your Synopsys tools.

XSI Documentation

The XSI documentation set includes the following manuals.

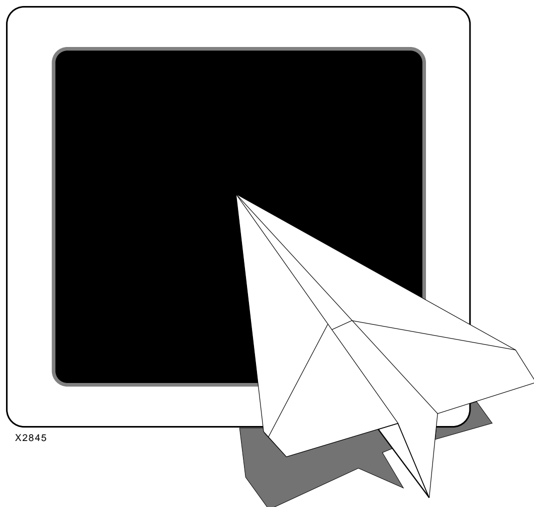
- *XSI Release Notes* provide detailed instructions on installing the XSI software and additional platform-specific information, as well as known issues and workarounds.
- *Xilinx Synopsys Interface FPGA User Guide*, this guide, contains information on how to use your Synopsys tools with the XACT Development System to create FPGA designs. The “Preface” describes the contents of each chapter.

- *Xilinx Synopsys Interface EPLD User Guide* contains information on how to use your Synopsys tools with the XACT Development System to create EPLD designs.

XACT Documentation

The XACT documentation set includes the following manuals.

- *XACT User Guide* contains an overview of the XACT Development software, including general design implementation flows and configuration hints.
- *XACT Reference Guide* provides detailed information on the programs that XMake invokes during the design implementation and design verification stages.
- *X-BLOX User Guide* describes the X-BLOX synthesis tool, which consists of a library of modules you can use to describe high-level functions.
- *XACT Libraries Guide* presents information about the various Xilinx-provided primitives and macros.



Getting Started

Xilinx Synopsys Interface FPGA User Guide

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Chapter 2

Getting Started

This chapter enables you to verify that the Xilinx and XSI software is installed. This chapter also describes how to do the following.

- Modify your Synopsys startup file, .synopsys_dc.setup
- Use the Synlibs program to determine the correct XSI libraries for the FPGA Compiler or Design Compiler

Note: Read this chapter before you begin the FPGA Compiler tutorial or Design Compiler tutorial.

Software Configuration

Your XSI software (DS-401) must be installed on the same platform as the Synopsys software. However, the XACT software (DS-502) can be installed on the same network or on a different network (platform); for example, your XACT software might reside on a PC while your XSI and Synopsys tools reside on a UNIX-based workstation. For more information, consult the installation section of the release notes or your system administrator.

Verifying Software Installation

This section enables you to verify that XACT, X-BLOX, and XSI are installed on your system and that your .cshrc and .login files include the required environmental variables and search paths.

1. Go to the platform where the XACT software is installed.
2. To verify that your system has the XACT Development System software (DS-502), type **`which ppr`**↵ at the system prompt.

The full path for PPR appears onscreen. If the system cannot find PPR, refer to the installation instructions in the release notes or see your system administrator.

3. To verify that X-BLOX (DS-380) has been installed, type **`which xblox`** ↵ at the system prompt.

   The full path for X-BLOX appears onscreen. If the system cannot find X-BLOX, refer to the installation instructions in the release notes.

4. To verify that XSI (DS-401) has been installed, type **`which syn2xnf`** ↵ at the system prompt.

Note: If XSI is installed on a different platform, go to that platform before executing the Syn2XNF command.

   The full path for XSI appears onscreen. If the system cannot find Syn2XNF, refer to the installation instructions in the release notes or see your system administrator.

5. Change to the following directory.

   cd *DS401-directory*/**synopsys/libraries/dw/lib/fpga**

   DS401-Directory is the directory where XSI is installed.

6. List the contents of this directory to verify that the source X-BLOX DesignWare files were placed in this directory during installation.

   This directory should contain the object file for the X-BLOX DesignWare symbol modules (*xblox_dw_module*.syn) and the simulation modules (*xblox_dw_module*.sim).

Note: The variable *xblox_dw_module* refers to the X-BLOX DesignWare primitive name.

   If you do not find the SYN and SIM files in this directory, refer to the release notes or see your system administrator. The README file contains installation instructions and is located in the *DS401-Directory*/synopsys/libraries/dw/src/fpga directory. Refer to the installation notes for instructions on how to analyze the X-BLOX DesignWare modules.

Modifying the Default Synopsys Startup File

The .synopsys_dc.setup file is the startup file for the Synopsys synthesis tools. This file contains the search path for the XSI libraries, Synopsys libraries, and user libraries. XSI provides a default Synopsys startup file.

This section describes how to modify this default setup file to include the path to the Xilinx link and target libraries for the FPGA Compiler and Design Compiler as well as the other required libraries.

XSI provides a default Synopsys startup file in the following directory.

DS401-directory`/synopsys`

DS401-Directory is the directory where XSI is installed. If you do not know the location of this directory, type the following at the system prompt.

```
echo $XACT
```

The system displays the paths set for the XACT environment variable and lists the path for XSI first.

If you already have a .synopsys_dc.setup file, you must modify your file to include the commands found in the Xilinx-supplied default startup file.

If you do not already have a Synopsys startup file, copy the appropriate Xilinx-supplied startup file to your home or working directory and rename it as follows.

```
cp DS401-Directory/synopsys/tech.synopsys_dc.setup \
  .synopsys_dc.setup
```

Substitute *tech* with one of the following options.

- `fc4k` if you are using the FPGA Compiler
- `dc4k` if you are using the Design Compiler with XC4000 devices
- `dc3k` if you are using the Design Compiler with XC3000/A/L or XC3100/A devices

The following sections describe how to modify your setup file for the selected compiler.

Using the FPGA Compiler

The FPGA Compiler requires five libraries for synthesis to Xilinx devices. The libraries are separated to reduce disk space.

This section describes how to modify the search path, modify the DesignWare library search path, and use Synlibs to display the appropriate target and link libraries.

Figure 2-1 is an example of the default startup file provided with the XSI software — fc4k.synopsys_dc.setup. Refer to your Synopsys documentation for more information about the Synopsys startup file.

```
/* EXAMPLE FPGA COMPILER STARTUP FILE - .synopsys_dc.setup */
/* FOR XC4000/A/H/D PARTYPES                               */

search_path = { .       \
        <DS401-XACT-Directory>/synopsys/libraries/syn \
        <SYNOPSYS_Directory>/libraries/syn}

link_library = {xprim_4005-5.db xprim_4000-5.db xgen_4000.db \
        xio_4000-5.db xfpga_4000-5.db}

target_library = {xprim_4005-5.db xprim_4000-5.db xgen_4000.db \
        xio_4000-5.db xfpga_4000-5.db}

symbol_library = xc4000.sdb

define_design_lib WORK -path ./WORK

define_design_lib xblox_4000 -path \
        <DS401-XACT-Directory>/synopsys/libraries/dw/lib/fpga

synthetic_library = {xblox_4000.sldb standard.sldb}

compile_fix_multiple_port_nets = true

xlnx_hier_blknm = 1

xnfout_library_version = "2.0.0"

bus_naming_style = "%s<%d>"
bus_dimension_separator_style = "><"
bus_inference_style = "%s<%d>"
```

Figure 2-1 Synopsys Startup File

Generic FPGA Compiler Startup File Contents

This section describes the sample .synopsys_dc.setup file illustrated in Figure 2-1.

- ```
  search_path = { . \
     <DS401-XACT-Directory>/synopsys/libraries/syn \
     <SYNOPSYS-Directory>/libraries/syn}
  ```

  This line sets the search path for Xilinx and Synopsys-supplied library files.

- ```
  link_library = {xprim_4005-5.db xprim_4000-5.db xgen_4000.db
      xio_4000-5.db xfpga_4000-5.db}

  target_library = {xprim_4005-5.db xprim_4000-5.db \
     xgen_4000.db xio_4000-5.db xfpga_4000-5.db}
  ```

  These lines specify the *default* target and link libraries that indicate which compiler is used. The sample libraries are for an XC4005–5 device. You can use these libraries for the FPGA Compiler tutorial.

  The Synlibs program determines the correct libraries for the different device types and speed grades. Refer to the "Using Synlibs with the FPGA Compiler" section that follows for more information on how to use Synlibs to change the link and target libraries to a different part type and speed grade.

  The target and link libraries are device-specific. Therefore, you might not want to specify target and link libraries in a generic startup file.

- ```
  symbol_library = xc4000.sdb
  ```

  This line specifies the symbol libraries.

- ```
  define_design_lib WORK –path ./WORK
  ```

  This line creates a directory to store intermediate files created by the Analyze command for VSS users.

- ```
  define_design_lib xblox_4000 –path
     <DS401-XACT-Directory>/synopsys/libraries/dw/lib/fpga
  ```

  This line specifies the directory where the X-BLOX DesignWare components reside.

- ```
  synthetic_library = {xblox_4000.sldb standard.sldb}
  ```

  This line specifies the synthetic library.

- `compile_fix_multiple_port_nets = true`

  This line allows the Synopsys optimization algorithm to insert extra logic into the design to ensure that there are no feedthroughs and that no two output ports are connected to the same net.

- `xlnx_hier_blknm = 1`

  This line creates unique names for each instance of the sub-module for hierarchical design that have more than one instance of the same module.

- `xnfout_library_version = “2.0.0”`

  This line allows Synopsys to write a version 5 XNF file. By default, Synopsys writes a version 4 XNF file. You must specify this command if you are using XSI V3.1 or later libraries and XACT V5.0 or later software.

- ```
  bus_naming_style = "%s<%d>"
  bus_dimension_separator_style = "><"
  bus_inference_style = "%s<%d>"
  ```

  These lines set the READ parameters for the Xilinx netlist formats. All bus indexing is of the form “bus<index>.”

Modifying the Search Paths

Modify the search path line in the default .synopsys_dc.setup file to include the path to the XSI and Synopsys installation directories as follows.

1. To determine the Synopsys installation path, enter the following at the command line.

   `echo $SYNOPSYS`

2. Open the setup file in a text editor and replace *DS401-Directory* with the full path of the directory where the XSI software is installed (the $XACT environment variable contains this path).

3. Replace *SYNOPSYS-Directory* with the path where your Synopsys software is installed (usually stored in the environment variable $SYNOPSYS).
  ```

Modifying the DesignWare Library Search Path

The X-BLOX DesignWare library contains descriptions of adders, subtracters, comparators, incrementers, and decrementers that map to X-BLOX modules. X-BLOX also generates Xilinx-optimized implementations of common functions. If you have the X-BLOX package, follow this procedure.

Modify the following line in your .synopsys_dc.setup file to use the X-BLOX DesignWare library.

```
define_design_lib xblox_4000 -path \
  <DS401-XACT-Directory>/synopsys/libraries/dw \
  /lib/fpga
```

The *DS401-XACT-Directory* is the full path of the directory where the XSI software is installed (the $XACT environment variable contains this path).

If you *do not* use X-BLOX, comment out or remove the following lines in the Synopsys startup file. Enclose your comments with a /* (slash, asterisk) and an */ (asterisk, slash), as illustrated by the following example.

```
/* define_design_lib xblox_4000 -path \
  <DS401-XACT-Directory>/synopsys/libraries/dw \
  /lib/fpga */

/* synthetic_library = {xblox_4000.sldb \
  standard.sldb} */
```

Using Synlibs with the FPGA Compiler

Synlibs displays the link and target libraries for the specified part type and speed grade. You can run Synlibs from any directory as follows.

synlibs *parttype–speedgrade*

For example, to list the link and target libraries for the XC4005–5 device, you would enter the following.

```
synlibs 4005-5
```

The system displays the output onscreen, as illustrated in Figure 2-2.

```
link_library = {xprim_4005-5.db xprim_4000-5.db \
   xgen_4000.db xio_4000-5.db xfpga_4000-5.db}

target_library = {xprim_4005-5.db xprim_4000-5.db \
   xgen_4000.db xio_4000-5.db xfpga_4000-5.db}
```

Figure 2-2 Synlibs Output for Use with FPGA Compiler

Note: Use the target and link libraries in the default startup file if you plan to perform the FPGA Compiler tutorial.

You must copy the output from Synlibs into your Synopsys startup file. You can use the UNIX Append (>>) command to redirect the output of Synlibs to your .synopsys_dc.setup file as follows.

synlibs *parttype–speedgrade* **>> .synopsys_dc.setup**

After you redirect the output, use a text editor to delete the default target and link libraries.

Warning: You must list the libraries in your setup file in the order that they appear in the Synlibs output.

Using the Design Compiler for XC4000 Designs

The Design Compiler requires five libraries for synthesis to Xilinx XC4000 devices. The libraries are separated to reduce disk space.

This section describes how to modify the search path, modify the DesignWare Library search path, and use Synlibs to display the appropriate target and link libraries.

Figure 2-3 is an example of the generic startup file for an XC4000 design, dc4k.synopsys_dc.setup, using the Design Compiler.

```
/* EXAMPLE DESIGN COMPILER STARTUP FILE - .synopsys_dc.setup */
/* FOR XC4000/H/A/D PARTYPES                                  */

search_path = { .       \
        <DS401-XACT-Directory>/synopsys/libraries/syn \
        <SYNOPSYS-Directory/libraries/syn}

link_library = {xprim_4005-5.db xprim_4000-5.db xgen_4000.db \
        xdc_4000-5.db xio_4000-5.db}

target_library = {xprim_4005-5.db xprim_4000-5.db xgen_4000.db \
        xdc_4000-5.db xio_4000-5.db}

symbol_library = xc4000.sdb

define_design_lib WORK -path ./WORK

define_design_lib xblox_4000 -path \
        <DS401-XACT-Directory>/synopsys/libraries/dw/lib/fpga

synthetic_library = {xblox_4000.sldb standard.sldb}

compile_fix_multiple_port_nets = true

bus_naming_style = "%s<%d>"
bus_dimension_separator_style = "><"
bus_inference_style = "%s<%d>"

edifout_netlist_only = true
edifout_power_and_ground_representation = cell
edifout_write_properties_list = "instance_number port_location part"
```

Figure 2-3 Generic XC4000 Design Compiler Startup File

Generic Design Compiler Startup File Contents

This section describes the sample .synopsys_dc.setup file illustrated in Figure 2-3.

- ```
  search_path = { . \
     <DS401-XACT-Directory>/synopsys\libraries/syn \
     <SYNOPSYS-Directory>/libraries/syn}
  ```

  This line sets the search path for Xilinx and Synopsys-supplied library files.

- ```
  link_library = {xprim_4005-5.db xprim_4000-5.db \
     xgen_4000.db xdc_4000-5.db xio_4000-5.db}

  target_library = {xprim_4005-5.db xprim_4000-5.db \
     xgen_4000.db xdc_4000-5.db xio_4000-5.db}
  ```

  These lines specify the *default* target and link libraries, which are for an XC4005–5 device.

The Synlibs program determines the correct libraries for the different device types and speed grades. Refer to the “Using Synlibs with the Design Compiler” section that follows for more information on how to use Synlibs to change the link and target libraries to a different part type and speed grade.

The target and link libraries are device-specific. Therefore, you might not want to specify target and link libraries in your generic startup file.

- `symbol_library = xc4000.sdb`

  This line specifies the symbol libraries.

- `define_design_lib WORK –path ./WORK`

  This line creates a directory to store intermediate files created by the Analyze command for VSS users.

- ```
  define_design_lib xblox_4000 –path \
     <DS401-XACT-Directory>/synopsys/libraries/dw/lib/fpga
  ```

  This line specifies the directory where the X-BLOX DesignWare components reside.

- `synthetic_library = {xblox_4000.sldb standard.sldb}`

  This line specifies the synthetic library.

- `compile_fix_multiple_port_nets = true`

  This line allows the Synopsys optimization algorithm to insert extra logic into the design to ensure that there are no feedthroughs and that no two output ports are connected to the same net.

- ```
  bus_naming_style = "%s<%d>"
  bus_dimension_separator_style = "><"
  bus_inference_style = "%s<%d>"
  ```

  These lines set the READ parameters for the Xilinx netlist formats. All bus indexing is of the form ‘‘bus<index>.’’ ’

- ```
  edifout_netlist_only = true
  edifout_power_and_ground_representation = cell
  edifout_write_properties_list = “instance_number
      port_location part”
  ```

  These lines set the EDIF parameters for Xilinx devices.
  ```

Modifying the Search Path

Modify the search path line in the default .synopsys_dc.setup file to include the path to the XSI and Synopsys installation directories as follows.

1. To determine the Synopsys installation path, enter the following at the command line.

   echo $SYNOPSYS

2. Open the setup file in a text editor and replace *DS401-Directory* with the full path of the directory where the XSI software is installed (the $XACT environment variable contains this path).

3. Replace *SYNOPSYS-Directory* with the path where your Synopsys software is installed (usually stored in the environment variable $SYNOPSYS).

Modifying the DesignWare Library Search Path

The X-BLOX DesignWare library contains descriptions of adders, subtracters, comparators, incrementers, and decrementers that map to X-BLOX modules. X-BLOX also generates Xilinx-optimized implementations of common functions. If you have the X-BLOX package, add the following line to your .synopsys file to use the X-BLOX DesignWare library.

```
define_design_lib xblox_4000 –path \
  <DS401-XACT-Directory>\synopsys\libraries/dw/ \
  /lib/fpga
```

Verify that your .synopsys_dc.setup file contains the following statement.

```
synthetic_library = {xblox_4000.sldb \
  standard.sldb}
```

Using Synlibs with the Design Compiler

Synlibs displays the link and target libraries for the specified part type and speed grade. You can run Synlibs from any directory as follows.

synlibs –dc *parttype–speedgrade*

You must specify the –dc option to list the link and target libraries for use with the Design Compiler.

For example, to list the link and target libraries for an XC4005–5 device for use with the Design Compiler, you would enter the following.

```
synlibs –dc 4005–5
```

The system displays the output onscreen as illustrated by Figure 2-4.

```
link_library = {xprim_4005-5.db xprim_4000-5.db \
   xgen_4000.db xdc_4000-5.db xio_4000-5.db}

target_library = {xprim_4005-5.db xprim_4000-5.db \
   xgen_4000.db xdc_4000-5.db xio_4000-5.db}
```

Figure 2-4 Synlibs Output for Use with Design Compiler (XC4000)

You must copy the output from Synlibs into your Synopsys startup file. You can use the UNIX Append (>>) command to redirect the output of Synlibs to your .synopsys_dc.setup file as follows.

synlibs –dc ***parttype–speedgrade*** **>> .synopsys_dc.setup**

After you redirect the output, use a text editor to delete the default target and link libraries.

Warning: You must list the libraries in your setup file in the order that they appear in the Synlibs output.

Using the Design Compiler for XC3000 Designs

The Design Compiler requires four libraries for synthesis to Xilinx XC3000/A/L and XC3100/A devices. The libraries are separated to reduce disk space.

This section describes how to modify the search path and use Synlibs to display the appropriate target and link libraries.

Figure 2-5 is a example of the generic startup file, dc3k.synopsys_dc.setup, for an XC3000A design using the Design Compiler.

```
/* EXAMPLE DESIGN COMPILER STARTUP FILE - .synopsys_dc.setup */
/* For XC3000/A/L and XC3100/A/L PARTYPES                    */

search_path = {.        \
        <DS401-XACT-Directory>/synopsys/libraries/syn \
        <SYNOPSYS-Directory/libraries/syn}

link_library = {xprim_3020a-6.db xprim_3000a-6.db \
        xgen_3000.db xdc_3000a-6.db}

target_library = {xprim_3020a-6.db xprim_3000a-6.db \
        xgen_3000.db xdc_3000a-6.db}

symbol_library = xc3000.sdb

define_design_lib WORK -path ./WORK

compile_fix_multiple_port_nets = true

bus_naming_style = "%s<%d>"
bus_dimension_separator_style = "><"
bus_inference_style = "%s<%d>"

edifout_netlist_only = true
edifout_power_and_ground_representation = cell
edifout_write_properties_list = "instance_number port_location part"
```

Figure 2-5 Generic XC3000A Design Compiler Startup File

Generic Design Compiler Startup File Contents

This section describes the sample .synopsys_dc.setup file illustrated in Figure 2-5.

- ```
  search_path = { . \
     <DS401-XACT-Directory>/synopsys/libraries/syn \
     <SYNOPSYS-Directory>/libraries/syn}
  ```

  This line sets the search path for Xilinx and Synopsys-supplied library files.

- ```
  link_library = {xprim_3020a-6.db xprim_3000a-6.db \
    xgen_3000.db xdc_3000a-6.db}

  target_library = {xprim_3020a-6.db xprim_3000a-6.db \
    xgen_3000.db xdc_3000a-6.db}
  ```

  These lines specify the *default* target and link libraries that indicate which compiler Synopsys uses. The link and target libraries are for an XC3020A–6 device. You can use these libraries for the Design Compiler tutorial for XC3000A devices.

The Synlibs program determines the correct libraries for the different device types and speed grades. Refer to the “Using Synlibs for XC3000 Devices” section that follows for more information on how to use Synlibs to change the link and target libraries to a different part type and speed grade.

The target and link libraries are device-specific. Therefore, you might not want to specify target and link libraries in your generic startup file.

- `symbol_library = xc3000.sdb`

  This line specifies the symbol libraries.

- `define_design_lib WORK –path ./WORK`

  This line creates a directory to store intermediate files created by the Analyze command for VSS users.

- `compile_fix_multiple_port_nets = true`

  This line allows the Synopsys optimization algorithm to insert extra logic into the design to ensure that there are no feedthroughs and that no two output ports are connected to the same net.

- ```
  bus_naming_style = "%s<%d>"
  bus_dimension_separator_style = "><"
  bus_inference_style = "%s<%d>"
  ```

  These lines set the READ parameters for the Xilinx netlist formats. All bus indexing is of the form “bus<index>.”

- ```
  edifout_netlist_only = true
  edifout_power_and_ground_representation = cell
  edifout_write_properties_list = “instance_number \
      port_location part”
  ```

  These lines set the EDIF parameters for Xilinx devices.

Modifying the Search Path

Modify the search path line in the default .synopsys_dc.setup file to include the path to the XSI and Synopsys installation directories as follows.

1. To determine the Synopsys installation path, enter the following at the command line.

   echo $SYNOPSYS

2. Open the setup file in a text editor and replace *DS401-Directory* with the full path of the directory where the XSI software is installed (the $XACT environment variable contains this path).
3. Replace *SYNOPSYS-Directory* with the path where your Synopsys software is installed (usually stored in the environment variable $SYNOPSYS).

Using Synlibs for XC3000 Devices

Synlibs displays the link and target libraries for the specified part type and speed grade. You can run Synlibs from any directory as follows.

synlibs *parttype–speedgrade*

For example, to list the target and link libraries for an XC3020A–6 device, enter the following.

```
synlibs 3020a-6
```

The system displays the output onscreen as illustrated by Figure 2-6.

```
link_library = {xprim_3020a-6.db xprim_3000a-6 \
   xgen_3000.db xdc_3000a-6.db}

target_library = {xprim_3020a-6.db xprim_3000a-6 \
xgen_3000.db xdc_3000a-6.db}
```

Figure 2-6 Synlibs Output for Use with Design Compiler (XC3000)

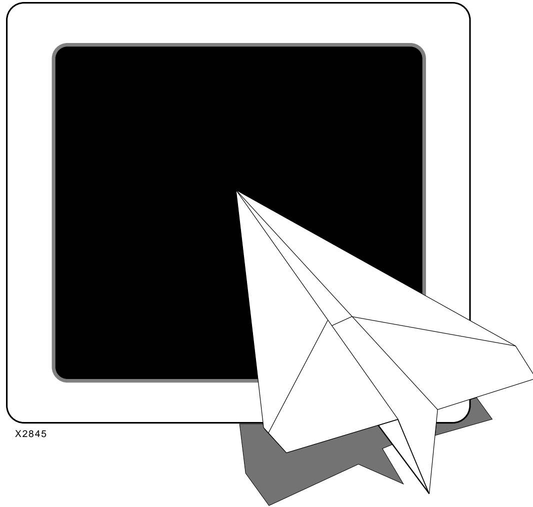
Note: Use the target and link libraries in the default setup file if you plan to perform the Design Compiler tutorial for XC3000A devices.

You must copy the output from Synlibs into your Synopsys startup file. You can use the UNIX Append (>>) command to redirect the output of Synlibs to your .synopsys_dc.setup file as follows.

synlibs *parttype–speedgrade* **>> .synopsys_dc.setup**

After you redirect the output, use a text editor to delete the default target and link libraries.

Warning: You must list the libraries in your setup file in the order that they appear in the Synlibs output.



FPGA Compiler Tutorial

Xilinx Synopsys Interface FPGA User Guide

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Chapter 3

FPGA Compiler Tutorial for XC4000 Designs

XSI provides an interface between Synopsys synthesis tools and the Xilinx XACT Development System. This interface enables you to use an HDL description to create your design and the XACT tools to map, place, and route the design.

This tutorial provides step-by-step information on how to run the FPGA Compiler for XC4000 designs and takes approximately one hour to complete. The FPGA Compiler understands the XC4000 architecture and maps to XC4000 CLBs.

Note: For XC3000/A/L or XC3100/A designs, the FPGA Compiler and Design Compiler results are the same. Refer to the "Design Compiler Tutorial for XC3000A Designs" chapter.

Before You Begin

Before starting this tutorial, make sure that the Xilinx Synopsys Interface (DS-401), XACT Development System (DS-502), X-BLOX (DS-380), and Synopsys FPGA Compiler are installed.

Note: X-BLOX must be installed if you plan to use the DesignWare library.

To verify the correct installation of these tools, refer to the "Getting Started" section at the beginning of this user guide, which describes how to modify the default Synopsys startup file to include the appropriate libraries and search path.

Required Files

To access the files you need to perform this tutorial, follow these steps. Replace *DS401-Directory* with the directory where the XSI software is installed.

The files you need are in one of the following directories.

VHDL users *DS401-Directory*`/tutorial/synopsys/fpga \`
`/x4000/vhd`

Verilog users *DS401-Directory*`/tutorial/synopsys/fpga \`
`/x4000/verilog`

In this tutorial, you use a design called count8, which is a modulo 256 (8-bit) counter. The vhd directory contains the VHDL version, count8.vhd, and the verilog directory contains the Verilog HDL version, count8.v.

1. Change to your working directory.
2. Create a directory called count8 and change to that directory.

   ```
   mkdir count8
   cd count8
   ```

3. Copy the files from either the VHDL or Verilog tutorial directory into the count8 directory.

   To use the VHDL count8 design, enter the following on the command line.

   cp -r *DS401-Directory***/tutorial/synopsys/fpga/x4000 **
   /vhd .

   To use the Verilog HDL count8 design, enter the following on the command line.

   cp -r *DS401-Directory***/tutorial/synopsys/fpga/x4000 **
   /verilog .

Note: The backslash (\) is a continuation character; do not enter it on the command line.

   If you do not know the location of the *DS401-Directory*, type the following, which displays the paths set for the XACT environment variable. The XSI path appears first.

   ```
   echo $XACT
   ```

Exiting the Tutorial

You can exit or stop the tutorial at any time. For best results, complete all steps in a section before quitting. If you must exit the Design

Analyzer before completing the tutorial, you must re-run the tutorial from the beginning.

Design Flow

This section illustrates the Xilinx implementation flow for the count8 tutorial design. Generally, the design process starts with an HDL description of the desired circuit functions and ends with a BIT file, a binary file that contains the configuration data for your design, and an LCA file, which you can use for back-annotation and simulation.

Figure 3-1 illustrates the Xilinx XC4000 implementation flow for synthesis. Use it as a checklist as you proceed with your XC4000 design.

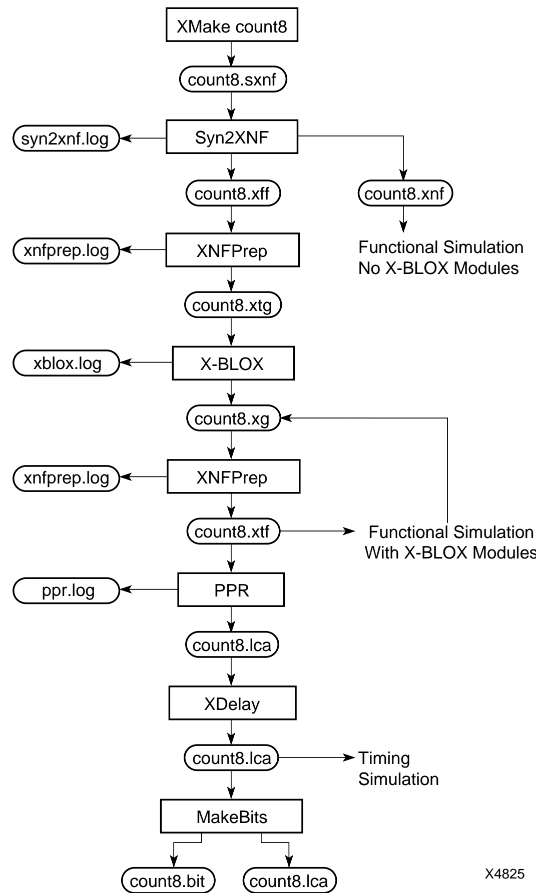


Figure 3-1 Xilinx XC4000 Implementation Flow for Synthesis

Note: For the XSI design flow, which precedes running XMake, see the beginning of the “Using the FPGA Compiler” chapter.

Count8 Design Description

This section contains a description of the count8 design used in this tutorial. Figure 3-2 shows the VHDL code and Figure 3-3 shows the Verilog HDL code.

The count8 design counts up to 255, then starts again at zero. It can count only when Enable is High and Clear is Low. If Clear is High, the counter resets synchronously. If Enable is Low, the counter is disabled. The output signal is COUT.

```
-- Count8 - Behavioral Model
-- 8-bit Counter with Enable and Clear
-- XSI v3.2
--

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

entity count8 is
        port (CLOCK, CLEAR, ENABLE: in STD_LOGIC;
              COUT: out STD_LOGIC_VECTOR (7 downto 0));
end count8;

architecture BEHAVIORAL of count8 is
signal QOUT: STD_LOGIC_VECTOR (7 downto 0);
begin
    process (CLEAR, CLOCK, ENABLE)
        begin
        if (CLEAR = '1') then
            QOUT <= "00000000";
        elsif (CLOCK'event and CLOCK='1') then
            if (ENABLE = '1') then
                QOUT <= QOUT + "00000001";
            end if;
        end if;
    end process;
COUT <= QOUT;
end BEHAVIORAL;
```

Figure 3-2 VHDL Code for Count8

```
/*
 *
 * Count8 - Behavioral Model
 *
 * Model originally developed by Seva Technologies, Inc.
 *
 *       %W%     %G%
 *
 */

module count8(clock, clear, enable, cout) ;
input           clock;
input           clear;
input           enable;
output  [7:0]   cout;

reg     [7:0]   cout;

always @(posedge clear or posedge clock)
begin
   if (clear == 1´b1)
      cout = 8´h00 ;
   else if (enable == 1´b1)
      cout = cout + 1´b1 ;
end

endmodule
```

Figure 3-3 Verilog HDL Code for Count8

Invoking the Design Analyzer

In this section you learn how to invoke the Design Analyzer and verify that the Synopsys startup file (.synopsys_dc.setup) has been properly installed and modified as described in the release notes and the "Getting Started" chapter of this user guide.

Perform the following steps.

1. From the count8 directory, run the Synopsys Design Analyzer in the background by entering the following command.

   `design_analyzer &`

   If the .synopsys_dc.setup file generates any errors or warnings, the system displays them onscreen. If you receive any error or warning messages, refer to the "Getting Started" chapter.

Note: The command.log file in your working directory lists the variable settings for the Design Analyzer. To verify that Synopsys

read the correct .synopsys_dc.setup file, you can view the command.log file.

2. Verify that your Synopsys options were set correctly.

   Setup → Defaults...

   The system displays the following dialog box.

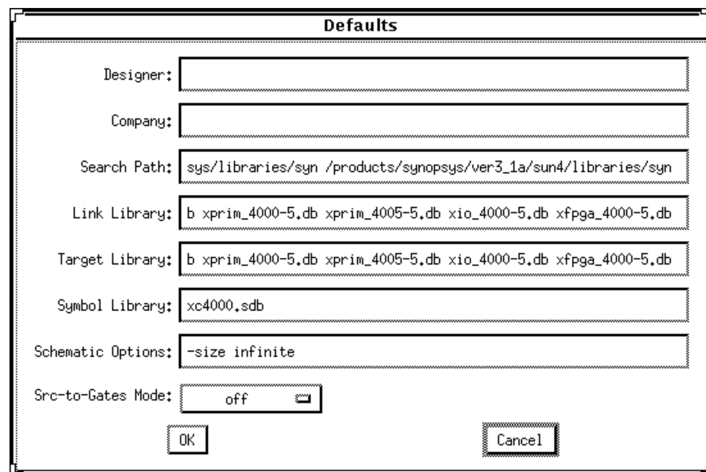


Figure 3-4 Defaults Dialog Box

3. Verify that your settings match the following.

   search_path = *DS401-Directory*`/synopsys/libraries/syn`
   SYNOPSYS-Directory`/libraries/syn`

   link_library = `xprim_4005-5.db xprim_4000-5.db xgen_4000.db xio_4000-5.db xfpga_4000-5.db`

   target_library = `xprim_4005-5.db xprim_4000-5.db xgen_4000.db xio_4000-5.db xfpga_4000-5.db`

   symbol_library = `xc4000.sdb`

   The fields in the dialog box are not long enough to show all the default information. To view hidden information, position your

cursor in a specific field and use the left arrow key or enlarge the width of the Defaults window.

Note: *DS401-Directory* is the directory where the Xilinx Synopsys Interface software is installed, and the *SYNOPSYS-Directory* is where the Synopsys FPGA Compiler is installed.

4. Select `Cancel` to close the window.

Reading the Design File

In this section you learn how to use the Design Analyzer to analyze and create the design file.

Analyzing the Design File

The Analyze command checks the syntax and logic, and converts the HDL file to an intermediate format for use during simulation. To analyze the design file, perform the following steps.

1. Select `File → Analyze...` from the Design Analyzer menu.

   The system displays the Analyze File dialog box as shown in Figure 3-5.

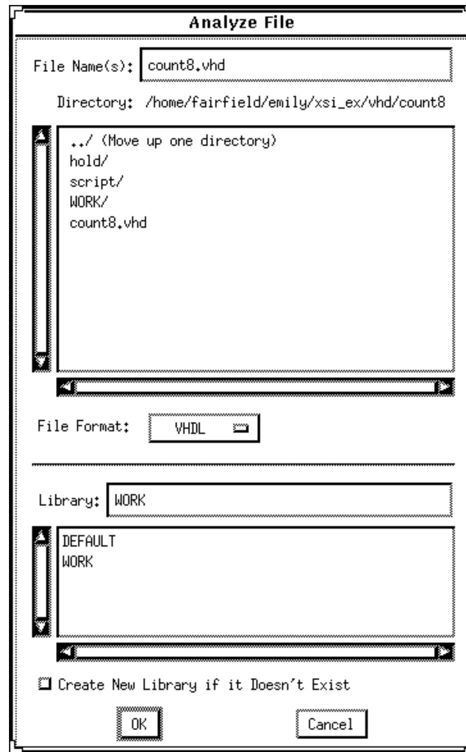


Figure 3-5 Analyze File Dialog Box

2. Use the left mouse button to click once on **`count8.vhd`** for VHDL users, or **`count8.v`** for Verilog HDL users.

   The system displays count8.vhd or count8.v in the File Name(s) field.

3. Click **`OK`**.

   The Analyze window displays informational, error, and warning messages. The system also displays processing messages in the Command Window. (To display the Command window, select **`Setup`** → **`Command Window ...`** from the Design Analyzer menu.)

   Figure 3-6 illustrates the Analyze window output.

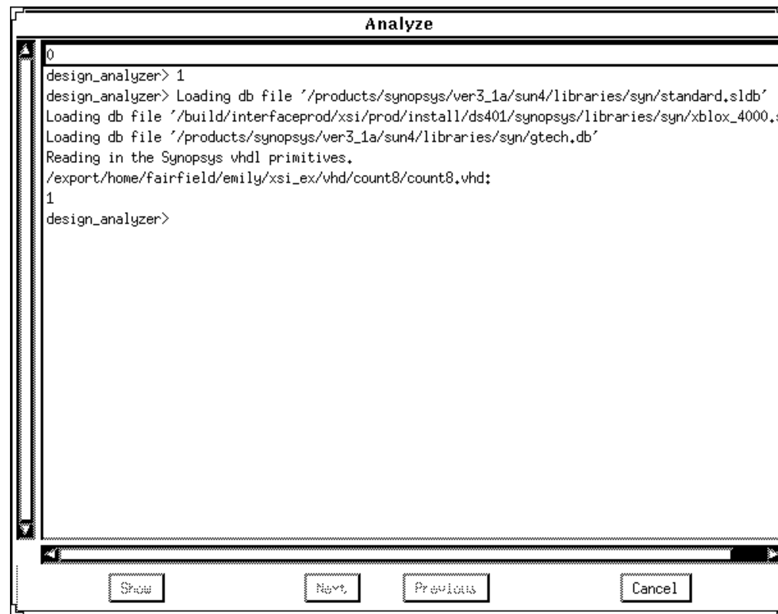


Figure 3-6 Analyze Window

4. Click **`Cancel`** to close the Analyze window.

Creating the Design File

To create the design file, perform the following steps.

1. Use the Elaborate command.

   `File → Elaborate...`

   The Elaborate Design dialog box appears as follows.

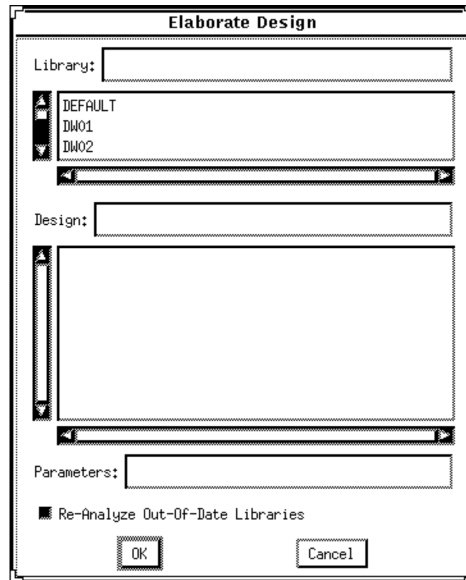


Figure 3-7 Elaborate Design Dialog Box

2. Scroll the library list and click on **WORK**.
3. Click once on **count8(BEHAVIORAL)**.

   The system displays `count8(BEHAVIORAL)` in the Design field.
4. Click **OK**.

   The system displays informational messages in the Elaborate window as illustrated by Figure 3-8.

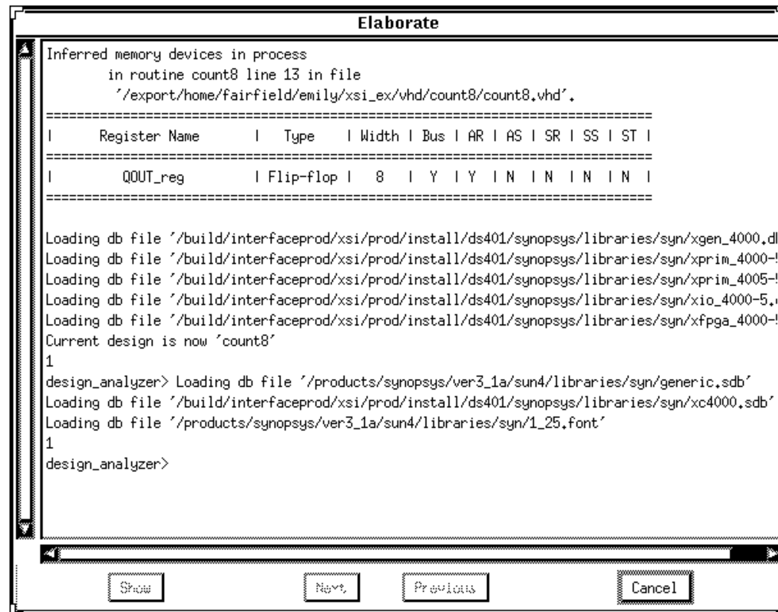


Figure 3-8 Elaborate Window

5. Click **`Cancel`** to close the Elaborate window.

   A symbol that represents the count8 design appears in the Design Analyzer main screen as follows.

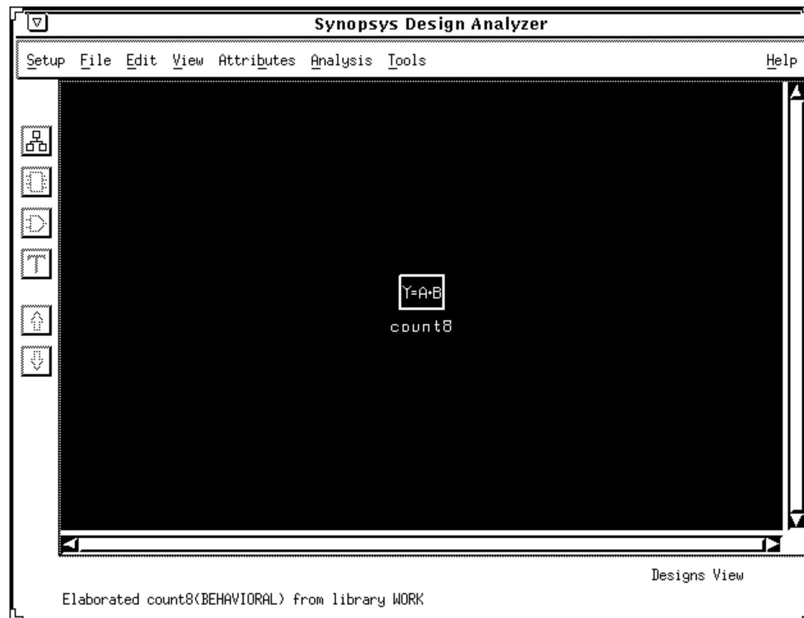


Figure 3-9 Top-Level Symbol for Count8 Design

Inserting I/O Buffers

In this section you define the ports of the top-level design as inputs, outputs, clock ports, or bidirectional ports. Also, you use the Insert Pads commands to add the necessary I/O buffers to the top-level design. Defining a port as a pad causes the Insert Pads command to attach a buffer to that port, which the Xilinx tools can then recognize.

Note: Count8 is a one-level design.

The FPGA Compiler can optimize registers and 3-state functions into IOBs. Refer to the "Using the FPGA Compiler" chapter in this user guide for more information.

The following procedures describe how to define the input ports, CLEAR and ENABLE; the input clock, CLOCK; and the output bus, COUT <7:0>. The actual buffers are not added to the design until the pads are inserted.

Note: The procedures in this section only apply to inserting IBUFs, ILDs, IFDs, OBUFs, IOBUFs, OFDS, and OFDTs. For any other IOB configurations, you must instantiate the buffers into a design. See the "XC3000/A/L and XC3100/A Primitives" and "XC4000/A/D/H Primitives and Hard Macros" appendixes for information on other available buffers.

Defining Input Ports as Pads

To define the input ports as pads, perform the following steps.

1. Click the left mouse button on the count8 icon as illustrated by Figure 3-9.

   The system changes the solid line to a dotted line to indicate that the icon is selected.

2. Click on the down arrow icon to display the design in Symbol View.

   The system displays the count8 design in Symbol View as illustrated by Figure 3-10.

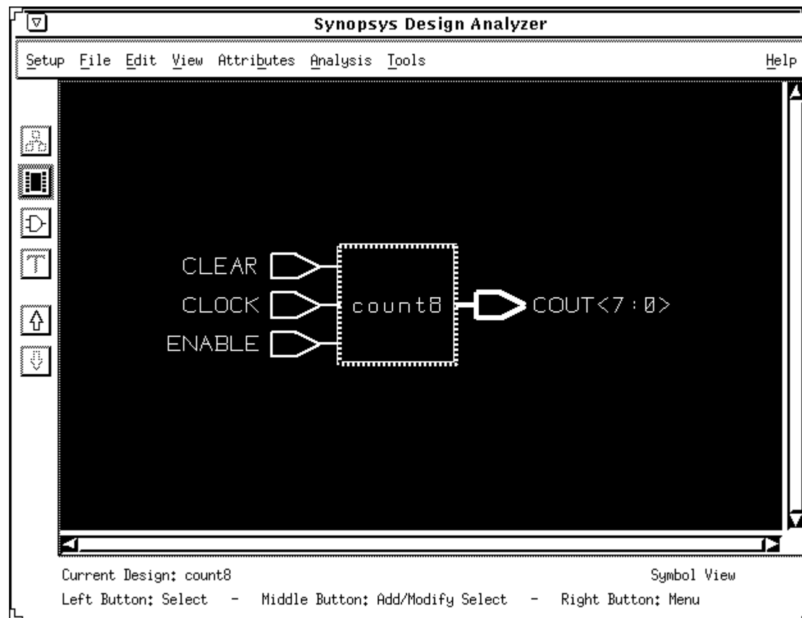


Figure 3-10 Symbol View

3. Select the CLEAR, CLOCK, and ENABLE input ports by clicking on one with the left mouse button, and the other two with the middle mouse button.

   The middle mouse button extends the selection. A dotted rectangle indicates that the ports are selected.

Note: To deselect an input port, click on it again with the middle mouse button.

4. Select `Attributes` → `Optimization Directives` → `Input Port...` from the Design Analyzer menu.

   The Input Port Attributes dialog box appears as shown in Figure 3-11.

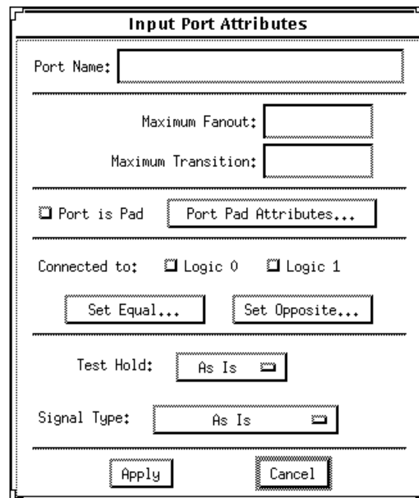


Figure 3-11 Input Port Attributes Dialog Box

5. Click on the box next to **`Port is Pad`**.
6. Select **`Apply`**.

   The system sets the attributes for the CLEAR, CLOCK, and ENABLE ports.
7. Click on **`Cancel`** to close the dialog box.

Defining the Output Port as a Pad

To define the output port as a pad, perform the following steps.

1. Select the **`COUT [7:0]`** bus by clicking on it with the left mouse button.

   A dotted rectangle indicates that the output port is selected.
2. Select **`Attributes`** → **`Optimization Directives`** → **`Output Port...`** from the Design Analyzer menu.

   The Output Port Attributes window appears first, then the Bus Selector dialog box appears over it as illustrated by Figure 3-12.

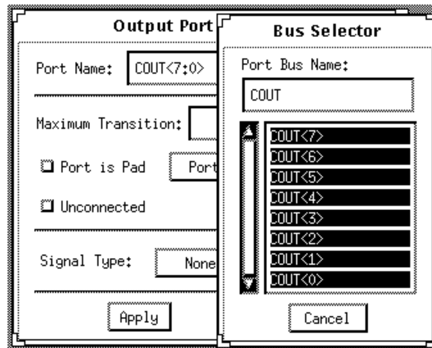


Figure 3-12 Bus Selector and Output Port Attributes Dialog Boxes

3. In the Bus Selector window, select **`Cancel`**.

   The Bus Selector dialog box disappears.

4. In the Output Port Attributes dialog box, click on the box labeled **`Port is Pad`**.

5. Select **`Apply`**.

6. Select **`Cancel`** to close the dialog box.

Note: You can also define the inputs, outputs, and clock buffers using the Set Port Is Pad command at the Synopsys DC-shell prompt or in the Design Analyzer command window as follows. This command sets all the ports as pads in one simple step.

```
set_port_is_pad "*"
```

Using the Insert Pads Command

After the ports are defined as pads, you can insert the I/O buffers using the following procedure.

1. If the Command window is not open, select **`Setup → Command Window...`** from the Design Analyzer menu.

   The Command window appears.

2. At the Design analyzer prompt in the Command window, type **`insert_pads`**↵.

The Command window displays informational messages. You may want to move the Command window to a place on your desktop where it does not obscure the Design Analyzer main window.

Figure 3-13 illustrates the Command window output after running the Insert Pads command.

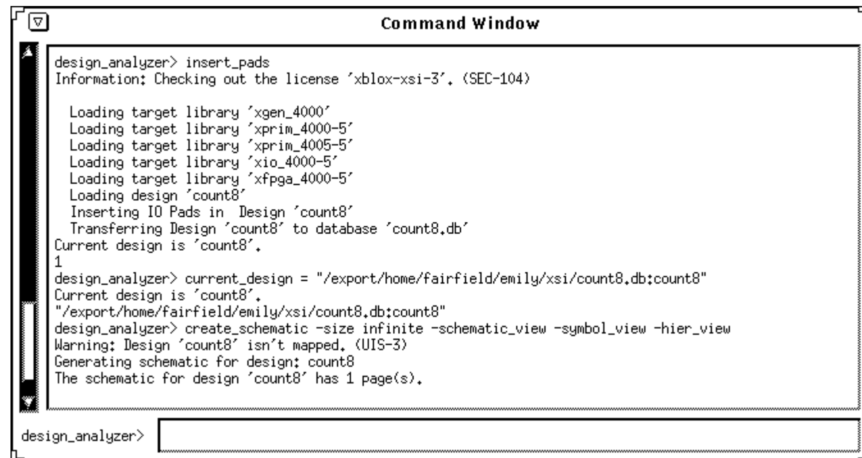


Figure 3-13 Command Window Output for Insert Pads Command

Estimating Pre-Layout Timing

The XSI libraries contain operating conditions and wire-load models that are used to provide a pre-layout timing estimate of your design.

Selecting the Operating Condition

XSI offers a set of operating condition parameters called worst-case commercial (WCCOM). The operating conditions are selected automatically if you used Synlibs to generate the link and target libraries. For more information on the Synlibs command, refer to the “Getting Started” chapter at the beginning of this user guide.

Setting the Wire-Load Models

The XSI libraries offer worst-case and average wire-load models. Wire loads are the estimated net delays for a design that has been partitioned into CLBs and IOBs. Refer to the "Using the FPGA Compiler" chapter in this user guide for more information.

Synopsys uses these estimates as guidelines to optimize your design for an FPGA. The actual wire loads cannot be determined until after the design has been placed and routed.

The models are device and speed-grade dependent, with an average wire-load model (*parttype–speedgrade*_avg) and a worst-case wire-load model (*parttype–speedgrade*_wc) for each. The average wire-load model is the mean of the test suite and the worst-case is the average plus one standard deviation. Therefore, the worst-case model is more conservative.

The average wire-load model is selected automatically if you used Synlibs to generate the link and target libraries.

Optimizing for Speed

Before compiling a design, you can set area and speed constraints to improve results. In this section you set a timing constraint. For the most effective results from the FPGA Compiler, the constraints must be accurate and achievable. For example, if a timing goal of 0 ns is set on all ports, the FPGA Compiler adds buffers to critical paths or duplicates logic on heavily loaded nets, attempting to achieve this goal. An unrealistic goal might cause significant and unwarranted area increases. Refer to the *Synopsys Design Compiler Reference Manual* for details on optimization techniques.

Path timing includes both logic and net delays. All gate, CLB, and IOB timing delays are worst-case commercial estimates and are specified in nanoseconds. The wire-load delays are either average estimates or worst-case estimates. Actual delays are determined only after you use PPR.

Additional timing information about primitives is included in the "XC4000/A/D/H Primitives and Hard Macros" appendix in this user guide and *The Programmable Logic Data Book.*

To set a clock constraint, follow these steps.

1. Select the CLOCK pin by placing the cursor on the CLOCK port and pressing the left mouse button.
2. Select the following menu options from the Attributes menu.

   Attributes → Clocks → Specify...

   The system displays the Specify Clock dialog box as follows. The default clock period is 50.

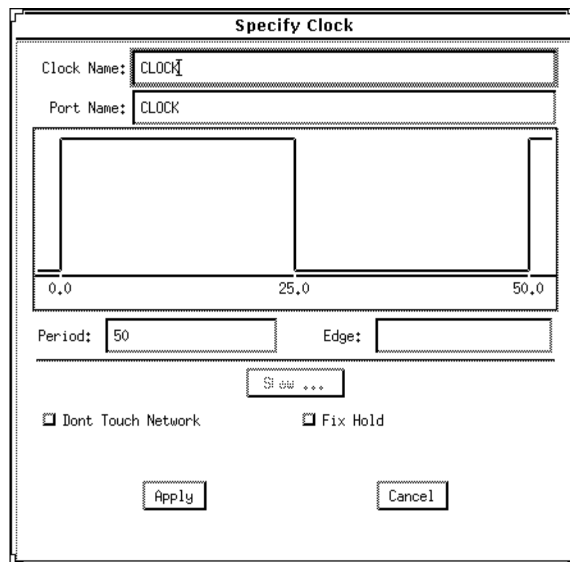


Figure 3-14 Specify Clock Dialog Box

3. Select **Apply → Cancel**.

   A waveform appears above the CLOCK pin to indicate the setting of a timing constraint.

Compiling the Design

In this section you learn how to compile a design with the recommended options.

The optimization process is part of the Compile command. Optimization is a complex series of transformations guided by constraints that you specify. One of the optimization steps is technology mapping, which transforms the Boolean logic network representation of your design into interconnected gates that are selected from the target technology library. You can set the mapping as Low, Medium, or High. Refer to the *Synopsys Design Compiler Reference Manual* for more details about mapping and other optimization techniques.

To compile the count8 design, do the following.

1. Select **`Tools`** → **`FPGA Compiler...`** from the Design Analyzer menu.

   The FPGA Compiler dialog box appears as follows.

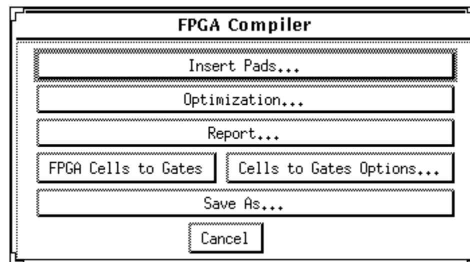


Figure 3-15 FPGA Compiler Dialog Box

2. Click on **`Optimization...`**

   The Design Optimization dialog box appears as follows.

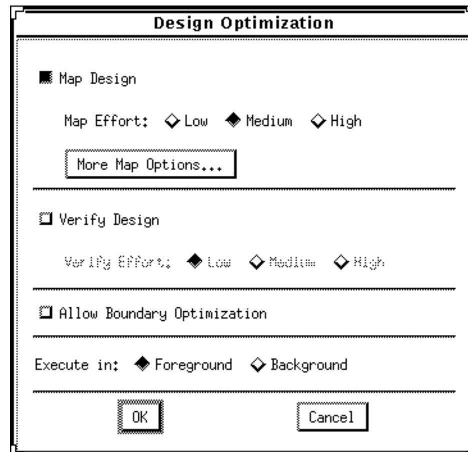


Figure 3-16 Design Optimization Dialog Box

3. Make sure the Map Design box is shaded and the Map Effort is Medium.
4. Click **OK**.

   The system displays any informational messages and compilation errors in the Compile Log window and the Command window.
5. Once the design is compiled, click **Cancel** to close the Compile Log window.
6. Select **Cancel** to close the FPGA Compiler window.

Evaluating the Results

The design is now optimized for the XC4000 architecture and mapped into CLBs and IOBs.

The XSI libraries contain both area and timing information. In this section you view an area report on the estimated CLB and IOB utilization and a timing report on the estimated delays. You also learn how to redirect the report output from the screen to a file.

1. View a schematic of the design by selecting the gate picture icon on the left side of the Synopsys Design Analyzer window.

The system displays a schematic view of the count8 design.

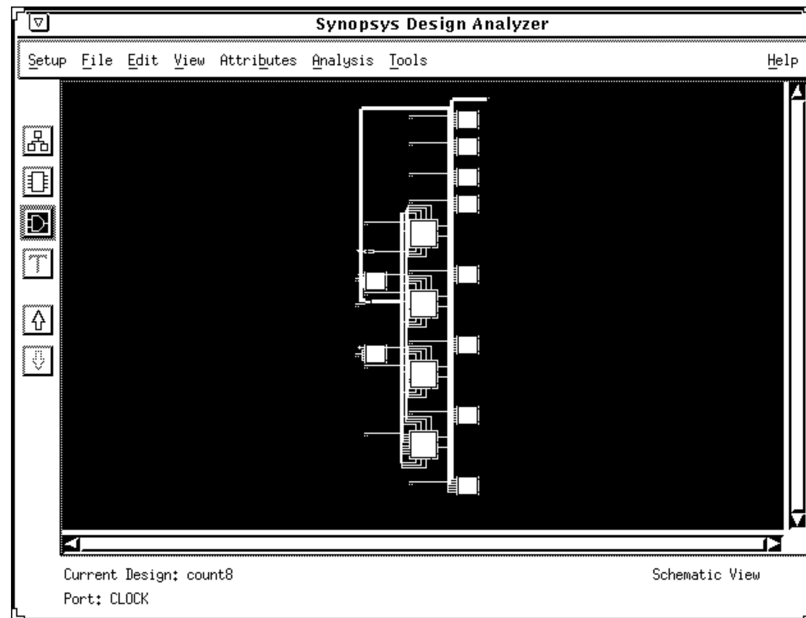


Figure 3-17 Schematic View

2. When you finish viewing the schematic, click on the up arrow icon to switch to the Designs View as illustrated by Figure 3-18.

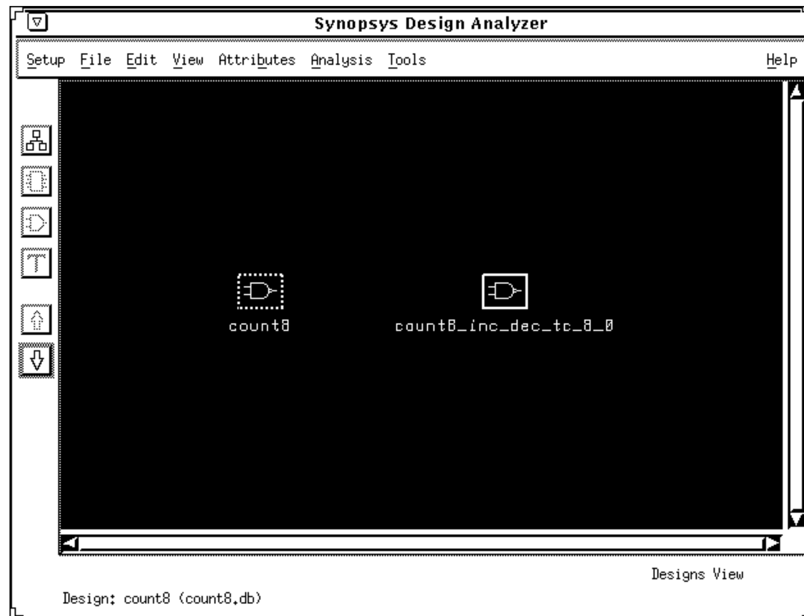


Figure 3-18 Designs View

Viewing the Estimated Area Results

To evaluate the estimated area results, perform the following steps.

1. Click on the count8 icon.
2. Select **`Tools`** → **`FPGA Compiler...`** from the Design Analyzer menu.

   The FPGA Compiler window appears.
3. Select **`Report...`**.

   The Report dialog box appears as follows.

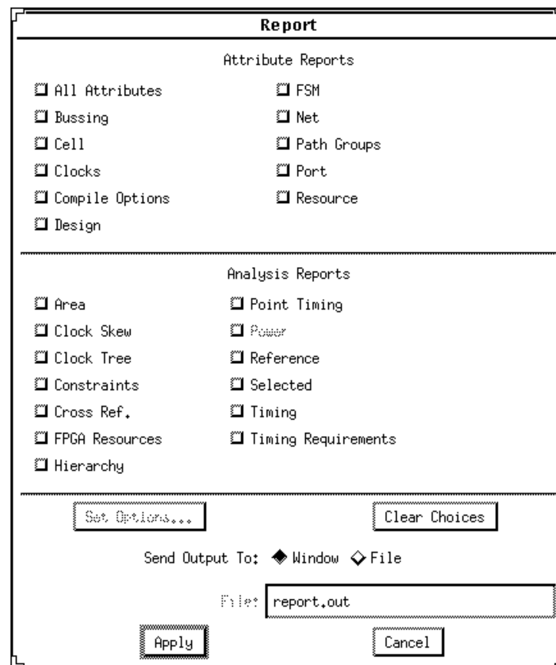


Figure 3-19 Report Dialog Box

4. In the Analysis Reports section, select the boxes next to FPGA Resources and Timing.
5. Select **`Apply`**.

   The Report Output window appears.
6. Use the scroll bar in the Report Output window to view the design statistics.
7. Select **`Cancel`** to close the Report Output window.

Note: Do not close the Report dialog box.

Figure 3-20 illustrates an example of a report file. This report shows area utilization (CLBs used) for the count8 design.

```
Information: Updating design information... (UID-85)

****************************************
Report : fpga
Design : count8
Version: v3.1b-20502
Date   : Sun Oct  2 16:35:19 1994
****************************************

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

    FG Function Generators:          4
    H Function Generators:           0
    Number of CLB cells:             4
    Number of Hard Macros and
        Other Cells:                 1
    Number of CLBs in
        Other Cells:                 4
    Total Number of CLBs:            8

    Number of Ports:                11
    Number of Clock Pads:            1
    Number of IOBs:                 10

    Number of Flip Flops:            8
    Number of 3-State Buffers:       0

    Total Number of Cells:          16
```

Figure 3-20 Area Utilization Report

Note: Clock pads are IOBs, yet they are listed separately in this report.

Viewing the Estimated Timing Results

To evaluate the timing results, perform the following steps.

1. In the Analysis Reports section of the Report dialog box, click on the box to the left of Timing with the left mouse button.
2. Deselect the FPGA Resources box.
3. Select **`Apply`**.

   The Report Output window opens. The results reported are worst-case timing delay estimates. The final results cannot be determined until after you run PPR.

4. When you are finished reviewing the Report Output window, select **Cancel**.

Note: Do not close the Report dialog box.

Saving the Area and Timing Results to a File

To save the estimated area and timing results to a report file, perform the following steps.

1. Locate the Send Output To field at the bottom of the Report dialog box.
2. Select **File**.
3. Place your cursor in the File field.
4. Double-click to highlight the default report file name.
5. Type **count8.timing**
6. Select **Apply** → **Cancel**.

Note: Do not close the FPGA Compiler window.

The Xilinx libraries use worst-case delays. Synopsys timing delay estimates include wire-load delays in addition to gate delays. In most cases, actual results are better than the pre-placement and routing Synopsys estimates.

Figure 3-21 is a complete timing report for the count8 design, which is called count8.timing.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : count8
Version: v3.1b-20502
Date   : Sun Oct  2 16:35:19 1994
****************************************

Operating Conditions: WCCOM   Library: xprim_4005-5
Wire Loading Model Mode: top

Design           Wire Loading Model      Library
------------------------------------------------
count8                4005-5_avg       xprim_4005-5


  Startpoint: U83 (rising edge-triggered flip-flop clocked by CLOCK)
  Endpoint: U89 (rising edge-triggered flip-flop clocked by CLOCK)
  Path Group: CLOCK
  Path Type: max

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock CLOCK (rise edge)                                 0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  U83/K (clb_4000)                                        0.00       0.00 r
  U83/XQ (clb_4000)                                       5.48       5.48 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/A_1 (count8_inc_dec_ub_8_0)
                                                          0.00       5.48 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/u8/FUNC<6> (INC_DEC_UBIN_8)
                                                         16.29      21.77 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/S_6 (count8_inc_dec_ub_8_0)
                                                          0.00      21.77 r
  U89/G1 (clb_4000)                                       0.00      21.77 r
  data arrival time                                                 21.77

  clock CLOCK (rise edge)                                50.00      50.00
  clock network delay (ideal)                             0.00      50.00
  U89/K (clb_4000)                                        0.00      50.00 r
  library setup time                                     -4.50      45.50
  data required time                                                45.50
  --------------------------------------------------------------------------
  data required time                                                45.50
  data arrival time                                                -21.77
  --------------------------------------------------------------------------
  slack (MET)                                                       23.73
```

Figure 3-21 Timing Report (count8.timing)

Saving the Design

In this section you learn how to save your design as a DB (Synopsys Database file) file, replace CLBs and IOBs with gates, set the design part type and speed grade, and save the design into an SXNF file.

Writing the DB File

To save the design to a DB file, do the following.

1. Select the count8 icon with the left mouse button.
2. Select **`File`** → **`Save`** from the Design Analyzer menu.

The system saves the file as count8.db.

Replacing CLBs and IOBs with Gates

After a design is compiled, it contains CLB and IOB elements. To create an SXNF file, the FPGA Compiler must convert these CLBs and IOBs to gates. Perform the following steps.

1. In the FPGA Compiler window, select **`FPGA Cells to Gates`**.

   The system displays the FPGA Log window.

Note: The mapping of logic into CLBs is written to the SXNF file and is retained by PPR. Refer to the "Using the FPGA Compiler" chapter in this user guide for information about how to remove the mapping information.

2. Select **`Cancel`** to close the FPGA Log window.
3. Select **`Cancel`** to close the FPGA Compiler window.

Setting the Design Part Type

To select a particular part for the count8 design, type the following command at the Design Analyzer prompt in the Command window.

Note: The \ (backslash) is a line continuation marker. Do not type it on the command line.

```
set_attribute count8 "part" -type \
string "4005pc84-5"↵
```

Removing BLKNM Attributes

To allow the XACT software more freedom during placement and routing, Xilinx recommends not writing the block names to the SXNF

file. To prohibit the writing of block names, enter the following command at the Design Analyzer prompt in the Command window.

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

Saving the Design File as an SXNF File

The next step is to save the design file as an SXNF file as follows.

1. Select the following menu options.

   File → Save As...

   The Save File dialog box appears as illustrated by Figure 3-22.

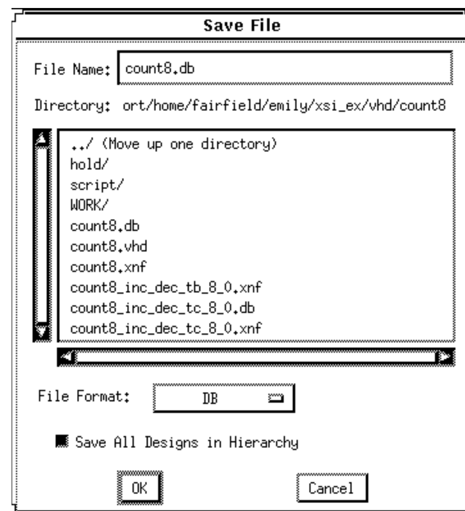


Figure 3-22 Save File Dialog Box

2. Click in the field next to File Name.
3. Change the .db extension to .sxnf.

   Place your cursor to the right of **count8.db**, backspace to delete **db**, then replace it with **sxnf**.
4. Click on the bar next to File Format.

   The system displays a list of formats.

5. Select **XNF**.
6. Make sure the Save All Designs in Hierarchy box is shaded.
7. Select **OK**.

Exiting the Design Analyzer

You are now done with the Synopsys FPGA Compiler and are ready to use the XACT Development System. To exit the Design Analyzer, do the following.

1. Select **File → Quit** from the Design Analyzer main menu.

   The Quit Design Analyzer? window appears.
2. Click **OK** to exit.

The following section is a reference section that describes running a script that invokes the Synopsys tools.

To continue with the tutorial, skip to the "Placing and Routing Your Design Using XMake" section.

Executing the Commands from a Script File

Warning: Do not execute the commands in this section. Use this section as a reference for how to execute a script file. You have already executed these commands through the Design Analyzer menus.

You can use a script file to compile your design instead of using pull-down menus. The commands illustrated in this tutorial are all listed in a script file, count8.script.

You can execute this script either from the Design Analyzer or DC-Shell. Each command is annotated in the script file. Comments start and end with /* and */. Each command corresponds to a command already executed in this tutorial.

The procedures to execute the count8.script file from the Design Analyzer are the following.

1. Invoke the Synopsys Design Analyzer in the background.

   ```
   design_analyzer &
   ```

2. Open the Command window to view the script as it executes.

   Setup → Command Window...

3. Execute the count8.script file.

   Setup → Execute Script...

   The Execute File dialog box appears as shown in Figure 3-23.

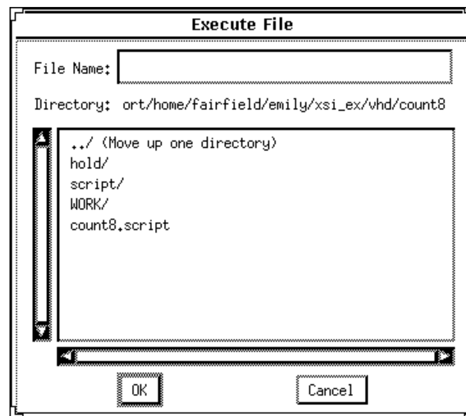


Figure 3-23 Execute File Dialog Box

4. Select **count8.script**.

   The system displays count8.script in the File Name field.

5. Select **OK**.

6. Exit the Design Analyzer.

Figure 3-24 and Figure 3-25 illustrate the actual text for the count8.script file for VHDL and Verilog HDL, respectively.

```
/* ================================================*/
/* Script for Synopsys to Xilinx FPGA Compiler v3.1  */
/*             Count8 VHDL Tutorial                 */
/* ================================================*/
   TOP = count8
   PART = "4005pc84-5"
   designer = "XSI Team"
   company  = "Xilinx, Inc"

   analyze -format vhdl TOP + ".vhd"
   elaborate TOP

   current_design TOP

   set_port_is_pad "*"
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads

   remove_constraint -all
   create_clock "CLOCK" -period 50

   compile

   report_fpga > TOP + ".fpga"
   report_timing > TOP + ".timing"

   write -format db -hierarchy -output TOP + ".db"

   replace_fpga

   set_attribute TOP "part" -type string PART
   set_attribute find(design,"*") "xnfout_use_blknames" \
     -type boolean FALSE

   write -format xnf -hierarchy -output TOP + ".sxnf"

   exit
```

Figure 3-24 VHDL Script File for Count8

```
/* ================================================*/
/* Script for Synopsys to Xilinx FPGA Compiler v3.1  */
/*             Count8 Verilog Tutorial               */
/* ================================================*/
   TOP = count8
   PART = "4005pc84-5"
   designer = "XSI Team"
   company  = "Xilinx, Inc"

   analyze -format verilog TOP + ".v"
   elaborate TOP

   current_design TOP

   set_port_is_pad "*"
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads

   remove_constraint -all
   create_clock "clock" -period 50

   compile

   report_fpga > TOP + ".fpga"
   report_timing > TOP + ".timing"

   write -format db -hierarchy -output TOP + ".db"

   replace_fpga

   set_attribute TOP "part" -type string PART
   set_attribute find(design,"*") "xnfout_use_blknames" \
   -type boolean FALSE

   write -format xnf -hierarchy -output TOP + ".sxnf"

   exit
```

Figure 3-25 Verilog HDL Script File for Count8

Placing and Routing Your Design Using XMake

XMake automates the translation portion of the Xilinx design flow, which makes processing a complex design as simple as running one program.

Given the name of the top-level SXNF file, XMake finds and processes all lower-level drawings. It produces an LCA file that is placed and routed, as well as a BIT file ready for downloading to an FPGA. You can invoke XMake from within the XACT Design Manager (XDM) or from a shell tool window.

In this section you translate the count8 design using XMake from the shell tool window. Refer to the *XACT Reference Guide* for details about each program that XMake runs or about running XMake from XDM.

The procedure for translating the count8 design is slightly different if XSI is installed on a different platform or network than that of the XACT Development System. Follow the procedures that apply to your specific configuration.

If XSI Is on Same Network as XACT Software

Follow the procedures in this section if the XSI software is installed on the same network or platform as the XACT Development System software. You can find the command-line options that XMake used in the XMake output file, count8.out.

To run XMake from a shell tool window, type the following.

```
xmake count8.sxnf
```

Refer to Figure 3-1 for a flow diagram that illustrates the Xilinx implementation flow for synthesis.

If XSI Is on Different Network Than XACT Software

Follow the procedures in this section if the XSI software is installed on a different network or platform than the XACT Development System software.

If XSI is installed on a machine that does not have access to both the XSI and XACT Development System executable files, you must run Syn2XNF first and then copy the output XNF and XFF files to the platform where the XACT executable files reside.

Refer to Figure 3-1 for a flow diagram that illustrates the Xilinx implementation flow for synthesis.

The following sections describe how to run the Syn2XNF and XMake programs.

Running Syn2XNF

Because the XSI software is installed on a different platform than the XACT software, you must first run Syn2XNF to translate your design into an XFF file, as follows.

1. Change to the directory where the count8.sxnf file is located and execute the following command.

   `syn2xnf count8.sxnf`

   The SYN2XNF software might display the following message, which prompts you to overwrite any existing XFF file that has the same design name.

   ```
   WARNING: The file count8.xff already exists.
   Do you want to overwrite it? (yes or no)
   ```

   If the system displays the previous message, enter `y` ↵.

   Syn2XNF creates the following output files: count8.xff, count8.xnf and syn2xnf.log.

2. Copy the count8.xff and count8.xnf files to the platform or network where the XACT software is installed.

Note: Use the –p option with the Copy command to preserve the files' time stamp.

Running XMake

Perform the following steps to translate the count8 design using XMake.

1. Go to the platform where the XACT Software is installed.
2. Open a shell tool window.
3. Enter the following on the command line.

   `xmake –x count8`

   The –x option causes XMake to start with the XNF file and skip the translation process. The XMake program processes all the necessary design files, displaying its progress on the screen. If the translation is successful, XMake issues this message.

   ```
   'count8.bit' has been made, check output in
   'count8.out'
   ```

4. Be sure to examine the count8.out, count8.prp, and count8.rpt files for warnings and errors, as described in the "Examining XMake Output Files" section in this manual.

Examining XMake Output Files

In addition to the routed LCA file and the bitstream BIT file, XMake generates three very useful output files. In this section you open each file and familiarize yourself with its contents.

- The OUT file, count8.out, contains all the output from the programs that XMake invokes. This information is also displayed onscreen during processing.
- The PRP file, count8.prp, is the DRC (Design Rules Checker) report file generated by XNFPrep.
- The RPT file, count8.rpt, contains the PPR placement and routing results. This report also contains a listing of any unrouted pins or nets.

Reviewing the XMake OUT File

When you run XMake, the output of the XMake program appears on the screen. The OUT file shows every program run by XMake, the command options selected, and the output of each individual program.

Any warnings or errors produced by the programs run by XMake appear in the OUT file. You should always review the OUT file after running XMake, even if you did not see any warnings or error messages during design processing. If any warnings or errors do occur, you can save yourself some time by catching the problem now instead of later in the design process.

Examine the count8.out file for the count8 design as follows.

1. Open a shell tool window.
2. Change to the project directory.
3. Use a text editor to view count8.out.

Checking for Warnings and Errors in the PRP File

If XNFPrep finds any errors or warnings, the OUT file directs you to examine the PRP file. The PRP file also contains a detailed list of all logic trimmed by XNFPrep and why it was unnecessary. This file can be a useful debugging tool.

You should expect to see some warning messages in the count8.prp files but no errors.

Examine the count8.prp file for the count8 design.

The following headers correspond to the table of contents found in the count8.prp.

- XNFPrep Errors — lists all errors found in the design. Errors are problems with the design that cause XMake to terminate. You must fix any reported errors.
- XNFPrep Warnings — lists all warnings found in the design. Warnings notify you of any unusual aspects in your design. You should correct all warnings; however, it is not mandatory.
- Clock Signals Report — contains a summary of all the clock signals and/or global buffers to assist you in determining the best use of the global buffers. This section also contains a list of guidelines to consider when assigning signals to a global buffer.
- Timing Specification Summary — contains a list of the XACT-Performance timing specifications used in the design.
- Logic Trimming — shows the logic removed from your design due to sourceless or loadless signals and V_{CC} or ground connection. You should review this section to verify that logic required in your design has not been removed due to design error.

Checking the RPT File

After XMake runs PPR, PPR generates a report file with an .rpt extension, which contains important information in the following categories.

- Partition, Place, and Route Summary — includes the number of occupied CLBs that approximately corresponds to the total area provided in the Synopsys Report.
- Chip Pinout Description — contains a list of the pins used in the design and any pin locations specified in the constraints file.
- Critical Nets — indicates any nets that were assigned a constraint.
- Feedthrough Split Nets — indicates any nets with names that were modified so the net could be re-powered. Re-powering or

signal regeneration is accomplished by the net using a special function in a CLB.

- Deletion Traceback — enables you to check for any nets or cells that were removed that should remain. The Synopsys Check Design tool detects any unconnected pins or unused cells.

Examine the count8.rpt file to make sure there are no unrouted pins or nets. Use a text editor to view this file. Figure 3-26 illustrates each page of the RPT file.

```
PPR RESULTS FOR DESIGN COUNT8                                     Page   1

Design Statistics and Device Utilization
----------------------------------------


Partitioned Design Utilization Using Part 4005PC84-5

                                 No. Used   Max Available   % Used
  ----------------------------   -------    -------------   ------
  Occupied CLBs                     6           196           3%
  Packed CLBs                       4           196           2%
  ----------------------------   -------    -------------   ------
  Bonded I/O Pins:                 11           61           18%
  F and G Function Generators:      8           392           2%
  H  Function Generators:           0           196           0%
  CLB Flip Flops:                   8           392           2%
  IOB Input Flip Flops:             0           112           0%
  IOB Output Flip Flops:            0           112           0%
  Memory Write Controls:            0           196           0%
  3-State Buffers:                  0           448           0%
  3-State Half Longlines:           0           56            0%
  Edge Decode Inputs:               0           168           0%
  Edge Decode Half Longlines:       0           32            0%


Routing Summary

  Number of unrouted connections: 0


PPR Parameters

  Design           = count8.xtf
  Parttype         = 4005PC84-5
  Guide_cell       =
  Seed             = 781116564
  Estimate         = FALSE
  Complete         = TRUE
  Placer_effort    = 2
  Router_effort    = 2
  Path_timing      = TRUE
  Stop_on_miss     = FALSE
  DC2S             = none
  DP2S             = none
  DC2P             = none
  DP2P             = none
  Guide_only       = FALSE
  Ignore_maps      = FALSE
  Ignore_rlocs     = FALSE
  Outfile          = <design name>
```

```
PPR RESULTS FOR DESIGN COUNT8                                       Page   2

CPU Times

  Partition:                      00:00:00
  Placement:                      00:00:38
  Routing:                        00:00:10
  Total:                          00:00:55


PPR RESULTS FOR DESIGN COUNT8                                       Page   3

Xact Performance Summary
----------------------------------------

    Deadline  Actual(*)  Specification
    --------  ---------  -------------
      50.0ns    24.8ns    TS0=clock to setup:
      <auto>    11.6ns    <default> pad to setup
      50.0ns    10.1ns    TS7=pad to setup:ENABLE
      50.0ns    10.1ns    TS6=pad to setup:ENABLE
      50.0ns    10.1ns    TS5=pad to setup:ENABLE
      50.0ns    10.1ns    TS4=pad to setup:ENABLE
      50.0ns    10.1ns    TS3=pad to setup:ENABLE
      50.0ns    10.1ns    TS2=pad to setup:ENABLE
      50.0ns    10.1ns    TS1=pad to setup:ENABLE
      50.0ns    10.1ns    TS8=pad to setup:ENABLE
      <auto>    16.5ns    DEFAULT_FROM_FFS_TO_PADS=FROM:ffs:TO:pads

(*) Note: the actual path delays computed by PPR indicate that ALL
timing  specifications you provided have been met.  Please use the
-FailedSpec  and/or -TSMaxpaths options of the Xdelay-TimeSpec
command, accessible  through the XDE or XDelay program, as a final
confirmation of the performance of your design.

    *** PPR: WARNING 7028:
The design has flip-flops with asynchronous set/reset controls
(PRE/SD or CLR/RD pins).  When PPR analyzes design timing, it does
not trace paths through the asynchronous set/reset input and on
through the Q output.

If you want PPR to control the delay on paths through asynchronous
set/reset pins, you must split the delay requirement into two
segments: one ending at the set/reset input, and the other beginning
at the flip-flop output. If you want PPR not to analyze paths that
lead to asynchronous set/reset pins, attach an IGNORE specification
to the pin(s) or signal(s).

By default, XDelay reports all paths through asynchronous set/reset
pins. To prevent XDelay from showing these paths, use FlagBlk
CLB_Disable_SR_Q on the appropriate flip-flops.
```

```
PPR RESULTS FOR DESIGN COUNT8                                   Page   4

Chip Pinout Description
-----------------------

This chapter describes where your design´s pins were placed in terms
of the package pins.  This first list is sorted by package pin
location.  The second list presents the same data sorted by your
design´s pin names.

Package Pin Location   Pin Name
--------------------   -----------------
P4                   : COUT<7>
P5                   : CLEAR
P6                   : ENABLE
P7                   : COUT<4>
P15                  : COUT<6>
P16                  : COUT<5>
P17                  : COUT<2>
P18                  : COUT<3>
P19                  : COUT<0>
P20                  : COUT<1>
P29                  : CLOCK

This list describes where your design´s pins are in terms of the
package pins; it is sorted by your design´s pin name.  The list
presented above has the same data sorted by package pin location.

Package Pin Location   Pin Name
--------------------   -----------------
P5                   : CLEAR
P29                  : CLOCK
P19                  : COUT<0>
P20                  : COUT<1>
P17                  : COUT<2>
P18                  : COUT<3>
P7                   : COUT<4>
P16                  : COUT<5>
P15                  : COUT<6>
P4                   : COUT<7>
P6                   : ENABLE
```

```
PPR RESULTS FOR DESIGN COUNT8                                   Page   5

  Split Nets
  -------------------
  The list below identifies those signals which were routed through
  CLBs or other blocks. In XDE and XDelay reports, these signals will
  have two or more segments. An underscore (_) and a number will be
  added to the end of the original signal name to identify the
  different segments. To analyze all segments of a signal in XDelay or
  QueryNet reports, append the original name with "_*" when prompted
  for a signal name.

  PPR may route signals through CLBs or other blocks in any of the
  following situations:

  * The delay on a signal might be reduced by routing it through a CLB,
  given the extra flexibility in routing resources and the reduced
  capacitive loading on the signal. PPR takes this into consideration.

  * The delay on a signal might be reduced by sourcing it from two
  block outputs instead of one, which is possible in some block
  configurations. PPR will do this where possible.

  * A signal on a global buffer may not be able to connect directly to
  a load pin, given the placement of that load pin and the other global
  resources which are used. PPR will pass the signal through another
  CLB and route the load pin using general-purpose interconnect.

  * In an XC4000 design, A BUFGP can be sourced only from an IOB. If
  the design indicates that a BUFGP is driven from an internal source,
  PPR will route the signal through the output path of the IOB in order
  to access the BUFGP input.

  Segments   Original Signal Name
  --------   -------------------------------------------------------
```

```
PPR RESULTS FOR DESIGN COUNT8                                    Page   6

Information in Other Reports
----------------------------


Since not all pertinent design information is listed in this PPR
report file, this section describes where additional information can
be found.

Information                                 Report File  Created By
-----------------------------------------   -----------  ----------


Connection of signals between levels of     design.mrg   XNFMERGE
design hierarchy

Resolution of relative location (RLOC)      design.mrg   XNFMERGE
constraints through design hierarchy

X-BLOX designs: design rule check for       design.prx   XNFPREP
pre-expanded design

X-BLOX designs: results of optimization     design.blx   XBLOX
and module expansion in X-BLOX

Design rule check for invalid and/or        design.prp   XNFPREP
inefficient use of LCA architecture

Unused or disabled logic removed from       design.prp   XNFPREP
design, due to sourceless or loadless
signals and VCC or ground connections

XC3000A/L designs: Mapping of design        design.crf   XNFMAP
logic into each CLB or IOB

XC3000A/L designs: Summary of guided        design.crf   XNFMAP
partitioning results
```

Figure 3-26 RPT File

Comparing Actual Versus Estimated Area Results

The RPT file contains a partition, place, and route summary that includes the number of occupied CLBs that approximately corresponds to the total area number provided in the Synopsys Report.

In this section you compare how accurate the FPGA Compiler pre-place and route estimates were to the actual results.

Figure 3-20 shows the estimated area results from the FPGA Compiler, and Figure 3-26 shows the actual area results from PPR. The following table summarizes the area utilization results.

Table 3-1 Area Utilization Summary

Partitioned Design Utilization Using Part 4005PC84-5		
	Actual No. Used	Estimated No. Used
Occupied CLBs	6	8
Packed CLBs	4	N/A
Bonded I/O Pins	11	11
F and G Function Generators	8	4*
H Function Generators	0	0
CLB Flip-flops	8	8
Clock Pads	1	1

* The reported number of CLBs in other cells was 4.

The actual area utilization is accurate because the FPGA Compiler mapped the design and passed this information to PPR.

Note: Your actual area numbers may vary from the area utilization reported in the FPGA Compiler since PPR adds additional CLBs as feedthrough split nets. Refer to page 5 of the RPT file, illustrated by Figure 3-26, for more information on split nets for the count8 design.

Using XDelay

The XDelay command allows you to obtain detailed post-placement and post-routing timing information about your design.

The XDelay results summarize the worst paths for the design, not necessarily the paths that concern you. XDelay also has an interactive mode, which enables you to extract information about specific paths in the design, for example, to generate a timing report for a subset of the design. You can choose specific paths by selecting individual starting and ending points or by indicating a specific path type. For more information about XDelay and its options, refer to the *XACT Reference Guide.*

In this section you use XDelay to report the worst-case paths and the maximum clock frequency of the design. You also compare the output of XDelay to the estimated timing reported by the FPGA Compiler.

Invoking XDelay

Enter the following command at the command line to run XDelay, which creates a short report, count8.dly.

```
xdelay count8
```

XDelay produces the following output as shown in Figure 3-27.

```
XDelay Report File:

    Design: count8.lca (4005PC84-5)
    Program: xdelay 5.0.0
    Speedsfile: File 4005.spd, Version 4000.1, Revision 4005.8

Xdelay timing analysis options:

From all.
To all.
 Worst case Pad to Pad path delay        :   28.6ns   (1 block level)
  Pad "CLEAR" (P5) to Pad "COUT<7>" (P4.0)


Clock net "N107" path delays:

 Pad to Setup                            :   37.0ns   (5 block levels)
  (Includes an external input margin of 0.0ns.)
  Pad "CLEAR" (P5) to FF Setup (D) at "N163.C4"
   Target FFY drives output net "N163"

 Clock to Pad                            :   16.5ns   (0 block levels)
  (Includes an external output margin of 0.0ns.)
  Clock to Q, net "N163" to Pad "COUT<7>" (P4.0)

 Clock to Setup (same edge)              :   24.9ns   (4 block levels)
  Clock to Q, net "N169" to FF Setup (D) at "N163.C4"
   Target FFY drives output net "N163"


                    Minimum Clock Period :   37.0ns

           Estimated Maximum Clock Speed :   27.0Mhz
```

Figure 3-27 XDelay Short Report

Comparing Actual Versus Estimated Timing Results

You can often get better timing estimates by looking at the number of block levels that the critical or longest path must traverse rather than using the estimated delays listed in the count8.timing report, illustrated in Figure 3-21.

Block levels are the number of CLBs and IOBs. The longest path reported in the FPGA Compiler was a clock-to-clock delay from a

register through the incrementer. This delay was reported as 26.27 ns in the count8.timing report and included the clock-to-output delay, the delay though the X-BLOX incrementer, the clock-to-setup delay, the average wire load, and the flip-flop setup time.

The XDelay report, illustrated in Figure 3-27, reports the longest clock-to-setup delay as 24.9 ns with four block levels. The wire-load models and the mapping of the X-BLOX modules account for the difference in delay from that of the timing report.

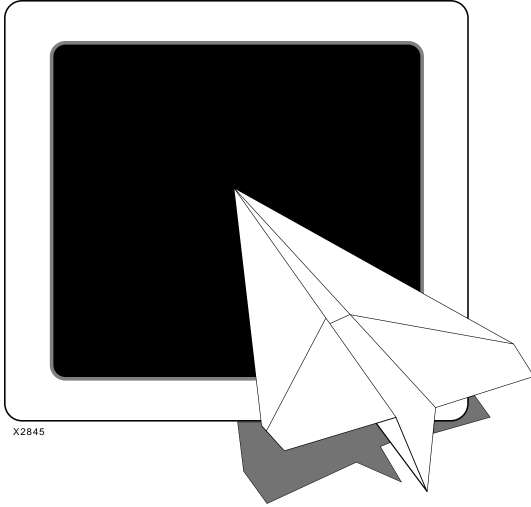
Note: The FPGA Compiler does not provide estimated block levels for X-BLOX components, and X-BLOX timing assumes the modules fit in a single column of CLBs.

Verifying Your Design Using XChecker

This section describes the function of the XChecker Download/Readback cable. You do not actually download the count8 design in this tutorial.

To verify that your design works in your system, you can use the XChecker Download/Readback cable and associated software. With XChecker, you can load a configuration bitstream generated by the MakeBits program. The MakeBits file defines the internal logic functions and interconnections of the target FPGA. For more information on the XChecker cable or the MakeBits program, refer to the *XACT Hardware and Peripherals Guide* or the *XACT Reference Guide, Volume 2*, respectively.

You can store the BIT file in your system memory or in a PROM. Refer to the XPP section of the *XACT Hardware and Peripherals Guide* for more information about storing BIT files in PROMs.



Design Compiler Tutorial

Xilinx Synopsys Interface FPGA User Guide

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Chapter 4

Design Compiler Tutorial for XC3000A Designs

XSI provides an interface between Synopsys synthesis tools and the Xilinx XACT Development System. This interface enables you to use an HDL description to create your design and the XACT tools to map, place, and route the design.

This tutorial provides step-by-step information on how to run the Design Compiler for XC3000A designs and takes approximately one hour to complete.

Note: You can also perform this tutorial with XC4000 designs, yet you need to specify different libraries in your .synopsys_dc.setup file. Refer to the "Using the Design Compiler for XC4000 Designs" in the "Getting Started" chapter for more information, including how to access the X-BLOX DesignWare library.

Before You Begin

Before starting this tutorial, make sure that the Xilinx Synopsys Interface (DS-401), XACT Development System (DS-502), and Synopsys Design Compiler are installed.

To verify the correct installation of these tools, refer to the "Getting Started" section at the beginning of this user guide, which describes how to modify the default Synopsys start-up file to include the appropriate libraries and search path.

Required Files

To access the files you need to perform this tutorial, follow these steps. Replace *DS401-Directory* with the directory where the XSI software is installed.

The files you need are in one of the following directories.

VHDL users *DS401-Directory*`/tutorial/synopsys/dc \`
`/x3000a/vhd`

Verilog users *DS401-Directory*`/tutorial/synopsys/dc \`
`/x3000a/verilog`

In this tutorial, you use a design called count8, which is a modulo 256 (8-bit) counter. The vhd directory contains the VHDL version, count8.vhd, and the verilog directory contains the Verilog HDL version, count8.v.

1. Change to your working directory.
2. Create a directory called count8 and change to that directory.

   ```
   mkdir count8
   cd count8
   ```

3. Copy the files from either the VHDL or Verilog tutorial directory into the count8 directory.

   To use the VHDL count8 design, enter the following on the command line.

   ```
   cp -r DS401-Directory/tutorial/synopsys/dc/x3000a \
   /vhd .
   ```

   To use the Verilog HDL count8 design, enter the following on the command line.

   ```
   cp -r DS401-Directory/tutorial/synopsys/dc/x3000a \
   /verilog .
   ```

Note: The backslash (\) is a continuation character; do not enter it on the command line.

If you do not know the location of the *DS401-Directory*, type the following, which displays the paths set for the XACT environment variable. The XSI path appears first.

```
echo $XACT
```

Exiting the Tutorial

You can exit or stop the tutorial at any time. For best results, complete all steps in a section before quitting. If you must exit the Design

Analyzer before completing the tutorial, you must re-run the tutorial from the beginning.

Design Flow

This section illustrates the Xilinx implementation flow for the count8 tutorial design. Generally, the design process starts with an HDL description of the desired circuit functions and ends with a BIT file, a binary file that contains the configuration data for your design, and an LCA file, which you can use for back-annotation and simulation.

Figure 4-1 illustrates the Xilinx XC3000A implementation flow for synthesis. Use it as a checklist as you proceed with your XC3000A design.

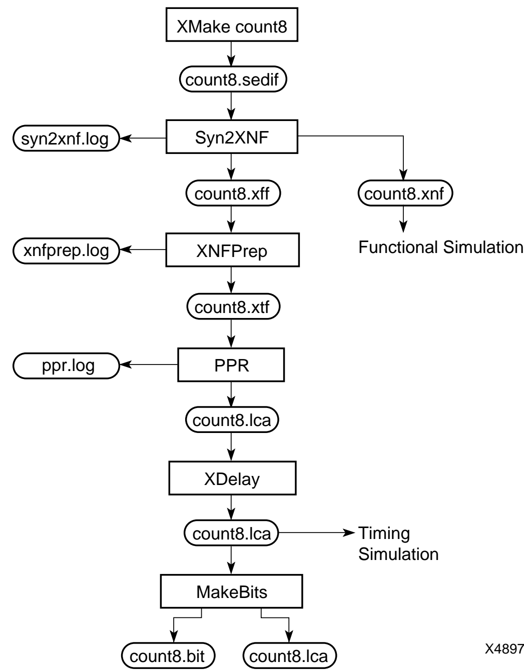


Figure 4-1 XC3000A Implementation Flow for Synthesis

Note: For the XSI design flow, which precedes running XMake, see the beginning of the “Using the Design Compiler” chapter.

Count8 Design Description

This section describes the count8 design used in this tutorial. Figure 4-2 shows the VHDL code and Figure 4-3 shows the Verilog HDL code for count8.

The count8 design counts up to 255, then starts again at zero. It can count only when Enable is High and Clear is Low. If Clear is High, the counter resets synchronously. If Enable is Low, the counter is disabled. The output signal is COUT.

```
-- Count8 - Behavioral Model
-- 8-bit Counter with Enable and Clear
-- XSI v3.2
--

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

entity count8 is
        port (CLOCK, CLEAR, ENABLE: in STD_LOGIC;
              COUT: out STD_LOGIC_VECTOR (7 downto 0));
end count8;

architecture BEHAVIORAL of count8 is
signal QOUT: STD_LOGIC_VECTOR (7 downto 0);
begin
    process (CLEAR, CLOCK, ENABLE)
        begin
        if (CLEAR = '1') then
            QOUT <= "00000000";
        elsif (CLOCK'event and CLOCK='1') then
            if (ENABLE = '1') then
                QOUT <= QOUT + "00000001";
            end if;
        end if;
    end process;
COUT <= QOUT;
end BEHAVIORAL;
```

Figure 4-2 VHDL Code for Count8

```
/*
 *
 * Count8 - Behavioral Model
 *
 * Model originally developed by Seva Technologies, Inc.
 *
 *      %W%     %G%
 *
 */

module count8(clock, clear, enable, cout) ;
input           clock;
input           clear;
input           enable;
output  [7:0]   cout;

reg     [7:0]   cout;

always @(posedge clear or posedge clock)
begin
   if (clear == 1´b1)
      cout = 8´h00 ;
   else if (enable == 1´b1)
      cout = cout + 1´b1 ;
end

endmodule
```

Figure 4-3 Verilog HDL Code for Count8

Invoking the Design Analyzer

In this section you learn the following.

- How to invoke the Design Analyzer.
- How to verify that the Synopsys start-up file (.synopsys_dc.setup) has been properly installed and modified as described in the release notes and the "Getting Started" chapter at the beginning of this user guide.

Perform the following steps.

1. From the count8 directory, run the Synopsys Design Analyzer in the background by entering the following command.

   `design_analyzer &`

   If the .synopsys_dc.setup file generates any errors or warnings, the system displays them onscreen. If you receive any error or warning messages, refer to the "Getting Started" chapter.

Note: The command.log file in your working directory lists the variable settings for the Design Analyzer, which you can view to verify that the Synopsys tools read the correct .synopsys_dc.setup file.

2. Verify that your Synopsys options were set correctly.

   `Setup → Defaults...`

   The system displays the following dialog box.

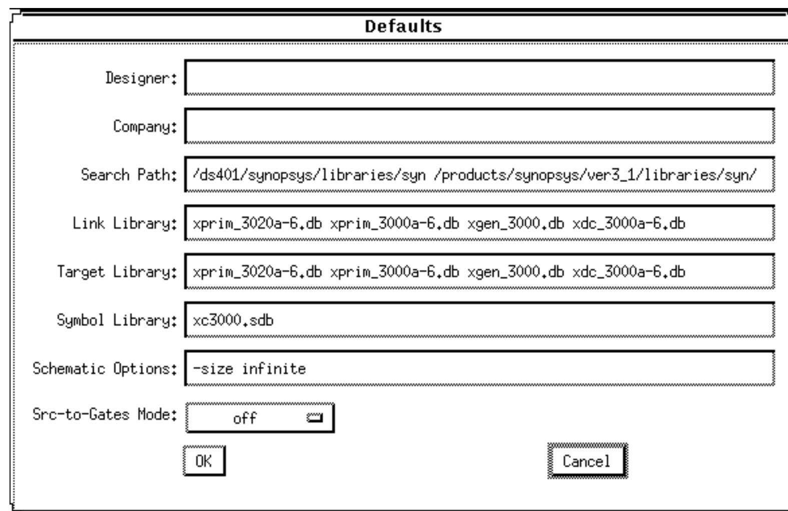


Figure 4-4 Defaults Dialog Box

3. Verify that your settings match the following.

search_path =	*DS401-Directory*`/synopsys/libraries /syn` *SYNOPSYS-Directory*`/libraries /syn`
link_library =	`xprim_3020a-6.db xprim_3000a-6.db xgen_3000.db xdc_3000a-6.db`
target_library =	`xprim_3020a-6.db xprim_3000a-6.db xgen_3000.db xdc_3000a-6.db`
symbol_library =	`xc3000.sdb`

   The fields in the dialog box are not long enough to show all the default information. To view hidden information, position your

cursor in a specific field and use the left arrow key or enlarge the width of the Defaults window.

Note: *DS401-Directory* is the directory where the Xilinx Synopsys Interface software is installed, and the *SYNOPSYS-Directory* is where the Synopsys Design Compiler is installed.

4. Select **`Cancel`** to close the window.

Reading the Design File

In this section you learn how to use the Design Analyzer to analyze and create the design file.

Analyzing the Design File

The Analyze command checks the syntax and logic, and converts the HDL file to an intermediate format for use during simulation. To analyze the design file, perform the following steps.

1. Select **`File`** → **`Analyze...`** from the Design Analyzer menu.

   The system displays the Analyze File dialog box as shown in Figure 4-5.

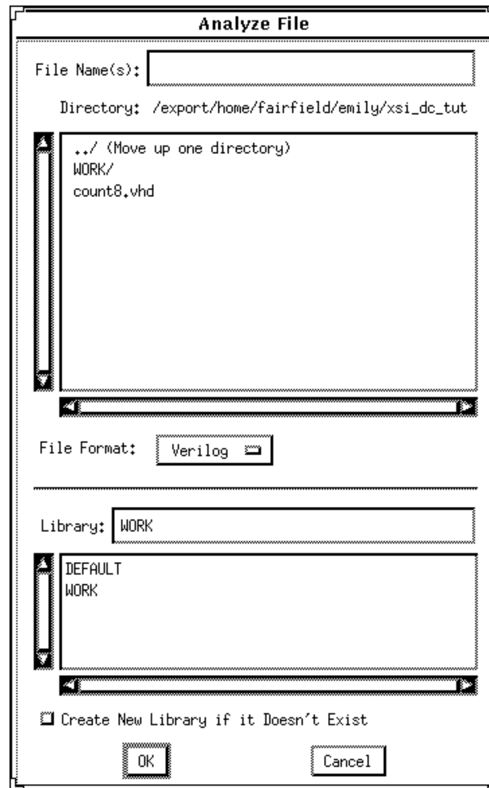


Figure 4-5 Analyze File Dialog Box

2. Use the left mouse button to click once on **`count8.vhd`** for VHDL users, or **`count8.v`** for Verilog HDL users.

   The system displays count8.vhd or count8.v in the File Name(s) field.

3. Click **`OK`**.

   The Analyze window displays informational, error, and warning messages. The system also displays processing messages in the Command window. (To display the Command window, select **`Setup`** → **`Command Window ...`** from the Design Analyzer menu.)

Figure 4-6 illustrates the Analyze window output.

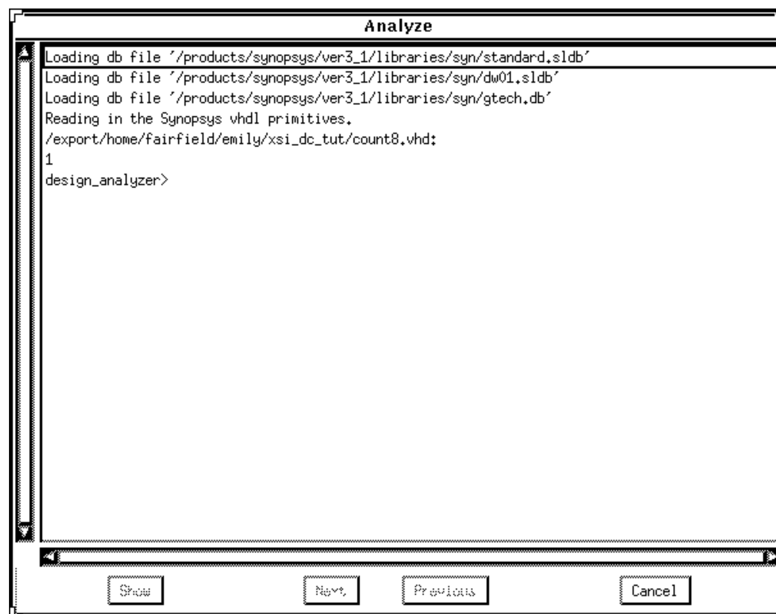


Figure 4-6 Analyze Window

4. Click **`Cancel`** to close the Analyze window.

Creating the Design File

To create the design file, perform the following steps.

1. Use the Elaborate command.

   `File → Elaborate...`

   The Elaborate Design dialog box appears as follows.

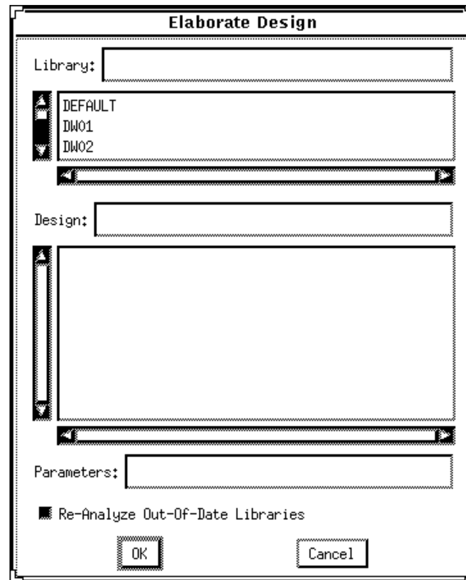


Figure 4-7 Elaborate Design Dialog Box

2. Scroll the library list and click on **WORK**.
3. Click on **count8(BEHAVIORAL)**.

   The system displays count8(BEHAVIORAL) in the Design field.
4. Click **OK**.

   The system displays informational messages in the Elaborate window as illustrated by Figure 4-8.

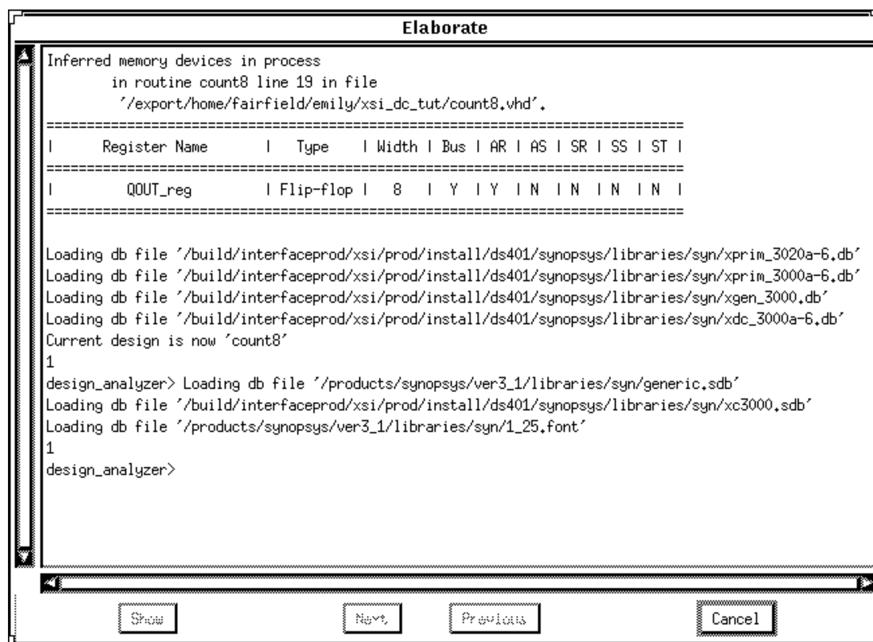


Figure 4-8 Elaborate Window

5. Click **`Cancel`** to close the Elaborate window.

   A symbol that represents the count8 design appears in the Design Analyzer main screen as follows.

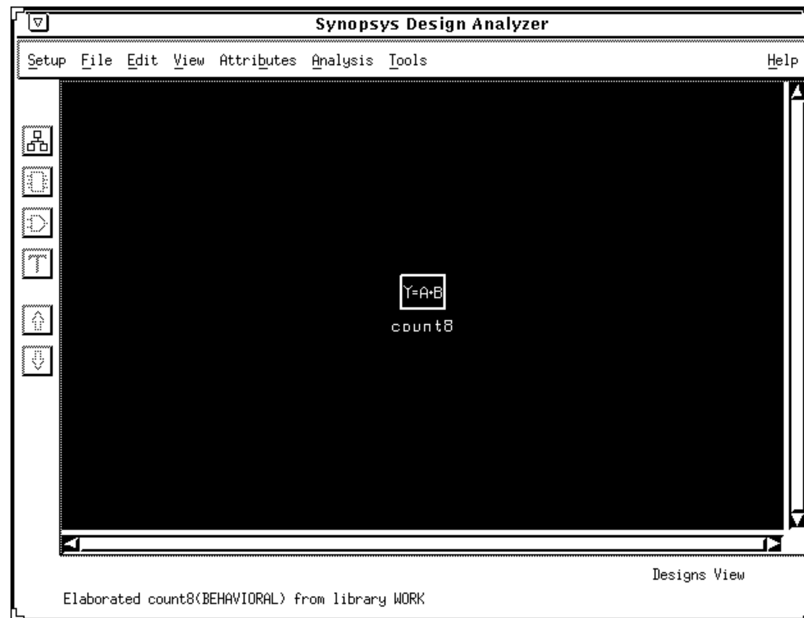


Figure 4-9 Top-Level Symbol for Count8 Design

Inserting I/O Buffers

In this section you define the ports of the top-level design as inputs, outputs, clock ports, or bidirectional ports. Also, you use the Insert Pads commands to add the necessary I/O buffers to the top-level design. Defining a port as a pad causes the Insert Pads command to attach a buffer to that port, which the Xilinx tools can then recognize.

Note: Count8 is a one-level design.

The Design Compiler can optimize registers and 3-state functions into IOBs. Refer to the "Using the Design Compiler" chapter in this user guide for more information.

The following procedures describe how to define the input ports, CLEAR and ENABLE; the input clock, CLOCK; and the output bus, COUT <7:0>. The actual buffers are not added to the design until the pads are inserted.

Note: The procedures in this section only apply to inserting IBUFs, OBUFs, IOBUFs, IFDs, OFDs, and ILDs. For any other IOB configurations, you must instantiate the buffers into a design. See the "XC3000/A/L and XC3100/A Primitives" appendix for information on other available buffers.

Defining Input Ports as Pads

To define the input ports as pads, perform the following steps.

1. Click the left mouse button on the count8 icon illustrated by Figure 4-9.

   The system changes the solid line to a dotted line to indicate the icon is selected.

2. Click on the down arrow icon to display the design in Symbol View.

   The system displays the count8 design in Symbol View as illustrated by Figure 4-10.

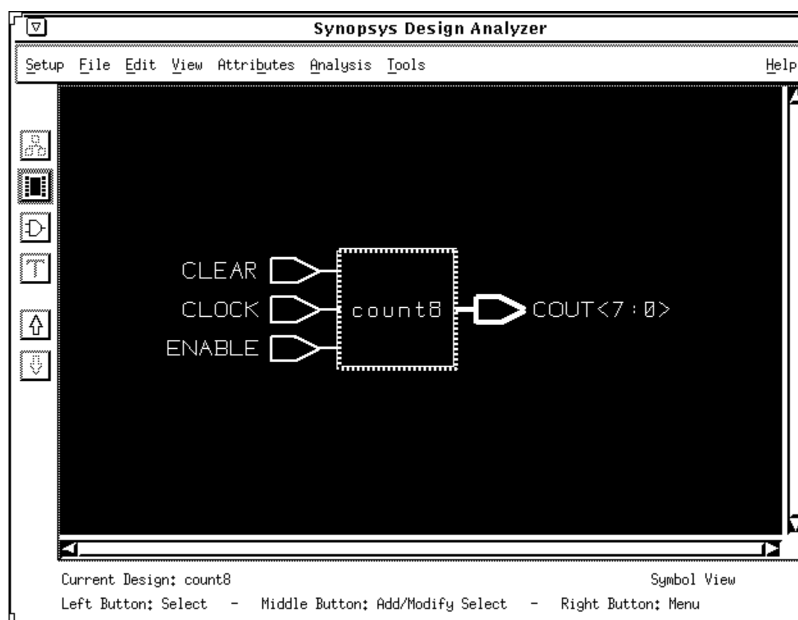


Figure 4-10 Symbol View

3. Select the CLEAR, CLOCK, and ENABLE input ports by clicking on one with the left mouse button, and the other two with the middle mouse button.

   The middle mouse button extends the selection. A dotted rectangle indicates that the ports are selected.

Note: To deselect an input port, click on it again with the middle mouse button.

4. Select **`Attributes`** → **`Optimization Directives`** → **`Input Port...`** from the Design Analyzer menu.

   The Input Port Attributes dialog box appears as shown in Figure 4-11.

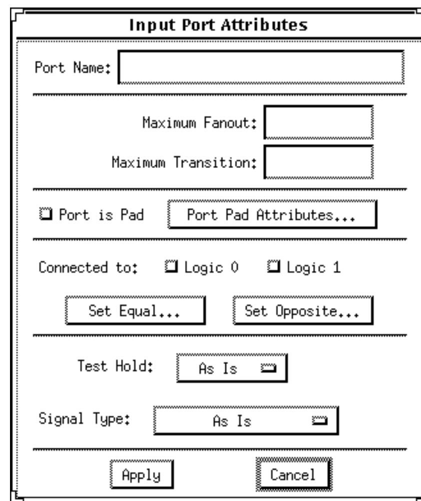


Figure 4-11 Input Port Attributes Dialog Box

5. Click on the box next to **`Port is Pad`**.
6. Select **`Apply`**.

   The system sets the attributes for the CLEAR, CLOCK, and ENABLE ports.

7. Click on **`Cancel`** to close the dialog box.

Defining the Output Port as a Pad

To define the output port as a pad, perform the following steps.

1. Select the **`COUT [7:0]`** bus by clicking on it with the left mouse button.

   A dotted rectangle indicates that the output port is selected.

2. Select **`Attributes`** → **`Optimization Directives`** → **`Output Port...`** from the Design Analyzer menu.

   The Output Port Attributes window appears first, then the Bus Selector dialog box appears over it as illustrated by Figure 4-12.

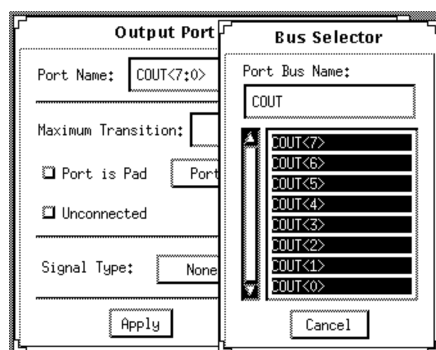


Figure 4-12 Bus Selector and Output Port Attributes Dialog Boxes

3. In the Bus Selector window, select **`Cancel`**.

   The Bus Selector dialog box disappears.

4. In the Output Port Attributes dialog box, click on the box labeled **`Port is Pad`**.

5. Select **`Apply`** → **`Cancel`**.

Note: You can also define the inputs, outputs, and clock buffers using the Set Port Is Pad command at the Synopsys DC-shell prompt or in the Design Analyzer command window as follows. This command sets all the ports as pads in one simple step.

```
set_port_is_pad "*"
```

Using the Insert Pads Command

After the ports are defined as pads, you can insert the I/O buffers using the following procedure.

1. If the Command window is not open, select **Setup → Command Window...** from the Design Analyzer menu.

   The Command window appears.

2. At the Design Analyzer prompt in the Command window, type **insert_pads↵**.

   The Command window displays informational messages. You may want to move the Command window to a place on your desktop where it does not obscure the Design Analyzer main window.

   Figure 4-13 illustrates the Command window output after running the Insert Pads command.

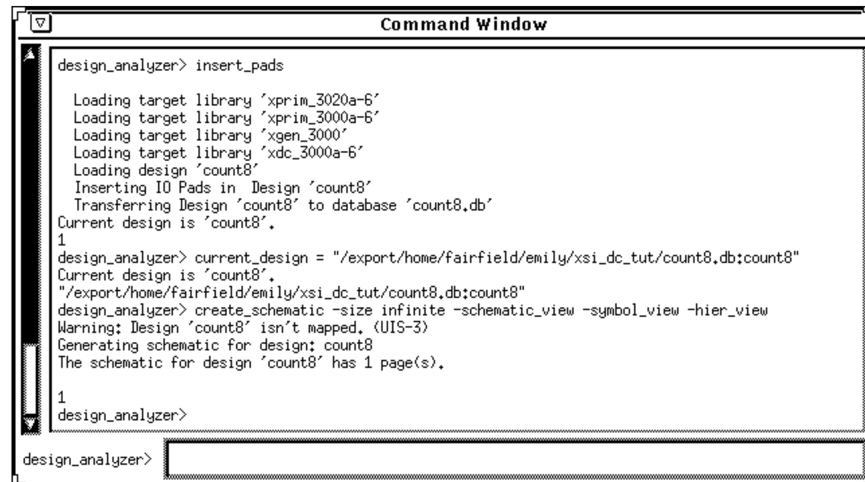


Figure 4-13 Command Window Output for Insert Pads Command

Estimating Pre-Layout Timing

The XSI libraries contain operating conditions and wire-load models that are used to provide a pre-layout timing estimate of your design.

Selecting the Operating Condition

XSI offers a set of operating condition parameters called worst-case commercial (WCCOM). The operating conditions are selected automatically if you used Synlibs to generate the link and target libraries. For more information on the Synlibs command, refer to the "Getting Started" chapter at the beginning of this user guide.

Setting the Wire-Load Models

The XSI libraries offer worst-case and average wire-load models. Wire loads are the estimated net delays for a design that has been partitioned into CLBs and IOBs. Refer to the "Using the Design Compiler" chapter in this user guide for more information.

Synopsys uses these estimates as guidelines to optimize your design for an FPGA. The actual wire loads cannot be determined until after the design has been placed and routed.

The models are device and speed-grade dependent, with an average wire-load model (*parttype-speedgrade*_avg) and a worst-case wire-load model (*parttype-speedgrade*_wc) for each. The average wire-load model is the mean of the test suite and the worst-case is the average plus one standard deviation. Therefore, the worst-case model is more conservative.

The average wire-load model is selected automatically if you used Synlibs to generate the link and target libraries.

Optimizing for Speed

Before compiling a design, you can set area and speed constraints to improve results. In this section you set a timing constraint. For the most effective results from the Design Compiler, the constraints must be accurate and achievable. For example, if a timing goal of 0 ns is set on all ports, the Design Compiler adds buffers to critical paths or duplicates logic on heavily loaded nets, attempting to achieve this goal. An unrealistic goal might cause significant and unwarranted area increases. Refer to the *Synopsys Design Compiler Reference Manual* for details on optimization techniques.

Path timing includes both logic and net delays. All gate timing delays are worst-case commercial estimates and are specified in

nanoseconds. The wire-load delays are either average estimates or worst-case estimates. Actual delays are determined only after you use PPR.

Additional timing information about primitives is included in the "XC3000/A/L and XC3100/A Primitives" appendix in this user guide and *The Programmable Logic Data Book.*

To set a clock constraint, follow these steps.

1. Select the CLOCK pin by placing the cursor on the CLOCK port and pressing the left mouse button.
2. Select the following menu options from the Attributes menu.

   Attributes → Clocks → Specify...

   The system displays the Specify Clock dialog box as follows. The default clock period is 50.

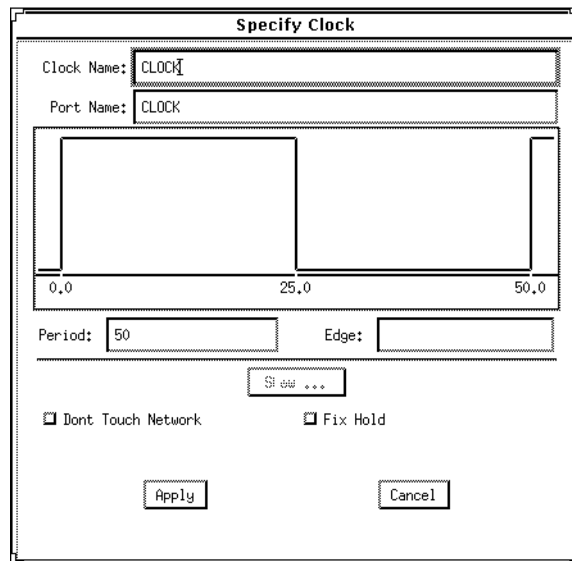


Figure 4-14 Specify Clock Dialog Box

3. Select **`Apply`**.

   A waveform appears above the CLOCK pin to indicate the setting of a timing constraint.

4. Select **`Cancel`** to close the dialog box.

Compiling the Design

In this section you learn how to compile a design with the recommended options.

The optimization process is part of the Compile command. Optimization is a complex series of transformations guided by constraints that you specify. One of the optimization steps is technology mapping, which transforms the Boolean logic network representation of your design into interconnected gates that are selected from the target technology library. You can set the mapping as Low, Medium, or High. Refer to the *Synopsys Design Compiler Reference Manual* for more details about mapping and other optimization techniques.

To compile the count8 design, do the following.

1. Select **`Tools → Design Optimization...`** from the Design Analyzer menu.

The Design Optimization dialog box appears as follows.

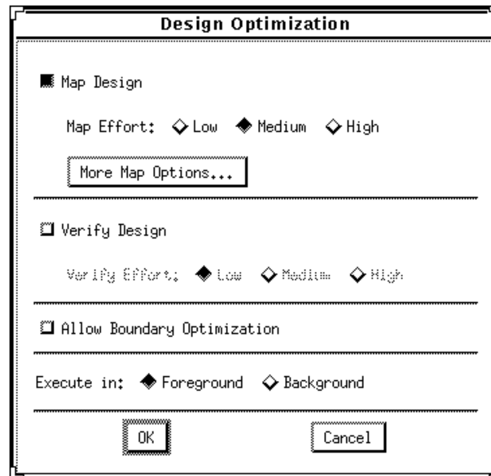


Figure 4-15 Design Optimization Dialog Box

2. Make sure the Map Design box is shaded and the Map Effort is Medium.
3. Click **OK**.

   The system displays any informational messages and compilation errors in the Compile Log window and the Command window.
4. Scroll through the Compile Log window to view the compilation messages.
5. Once the design is compiled, click **Cancel** to close the Compile Log window.

Evaluating the Results

The design is now optimized for the XC3000A architecture and mapped into primitive gates and registers.

The XSI libraries contain both area and timing information. In this section you view an area report on the estimated CLB and IOB utilization and a timing report on the estimated delays. You also learn how to redirect the report output from the screen to a file.

1. View a schematic of the design by selecting the gate picture icon on the left side of the Synopsys Design Analyzer window.

   The system displays a schematic view of the count8 design.

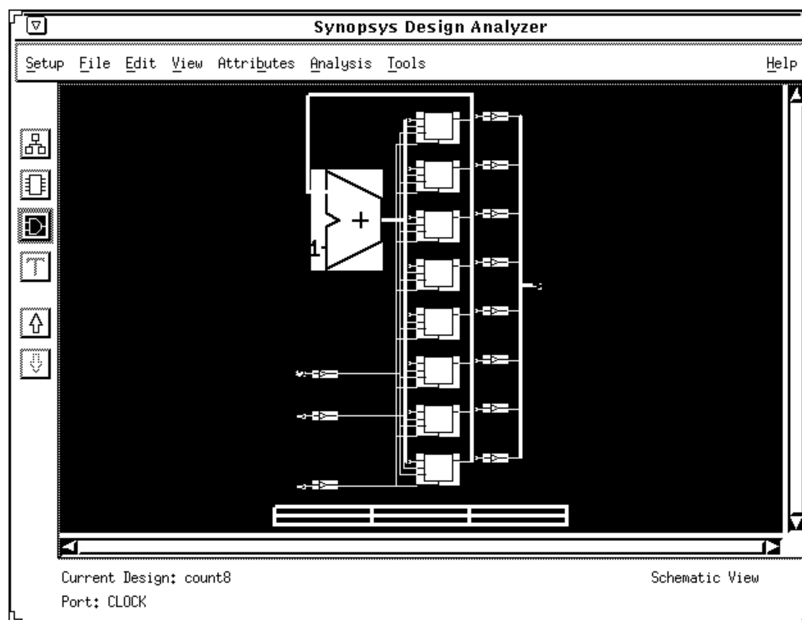


Figure 4-16 Schematic View

2. When you finish viewing the schematic, click on the up arrow icon to switch to the Designs View as illustrated by Figure 4-17.

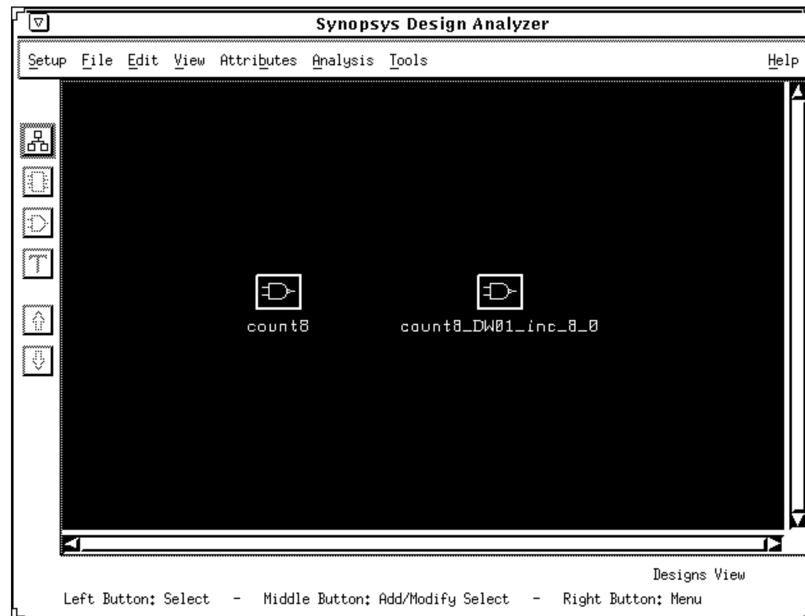


Figure 4-17 Designs View

Viewing the Estimated Area Results

To evaluate the estimated area results, perform the following steps.

1. Click on the count8 icon.
2. Select the following commands from the Design Analyzer menu.

   Analysis → Report...

   The Report dialog box appears as follows.

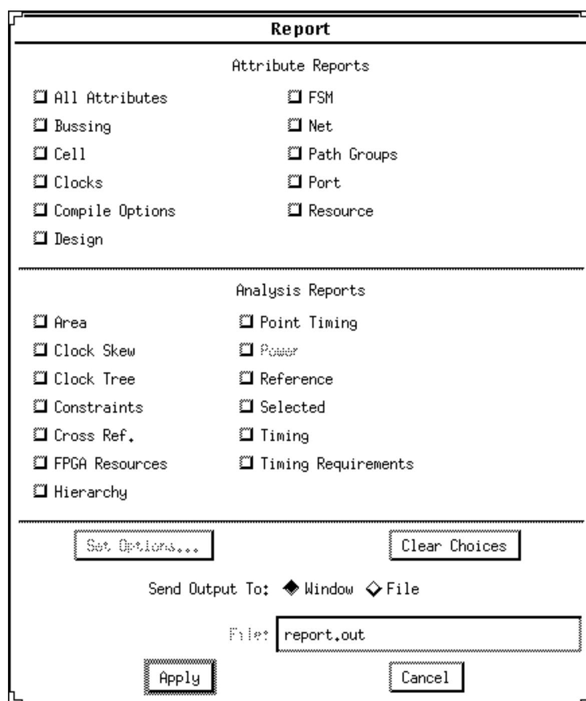


Figure 4-18 Report Dialog Box

3. In the Analysis Reports section, select the box next to Area.
4. Select `Apply`.

   The Report Output window appears as illustrated by Figure 4-19.

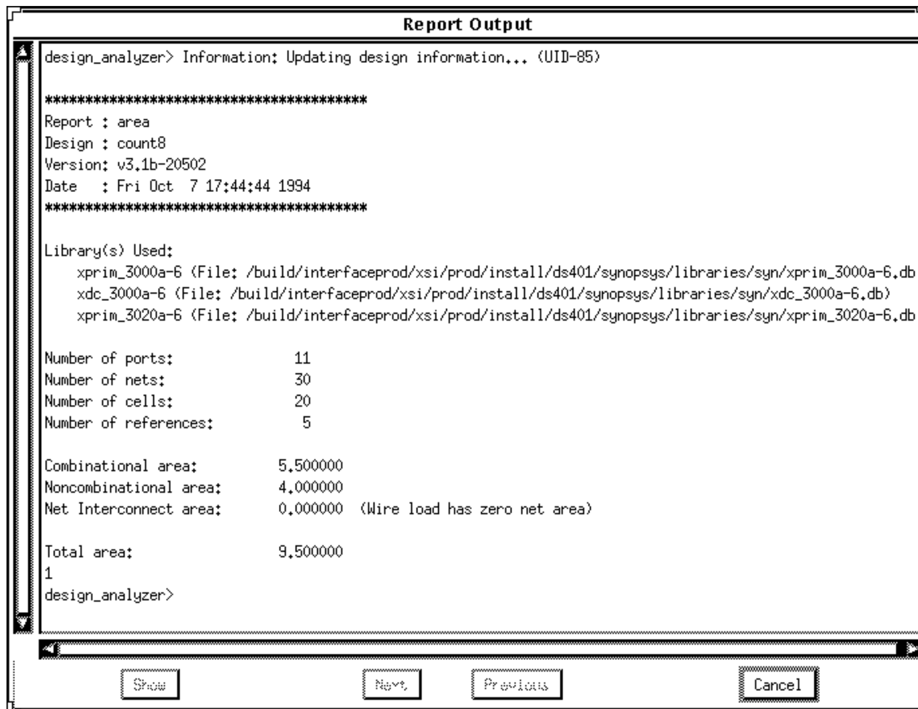


Figure 4-19 Report Output Window (Area)

5. Use the scroll bar in the Report Output window to view the design statistics.
6. Select `Cancel` to close the Report Output window.

Note: Do not close the Report dialog box.

Synopsys reports area in three parts — combinatorial area, non-combinatorial area, and total area. The area reported is in terms of the number of Xilinx CLBs used. Each CLB contains two 4-input function generators and two flip-flops. The flip-flops and 2-, 3-, and 4-input Boolean functions are weighted for area at 0.5 CLB. The 5-input primitives are weighted for area at 1 CLB.

If your design is register-intensive, the number of CLBs required is roughly equal to the non-combinatorial area reported. If the design is heavily combinatorial, the number of CLBs required is roughly equal

to the combinatorial area reported. However, the CLBs actually used are usually less than what Synopsys reports.

As a rule of thumb with Xilinx *mapped* libraries (Syn2XNF –map option) the minimum CLBs required is the larger of the combinatorial and non-combinatorial areas reported. The maximum number of CLBs required is the total number reported for both. The number of CLBs actually required is usually less than the total area, because the function generators and flip-flops often share the same CLB.

The rule of thumb with Xilinx *unmapped* libraries (Syn2XNF without options) is similar. The main difference is that the minimum number of CLBs required could be less than the combinatorial area reported, depending if PPR performs any local optimization.

Only PPR can accurately compute the actual number of required CLBs.

Viewing the Estimated Timing Results

To evaluate the timing results, perform the following steps.

1. In the Analysis Reports section of the Report dialog box, click on the box to the left of Timing with the left mouse button.
2. Deselect the Area box.
3. Select **`Apply`**.

   The Report Output window opens. The results reported are worst-case timing delay estimates. The final results cannot be determined until after you run PPR.
4. When you are finished reviewing the Report Output window, select **`Cancel`**.

Note: Do not close the Report dialog box.

Saving the Report Results to a File

To save the estimated timing results to a report file, perform the following steps.

1. Make sure only the Timing box is selected in the Analysis Reports section of the Report dialog box.

2. Locate the Send Output To field at the bottom of the Report dialog box and select **`File`**.
3. Place your cursor in the File field.
4. Double-click to highlight the default report file name.
5. Type **`count8.timing`**
6. Select **`Apply`** → **`Cancel`**.

Worst-case delays are used in Xilinx libraries, which assume that function generators and flip-flops are not in the same CLB. Synopsys timing delay estimates include wire-load delays in addition to gate delays. In most cases, actual results are better than the pre-placement and routing Synopsys estimates.

Figure 4-20 is the complete timing report for the count8 design, count8.timing.

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : count8
Version: v3.1b-20502
Date   : Thu Oct  6 18:26:05 1994
****************************************

Operating Conditions: WCCOM   Library: xprim_3020a-6
Wire Loading Model Mode: top

Design          Wire Loading Model      Library
------------------------------------------------
count8                3020a-6_avg       xprim_3020a-6


  Startpoint: QOUT_reg<0>
              (rising edge-triggered flip-flop clocked by CLOCK)
  Endpoint: QOUT_reg<6>
            (rising edge-triggered flip-flop clocked by CLOCK)
  Path Group: CLOCK
  Path Type: max

  Point                                                    Incr       Path
  --------------------------------------------------------------------------
  clock CLOCK (rise edge)                                  0.00       0.00
  clock network delay (ideal)                              0.00       0.00
  QOUT_reg<0>/C (FDCE)                                     0.00       0.00 r
  QOUT_reg<0>/Q (FDCE)                                     9.75       9.75 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/A_0 (count8_DW01_inc_8_0)
                                                           0.00       9.75 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/U4/O (AND3)            7.14      16.89 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/U5/O (AND2)            7.14      24.03 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/U6/O (AND2)            7.14      31.17 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/U7/O (AND2)            7.14      38.31 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/U9/O (XOR2)            6.24      44.55 r
  add_25/plus/LEFT_UNSIGNED_ARG_799/SUM_6 (count8_DW01_inc_8_0)
                                                           0.00      44.55 r
  QOUT_reg<6>/D (FDCE)                                     0.00      44.55 r
  data arrival time                                                  44.55

  clock CLOCK (rise edge)                                 50.00      50.00
  clock network delay (ideal)                              0.00      50.00
  QOUT_reg<6>/C (FDCE)                                     0.00      50.00 r
  library setup time                                       0.00      50.00
  data required time                                                 50.00
  --------------------------------------------------------------------------
  data required time                                                 50.00
  data arrival time                                                 -44.55
  --------------------------------------------------------------------------
  slack (MET)                                                         5.45
```

Figure 4-20 Timing Report (count8.timing)

Saving the Design

In this section you learn how to save your design as a DB (Synopsys Database file) file, set the design part type and speed grade, and save the design into an SEDIF file.

Writing the DB File

To save the design to a DB file, do the following.

1. Select the count8 icon with the left mouse button.
2. Select **File → Save** from the Design Analyzer menu.

The system saves the file as count8.db.

Setting the Design Part Type

To select a particular part for the count8 design, type the following command at the Design Analyzer prompt in the Command window.

Note: The \ (backslash) is a line continuation marker. Do not type it on the command line.

```
set_attribute count8 "part" -type \
string "3020apc84-6"↵
```

Saving the Design File as an SEDIF File

Next, save the design file as an SEDIF file as follows.

1. Select the following menu options.

   File → Save As...

   The Save File dialog box appears as follows.

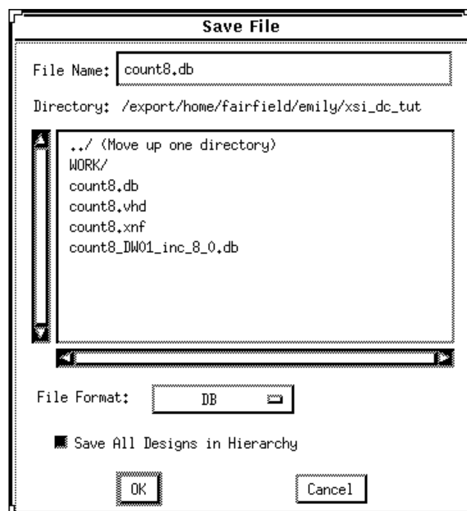


Figure 4-21 Save File Dialog Box

2. Click in the field next to File Name.
3. Change the .db extension to .sedif.

   Place your cursor to the right of **`count8.db`**, backspace to delete **`db`**, then type **`sedif`**.
4. Click on the bar next to File Format.

   The system displays a list of formats.
5. Select **`EDIF`**.
6. Make sure the Save All Designs in Hierarchy box is shaded.
7. Select **`OK`**.

Exiting the Design Analyzer

You are now done with the Synopsys Design Compiler and are ready to use the XACT Development System. To exit the Design Analyzer, do the following.

1. Select **`File`** → **`Quit`** from the Design Analyzer main menu.

   The Quit Design Analyzer? dialog box appears.

2. Click **OK** to exit.

The following section is a reference section that describes running a script that invokes the Synopsys tools.

To continue with the tutorial, skip to the “Placing and Routing Your Design Using XMake” section.

Executing the Commands from a Script File

Warning: Do not execute the commands in this section. Use this section as a reference for how to execute a script file. You have already executed these commands through the Design Analyzer menus.

You can use a script file to compile your design instead of using pull-down menus. The commands illustrated in this tutorial are all listed in a script file, count8.script.

You can execute this script either from the Design Analyzer or DC Shell. Each command is annotated in the script file. Comments start and end with /* and */. Each command corresponds to a command already executed in this tutorial.

The procedures to execute the count8.script file from the Design Analyzer are the following.

1. Invoke the Synopsys Design Analyzer in the background.

   design_analyzer &

2. Open the Command window to view the script as it executes.

   Setup → Command Window...

3. Execute the count8.script file.

   Setup → Execute Script...

   The Execute File dialog box appears as shown in Figure 4-22.

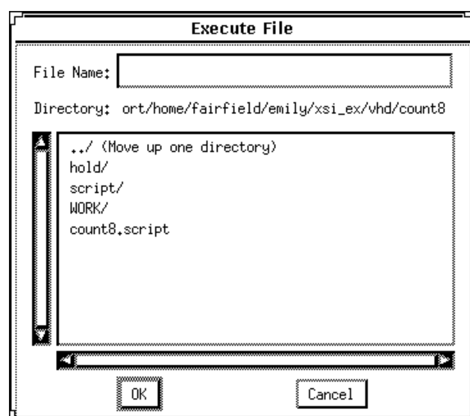


Figure 4-22 Execute File Dialog Box

4. Select **count8.script**.

   The system displays count8.script in the File Name field.

5. Select **OK**.

6. Exit the Design Analyzer.

Figure 4-23 and Figure 4-24 illustrate the actual text for the count8.script file for VHDL and Verilog HDL, respectively.

```
/* ===================================================*/
/* Script for Synopsys to Xilinx Design Compiler     */
/*             Count8 VHDL Tutorial                  */
/* ===================================================*/
   TOP = count8
   PART = "3020apc84-6"
   designer = "XSI Team"
   company  = "Xilinx, Inc"

   analyze -format vhdl TOP + ".vhd"
   elaborate TOP

   current_design TOP

   set_port_is_pad "*"
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads

   remove_constraint -all
   create_clock "CLOCK" -period 50

   compile

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

   set_attribute TOP "part" -type string PART
   write -format db -hierarchy -output TOP + ".db"
   write -format edif -hierarchy -output TOP + ".sedif"

   exit
```

Figure 4-23 VHDL Script File for Count8

```
/* ===================================================*/
/* Script for Synopsys to Xilinx Design Compiler     */
/*              Count8 Verilog Tutorial              */
/* ===================================================*/
   TOP = count8
   PART = "3020apc84-6"
   designer = "XSI Team"
   company  = "Xilinx, Inc"

   analyze -format verilog TOP + ".v"
   elaborate TOP

   current_design TOP

   set_port_is_pad "*"
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads

   remove_constraint -all
   create_clock "CLOCK" -period 50

   compile

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

   set_attribute TOP "part" -type string PART
   write -format db -hierarchy -output TOP + ".db"
   write -format edif -hierarchy -output TOP + ".sedif"


   exit
```

Figure 4-24 Verilog HDL Script File for Count8

Placing and Routing Your Design Using XMake

XMake automates the translation portion of the Xilinx design flow, which makes processing a complex design as simple as running one program.

Given the name of the top-level SEDIF file, XMake finds and processes all lower-level designs. It produces an LCA file that is placed and routed, as well as a BIT file ready for downloading to an FPGA. You can invoke XMake from within the XACT Design Manager (XDM) or from a shell tool window.

In this section you translate the count8 design using XMake from the shell tool window. Refer to the *XACT Reference Guide* for details about each program that XMake runs or about running XMake from XDM.

The procedure for translating the count8 design is slightly different if XSI is installed on a different platform or network than that of the

XACT Development System. Follow the procedures that apply to your specific configuration.

If XSI Is on Same Network as XACT Software

Follow the procedures in this section if the XSI software is installed on the same network or platform as the XACT Development System software. You can find the command-line options that XMake used in the XMake output file, count8.out.

To run XMake from a shell tool window, type the following.

```
xmake count8.sxnf
```

Refer to Figure 4-1 for a flow diagram that illustrates the Xilinx XC3000A implementation flow for synthesis.

If XSI Is on Different Network Than XACT Software

Follow the procedures in this section if the XSI software is installed on a different network or platform than the XACT Development System software.

If XSI is installed on a machine that does not have access to both the XSI and XACT Development System executable files, you must run Syn2XNF first and then copy the output XNF and XFF files to the platform where the XACT executable files reside.

Refer to Figure 4-1 for a flow diagram that illustrates the Xilinx XC3000A implementation flow for synthesis.

The following sections describe how to run the Syn2XNF and XMake programs.

Running Syn2XNF

Because the XSI software is installed on a different platform than the XACT software, you must first run Syn2XNF to translate your design into an XFF file, as follows.

1. Change to the directory where the count8.sedif file is located and execute the following command.

   ```
   syn2xnf count8
   ```

The SYN2XNF software might display the following message that prompts you to overwrite any existing XFF file that has the same design name.

```
WARNING: The file count8.xff already exists.
Do you want to overwrite it? (yes or no)
```

If the system displays the previous message, enter `y` ↵.

Syn2XNF creates the following output files: count8.xff, count8.xnf and syn2xnf.log.

2. Copy the count8.xff and count8.xnf files to the platform or network where the XACT software is installed.

Note: Use the Copy command with the –p option to preserve the time stamp information.

Running XMake

Perform the following steps to translate the count8 design using XMake.

1. Go to the platform where the XACT Software is installed.
2. Open a shell tool window.
3. Enter the following on the command line.

   `xmake -x count8`

   The –x option causes XMake to start with the XNF file and skip the translation process. The XMake program processes all the necessary design files, displaying its progress on the screen. If the translation is successful, XMake issues this message.

   ```
   'count8.bit' has been made, check output in
   'count8.out'
   ```

4. Be sure to examine the count8.out, count8.prp, and count8.rpt files for warnings and errors, as described in the "Examining XMake Output Files" section in this manual.

Examining XMake Output Files

In addition to the routed LCA file and the bitstream BIT file, XMake generates three very useful output files. In this section you open each file and familiarize yourself with its contents.

- The OUT file, count8.out, contains all the output from the programs that XMake invokes. This information is also displayed onscreen during processing.
- The PRP file, count8.prp, is the DRC (Design Rules Checker) report file generated by XNFPrep.
- The RPT file, count8.rpt, contains the PPR placement and routing results. This report also contains a listing of any unrouted pins or nets.

Reviewing the XMake OUT File

When you run XMake, the output of the XMake program appears on the screen. The OUT file shows every program run by XMake, the command options selected, and the output of each individual program.

Any warnings or errors produced by the programs run by XMake appear in the OUT file. You should always review the OUT file after running XMake, even if you did not see any warnings or error messages during design processing. If any warnings or errors do occur, you can save yourself some time by catching the problem now instead of later in the design process.

Examine the count8.out file for the count8 design as follows.

1. Open a shell tool window.
2. Change to the project directory.
3. Use a text editor to view count8.out.

Checking for Warnings and Errors in the PRP File

If XNFPrep finds any errors or warnings, the OUT file directs you to examine the PRP file. The PRP file also contains a detailed list of all logic trimmed by XNFPrep and why it was unnecessary. This file can be a useful debugging tool.

You should expect to see some warning messages in the count8.prp files but no errors.

Examine the count8.prp file for the count8 design.

The following headers correspond to the table of contents found in the count8.prp.

- XNFPrep Errors — lists all errors found in the design. Errors are problems with the design that cause XMake to terminate. You must fix any reported errors.
- XNPPrep Warnings — lists all warnings found in the design. Warnings notify you of any unusual aspects in your design. You should correct all warnings; however, it is not mandatory.
- Clock Signals Report — contains a summary of all the clock signals and/or global buffers to assist you in determining the best use of the global buffers. This section also contains a list of guidelines to consider when assigning signals to a global buffer.
- Timing Specification Summary — contains a list of the XACT-Performance timing specifications used in the design.
- Logic Trimming — shows the logic removed from your design due to sourceless or loadless signals and V_{CC} or ground connection. You should review this section to verify that logic required in your design has not been removed due to design error.

Checking the RPT File

After XMake runs PPR, PPR generates a report file with an .rpt extension. This file contains important information in the following categories.

- Partition, Place, and Route Summary — includes the number of occupied CLBs that approximately corresponds to the total area provided in the Synopsys Report.
- Chip Pinout Description — contains a list of the pins used in the design and any pin locations specified in the constraints file.
- Critical Nets — indicates any nets that were assigned a constraint.
- Feedthrough Split Nets — indicates any nets with names that were modified so the net could be re-powered. Re-powering or signal regeneration is accomplished by the net using a special function in a CLB.
- Deletion Traceback — enables you to check for any nets or cells that were removed that should remain. The Synopsys Check Design tool detects any unconnected pins or unused cells.

Examine the count8.rpt file to make sure there are no unrouted pins or nets. Use a text editor to view this file, which contains five pages. Figure 4-25 illustrates each page of the RPT file.

```
PPR RESULTS FOR DESIGN COUNT8                                    Page   1

    Design Statistics and Device Utilization
    ----------------------------------------

        Partitioned Design Utilization Using Part 3020APC84-6


                                          No. Used   Max Available   % Used
      ----------------------------      -------    -------------   ------
      Occupied CLBs                         8           64            12%
      ----------------------------      -------    -------------   ------
      Bonded I/O Pins:                      11          64            17%
      CLB Function Generators: (*)          12          128           9%
      CLB Flip Flops:                       8           128           6%
      IOB Input Flip Flops:                 0           64            0%
      IOB Output Flip Flops:                0           64            0%
      3-State Buffers:                      0           144           0%
      3-State Longlines:                    0           16            0%
      ----------------------------      -------    -------------   ------
      (*) Each base F or FGM function counts as two


    Routing Summary

      Number of unrouted connections: 0


    PPR Parameters

      Design          = count8.map
      Parttype        = 3020APC84-6
      Guide_cell      =
      Seed            = 781468140
      Estimate        = FALSE
      Complete        = TRUE
      Placer_effort   = 2
      Router_effort   = 2
      Path_timing     = TRUE
      Stop_on_miss    = FALSE
      DC2S            = none
      DP2S            = none
      DC2P            = none
      DP2P            = none
      Guide_only      = FALSE
      Ignore_maps     = FALSE
      Ignore_rlocs    = FALSE
      Outfile         = <design name>


    CPU Times

      Partition:                      00:00:00
      Placement:                      00:00:11
      Routing:                        00:00:07
      Total:                          00:00:25
```

```
PPR RESULTS FOR DESIGN COUNT8                                     Page   2

Xact Performance Summary
----------------------------------------

    Deadline  Actual(*)  Specification
    --------  ---------  -------------
      <auto>    29.2ns    DEFAULT_FROM_FFS_TO_FFS=FROM:ffs:TO:ffs
      <auto>    12.3ns    DEFAULT_FROM_PADS_TO_FFS=FROM:pads:TO:ffs
      <auto>    11.3ns    DEFAULT_FROM_FFS_TO_PADS=FROM:ffs:TO:pads

(*) Note: please use the -FailedSpec and/or -TSMaxpaths options of
the  Xdelay-TimeSpec command, accessible through the XDE or XDelay
program, to confirm the actual path delays computed by PPR.

    *** PPR: WARNING 7028:
The design has flip-flops with asynchronous set/reset controls
(PRE/SD or CLR/RD pins).  When PPR analyzes design timing, it does
not trace paths through the asynchronous set/reset input and on
through the Q output.

If you want PPR to control the delay on paths through asynchronous
set/reset pins, you must split the delay requirement into two
segments: one ending at the set/reset input, and the other beginning
at the flip-flop output. If you want PPR not to analyze paths that
lead to asynchronous set/reset pins, attach an IGNORE specification
to the pin(s) or signal(s).

By default, XDelay reports all paths through asynchronous set/reset
pins. To prevent XDelay from showing these paths, use FlagBlk
CLB_Disable_SR_Q on the appropriate flip-flops.
```

```
PPR RESULTS FOR DESIGN COUNT8                                   Page   3

Chip Pinout Description
-----------------------

This chapter describes where your design's pins were placed in terms
of the package pins.  This first list is sorted by package pin
location.  The second list presents the same data sorted by your
design's pin names.

Package Pin Location   Pin Name
--------------------   -----------------
P13                  : CLOCK
P39                  : ENABLE
P42                  : COUT<1>
P44                  : COUT<0>
P45                  : COUT<3>
P46                  : CLEAR
P47                  : COUT<2>
P49                  : COUT<5>
P56                  : COUT<6>
P57                  : COUT<7>
P59                  : COUT<4>

This list describes where your design's pins are in terms of the
package pins; it is sorted by your design's pin name.  The list
presented above has the same data sorted by package pin location.

Package Pin Location   Pin Name
--------------------   -----------------
P46                  : CLEAR
P13                  : CLOCK
P44                  : COUT<0>
P42                  : COUT<1>
P47                  : COUT<2>
P45                  : COUT<3>
P59                  : COUT<4>
P49                  : COUT<5>
P56                  : COUT<6>
P57                  : COUT<7>
P39                  : ENABLE
```

```
PPR RESULTS FOR DESIGN COUNT8                                  Page   4

Split Nets
-------------------
The list below identifies those signals which were routed through
CLBs or other blocks. In XDE and XDelay reports, these signals will
have two or more segments. An underscore (_) and a number will be
added to the end of the original signal name to identify the
different segments. To analyze all segments of a signal in XDelay or
QueryNet reports, append the original name with "_*" when prompted
for a signal name.

PPR may route signals through CLBs or other blocks in any of the
following situations:

* The delay on a signal might be reduced by routing it through a CLB,
given the extra flexibility in routing resources and the reduced
capacitive loading on the signal. PPR takes this into consideration.

* The delay on a signal might be reduced by sourcing it from two
block outputs instead of one, which is possible in some block
configurations. PPR will do this where possible.

* A signal on a global buffer may not be able to connect directly to
a load pin, given the placement of that load pin and the other global
resources which are used. PPR will pass the signal through another
CLB and route the load pin using general-purpose interconnect.

* In an XC4000 design, A BUFGP can be sourced only from an IOB. If
the design indicates that a BUFGP is driven from an internal source,
PPR will route the signal through the output path of the IOB in order
to access the BUFGP input.

Segments   Original Signal Name
--------   -------------------------------------------------------
```

```
PPR RESULTS FOR DESIGN COUNT8                                 Page   5

Information in Other Reports
----------------------------


Since not all pertinent design information is listed in this PPR
report file, this section describes where additional information can
be found.

Information                                 Report File  Created By
-----------------------------------------   -----------  ----------


Connection of signals between levels of     design.mrg   XNFMERGE
design hierarchy

Resolution of relative location (RLOC)      design.mrg   XNFMERGE
constraints through design hierarchy

X-BLOX designs: design rule check for       design.prx   XNFPREP
pre-expanded design

X-BLOX designs: results of optimization     design.blx   XBLOX
and module expansion in X-BLOX

Design rule check for invalid and/or        design.prp   XNFPREP
inefficient use of LCA architecture

Unused or disabled logic removed from       design.prp   XNFPREP
design, due to sourceless or loadless
signals and VCC or ground connections

XC3000A/L designs: Mapping of design        design.crf   XNFMAP
logic into each CLB or IOB

XC3000A/L designs: Summary of guided        design.crf   XNFMAP
partitioning results
```

Figure 4-25 RPT File

Comparing Actual Versus Estimated Area Results

The RPT file contains a partition, place, and route summary that includes the number of occupied CLBs that approximately corresponds to the total area number provided in the Synopsys Report.

In this section you compare how accurate the Design Compiler pre-placement and pre-routing estimates were to the actual results.

Figure 4-19 shows the estimated area results from the Design Compiler and Figure 4-25 shows the actual area results. The following table summarizes the area utilization results.

Table 4-1 Area Utilization Summary

Partitioned Design Utilization Using Part 3020APC84-6		
	Actual No. Used	Estimated No. Used
Occupied CLBs	8	9.5
Packed CLBs	N/A	N/A
Bonded I/O Pins	11	11
CLB Function Generators	12	N/A
CLB Flip-Flops	8	8
Clock Pads	1	1

The actual area utilization and timing may vary from the results reported by the Design Compiler because PPR performs the actual logic mapping.

Note: Your actual area numbers also may vary from the area utilization reported in the Design Compiler because PPR adds additional CLBs as feedthrough split nets. Refer to page 4 of the RPT file, illustrated by Figure 4-25, for more information on split nets for the count8 design.

Using XDelay

The XDelay command allows you to obtain detailed post-placement and post-routing timing information about your design.

The XDelay results summarize the worst paths for the design, not necessarily the paths that concern you. XDelay also has an interactive mode, which enables you to extract information about specific paths in the design, for example, to generate a timing report for a subset of the design. You can choose specific paths by selecting individual starting and ending points or by indicating a specific path type. For more information about XDelay and its options, refer to the *XACT Reference Guide.*

In this section you use XDelay to report the worst-case paths and the maximum clock frequency of the design. You also compare the output of XDelay to the estimated timing reported by the Design Compiler.

Invoking XDelay

Enter the following command at the command line to run XDelay, which creates a short report, count8.dly.

```
xdelay count8
```

XDelay produces the following output as shown in Figure 4-26.

```
XDelay Report File:

    Design: count8.lca (3020APC84-6)
    Program: xdelay 5.0.0
    Speedsfile: File 3020a.spd, Version 3000A.1, Revision 3020A.5

Xdelay timing analysis options:

From all.
To all.
 Worst case Pad to Pad path delay        :   27.3ns   (1 block level)
  Pad "CLEAR" (P76) to Pad "COUT<3>" (P67.O)


Clock net "N111" path delays:

 Pad to Setup                            :   38.5ns   (5 block levels)
  (Includes an external input margin of 0.0ns.)
  Pad "CLEAR" (P76) to FF Setup (D) at "N126.A"
   Target FFX drives output net "N126"
   Target FFY drives output net "N125"
   Target FFX drives its own D input
   Target FFX drives D input to FFY driving net "N125"

 Clock to Pad                            :   19.2ns   (0 block levels)
  (Includes an external output margin of 0.0ns.)
  Clock to Q, net "N129" to Pad "COUT<3>" (P67.O)

 Clock to Setup (same edge)              :   30.4ns   (4 block levels)
  Clock to Q, net "N132" to FF Setup (D) at "N126.A"
   Target FFX drives output net "N126"
   Target FFY drives output net "N125"
   Target FFX drives its own D input
   Target FFX drives D input to FFY driving net "N125"


                    Minimum Clock Period :   38.5ns

           Estimated Maximum Clock Speed :   26.0Mhz
```

Figure 4-26 XDelay Short Report

Comparing Actual Versus Estimated Timing Results

You can often get better timing estimates by looking at the number of block levels that the critical or longest path must traverse rather than using the estimated delays listed in the count8.timing report, illustrated in Figure 4-20.

Block levels are the number of CLBs and IOBs. The longest path reported in the Design Compiler was a clock-to-clock delay from a register through the incrementer. This delay was reported as 44.55 ns in the count8.timing report and included the clock-to-output delay, the clock-to-setup delay, and the average wire load.

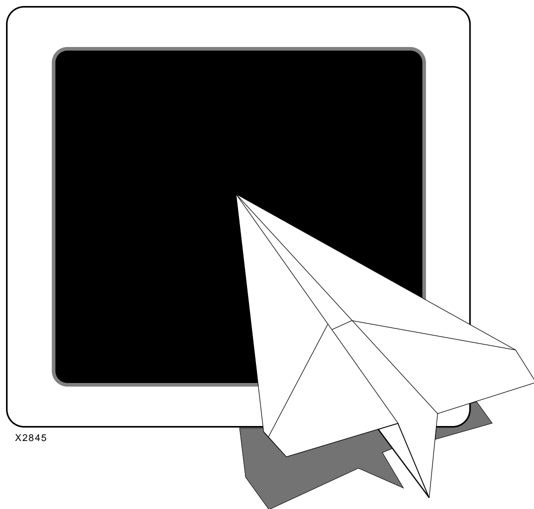
The XDelay report, illustrated in Figure 4-26, reports the longest clock-to-setup delay as 30.4 ns with four block levels. However, the short report does not include the delays from the clock pad to the clock net. To view a detailed timing report, refer to the XDelay chapter in the *XACT Reference Guide, Volume3*.

Verifying Your Design Using XChecker

This section describes the function of the XChecker Download/Readback cable. You do not actually download the count8 design in this tutorial.

To verify that your design works in your system, you can use the XChecker Download/Readback cable and associated software. With XChecker, you can load a configuration bitstream generated by the MakeBits program. The MakeBits file defines the internal logic functions and interconnections of the target FPGA. For more information on the XChecker cable or the MakeBits program, refer to the *XACT Hardware and Peripherals Guide* or the *XACT Reference Guide, Volume 2*, respectively.

You can store the BIT file in your system memory or in a PROM. Refer to the XPP section of the *XACT Hardware and Peripherals Guide* for more information about storing BIT files in PROMs.



Xilinx Synopsys Interface FPGA User Guide

Using the FPGA Compiler

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Chapter 5

Using the FPGA Compiler

The FPGA Compiler enables you to synthesize and implement your HDL design for Xilinx FPGA devices. The FPGA Compiler offers high-performance features that deliver efficient results and accurate timing and area reporting, as well as the following features.

- Logic optimization for the XC4000 family configurable logic block (CLB) and input/output block (IOB) architectures
- Timing and area constraints evaluations in terms of actual CLB and IOB utilization
- Area reports that list device usage, for example, CLBs, IOBs, and 3-state buffers
- Timing constraints passed to the XACT-Performance utility
- XNF (Xilinx Netlist Format) file reader for design reuse and back-annotation of post-route results
- DesignWare library that maps to X-BLOX modules, which generate optimized implementations of arithmetic functions

In addition, the FPGA Compiler optimizes flip-flops and latches into the I/O block, optimizes 3-state buffers into the I/O block, encodes for one-hot state machines, and uses the CLB Clock Enable pin automatically.

For best results, use the FPGA Compiler for XC4000/A/D/H devices. You can use the FPGA Compiler for XC3000 and XC3100 devices; however, the libraries for these devices use the Design Compiler synthesis tools so the results are the same.

Before You Begin

Before beginning a Xilinx design using the Synopsys tools, read the "Getting Started" chapter at the beginning of this manual, which describes how to do the following.

- Verify that the XSI and XACT Development System software is installed on your system
- Modify the Xilinx-provided default Synopsys startup file, if applicable

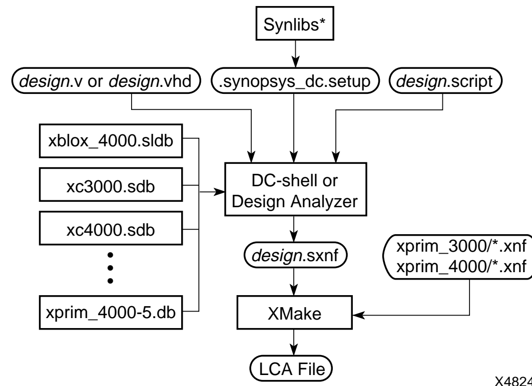
FPGA Compiler Design Flow

This section describes the FPGA Compiler design flow, which varies slightly depending on whether XSI is installed on the same platform or on a different platform than the XACT Development System software.

Proceed to the following section that applies to your system configuration.

XSI on Same Platform as XACT Software

Figure 5-1 shows the design flow for the FPGA Compiler based on a Xilinx XC4000 device and the FPGA Compiler-specific libraries when XSI software is installed on the same platform or network as the XACT Development System software.



*Append the Synlibs output to the .synopsys_dc.setup file. Refer to the "Getting Started" chapter for more information.

Figure 5-1 Design Flow with XSI Installed on the Same Platform

XSI on Different Platform than XACT Software

If XSI is installed on a machine that does not have access to both the XSI and XACT Development System executable files, you must copy or move the Syn2XNF output files, XFF and XNF, to the platform where the XACT executable files reside, as illustrated by Figure 5-2.

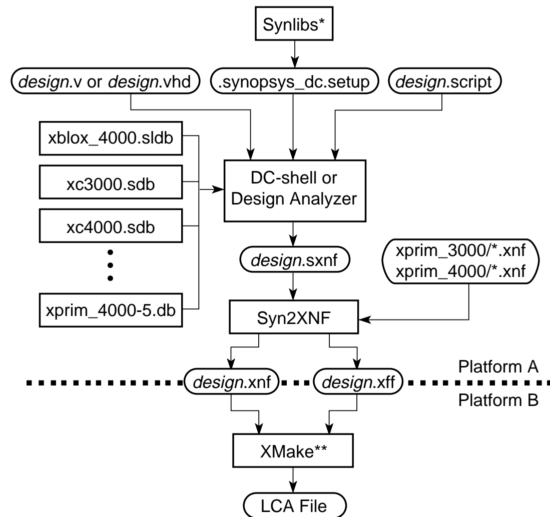


Figure 5-2 Design Flow with XSI Installed on a Different Platform

The basic flow for the different devices is the same. XMake automatically runs the appropriate mapping, placement, and routing tools. The FPGA Compiler-specific libraries, such as xfpga_4000–5.db, allow the FPGA Compiler to map directly to CLBs and IOBs. This direct mapping allows the most compatibility between the area and timing analysis created within the Synopsys environment for the final implementation in an FPGA.

See the “Files, Programs, and Libraries” chapter for additional library information.

Note: Although you can use the FPGA Compiler for XC3000A/L or XC3100/A designs, many of the commands in this section are specifically for XC4000 devices and are not available for the XC3000A/L and XC3100/A devices.

Setting the Wire-Load Model

Each primitive library contains estimated pre-layout and routing wire-load models that are device and speed-grade specific. The Synopsys tools can use these estimates when optimizing your design for an FPGA. XSI provides two wire-load models per device-speed grade combination — an average model and a worst-case model, designated by "_avg" and "_wc," respectively. The default wire load is *average*.

Wire-Load Models for Xilinx FPGAs

The following tables list the wire-load models for each Xilinx device. Substitute "_avg" or "_wc" for *a/w*, for example, 4003–4_wc.

Table 5-1 XC4000/A/D/H Wire-Load Models

–4 Speed Grade	–5 Speed Grade	–6 Speed Grade	–10 Speed Grade
	4002–5_*a/w*	4002–6_*a/w*	
4003–4_*a/w*	4003–5_*a/w*	4003–6_*a/w*	
	4004–5_*a/w*	4004–6_*a/w*	
4005–4_*a/w*	4005–5_*a/w*	4005–6_*a/w*	4005–10_*a/w*
4006–4_*a/w*	4006–5_*a/w*	4006–6_*a/w*	
4008–4_*a/w*	4008–5_*a/w*	4008–6_*a/w*	
4010–4_*a/w*	4010–5_*a/w*	4010–6_*a/w*	4010–10_*a/w*
4013–4_*a/w*	4013–5_*a/w*	4013–6_*a/w*	

Table 5-2 XC3000 Wire-Load Models

–50 Speed Grade	–70 Speed Grade	–100 Speed Grade	–125 Speed Grade
3020–50_*a/w*	3020–70_*a/w*	3020–100_*a/w*	3020–125_*a/w*
3030–50_*a/w*	3030–70_*a/w*	3030–100_*a/w*	3030–125_*a/w*
3042–50_*a/w*	3042–70_*a/w*	3042–100_*a/w*	3042–125_*a/w*
3064–50_*a/w*	3064–70_*a/w*	3064–100_*a/w*	3064–125_*a/w*
3090–50_*a/w*	3090–70_*a/w*	3090–100_*a/w*	3090–125_*a/w*

Table 5-3 XC3000A/L Wire-Load Models

–6 Speed Grade	–7 Speed Grade	–8 Speed Grade
3020a–6_*a/w*	3020a–7_*a/w*	3020l–8_*a/w*
3030a–6_*a/w*	3030a–7_*a/w*	3030l–8_*a/w*
3042a–6_*a/w*	3042a–7_*a/w*	3042l–8_*a/w*
3064a–6_*a/w*	3064a–7_*a/w*	3064l–8_*a/w*
3090a–6_*a/w*	3090a–7_*a/w*	3090l–8_*a/w*

Table 5-4 XC3100/A Wire-Load Models

–3 Speed Grade	–4 Speed Grade	–5 Speed Grade
3120–3_*a/w*	3120–4_*a/w*	3120–5_*a/w*
3120a–3_*a/w*	3120a–4_*a/w*	3120a–5_*a/w*
3130–3_*a/w*	3130–4_*a/w*	3130–5_*a/w*
3130a–3_*a/w*	3130a–4_*a/w*	3130a–5_*a/w*
3142–3_*a/w*	3142–4_*a/w*	3142–5_*a/w*
3142a–3_*a/w*	3142a–4_*a/w*	3142a–5_*a/w*
3164–3_*a/w*	3164–4_*a/w*	3164–5_*a/w*
3164a–3_*a/w*	3164a–4_*a/w*	3164a–5_*a/w*
3190–3_*a/w*	3190–4_*a/w*	3190–5_*a/w*
3190a–3_*a/w*	3190a–4_*a/w*	3190a–5_*a/w*
3195–3_*a/w*	3195–4_*a/w*	3195–5_*a/w*
3195a–3_*a/w*	3195a–4_*a/w*	3195a–5_*a/w*

Changing the Wire-Load Model

To change the wire load from average to worst case, use the Set Wire Load command as follows.

set_wire_load "*parttype–s_wc***"**

The speed grade for the wire-load model must match the speed grade of the primitive library as listed in the previous wire-load model tables as illustrated by this example.

```
set_wire_load "4005–5_wc"
```

If you want to evaluate the block delays of the design without the wire load, set the wire load to None by using the Set Wire Load command as follows.

```
set_wire_load none
```

How Wire-Load Models Are Determined

Average and worst-case models are derived from over 6000 designs that were placed and routed on the different Xilinx parts for each of the different speed grades.

The average wire-load model for a given part and speed grade is calculated by collecting all signal nets of a given fanout for all designs using the part type and speed grade. For a given fanout, 50 percent of the nets from the test suite are slower and 50 percent of the nets are faster than the delay number in the average wire-load model.

The worst-case wire-load models add one standard deviation to each average fanout value. For a given fanout, 68 percent of the nets from the test suite are faster and 22 percent are slower than the delay number in the worst-case wire-load model; therefore, the worst-case wire-load models are more conservative than the average wire-load models. You can determine the actual wire-load delays after placing and routing the design.

In all cases, the wire-load delays increase as the die size and fanout of the net increase. The delays decrease with faster device speed grades.

The Report Timing command combines the wire-load delay with the block delay. For more information on the Report Timing command, refer to the “Evaluating Timing Delays” section at the end of this chapter.

Operating Conditions

Only one set of operating condition parameters is available — worst-case commercial (WCCOM) — which is the default in the Xilinx libraries.

Configuring IOBs

The following section describes how to configure XC4000/A/D/H IOBs. The FPGA Compiler performs some optimization

automatically; however, you must implement some features manually. The FPGA Compiler performs the following functions automatically.

- Inserts input (IBUF) and output buffers (OBUF)
- Inserts input and 3-state output buffers for bidirectional I/O (IBUF, OBUFT, or IOBUF)
- Optimizes a flip-flop (IFD) or latch (ILD_1) attached to input buffers into the IOB
- Optimizes a flip-flop attached to output buffers into the IOB (OFD)
- Inserts a clock buffer for signals driving clock pins (BUFG)

First, you must enter these commands at either the DC shell or the Command window prompt to set the ports on the top-level design as pads and to insert the pads.

```
set_port_is_pad "*"
insert_pads
```

The Insert Pads command adds the correct buffers to the ports declared as pads with the Set Port Is Pad command.

The following sections provide a general description of XC4000/A/D/H devices and describes how to implement additional I/O features manually.

XC4000/A/D IOBs

This section describes how to configure the input and output signals, as well as how to set the output slew rate. You can configure XC4000/A/D IOBs as input, output, or bidirectional signals with or without a pull-up or pull-down resistor, independent of the pin usage.

Inputs

You can configure the buffered input signal that drives the data input of a storage element as either a flip-flop or a latch. You can use the buffered signal in conjunction with the input flip-flop or latch.

A delay buffer added to the signal feeding the data input of the input flip-flop/latch avoids a possible hold time violation. Instantiating a

flip-flop or latch, such as an IFD_F or ILD_1F, removes this delay because these cells include a NODELAY attribute. Refer to the “XC4000/A/H Primitives and Hard Macros” appendix for a complete list of primitives that include NODELAY attributes.

The FPGA Compiler optimizes any flip-flops connected to an input port into the IOB if the flip-flop or latch does not use the Clock Enable, Direct Clear, or Preset pin.

Outputs

The output signals, which can drive the programmable 3-state output buffer, can be registered or direct. The register is a positive-edge triggered flip-flop, and the clock polarity can be inverted inside the IOB. (PPR automatically optimizes any inverters in the IOB.)

The FPGA Compiler has the ability to optimize flip-flops attached to output pad in the IOB. However, the FPGA Compiler cannot optimize flip-flops in an IOB configured as a bidirectional pad. XC4000 output buffers can sink 12 mA and XC4000A output buffers can sink 24 mA.

XC4000/D Slew Rate

The XC4000 output buffers have a default slow slew rate that alleviates ground-bounce problems and the option of a fast slew rate that reduces the output delay. The SLOW option increases the transition time and reduces the noise level. The FAST option decreases the transition time and increases the noise level.

Warning: Synopsys and Xilinx define slew rate using opposite terms. Synopsys uses *slew control*, whereas Xilinx uses *slew rate*. For example, the Synopsys HIGH slew control is equivalent to the Xilinx SLOW slew rate.

There are two types of output buffers in the XSI libraries. The default output buffer has a FAST attribute assigned to it, that is, OBUF_F (output buffer) and OBUFT_F (3-state output buffer). However, to avoid a possible ground-bounce problem, Xilinx recommends that you select SLOW as the default slew rate. Assign a FAST slew rate only to output buffers that require additional speed.

The FPGA Compiler V3.1 or later automatically infers a FAST output slew rate. To set the default slew rate to SLOW (high control), use the following command.

```
set_pad_type -slewrate HIGH all_outputs()
```

Set this command before implementing the Insert Pads commands.

To change any output port to a FAST slew rate after changing the default to SLOW, use the following command. Replace *port* with the name of the output port.

set_pad_type -slewrate NONE {*port*}

Table 5-5 XC4000 Slew Rate Settings

Xilinx Slew Rate	Synopsys Slew Control Attribute	FPGA Compiler Command
SLOW	HIGH	**set_pad_type -slewrate HIGH** {*port*}
FAST	NONE	**set_pad_type -slewrate NONE** {*port*}

XC4000A Slew Rate

The XC4000A family offers more output slew-rate control options for each individual output drive: 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.

Warning: Synopsys and Xilinx define slew rate using opposite terms. Synopsys uses *slew control*, whereas Xilinx uses *slew rate*. For example, the Synopsys HIGH slew control is equivalent to the Xilinx SLOW slew rate.

The FPGA Compiler V3.1 or later automatically infers a FAST output slew rate. To set the default slew rate to SLOW (high control), use the following command.

```
set_pad_type -slewrate HIGH all_outputs()
```

Set this command before using the Insert Pads command.

To change an output to a FAST, MEDFAST or MEDSLOW slew rate after setting the default to SLOW, use the slew rate options found in the following table. Replace *port* with the name of the output port.

Table 5-6 XC4000A Slew Rate Settings

Xilinx Slewrate	Synopsys Slew Control Attribute	FPGA Compiler Command
SLOW	HIGH	**set_pad_type -slewrate HIGH** {*port*}
MEDSLOW	MEDIUM	**set_pad_type -slewrate MEDIUM** {*port*}
MEDFAST	LOW	**set_pad_type -slewrate LOW** {*port*}
FAST	NONE	**set_pad_type -slewrate NONE** {*port*}

The buffers have an _F suffix for FAST slew rate, an _MF suffix for MEDFAST, and an _MS suffix for MEDSLOW. Refer to the “XC4000/A/D/H Primitives and Hard Macros” appendix at the end of this user guide for a full listing of all cells that you can instantiate into a design.

Warning: The reported IOB timing delays reflect the delays for an XC4000 device, not an XC4000A device. XC4000A delays vary slightly from XC4000 delays. You can find the actual IOB delay numbers for the XC4000A devices in *The Programmable Logic Data Book*. You can use the Report Timing command to generate a timing report after invoking the Replace FPGA command to get accurate I/O cell delays. However, the delays for the internal gates are not accurate because no mapping information exists.

XC4000H IOBs

Because the XC4000H family almost doubles the number of input/output pins of XC4000 devices, the output drivers are more powerful and flexible. You can configure the XC4000H IOBs as input, output, or bidirectional signals. You can configure each I/O pad with or without a pull-up or pull-down resistor, independent of the pin usage.

Inputs

XC4000H devices contain no input flip-flops. You can configure each input individually with TTL or CMOS input thresholds. You must set the threshold level for each input. The buffers have a _CMOS suffix for the CMOS input threshold and a _TTL suffix for the TTL-input threshold. To set the input threshold, you must instantiate an input buffer with a CMOS or TTL threshold, or use the Set Pad Type command with the –exact option. Refer to the Synopsys documentation for more information on the Set Pad Type command.

Refer to the “XC4000/A/D/H Primitives and Hard Macros” appendix at the end of this user guide for a full listing of all cells.

Warning: If you do not specify the threshold, Synopsys assigns each input a random input threshold.

Use the following commands to set all inputs to CMOS or TTL.

- For the CMOS threshold, enter the following on the command line.

  set_pad_type –vih 3.33 –vil 1.05 all_inputs()

- For the TTL threshold, enter the following on the command line.

  set_pad_type –vih 2.0 –vil 0.8 all_inputs()

Note: You can use the All Inputs command to specify the names of all input ports; refer to your Synopsys documentation for more information.

You must set the input threshold *after* you compile the design. The following is an example set of commands you can use to compile, set the input threshold, set the output threshold, insert the pads, and then replace the CLBs and IOBs with gates.

```
compile
set_pad_type –vih 3.33 –vil 1.05 all_inputs()
set_pad_type –voh 4.75 –voh 0.6 all_outputs()
set_port_is_pad
insert_pads
replace_fpga
```

Figure 5-3 Example Compilation Flow for Setting Input and Output Thresholds

Outputs

XC4000H devices contain no output flip-flops. You can individually configure the outputs as either TTL- or CMOS-compatible. TTL-level outputs are the best choice for systems that use TTL-level input thresholds. CMOS-level outputs are ideal for systems that use CMOS input thresholds. The default output threshold is CMOS. To change the output threshold, you must instantiate an output or bidirectional buffer with a TTL or CMOS threshold, or use the Set PadType command with the –exact option. Refer to the Synopsys documentation for more information on the Set Pad Type command. The output and bidirectional cells are listed in the "XC4000/A/D/H Primitives and Hard Macros" appendix.

You must set the threshold level for each output. If you do not specify the threshold, Synopsys assigns each output a random output threshold. Use the following commands to set the output threshold.

- For the CMOS threshold, enter the following on the command line.

  ```
  set_pad_type -voh 4.75 -vol 0.6 all_outputs()
  ```

- For the TTL threshold, enter the following on the command line.

  ```
  set_pad_type -voh 2.4 -vol 0.5 all_outputs()
  ```

Note: Use the All Outputs command to specify all output ports.

You must set the output threshold *after* you compile the design. Figure 5-3 is an example set of commands used to compile, set the input threshold, set the output threshold, insert the pads, and then replace the CLBs and IOBs with gates.

Warning: XC4000H devices do not have flip-flops in the IOBs. To prevent the FPGA Compiler from pulling any flip-flops into the IOBs, insert the pads after compiling the design.

Note: The IOB timing delays reported for XC4000H devices are not included. Execute the Report Timing command after running the Replace FPGA command to report accurate I/O cell delays. However, the reported internal gate delays are not accurate.

XC4000H Slew Rate

The XC4000H family offers a choice of CMOS- or TTL-level output and input thresholds that you can select per pin. XC4000H devices have a capacitive and a resistive slew rate. The XC4000H outputs sink 24 mA.

You can configure each output for either of two slew-rate options, which affect only the pull-down operation — resistive or capacitive.

The resistive load (RES) has a pull-down transistor that is driven hard, resulting in a practically constant on-resistance of about 10 ohms. Selecting the resistive load results in the fastest High-to-Low transition and the capability to sink 24 mA with a voltage of 500 mV. Many outputs switch High to Low simultaneously, especially when they are discharging a capacitive load, which might result in excessive ground bounce.

When the output is configured for a capacitive load (CAP) or soft edge, the High-to-Low transition starts as described previously, but the drive to the pull-down transistor is reduced as soon as the output voltage reaches a value around 1V. Selecting a capacitive load results in a higher resistance in the pull-down transistor, slowing down of the falling edge, and significantly reduced ground bounce. Refer to *The Programmable Logic Data Book* for more details.

To change any of the output ports to a capacitive slew rate, use the Set Pad Type command. Replace *port* with the name of the output port.

set_pad_type -slewrate HIGH {*port*}

Table 5-7 XC4000H Slew Rates

Xilinx Slew Rates	Synopsys Slew Control Attribute	FPGA Compiler Command
CAP	HIGH	**set_pad_type -slewrate HIGH** {*port*}
RES	NONE	**set_pad_type -slewrate NONE** {*port*}

Note: Set this command after specifying the Set Port Is Pad command and before using the Insert Pads command.

For bidirectional cells, the input threshold is listed before the output threshold and slew rate. Refer to the “XC4000/A/D/H Primitives

and Hard Macros" appendix for a complete listing of all cells that you can instantiate into a design.

Assigning and Prohibiting Pad Locations

You can specify pad locations by either using the PPR constraints file (CST) or typing the following in the Command window.

```
set_attribute pad "pad_location" -type string \
"pin number"
```

Refer to *The Programmable Logic Data Book* for the locations and name of the pins.

Note: Pin names do not always start with a P.

For more information about the PPR constraints file, refer to the *XACT Reference Guide, Volume 2* or the *XACT Libraries Guide*.

Implementing 3-State Registered Output

For the FPGA Compiler to infer the use of 3-state output flip-flops, such as OFDT, two conditions must be met: the flip-flop must directly drive the 3-state signal and the HDL code of the flip-flop must be in the same process as the 3-state HDL code. The following sections illustrate a flip-flop that does not directly drive the 3-state signal and one that does directly drive the 3-state signal.

Not Directly Driving the 3-State Signal

The flip-flop must directly drive the 3-state signal. If any logic exists between the flip-flop and the 3-state signal connected to the output flip-flop, the FPGA Compiler does not infer a 3-state output flip-flop. Figure 5-4 and Figure 5-5 illustrate a flip-flop that is not directly driving a 3-state output flip-flop. Figure 5-6 is a schematic representation.

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

entity three_ex1 is
   port (BUS_IN, EN, CLK: in STD_LOGIC;
         BUS_OUT: out STD_LOGIC);
end three_ex1;

architecture BEHAVIORAL of three_ex1 is

signal BUS_IN_REG, BUS_OUT_REG: STD_LOGIC;

begin
sync: process (CLK)
   begin
        if (CLK´event and CLK=´1´) then
           BUS_IN_REG <= BUS_IN;
           BUS_OUT_REG <= BUS_IN_REG;
        end if;
   end process;

BUS_OUT <= BUS_OUT_REG when (EN=´0´) else ´Z´;

end BEHAVIORAL;
```

Figure 5-4 Register Not Directly Driving 3-State (VHDL)

```
/*
 * three_ex1 - Behavioral Model
 * Example of 3-state assignment NOT in clock process
 * XSI v3.2
 *      @(#)three_ex1.v 1.2     8/4/94
 */

module three_ex1(BUS_IN, EN, CLK, BUS_OUT) ;
input   BUS_IN ;
input   EN ;
input   CLK ;
output  BUS_OUT ;

reg     BUS_OUT_REG, BUS_IN_REG, BUS_OUT;


//assign BUS_OUT = (EN == 1´b0) ? BUS_OUT_REG : 1´bz ;

always @(posedge CLK)
begin
   BUS_OUT_REG = BUS_IN_REG ;
   BUS_IN_REG = BUS_IN ;
end

always @(EN or BUS_OUT_REG)
begin
   if (!EN)
     BUS_OUT = BUS_OUT_REG;
   else
     BUS_OUT = 1´bz;
end

endmodule
```

Figure 5-5 Register Not Directly Driving 3-State (Verilog HDL)

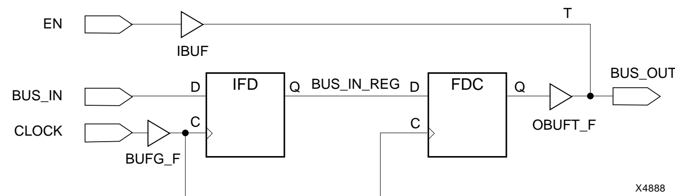


Figure 5-6 No Output Register Inferred

Directly Driving the 3-State Signal

The HDL code for the flip-flop must be in the same process as the 3-state HDL code and must directly drive the 3-state output, as shown in the “sync” process in Figure 5-7 and Figure 5-8. Having the flip-flop and the 3-state signal in separate processes causes the insertion of additional logic between the flip-flop and the 3-state

signal. If these two conditions are met, the FPGA Compiler infers a registered 3-state output, as illustrated by Figure 5-9.

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

entity three_ex2 is
   port (BUS_IN,EN,CLK: in STD_LOGIC;
         BUS_OUT: out STD_LOGIC);
end three_ex2;

architecture BEHAVIORAL of three_ex2 is

signal BUS_IN_REG: STD_LOGIC;

begin
sync: process (CLK, EN)
   begin
        if (CLK'event and CLK='1') then
           BUS_IN_REG <= BUS_IN;
           if (EN='0') then
                BUS_OUT <= BUS_IN_REG;
           else
                BUS_OUT <= 'Z';
           end if;
        end if;
   end process;

end BEHAVIORAL;
```

Figure 5-7 Register and 3-State Output in the Same Process (VHDL)

```
/*
 * three_ex2 - Behavioral Model
 * Example of 3-state assignment in the the clock process
 * XSi v3.2
 *      @(#)three_ex2.v         8/19/94
 */

module three_ex2(BUS_IN, EN, CLK, BUS_OUT) ;
input   BUS_IN ;
input   EN ;
input   CLK ;
output  BUS_OUT ;

reg     BUS_OUT ;
reg     BUS_IN_Q, BUS_IN_REG ;

always @(posedge CLK)
begin
   BUS_IN_REG = BUS_IN_Q ;
   BUS_IN_Q = BUS_IN ;
   if (!EN) BUS_OUT = BUS_IN_REG;
   else  BUS_OUT = 1´bz;
end

endmodule
```

Figure 5-8 Register and 3-State Output in the Same Process (Verilog HDL)

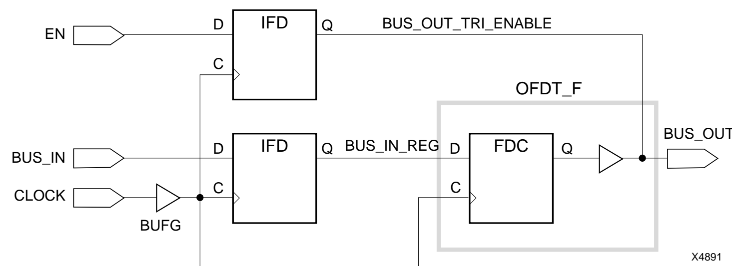


Figure 5-9 Output Register Inferred

Inserting Bidirectional I/Os

The FPGA Compiler has the ability to insert non-registered bidirectional ports. The 3-state signal that drives the output buffer must be described in the same hierarchy level as the input signal, as shown in Figure 5-10 and Figure 5-11.

Instantiating a Registered Bidirectional I/O

The VHDL design, bidi_reg.vhd, and the Verilog HDL design, bidi_reg.v, are examples of a top-level design that instantiates a core design, reg4. 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 the OFDT_F (3-state registered output buffers with a FAST slew rate) cells in bidirectional I/Os. Therefore, these cells and the IBUF are instantiated into the top-level design.

```
entity bidi_reg is
        port (SIGA: inout BIT_VECTOR (3 downto 0);
              LOADA, CLOCK1, CLOCK2, RST: in BIT);
end bidi_reg;

architecture BEHAV of bidi_reg is
        component reg4
           port (INX: in BIT_VECTOR (3 downto 0);
                 LOAD, CLOCK, RESET: in BIT;
                 OUTX: buffer BIT_VECTOR (3 downto 0));
        end component;

        component OFDT_F
           port (D: in BIT;
                 C: in BIT;
                 T: in BIT;
                 O: out BIT);
        end component;

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

signal INA, OUTA: BIT_VECTOR (3 downto 0);
begin
        U5: reg4 port map (INA, LOADA, CLOCK1, RST, OUTA);
        U0: OFDT_F port map (OUTA(0), CLOCK2, LOADA, SIGA(0));
        U1: OFDT_F port map (OUTA(1), CLOCK2, LOADA, SIGA(1));
        U2: OFDT_F port map (OUTA(2), CLOCK2, LOADA, SIGA(2));
        U3: OFDT_F port map (OUTA(3), CLOCK2, LOADA, SIGA(3));
        U4: IBUF port map (SIGA(0), INA(0));
        U6: IBUF port map (SIGA(1), INA(1));
        U7: IBUF port map (SIGA(2), INA(2));
        U8: IBUF port map (SIGA(3), INA(3));
end BEHAV;
```

Figure 5-10 Bidi_reg.vhd

```
/*
 * bidi_reg - Structural Model
 * Register Bidirectional I/O Example
 * XSI v3.2
 *      @(#)bidi_reg.v  8/19/94
 */

module bidi_reg (SIGA, LOADA, CLOCK1, CLOCK2, RST) ;
inout   [3:0]   SIGA ;
input           LOADA ;
input           CLOCK1 ;
input           CLOCK2 ;
input           RST;

wire    [3:0]   INA, OUTA ;
// Netlist

reg4 U5 (INA, LOADA, CLOCK1, RST, OUTA) ;


OFDT_F  U0 (.D(OUTA[0]), .C(CLOCK2), .T(LOADA), .O(SIGA[0])) ;
OFDT_F  U1 (.D(OUTA[1]), .C(CLOCK2), .T(LOADA), .O(SIGA[1])) ;
OFDT_F  U2 (.D(OUTA[2]), .C(CLOCK2), .T(LOADA), .O(SIGA[2])) ;
OFDT_F  U3 (.D(OUTA[3]), .C(CLOCK2), .T(LOADA), .O(SIGA[3])) ;
IBUF    U4 (.I(SIGA[0]), .O(INA[0])) ;
IBUF    U6 (.I(SIGA[1]), .O(INA[1])) ;
IBUF    U7 (.I(SIGA[2]), .O(INA[2])) ;
IBUF    U8 (.I(SIGA[3]), .O(INA[3])) ;

endmodule
```

Figure 5-11 Bidi_reg.v

Compiling Bidirectional I/O

Do not use the Set Port Is Pad command for the instantiated I/O cells. For example, in the bidi_reg.vhd example, you would use the following commands to insert the I/Os for the LOADA, RST, CLOCK1, and CLOCK2 signals only.

```
set_port_is_pad {LOADA RST CLOCK1 CLOCK2}

insert_pads
```

Before compiling the design, you must place a Don't Touch attribute on any instantiated I/O cells as follows, so that the I/O cells are not altered.

```
dont_touch {U0 U1 U2 U3 U5 U6 U7 U8}
```

The script files used to compile bidi_reg.vhd and bidi_reg.v are shown in Figure 5-12 and Figure 5-13, respectively.

```
/* ==================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*            the FPGA Compiler                     */
/*        Bidirectional Register Example.           */
/* ==================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Read in the design                    */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design     */

   TOP = bidi_reg
   SUB = reg4

/* Set the Designer and Company name for
   documentation.                                    */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                           */

   analyze -format verilog SUB + ".v"
   analyze -format verilog TOP + ".v"
   elaborate SUB
   elaborate TOP

/* Set the current design to the top level           */

   current_design TOP

/* Add pads to the design. Make sure the current
   design is the top-level module.                  */

   set_port_is_pad {LOADA RST CLOCK1 CLOCK2}
   insert_pads
   dont_touch {U0 U1 U2 U3 U4 U6 U7 U8}


/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Compile the design                  */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.             */

   remove_constraint -all

/* Synthesize and optimize the design                */

   compile -map_effort med
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Save the design                       */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                       */

   report_fpga > TOP + ".fpga"
   report_timing > TOP + ".timing"

/* Write out the design to a DB file                  */

   write -format db -hierarchy -output TOP + ".db"

/* Replace CLBs and IOBs with gates                   */

   replace_fpga

/* Set the part type                                  */

   set_attribute TOP "part" -type string "4005pc84-5"

/* Save design in XNF format as <design>.sxnf         */

   write -format xnf -hierarchy -output TOP + ".sxnf"


/* Exit the Compiler.                                 */

   exit
```

Figure 5-12 Bidi_reg.script (VHDL)

```
/* ==================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*            the FPGA Compiler                      */
/*       Bidirectional Register Example.             */
/* ==================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Read in the design                   */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design     */

   TOP = bidi_reg
   SUB = reg4

/* Set the Designer and Company name for
   documentation.                                    */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                           */

   analyze -format verilog SUB + ".v"
   analyze -format verilog TOP + ".v"
   elaborate SUB
   elaborate TOP

/* Set the current design to the top level           */

   current_design TOP

/* Add pads to the design. Make sure the current
   design is the top-level module.                  */

   set_port_is_pad {LOADA RST CLOCK1 CLOCK2}
   insert_pads
   dont_touch {U0 U1 U2 U3 U4 U6 U7 U8}


/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Compile the design                  */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.             */

   remove_constraint -all

/* Synthesize and optimize the design                */

   compile -map_effort med
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Save the design                        */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                       */

   report_fpga > TOP + ".fpga"
   report_timing > TOP + ".timing"

/* Write out the design to a DB file                  */

   write -format db -hierarchy -output TOP + ".db"

/* Replace CLBs and IOBs with gates                   */

   replace_fpga

/* Set the part type                                  */

   set_attribute TOP "part" -type string "4005pc84-5"

/* Save design in XNF format as <design>.sxnf         */

   write -format xnf -hierarchy -output TOP + ".sxnf"


/* Exit the Compiler.                                 */

   exit
```

Figure 5-13 Bidi_reg.script (Verilog HDL)

Using Unbonded IOBs (XC4000/A 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. Synopsys cannot infer unbonded primitives.

Unbonded primitives are indicated by a _U suffix. Refer to the "XC4000/A/D/H Primitives and Hard Macros" appendix for a complete listing of all unbonded cells.

Adding Pull-Up and Pull-Down Resistors

XC3000 and XC3100 devices have pull-up resistors that you can use to pull up an unconnected IOB. By default, all unused IOBs are configured as an input with a pull-up resistor. Refer to the "XC3000/A/L and XC3100/A Primitives" appendix for a listing of all cells and their pin names for instantiation.

The XC4000 family has high impedance pull-up and pull-down resistors that you can connect to an input or output buffer. You can

instantiate these cells, PULLUP and PULLDOWN, into your HDL design. Refer to the “XC4000/A/D/H Primitives and Hard Macros” appendix for a listing of all cells and their pin names for instantiation.

Removing the Default Input Delay

The XC4000 input flip-flops and latches have a default delay preceding the data to the input flip-flop or latch. This delay prevents any possible hold-time violations if you have a clock signal that is also coming into the device and clocking the input flip-flop or latch.

You can remove this delay by instantiating a cell that includes the NODELAY attribute if you need additional input speed and have no possibility of a hold-time violation. The “XC4000/A/D/H Primitives and Hard Macros” appendix lists all cells that include a NODELAY attribute. Input flip-flops or latches with an _F suffix have a NODELAY attribute assigned to the cell.

Initializing the IOB Flip-Flop to Preset

You can initialize XC4000 IOB flip-flops to either Clear or Preset. The default is Clear. To initialize an I/O flip-flop or latch to Preset, use the following command to attach an INIT=S attribute to the flip-flop.

```
set_attribute "register_name" xnf_init \
"S" type string
```

Replace *register_name* with the name of the I/O flip-flop.

You can instantiate I/O cells with the INIT=S attribute already assigned to them. Refer to the “XC4000/A/D/H Primitives and Hard Macros” appendix for a list of all cells and their pin names for instantiation.

Inserting Clock Buffers

Xilinx recommends that your design contain global clock buffers to take advantage of the low-skew, high-drive capabilities of the primary clock buffer, BUFGP, and the secondary clock buffer, BUFGS. When you use the Insert Pads command, the FPGA Compiler automatically inserts a BUFG generic clock buffer whenever an input signal drives a clock signal. XNFPrep selects a BUFGS cell. If you want to use a BUFGP, you must instantiate it.

However, you can instantiate a BUFGS or BUFGP if you understand the architecture and want to specify how the resources should be used. Each XC4000 device contains four primary and four secondary global buffers that share the same routing resources. Xilinx recommends that you use the generic global buffer, BUFG, for up to four low-skew, high-fanout clock signals. Both the primary and secondary clock buffers can be driven by signals sourced from inside the device; however, the difference is that the primary global buffer always uses the dedicated I/O pad.

You can use secondary global buffers to buffer high-fanout, low-skew signals that are sourced from inside the FPGA. To access the secondary global clock buffer for an internal signal, instantiate the BUFGS_F cell. The timing in the BUFGS does not include the pad delay.

Controlling Clock Buffer Insertion

Because the FPGA Compiler assigns a BUFG to any input signal that drives a clock signal, your design may contain too many clock buffers. The following examples illustrate how to control clock buffer insertion.

Figure 5-14 and Figure 5-15 illustrate a gated clock example using VHDL and Verilog HDL, respectively. By default, Synopsys assigns the signals IN1, IN2, IN3, IN4, and CLK to a BUFG since they are connected to a clock pin.

```
entity gate_clock is
   port (IN1,IN2,IN3,IN4,IN5,CLK,LOAD: in BIT;
         OUT1: buffer BIT);
end gate_clock;

architecture BEHAVIORAL of gate_clock is
signal GATECLK: BIT;
begin
GATECLK <= not((((IN1 and IN2) and IN3) and IN4) and CLK);
   process (GATECLK, IN5,LOAD)
   begin
        if (GATECLK´event and GATECLK=´1´) then
                   if (LOAD=´1´) then
                        OUT1 <= IN5;
                   else
                        OUT1 <= OUT1;
                   end if;
        end if;
   end process;
end BEHAVIORAL;
```

Figure 5-14 Gated Clock (VHDL)

```
/*
 * Gateclk - Behavioral Model
 * Gated Clock Example
 * XSI v3.2
 *      @(#)gate_clock.v        1.2     8/4/94
 */

module gate_clock(IN1, IN2, IN3, IN4, IN5, CLK, LOAD, OUT1) ;
input   IN1 ;
input   IN2 ;
input   IN3 ;
input   IN4 ;
input   IN5 ;
input   CLK ;
input   LOAD ;
output  OUT1;

reg     OUT1;

wire GATECLK ;

assign GATECLK = ~(IN1 & IN2 & IN3 & IN4 & CLK) ;

always @(posedge GATECLK)
begin
   if (LOAD == 1´b1)
      OUT1 = IN5 ;
end

endmodule
```

Figure 5-15 Gated Clock (Verilog HDL)

In Figure 5-16, the FPGA Compiler should *not* insert a clock buffer on the signals IN1 through IN4 and CLK because they do not directly drive the clock pin (C); for example, the clock is gated.

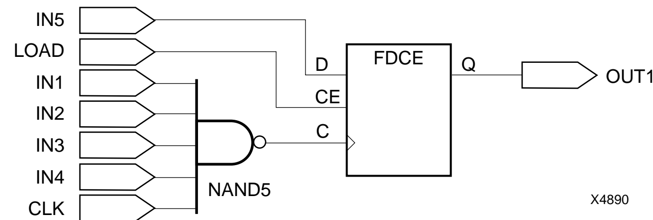


Figure 5-16 Gated Clock Schematic

Note: The FPGA Compiler identifies clock ports by tracing back from the clock pins on the flip-flops. If the clock signal is gated as illustrated in Figure 5-16, the gated signals are also assigned a clock buffer.

In Figure 5-17, the inputs to the 5-input NAND gate all have a BUFG inserted.

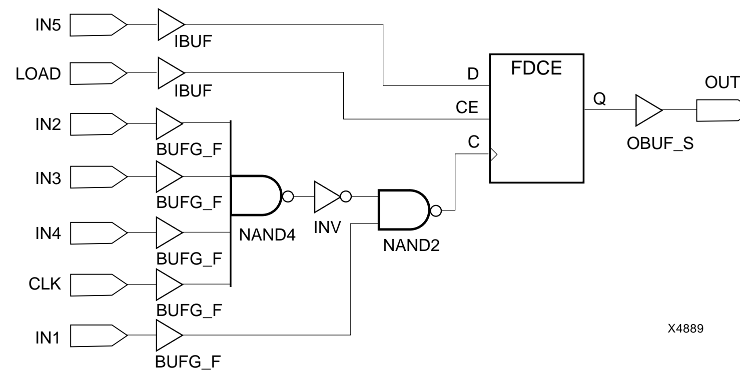


Figure 5-17 Gated Clock After Pad Insertion

If the design contains gated clocks or has more than four input pins that drive the clock pin in the design, you must disable the input pins from having a BUFG inserted.

Determining the Number of Clock Buffers

To determine how many clock buffers the FPGA Compiler will insert in the design, use the Report FPGA command as follows.

```
report_fpga
```

Figure 5-18 illustrates the report output of the Report FPGA command on the previous gated clock example.

```
****************************************
Report : fpga
Design : gate_clock
Version: v3.1b-20502
Date   : Thu Sep  8 09:22:21 1994
****************************************

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

    FG Function Generators:          1
    H Function Generators:           1
    Number of CLB cells:             1
    Number of Hard Macros and
        Other Cells:                 0
    Number of CLBs in
        Other Cells:                 0
    Total Number of CLBs:            1

    Number of Ports:                 8
    Number of Clock Pads:            5
    Number of IOBs:                  3

    Number of Flip Flops:            1
    Number of 3-State Buffers:       0

    Total Number of Cells:           9
```

Figure 5-18 Report FPGA for Gated Clock Example

Note: Clock pads are IOBs, yet they are listed separately in this report.

Preventing the Insertion of Clock Buffers

To prevent the FPGA Compiler from inserting the BUFG primitive, specify the Set Pad Type command with the following options before inserting the pads.

set_pad_type -no_clock {*clock_ports*}

Replace *clock_ports* with the name of the input pins on which you do not want a clock buffer inserted. For the gated clock example in Figure 5-14 and Figure 5-15, you would enter the following.

```
set_pad_type -no_clock {IN1, IN2, IN3, IN4, CLK}
```

Then follow the normal procedures to set the ports as pads and insert the pads as follows.

```
set_port_is_pad "*"
insert_pads
```

Using Memory

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

The XC4000 family can efficiently implement RAM and ROM using the CLB function generators. You can implement a ROM by describing it behaviorally as shown in Figure 5-19. The XSI XC4000 libraries contain 16 x 1 (16 deep x 1 wide) RAM and 32 x 1 (32 deep x 1 wide) RAM primitives, and 16 x 1 and 32 x 1 ROM primitives that you can instantiate.

You can also implement memory using the MemGen program, which is included in the XACT Development System. MemGen can create RAM and ROM between 1 to 32 bits wide and 2 to 256 bits deep. This section includes an example of using MemGen with XSI. Refer to the *XACT Reference Guide, Volume 1* for more information about using MemGen.

XC4000 RAMs

You can implement RAMs in your HDL by the following methods.

- You can instantiate 16 x 1 and 32 x 1 RAMs from the XSI primitive libraries.
- You can implement any other RAM size using MemGen.

Warning: Do not behaviorally describe RAMs in VHDL because compiling creates combinatorial loops.

XC4000 ROMs

You can implement ROMs in your HDL by the following methods.

- You can describe ROMs behaviorally.
- You can instantiate 16 x 1 and 32 x 1 ROMs primitives.
- You can implement other ROMs using MemGen.

To instantiate the ROM primitives ROM16X1 and ROM32X1 into your HDL design, use the Set Attribute command to define the ROM value.

set_attribute "*instance_name***" xnf_init "***rom_value***"**
type string

For example, if you gave the 16 x 1 ROM an instance name of "U1" and the value of the ROM is F5A3, you can use this command to set the ROM value as follows.

```
set_attribute "U1" xnf_init "F5A3" type string
```

For a 32 x 1 ROM, specify an 8-digit hexadecimal (hex) value in place of the 4-digit hex value as shown in the previous example.

Note: Instantiating ROM or RAM does not allow you to functionally simulate the design or easily migrate between FPGA families.

Figure 5-19 and Figure 5-20 illustrate how to define a ROM in VHDL and Verilog HDL, respectively. The FPGA Compiler creates ROMs from random logic gates that are implemented using function generators.

```
-----------------------------------------
--  Behavioral 16x4 ROM Example        --
--          rom16x4_4k.vhd             --
-----------------------------------------

entity rom16x4_4k is
        port (  ADDR: in INTEGER range 0 to 15;
                DATA: out BIT_VECTOR (3 downto 0));
end rom16x4_4k;

architecture BEHAV of rom16x4_4k is

        subtype ROM_WORD is BIT_VECTOR (3 downto 0);
        type ROM_TABLE is array (0 to 15) of ROM_WORD;
        constant ROM: ROM_TABLE := ROM_TABLE´(
                ROM_WORD´("0000"),
                ROM_WORD´("0001"),
                ROM_WORD´("0010"),
                ROM_WORD´("0100"),
                ROM_WORD´("1000"),
                ROM_WORD´("1000"),
                ROM_WORD´("1100"),
                ROM_WORD´("1010"),
                ROM_WORD´("1001"),
                ROM_WORD´("1001"),
                ROM_WORD´("1010"),
                ROM_WORD´("1100"),
                ROM_WORD´("1001"),
                ROM_WORD´("1001"),
                ROM_WORD´("1101"),
                ROM_WORD´("1111"));
begin
        DATA <= ROM(ADDR); -- Read from the ROM
end BEHAV;
```

Figure 5-19 Behavioral VHDL for 16 x 4 ROM

```
/*
 * rom16x4_4k - Behavioral Model
 * Behavioral Example of 16x4 ROM
 * XSI v3.2
 *      @(#)rom16x4_4k.v        1.2     8/4/94
 */

module rom16x4_4k(ADDR, DATA) ;
input [3:0] ADDR ;
output [3:0] DATA ;

reg [3:0] DATA ;

// A memory is not created because Synopsys will not synthesize it

always @(ADDR)
begin
   case (ADDR)
      4´b0000 : DATA = 4´b0000 ;
      4´b0001 : DATA = 4´b0001 ;
      4´b0010 : DATA = 4´b0010 ;
      4´b0011 : DATA = 4´b0100 ;
      4´b0100 : DATA = 4´b1000 ;
      4´b0101 : DATA = 4´b1000 ;
      4´b0110 : DATA = 4´b1100 ;
      4´b0111 : DATA = 4´b1010 ;
      4´b1000 : DATA = 4´b1001 ;
      4´b1001 : DATA = 4´b1001 ;
      4´b1010 : DATA = 4´b1010 ;
      4´b1011 : DATA = 4´b1100 ;
      4´b1100 : DATA = 4´b1001 ;
      4´b1101 : DATA = 4´b1001 ;
      4´b1110 : DATA = 4´b1101 ;
      4´b1111 : DATA = 4´b1111 ;
   endcase
end

endmodule
```

Figure 5-20 Behavioral Verilog HDL for 16 x 4 ROM

Using MemGen

Alternatively, you can implement ROMs using MemGen as follows.

1. Create the memory description file, for example, promdata.mem.

   You can use any file name. See Figure 5-21 for a sample memory description file.

2. Run MemGen on the memory description file to create the promdata.xnf file as follows.

   memgen promdata

3. Instantiate the memory submodule into the HDL design, as shown in Figure 5-22 or Figure 5-23.

The name of the address lines *must* be called A0 – A3 and the output data lines O0 – O3. When the design is compiled in the FPGA Compiler, the following warning occurs.

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

You can ignore this warning message.

4. Save the design to an SXNF file, for example, rom_memgen.sxnf.
5. Translate the output file, rom_memgen.sxnf, into an XNF file using Syn2XNF.

   The translator, Syn2XNF, automatically merges in the XNF file for the memory, for example, promdata.xnf. Refer to the “Translating SXNF Files to XNF Files Using Syn2XNF” section at the end of this chapter for more information.

The following figure illustrates a memory description file, promdata.mem.

```
; ==========================================
; Memory file for PROM symbol called ´PROM_SYM´ driving ´D_OUT_BUS´
; ==========================================
TYPE     ROM
WIDTH    4
DEPTH    16
DEFAULT  0  ; <== default value here
DATA
        2#0000#,
        2#0001#,
        2#0010#,
        2#0100#,
        2#1000#,
        2#1000#,
        2#1100#,
        2#1010#,
        2#1001#,
        2#1001#,
        2#1010#,
        2#1100#,
        2#1001#,
        2#1001#,
        2#1101#,
        2#1111#; <== END of ROM data
```

Figure 5-21 Memory Description File

Figure 5-22 and Figure 5-23 illustrate instantiating a ROM submodule using VHDL and Verilog HDL, respectively.

```
-----------------------------------------
-- Example of Instantiating a MemGen    --
-- Created Memory File                  --
--          rom_memgen.vhd              --
-----------------------------------------

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

architecture BEHAV of rom_memgen is

component promdata
        port (  A3,A2,A1,A0: in BIT;
                O3,O2,O1,O0: out BIT);
end component;

begin
        u1: promdata port map (A3=>ADDR(3),A2=>ADDR(2),A1=>ADDR(1),A0=>ADDR(0),
                O3=>DATA(3),O2=>DATA(2),O1=>DATA(1),O0=>DATA(0));

end BEHAV;
```

Figure 5-22 Instantiating 16 x 4 ROM Submodule (VHDL)

```
/*
 * rom_memgen - Structural Model
 * Example of Using MEMGEN to create a ROM
 * XSI v3.2
 *      @(#)rom_memgen.v        1.2     9/9/94
 */

module rom_memgen (ADDR,
                   DATA) ;
input [3:0] ADDR ;
output [3:0] DATA ;

promdata u1(.A3(ADDR[3]), .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
            .O3(DATA[3]), .O2(DATA[2]), .O1(DATA[1]), .O0(DATA[0])) ;

endmodule

module promdata(A3, A2, A1, A0, O3, O2, O1, O0);
input A3, A2, A1, A0;
output O3, O2, O1, O0;

endmodule
```

Figure 5-23 Instantiating 16 x 4 ROM Submodule (Verilog)

Performing Boundary Scan

The XC4000 FPGA devices contain boundary-scan facilities that are compatible with IEEE Standard 1149.1. Refer to the *XACT User Guide* for a detailed description of the XC4000 boundary scan capabilities.

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

Full access to the built-in boundary-scan logic is always available between power-up and the start of configuration. Optionally, the built-in logic is available after configuration if you specified boundary scan in the design. During configuration, you can use the Sample/Preload and Bypass instructions only.

In a configured FPGA device, the boundary-scan logic might not be active depending on the configuration data loaded into the part. Activation of the boundary-scan logic, if desired, is part of the design process. For HDL designs, you must instantiate the boundary-scan symbol, BSCAN, and the boundary scan I/O pins, TDI, TMS, TCK, and TDO to access the boundary scan logic after configuration. After configuring the device, you cannot activate or deactivate boundary scan without changing the configuration.

Warning: *Do not* use these FPGA Compiler boundary scan commands: Set JTAG Implementation, Set JTAG Instruction, and Set JTAG Port because they do not work with FPGA devices.

Figure 5-24 illustrates the BSCAN symbol instantiated into an HDL design.

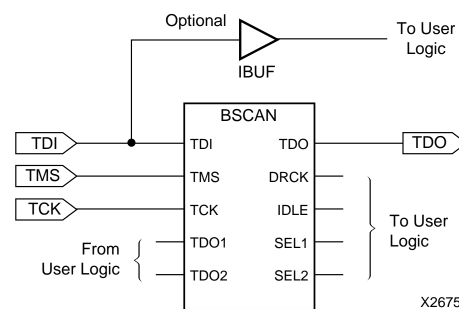


Figure 5-24 Boundary Scan Symbol

Using the Global Set/Reset Net

The Xilinx XC4000 devices have a dedicated Global Set/Reset (GSR) net that initializes all CLBs and IOBs. The function of the Global Set/Reset net is separate from the individual Preset (PRE) and Direct Clear (CLR) pin.

If the design has a Preset or Direct Clear signal, using the Global Set/Reset net increases the design's performance by reducing the overall routing congestion. You can remove the Preset or Direct Clear signal from the synthesized design and implement it using the dedicated Global Set/Reset net.

Startup State

The STARTUP symbol's Global Set/Reset pin drives the Set/Reset net and connects to each flip-flop's Preset and Direct Clear pin. When you connect a signal from a pad to the STARTUP symbol's GSR pin, the Global Set/Reset net is activated.

The Global Set/Reset net does not appear in the pre-placed and routed XNF file. When the GSR signal is asserted High (the default), every flip-flop and latch is set to the same state it had at the end of configuration as illustrated by Table 5-8. For XC3000 devices, all flip-flops and latches reset to 0 after configuration. When you simulate the routed design, the gate simulator's translation program correctly models the GSR function.

Table 5-8 Initialization State After Configuration (XC4000 Only)

Initializes to 0		Initializes to 1	
FDC	OFD_MF	FDP	OFDI_MF
FDCE	OFD_MS	FDPE	OFDI_MS
IFD	OFD_S	IFDI	OFDI_S
IFD_F	OFD_U	IFDI_F	OFDI_U
IFD_U	OFDT	IFDI_U	OFDTI
ILD_1	OFDT_F	IFDI_1	OFDTI_F
ILD_1F	OFDT_MF	IFDI_1F	OFDTI_MF
ILD_1U	OFDT_MS	IFDI_1U	OFDTI_MS

Initializes to 0		Initializes to 1	
OFD	OFDT_S	OFDI	OFDTI_S
OFD_F	OFDT_U	OFDI_F	OFDTI_U
OFD_FU			

Note: PPR implements inverters in the XC4000 devices without using additional CLB resources. You can connect any signal to drive the STARTUP symbol GSR pin.

Preset Versus Direct Clear

You can program each flip-flop and latch to be Preset or Direct Clear but *not* both. The flip-flops and latches can be Preset or Direct Clear upon completion of configuration by asserting the Global Set/Reset net and the individual Preset (PRE) and Direct Clear (CLR) pins of the flip-flop or latch. The value of the flip-flop (Preset or Direct Clear) is the same for all cases. The value of the flip-flop is determined by whether you used the PRE or CLR pin.

If the CLR or PRE pin on a non-I/O flip-flop cell is tied to an active signal, the state of that signal controls the startup state of those flip-flop cells; for example, if you use the PRE pin, the flip-flop starts up in Preset state. If you do not use the CLR and PRE pin, the default is to startup in a Clear state.

The following section describes how to change states. You must issue the commands to the FPGA Compiler after reading in the design but before writing the SXNF file.

You can use the Report Cell command to determine the instance names and the type of flip-flop used as follows.

- FDPE or FDP — Preset upon power-up
- FDCE or FDC — Direct Clear upon power-up

The COUT registers are implemented using FDCE. Upon power-up, these registers are cleared. For more information on the Report Cell output, refer to the “Generating Reports for Debugging” section at the end of this chapter.

Changing States

If you are not using the clear pin of a FDCE or FDC cell (grounded), you can override the initial state by issuing the Set Attribute command.

```
set_attribute "cell" fpga_xilinx_init_state \
-type string "S"
```

For IOBs, the default is to start up in a Direct Clear state. You can instantiate an I/O flip-flop and a latch with an INIT=S parameter to have the flip-flop start up in a Preset state.

The following illustrates using the Set Attribute command to change a flip-flop with the cell name of OUTX_reg<0> from a Reset-upon-powerup to a Set-upon-powerup as follows.

Warning: You can use the following command to change the initial state only if you are inferring a flip-flop without Clear and Preset pins. You cannot change the initialization state of instantiated flip-flops.

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

Refer to the Synopsys documentation on the FPGA Compiler for more information.

Increasing Performance with the GSR Net

Many designs have a net that initializes the majority of the design's flip-flops. If this signal can initialize *all* the flip-flops, you can use the Global Set/Reset net.

To have your HDL simulation match that of the resulting design, you should modify the HDL code so that every flip-flop and latch is preset or cleared when the Global Set/Reset signal is asserted. However, you must disconnect this signal with the Disconnect Net command after compiling the design and before saving it.

The FPGA Compiler cannot infer the usage of the Global Set/Reset net from the HDL code.

Note: The Xilinx optimizer, X-BLOX, has the ability to use the Global Set/Reset net automatically if every flip-flop and latch in the design has a common signal driving the Set Direct or Reset Direct pin, that is,

the CLR or PRE pin. You can run X-BLOX on any XNF, XTG, or XTF file.

Figure 5-25 and Figure 5-26 are examples of a design that can use the Global Set/Reset net for VHDL and Verilog HDL, respectively. The design contains two flip-flops. One flip-flop is reset and one is set when the signal “RST” is High.

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

entity gsr_ex is
        port (  CLK,RST : in STD_LOGIC;
                ST: buffer std_logic_vector (1 downto 0));
end gsr_ex;

architecture EXAMPLE of gsr_ex is

begin
        process (CLK, RST)
        begin
                if RST= ´1´ then
                        ST <= "01";
                elsif (CLK´event and CLK=´1´) then
                        ST <= ST + "01";
                end if;
        end process;

end EXAMPLE;
```

Figure 5-25 Before Using the GSR Net (VHDL)

```
/*
 * gsr_ex - Behavioral Model
 * XC4000 Global Set/Reset Example
 * XSI v3.2
 *      @(#)gsr_ex.v    1.2     8/22/94
 */

module gsr_ex (CLK, RST, ST) ;
input           CLK ;
input           RST ;
output  [1:0]   ST ;

reg     [1:0]   ST ;

always @(posedge CLK or posedge RST)
begin
   if (RST == 1´b1)
      ST = 2´b01 ;
   else
      ST = ST + 1´b1 ;
end

endmodule
```

Figure 5-26 Before Using the GSR Net (Verilog HDL)

To utilize the Global Set/Reset net, create a level of hierarchy that instantiates the STARTUP symbol and the core design as illustrated in Figure 5-27 and Figure 5-28 for VHDL and Verilog HDL, respectively. Use another signal name, such as “GSR” in the following design example, and route it to the STARTUP symbol GSR pin.

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

entity top_gsr is
        port(   CLK,GSR,RST: in STD_LOGIC;
                ST : buffer STD_LOGIC_VECTOR (1 downto 0));
end top_gsr;

architecture EXAMPLE of top_gsr is
        component STARTUP
                port (  GSR: in STD_LOGIC);
        end component;

        component gsr_ex
                port (  CLK,RST: in STD_LOGIC;
                        ST : buffer STD_LOGIC_VECTOR (1 downto 0));
        end component;

        begin

         U1 : STARTUP port map (GSR=>GSR);
         U2 : gsr_ex port map (CLK=>CLK,RST=>RST,ST=>ST);
end EXAMPLE;
```

Figure 5-27 Top_gsr.vhd

```
/*
 * top_gsr - Structural Model
 * Example of using the Global Set/Reset net
 * XSI v3.2
 *      @(#)top_gsr.v   1.2     8/4/94
 */

module top_gsr (CLK, GSR, RST, ST) ;
input           CLK ;
input           GSR ;
input           RST ;
output  [1:0]   ST ;

STARTUP U1 (.GSR(GSR)) ;
gsr_ex  U2 (.CLK(CLK), .RST(RST), .ST(ST)) ;

endmodule
```

Figure 5-28 Top_gsr.v

Figure 5-29 and Figure 5-30 contain the procedures for executing the top_gsr.vhd and top_gsr.v designs, respectively.

```
/* ================================================*/
/* Sample Script for Synopsys to Xilinx Using      */
/*            the FPGA Compiler                    */
/* ================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Read in the design                   */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design    */

   TOP = top_gsr
   SUB1= gsr_ex

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format vhdl TOP + ".vhd"
   analyze -format vhdl SUB1 + ".vhd"
   elaborate TOP
   elaborate SUB1

/* Set the current design to the top level          */

   current_design TOP

/* Since the STARTUP block does not have any outputs
   that are being used in this example, use the dont_
   touch command so that the compiler does not remove
   the STARTUP block.                              */

   dont_touch "U1"

/* Add pads to all ports except RST. Make sure the
   current design is the top-level module.
   Change the default slew rate to SLOW (HIGH slew
   control).                                       */

   set_port_is_pad {CLK GSR ST}
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Compile the design                 */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all

/* Synthesize and optimize the design               */

   compile -map_effort med
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Save the design                         */
/* +++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                        */

   report_fpga > TOP + ".fpga"
   report_timing > TOP + ".timing"

/* Write out the design to a DB file                   */

   write -format db -hierarchy -output TOP + ".db"

/* Replace CLBs and IOBs with gates                    */

   replace_fpga

/* Set the part type                                   */

   set_attribute TOP "part" -type string "4005pc84-5"

/* Remove the RST signal                               */

  disconnect_net RST -all

/* Save design in XNF format as <design>.sxnf          */

   write -format xnf -hierarchy -output TOP + ".sxnf"


/* Exit the Compiler.                                  */

   exit
```

Figure 5-29 Top_gsr Script File (VHDL)

```
/* ==================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*           the FPGA Compiler                      */
/* ==================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Read in the design                    */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design    */

   TOP = top_gsr
   SUB1= gsr_ex

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format vhdl TOP + ".vhd"
   analyze -format vhdl SUB1 + ".vhd"
   elaborate TOP
   elaborate SUB1

/* Set the current design to the top level          */

   current_design TOP

/* Since the STARTUP block does not have any outputs
   that are being used in this example, use the dont_
   touch command so that the compiler does not remove
   the STARTUP block.                              */

   dont_touch "U1"

/* Add pads to all ports except RST. Make sure the
   current design is the top-level module.
   Change the default slew rate to SLOW (HIGH slew
   control).                                       */

   set_port_is_pad {CLK GSR ST}
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Compile the design                  */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all

/* Synthesize and optimize the design               */

   compile -map_effort med
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Save the design                      */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                      */

   report_fpga > TOP + ".fpga"
   report_timing > TOP + ".timing"

/* Write out the design to a DB file                 */

   write -format db -hierarchy -output TOP + ".db"

/* Replace CLBs and IOBs with gates                  */

   replace_fpga

/* Set the part type                                 */

   set_attribute TOP "part" -type string "4005pc84-5"

/* Remove the RST signal                             */

  disconnect_net RST -all

/* Save design in XNF format as <design>.sxnf        */

   write -format xnf -hierarchy -output TOP + ".sxnf"


/* Exit the Compiler.                                */

   exit
```

Figure 5-30 Top_gsr Script File (Verilog HDL)

Read the top level (top_gsr) and the core design (gsr_ex) into the FPGA Compiler. Since the STARTUP block does not use any outputs, the design compiler removes the STARTUP block unless you specify the Don't Touch attribute for U1. You must issue this command before inserting the I/O pads. The FPGA Compiler then optimizes the design once you use the Replace FPGA command.

Before saving the design to an SXNF file, you must remove the RST signal from the design using the Disconnect Net command. PPR removes any unconnected gates from the design.

When the RST net is disconnected from the circuit, the PRE and CLR pin is no longer used. The flip-flop ST<0> is mapped to an FDPE that initializes to a Preset state (INIT=S), and the flip-flop ST<1> is mapped to an FDCE that initializes to a Direct Clear state (INIT=R).

Using the X-BLOX DesignWare Library

The XC4000 family DesignWare library describes adders, subtracters, comparators, incrementers, and decrementers that map to X-BLOX modules. Refer to “Getting Started” at the beginning of this user guide to ensure that you have X-BLOX installed on your system.

HDL Operators Using X-BLOX Modules

For XC4000 designs using the VHDL or Verilog arithmetic operators, Xilinx highly recommends that you use X-BLOX to take advantage of the X-BLOX DesignWare library. This DesignWare library contains the arithmetic functions that utilize the XC4000 dedicated carry logic to improve both the area and speed of the design.

The following is a list of the VHDL and Verilog arithmetic operators and the X-BLOX modules to which they map.

Table 5-9 Arithmetic Operators for X-BLOX Modules

Operators	X-BLOX Module
+	ADD_SUB
–	ADD_SUB
<, <=, >, >=	COMPARE
+ 1	INC_DEC
– 1	INC_DEC

X-BLOX is run on the output from the Synopsys-to-Xilinx translator, Syn2XNF. X-BLOX synthesizes these modules into XNF primitives and performs the necessary optimization and implementation.

The X-BLOX DesignWare library contains twos complement and unsigned binary modules of width 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, and 48. Sixty-four-bit widths are available for the COMPARE module only. Operands falling between bit ranges are mapped to the next higher bit-width module. X-BLOX removes any unused logic if implementing a smaller bit width. X-BLOX removes any unused logic if adding, subtracting, or comparing with a constant value.

The X-BLOX DesignWare modules contain path timing. The timing of the module depends on how many columns the module uses — the

larger the device, the more CLBs per column. The fastest implementation of an X-BLOX module is implemented in the fewest columns. If the XACT tools can implement the X-BLOX modules in one column and align the flip-flops and multiplexers, PPR can reduce routing congestion and improve overall design performance.

Table 5-10 shows the maximum bits that can be implemented in one column per device size.

Improving the Timing of X-BLOX Modules

To improve the timing of the X-BLOX module, choose a device type that requires the fewest columns. For example, if you wanted the fastest implementation of a 33-bit twos complement adder (without carry out), you should select a XC4008 or larger part type. Since the XC4008 can implement a 34-bit twos complement adder in one column, using a XC4008 or larger device would give you the fastest implementation since the adder would not have to wrap into the next column.

Note: The timing between device types does not vary because the X-BLOX modules do not contain routing delays.

In Table 5-10, replace _# with the number of bits, for example, add_sub_co_two_comp_14.

Table 5-10 Maximum Size of X-BLOX Module Before Wrapping

Device Type	4002	4003	4005	4006	4008	4010	4013
CLB Array Size	**8x8**	**10x10**	**12x12**	**16x16**	**18x18**	**20x20**	**24x24**
Add_Sub							
Twos Complement add_sub_two_comp_#	14	18	22	30	34	38	46
Unsigned Binary add_sub_ubin_#	14	18	22	30	34	38	46
Compare							
Greater Than or Equal To, Twos Complement comp_ge_two_comp_#	11	14	21	28	30	34	38

Device Type	4002	4003	4005	4006	4008	4010	4013
CLB Array Size	**8x8**	**10x10**	**12x12**	**16x16**	**18x18**	**20x20**	**24x24**
Compare (Cont'd)							
Greater Than or Equal To, Unsigned Binary comp_ge_ubin_#	13	17	21	29	33	37	45
Greater Than, Twos Complement comp_gt_two_comp_#	11	14	21	28	30	34	38
Greater Than, Unsigned Binary comp_gt_ubin_#	13	17	21	29	33	37	45
Less Than or Equal To, Twos Complement comp_le_two_comp_#	11	14	21	28	30	34	38
Less Than or Equal To, Unsigned Binary comp_le_ubin_#	13	17	21	29	33	37	45
Less Than, Twos Complement comp_lt_two_comp_#	11	14	21	28	30	34	38
Less Than, Unsigned Binary comp_lt_ubin_#	13	17	21	29	33	37	45
Not Equal, Twos Complement comp_ne_two_comp_#	32	38	48	64	N/A	N/A	N/A
Not Equal, Unsigned Binary comp_ne_ubin_#	32	38	48	64	N/A	N/A	N/A
Inc_Dec							
Twos Complement inc_dec_two_comp_#	16	20	24	32	36	38	46
Unsigned Binary inc_dec_ubin_#	16	20	24	32	36	38	46

Creating Timing Specifications

The Synopsys FPGA Compiler generates timing specifications that PPR analyzes. You can enter timing constraints in the Design Analyzer, Command window, or a script file. FPGA Compiler uses these timing constraints to determine the values and types of timing specifications it generates, which are based on the following path types.

Pad-to-pad (P2P)	Input port to an output port
Pad-to-setup (P2S)	Input port to the data pin of a flip-flop
Clock-to-setup (C2S)	Output pin of a flip-flop to the data pin of a flip-flop
Clock-to-pad (C2P)	Output pin of a flip-flop to an output port

Note: You can set or overwrite Synopsys-created timing specifications using PPR command-line options or the constraints (CST) file. Refer to the *XACT Libraries Guide* for more information on the constraints file.

Setting Timing Constraints

This section lists the Synopsys commands that enable you to create timing specifications for your Xilinx designs. Examples are provided to demonstrate how implemented Synopsys commands are passed to the XACT tools. For a complete listing of all options and arguments for each command, refer to the Synopsys online help pages.

Create Specifications for Input Ports and Clock Net

You can use the following commands to place a timing specification on all input ports and a specified clock net.

- Create Clock — This command creates a P2S specification on each input port, and a C2S specification on the specified clock net as follows.

  create_clock {*Clk*} -period 50

  Clk is the name of the clock net and “50” is the delay in nanoseconds.

- Max Period — This command creates a P2S specification on each input port, and a C2S specification on the specified clock net as follows.

  max_period 50 {*Clk*}

  The "50" indicates the net delay in nanoseconds, and *Clk* is the name of the clock net.

Create Specifications for Input and Output Ports

The Set Max Delay command creates a P2P specification on each input and output port. You can also use this command to affect P2S, C2S, and C2P timing specifications if you list the flip-flop cell names with either the –from or –to options.

set_max_delay *delay* **–from** {*input_port*} **–to** {*output_port*}

Create Tighter Constraints on Output Ports

The Set Output Delay command creates C2P specifications using the values from the Create Clock or Max Period constraints and creates tighter constraints for the output ports as follows.

set_output_delay 10 –clock {*Clk*} {*output_port*}

This command also changes the values of the P2P specifications created by the Set Max Delay command.

Create Tighter Constraints on Input Ports

The Set Input Delay command changes the values of the P2S specifications created by the Create Clock or Set Max Delay commands and creates tighter constraints on all input ports as follows.

set_input_delay 10 –clock {*Clk*} {*input_port*}

This command also changes the values of the P2P specifications created by the Set Max Delay command.

Prevent Specifications on Indicated Paths

The Set False Path command prevents the FPGA Compiler from generating timing specifications for specified paths as follows.

set_false_path –from {*input_port*}

Depending on the desired result, you can use this command in one of two ways.

- If you do not want the FPGA Compiler to generate timing specifications for the specified paths, invoke this command before running the Compile command.
- If you would like FPGA Compiler to optimize the path but not pass the timing specifications to PPR, invoke this command after running the Compile command but before running the Replace FPGA command.

Create Clocks on All Input Ports

The Derive Clocks command creates clocks on all input ports that source clock pins on flip-flops. If performing timing optimization, set constraints on all clocks in your design. You can run the Derive Clocks command to make sure that you have not missed any clocks in your design as follows.

```
derive_clocks
```

Controlling How Timing Specifications Are Written

Two variables control the way the FPGA Compiler writes timing specifications to the SXNF file.

Control the Number of Constraints Written

The XNFout Constraints Per Endpoint variable controls the number of constraints written per end point, for example, the number of timing specifications written per flip-flop. The default is 50 constraints.

Setting this variable to a lower number may over-constrain the design, whereas setting this variable to a higher number may under-constrain the design.

To prevent the FPGA Compiler from writing timing specifications to the SXNF file, set the following variable at the DC shell or Design Analyzer prompt as follows.

Setting this variable to 0 is useful if you want to create timing specifications using PPR constraints.

```
xnfout_constraints_per_endpoint = 0
```

Create Default Timing Constraints

The XNFout Default Time Constraints variable controls the constraints that exceed 50 or the specified number indicated using the XNFout Constraints Per Endpoint variable as follows. The default is True.

```
xnfout_default_time_constraints = [true|false]
```

The tightest constraint that was not written to the SXNF file becomes the default timing constraint. If the XNFout Constraints Per Endpoint variable is set to a low number, the design might be over-constrained.

If you set this variable to False, Synopsys does not write any default timing specifications to the SXNF file. The FPGA Compiler ignores the constraints that exceed the number specified with the XNFout Constraints Per Endpoint command. If the XNFout Constraints Per Endpoint variable is set to a low number, the design might be under-constrained.

Compiling the Design

Once you insert the I/O pads, you can optimize the design for area and/or speed. To get the most effective results from the FPGA Compiler, the constraints applied must be accurate and achievable. For example, if you set a timing goal of 0 ns on all ports, the FPGA Compiler attempts to meet this goal by duplicating logic to reduce critical paths, which can result in a significant and possibly unwarranted increase in CLB usage.

This following sections describe the commands you use to compile and optimize your HDL design.

Optimizing Logic Across Hierarchical Boundaries

CLBs contain both Boolean logic implemented in function generators and flip-flops. When you compile a hierarchical design or a design that uses a DesignWare module, the logic is *not* optimized across the hierarchical boundary. Therefore, some of the combinatorial logic has unused flip-flops, and the CLBs that implement the flip-flops have unused function generators. Also the Boolean logic from one hierarchy to another is not optimized to reduce the CLB area or logic levels.

The choice of hierarchical boundaries can have a significant impact on the quality — area or speed — of the synthesized design. Using the FPGA Compiler, you can optimize a design while preserving these hierarchical boundaries.

By default, the FPGA Compiler does not flatten a design. You must use the Compile command with the Ungroup All option to flatten the design; however, FPGA Compiler only partially optimizes logic across hierarchical modules. Full optimization is possible across those parts of the design hierarchy that are ungrouped in the FPGA Compiler. Follow the guidelines for controlling flattening in the online *Synopsys Design Compiler Reference Manual*.

Flattening the Design

Flattening can eliminate all the existing logic structure. In general, you can flatten random control logic because automatic structuring usually improves upon manual structuring. For FPGA designs, Xilinx recommends that you flatten designs when the number of CLBs needed to implement a Boolean function seems too high or there are too many logic levels. However, you should probably *not* flatten regular or highly structured designs, such as adders and ALUs that are designed with an explicit structure.

Flattening is especially useful for the FPGA CLB structure. The FPGA Compiler has a built-in optimizer for Boolean logic. For this algorithm to work efficiently, the structure must be sufficiently decomposed so that the Boolean logic can be mapped into the CLB function generators.

The TOP design, illustrated in Figure 5-31, references two sub-blocks: one that is completely combinatorial (block1) and one that is completely sequential (block2).

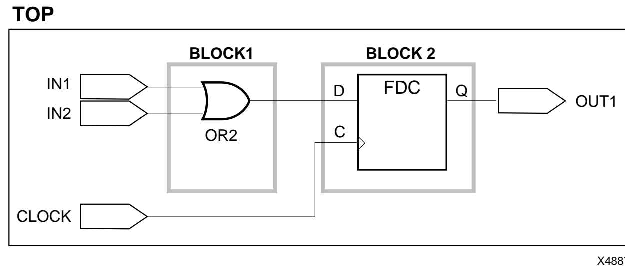


Figure 5-31 Sequential and Combinatorial Design

The FPGA Compiler cannot move logic across levels of hierarchy. If the hierarchy is maintained, two CLBs are required to implement the TOP design. The FPGA Compiler uses one CLB to implement the OR gate and another to implement the FDC flip-flop.

However, if the FPGA Compiler merges two subdesigns into a single level of hierarchy, only one CLB is required to implement the TOP design because the FPGA Compiler can merge the combinatorial and sequential logic into one CLB.

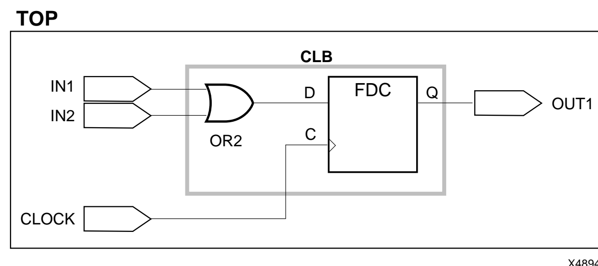


Figure 5-32 Merging into a Single Level of Hierarchy

To check if the FPGA Compiler can combine the combinatorial and sequential logic across hierarchical boundaries, optimize the design with and without hierarchy, and then compare the results as described in the following sections.

Compiling the Design with Hierarchy

To compile the design and maintain its hierarchy, enter the following command.

```
compile –map_effort [low|med|high] \
–boundary_optimization
```

This command enables some logic optimization across hierarchical boundaries. For more information on this option, refer to the *Synopsys Design Compiler Family Reference Manual*.

Even though your original design is flat, the design might end up containing hierarchical blocks after compiling. These hierarchical blocks contain either Synopsys DesignWare modules or X-BLOX DesignWare modules that were mapped during the optimization process.

Compiling the Design Without Hierarchy

To compile the design without hierarchy, enter the following command.

```
compile –map_effort [low|med|high] –ungroup_all
```

This command creates a flattened design and then optimizes it.

If your design contains Synopsys DesignWare modules (after the first compile), it is recommend that you re-compile the design using the Ungroup All option. Using this command does not optimize X-BLOX DesignWare modules; however, Synopsys DesignWare modules can be optimized because Synopsys DesignWare modules are entirely combinatorial. The CLBs that implement these DesignWare parts have unused flip-flops.

Warning: Using the Ungroup command with the All Flatten option and then compiling is not the same as invoking the Compile command with the Ungroup All option. If you run the Ungroup command before the using the Compile command, DesignWare components inferred during compilation retain their hierarchy and might cause the usage of unnecessary CLBs. See your Synopsys documentation for more information on the Ungroup command.

Creating Unique Names for Multiple Instances

For hierarchical designs that have more than one instance of the same module, the following command creates unique names for each instance of the submodule.

```
xlnx_hier_blknm=1
```

Compiling a Design That Contains Feedthroughs

You must set Compile Fix Multiple Port Nets to True before you compile to prevent PPR from deleting logic if the design contains feedthroughs, or if the same net is connected to more than one port.

```
compile_fix_multiple_port_nets = true
```

Compiling a Design with Instantiated I/O Cells

This section describes the design flow if your design contains instantiated I/O cells. If all I/O buffers are instantiated (the FPGA Compiler does not need to automatically insert I/O buffers), do not use the Set Port Is Pad and Insert Pads commands. Place a Don't Touch attribute on all instantiated I/O buffers.

If your design contains some instantiated I/O buffers and you want the FPGA Complier to automatically insert the rest of the I/O buffers, do the following.

- Use the Set Port Is Pad command only on the I/Os that you want the FPGA Compiler to insert.
- Place a Don't Touch attribute on all instantiated I/O buffers before the design is compiled.

See Figure 5-10 or Figure 5-11 for example designs that contain both instantiated I/Os and I/Os that are inserted using the FPGA Compiler. Figure 5-12 is an example script file illustrating the correct design flow.

Compiling XC4000 Designs

Figure 5-33 illustrates a sample script file that demonstrates how to compile your XC4000 design using the FPGA Compiler.

```
/* ==================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*            the FPGA Compiler                     */
/* ==================================================*/

/* Define the set-up variables either in the script  */
/*       or in the .synopsys_dc.setup file           */

/* Set the search path in your script or in your     */
/* .synopsys_dc.setup file                           */
/* Replace <DS401-XACT-Dir> with the directory       */
/* path where the DS-401 was installed and replace   */
/* $SYNOPSYS with the Synopsys installation directory*/

/*
   search_path = {. \
   <DS401-XACT-Dir>/synopsys/libraries/syn \
   $SYNOPSYS/libraries/syn}
*/

/* Set the link, target and synthetic library variable
   either in the script or in the .synopsys_dc.setup
   Use synlibs to determine the link and target
   libraries                                         */

   link_library = {xprim_4005-5.db xprim_4000-5.db \
      xgen_4000.db xio_4000-5.db xfpga_4000-5.db}

   target_library = {xprim_4005-5.db xprim_4000-5.db \
      xgen_4000.db xio_4000-5.db xfpga_4000-5.db}

   symbol_library = xc4000.sdb

   define_design_lib WORK -path ./WORK

/* Set the X-BLOX synthetic library path. Replace
   <DS401-XACT-Dir> with the directory path where
   the DS401 was installed                           */

/*
   define_design_lib xblox_4000 -path \
   <DS401-XACT-Dir>/synopsys/libraries/dw/lib/fpga
*/

   synthetic_library = {xblox_4000.sldb standard.sldb}
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Read in the design                     */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design     */

   TOP = <design_name>

/* Set the Designer and Company name for
   documentation.                                     */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                            */

   analyze -format vhdl TOP + ".vhd"
   elaborate TOP

/* Set the current design to the top level           */

   current_design TOP

/* Add pads to the design. Make sure the current
   design is the top-level module.                   */

   set_port_is_pad "*"
   insert_pads

/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Compile the design                   */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.             */

   remove_constraint -all

/* If setting timing constraints, do it here.
   For example:                                      */

/*
   create_clock <clock_pad_name> -period 50
*/

/* Synthesize and optimize the design                */

   compile -map_effort med
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Save the design                     */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                     */

   report_fpga > TOP + ".fpga"
   report_timing > TOP + ".timing"

/* Write out the design to a DB file                */

   write -format db -hierarchy -output TOP + ".db"

/* Replace CLBs and IOBs with gates                 */

   replace_fpga

/* Set the part type                                */

   set_attribute TOP "part" -type string "4005pc84-5"

/* Optional attribute to remove the FPGA Compilers
   mapping to CLBs and IOBs from all levels         */

/*
   set_attribute find(design,"*") "xnfout_write_map_symbols" \
   -type boolean FALSE
*/

/* Remove block names from all levels to allow more flexible routing */

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

/* Save design in XNF format as <design>.sxnf       */

   write -format xnf -hierarchy -output TOP + ".sxnf"

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Implement the Design                 */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Run xmake to process the design through the XACT */
/* tools.                                           */

/*
   sh xmake TOP
*/

/* Exit the Compiler.                               */

   exit
```

Figure 5-33 Sample Script File for Compiling XC4000 Designs

Compiling XC4000H Designs

Figure 5-34 and Figure 5-35 illustrate a script file that demonstrates how to compile an XC4000H design for VHDL and Verilog designs, respectively.

The design is compiled before pads are inserted to avoid pulling the flip-flops into the IOB. When compiling an XC4000H design, you must specify the input and output voltage levels. Refer to the "XC4000H IOBs" section for more information.

```
/* ==================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*            the FPGA Compiler                     */
/* ==================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Read in the design                    */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design     */

   TOP = three_ex2

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format vhdl TOP + ".vhd"
   elaborate TOP

/* Set the current design to the top level          */

   current_design TOP

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Compile the design                  */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all


/* Synthesize and optimize the design               */

   compile -map_effort med

/* +++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Insert Pads                          */
/* +++++++++++++++++++++++++++++++++++++++++++++++ */

/* Add pads to the design. Make sure the current
   design is the top-level module.
   Change the default slew rate to CAP (HIGH slew
   control;'CAP' applicable only to XC4000H devices). */

   set_port_is_pad "*"
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Save the design                       */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                       */

   report_fpga > TOP + ".fpga"
   report_timing > TOP + ".timing"

/* Write out the design to a DB file                  */

   write -format db -hierarchy -output TOP + ".db"

/* Replace CLBs and IOBs with gates                   */

   replace_fpga

/* Set the part type                                  */

   set_attribute TOP "part" -type string "4005hpq240-5"


/* Save design in XNF format as <design>.sxnf         */

   write -format xnf -hierarchy -output TOP + ".sxnf"


/* Exit the Compiler.                                 */

   exit
```

Figure 5-34 Sample Script File for Compiling XC4000H Design (VHDL)

```
/* ==================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*            the FPGA Compiler                     */
/* ==================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Read in the design                   */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design     */

   TOP = three_ex2

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format verilog TOP + ".v"
   elaborate TOP

/* Set the current design to the top level          */

   current_design TOP

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Compile the design                 */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all


/* Synthesize and optimize the design               */

   compile -map_effort med

/* +++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Insert Pads                         */
/* +++++++++++++++++++++++++++++++++++++++++++++++ */

/* Add pads to the design. Make sure the current
   design is the top-level module.
   Change the default slew rate to CAP (HIGH slew
   control;´CAP´ applicable only to XC4000H devices). */

   set_port_is_pad "*"
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Save the design                        */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                       */

   report_fpga > TOP + ".fpga"
   report_timing > TOP + ".timing"

/* Write out the design to a DB file                  */

   write -format db -hierarchy -output TOP + ".db"

/* Replace CLBs and IOBs with gates                   */

   replace_fpga

/* Set the part type                                  */

   set_attribute TOP "part" -type string "4005hpq240-5"


/* Save design in XNF format as <design>.sxnf         */

   write -format xnf -hierarchy -output TOP + ".sxnf"


/* Exit the Compiler.                                 */

   exit
```

Figure 5-35 Sample Script File for Compiling XC4000H Design (Verilog HDL)

Creating the Area Report

The FPGA Compiler reports area with the Report FPGA command as follows.

report_fpga

The statistics reported by this command include the number of the following elements used in your design.

- F, G, and H function generators
- X-BLOX cells
- Instantiated cells
- 3-state buffers
- Flip-flops
- IOBs

This command also reports the number of CLBs used for the design on the basis of the mapping performed by the FPGA compiler.

Run this command after the design has been compiled because the Compile command maps the logic into CLBs and IOBs. Run this command before replacing the CLB and IOBs with gates, that is, before running the Replace FPGA command.

Figure 5-36 illustrates the Report FPGA output for the bidi_reg design. The report shows the number of CLBs used.

```
****************************************
Report : fpga
Design : bidi_reg
Version: v3.1b-20502
Date   : Thu Sep  8 07:48:36 1994
****************************************

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

    FG Function Generators:          2
    H Function Generators:           0
    Number of CLB cells:             2
    Number of Hard Macros and
        Other Cells:                 4
    Number of CLBs in
        Other Cells:                 0
    Total Number of CLBs:            2

    Number of Ports:                 8
    Number of Clock Pads:            2
    Number of IOBs:                  2

    Number of Flip Flops:            4
    Number of 3-State Buffers:       4

    Total Number of Cells:          14
```

Figure 5-36 Area Utilization Report

Evaluating Timing Delays

The Synopsys tools report all delays in nanoseconds. The delays reported are pre-placement and routing estimates. You can use either average or worst-case wire-load models. The FPGA Compiler cannot determine the actual wire-load delays until after the design is placed and routed.

To evaluate the timing results, use the Report Timing command.

```
report_timing
```

Refer to the online *Synopsys Command Reference Manual* for information on other report options.

Run the Report Timing command after compiling the design since the Compile command maps the logic into CLBs and IOBs, and before running the Replace FPGA command, which replaces the CLBs and IOBs with gates.

If you specify no wire-load value, Synopsys assigns a default of None to all wire loads in the design. Refer to “Setting the Wire-Load Model” at the beginning of this chapter for more information.

Warning: Tpickd, the input flip-flop and latch setup time from pad to clock (IK) with delay, and Tikpid, the hold time from pad to clock (IK) with delay, for the XC4000 and XC4000A –4 speed grades might be inaccurate. The Tpickd and Tikpid timing parameters apply to the input IOB flip-flops that do *not* have the NODELAY parameter, which is the default configuration for the input flip-flops and latches. The –4 speed grade timing parameters vary with the size of the die. The delays reported are the maximum delays. The actual delay might be 10 percent less. For example, in the XC4000 libraries, the Tpickd and Tikpid delays are for the XC4013 devices; for XC4000A devices, XC4005A delays are used.

Generating Reports for Debugging

To assist you in debugging, XSI provides additional commands that furnish CLB and IOB information for debugging purposes.

Warning: Use the following commands before replacing the CLBs and IOBs with gates using the Replace FPGA command.

Generating a Configuration Report

You can generate a report that gives you CLB and IOB configuration information similar to the reports generated in the XACT Development System. This report contains information on how each cell is configured and the logic function it implements.

To generate a CLB and IOB configuration report, first generate a symbol or schematic view for the design using either of the following methods.

- From Design Analyzer Menu, select **`Tools`** → **`FPGA Compiler`** → **`Report`** → **`Cell`** → **`Apply`**.
- From the DC shell prompt, enter **`report_cell`**↵.

The system displays the following output in the Command window.

```
****************************************
Report : cell
Design : count8
Version: v3.1b-20502
Date   : Mon Oct 17 18:37:46 1994
****************************************

Attributes:
    b - black box (unknown)
   BO - reference allows boundary optimization
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Cell                      Reference       Library            Area  Attributes
--------------------------------------------------------------------------------
U62                       iob_4000        xfpga_4000-5       1.00  n
U64                       iob_4000        xfpga_4000-5       1.00  n
U66                       iob_4000        xfpga_4000-5       1.00  n
U68                       iob_4000        xfpga_4000-5       1.00  n
U70                       iob_4000        xfpga_4000-5       1.00  n
U72                       iob_4000        xfpga_4000-5       1.00  n
U74                       iob_4000        xfpga_4000-5       1.00  n
U76                       iob_4000        xfpga_4000-5       1.00  n
U78                       iob_4000        xfpga_4000-5       1.00  n
U80                       iob_4000        xfpga_4000-5       1.00  n
U82                       BUFG_F          xprim_4005-5       0.00
U83                       clb_4000        xfpga_4000-5       1.00  n
U85                       clb_4000        xfpga_4000-5       1.00  n
U87                       clb_4000        xfpga_4000-5       1.00  n
U89                       clb_4000        xfpga_4000-5       1.00  n
add_21/plus/LEFT_UNSIGNED_ARG_799
                          count8_inc_dec_ub_8_0              4.00  BO, h, n
--------------------------------------------------------------------------------
Total 16 cells                                              18.00

Detailed FPGA Configuration Information:
Cell Name: U62  TYPE: IOB
        OUT:O
        PAD:FAST        I1:     I2:     TRI:

Cell Name: U64  TYPE: IOB
        OUT:O
        PAD:FAST        I1:     I2:     TRI:

Cell Name: U66  TYPE: IOB
        OUT:O
        PAD:FAST        I1:     I2:     TRI:

Cell Name: U68  TYPE: IOB
        OUT:O
        PAD:FAST        I1:     I2:     TRI:

Cell Name: U70  TYPE: IOB
        OUT:O
        PAD:FAST        I1:     I2:     TRI:

Cell Name: U72  TYPE: IOB
        OUT:O
        PAD:FAST        I1:     I2:     TRI:
```

```
Cell Name: U74  TYPE: IOB
        OUT:O
        PAD:FAST        I1:     I2:     TRI:

Cell Name: U76  TYPE: IOB
        OUT:O
        PAD:FAST        I1:     I2:     TRI:

Cell Name: U78  TYPE: IOB
        OUT:O
        PAD:FAST        I1:PAD  I2:     TRI:

Cell Name: U80  TYPE: IOB
        OUT:O
        PAD:FAST        I1:PAD  I2:     TRI:

Cell Name: U83  TYPE: CLB
        X:            Y:          XQ:QX       YQ:QY
        H1:           DIN:C1      SR:C2       EC:C3
        DX:DIN        DY:G        FFX:EC:RESET:K
                                              FFY:EC:RESET:K
        EQUATE G = (G1)
        FFX_NAME:QOUT_reg<1>    FFY_NAME:QOUT_reg<0>

Cell Name: U85  TYPE: CLB
        X:            Y:          XQ:QX       YQ:QY
        H1:           DIN:C1      SR:C2       EC:C3
        DX:DIN        DY:G        FFX:EC:RESET:K
                                              FFY:EC:RESET:K
        EQUATE G = (G1)
        FFX_NAME:QOUT_reg<3>    FFY_NAME:QOUT_reg<2>

Cell Name: U87  TYPE: CLB
        X:            Y:          XQ:QX       YQ:QY
        H1:           DIN:C1      SR:C2       EC:C3
        DX:DIN        DY:G        FFX:EC:RESET:K
                                              FFY:EC:RESET:K
        EQUATE G = (G1)
        FFX_NAME:QOUT_reg<5>    FFY_NAME:QOUT_reg<4>

Cell Name: U89  TYPE: CLB
        X:            Y:          XQ:QX       YQ:QY
        H1:           DIN:C1      SR:C2       EC:C3
        DX:DIN        DY:G        FFX:EC:RESET:K
                                              FFY:EC:RESET:K
        EQUATE G = (G1)
        FFX_NAME:QOUT_reg<7>    FFY_NAME:QOUT_reg<6>
```

Figure 5-37 Report Cell Output

Generating a Hierarchical Schematic

As an alternative to interpreting the Report Cell output listing, you can direct the FPGA Compiler to replace all CLB and IOB cells with an equivalent set of logic from the target libraries. You can use the

generated schematic to determine what logic was used to implement the CLBs and IOBs.

To generate a hierarchical CLB and IOB schematic, perform the following steps.

1. Save your original design as a DB file because the commands used to generate the hierarchical CLB and IOB schematic change the original design.
2. Select **Tools → FPGA Compiler → FPGA Cells to Gates Options** from the Design Analyzer menu.

Create a hierarchy level for each CLB and IOB or a hierarchy level for each function generator as described in the following sections.

Warning: Once you select these options, the resulting logic does not accurately reflect the timing of the actual CLB and IOB implementation. Any timing or area reports will not be accurate.

3. After you finish viewing the hierarchical schematic, read in the original DB file.

Creating a Level for Each CLB and IOB

This section describes how to generate hierarchical schematics that contain CLBs and IOBs and the underlying logic that comprises them. Viewing hierarchical schematics can assist you in locating logic or signals for debugging purposes.

To create a level of hierarchy for each CLB and IOB, do one of the following.

- Select **Tools → FPGA Compiler → FPGA Cells to Gates Options → Create a Level of Hierarchy for each CLB and IOB** from the Design Analyzer menu.
- Enter the following at the DC shell prompt.

```
replace_fpga –group_cells
```

Creating a Level for Each Function Generator

This section describes how to generate hierarchical schematics that show the logic in each function generator it implements. This process replaces each CLB by an F, G, or H function generator, along with the

flip-flops, if they are being used. The function generators are an additional level of hierarchy.

To create a level of hierarchy for each function generator, do one of the following.

- Select **`Tools → FPGA Compiler → FPGA Cells to Gates Options → Create a Level of Hierarchy for each "Table-lookup"`** from the Design Analyzer menu.
- Enter the following at the DC shell prompt.

  `replace_fpga -group_tlus`

You can now view the implementation of the function generators.

Writing and Saving the Design

Once the design meets your timing and area requirements, you can save the design as a DB file; replace the CLBs and IOBs with gates; remove the Synopsys mapping; set the design part type; and write and save the design.

Before saving the design, set all desired variables to control how the design is optimized using the Synopsys timing options. Refer to the "Creating Timing Specifications"section for more information.

Before saving the design, set the appropriate variables to define the I/O pad locations, slew rates, and so on. Refer to the "Configuring IOBs" section for more information.

Saving the DB File

Save the Synopsys database file before replacing the design with gates by running the Replace FPGA command. If you are using the Replace FPGA command options for debugging, such as Group TLUS and Group Cell, Xilinx also recommends that you save the DB file before running these debugging options.

To save the DB file, you can choose one of the following methods.

- Enter the following from the Design Analyzer menu.

  `File → Save As`
  `File name:` *design_name*.**db**
  `File Format: db`

```
Save all Designs in Hierarchy: on
OK
```

- Type the following at the command line. (Make sure the top level of the design is selected.)

  write -format db -hierarchy -output *design_name*.**db**

Replacing CLBs and IOBs with Gates

After compiling, a design contains CLB and IOB elements that the FPGA Compiler uses to determine the best implementation of a design for a given set of constraints. Before creating the SXNF file, you must convert these CLBs and IOBs into gates that can be recognized by the XACT Development System. The mapping information is passed to the SXNF file using the FMAP, HMAP, and BLKNM parameters, so PPR can map the design.

Invoking the Replace FPGA Command

Enter the following command at the command line at the top level of your design.

```
replace_fpga
```

If Your Design Contains Hierarchy

Xilinx does not recommend running the Replace FPGA command with either the Group Cells or the Group TLUS option, and then writing the SXNF file. These option generate SXNF files for each level of hierarchy in the design. If you use the Group Cells option, each CLB is transformed into a level of hierarchy, and an SXNF file is created for each CLB. Similarly, if you use the Group TLUS option, each function generator is transformed into a level of hierarchy.

If you used these options, perform the following steps.

1. Delete the design from memory.
2. Read in the saved DB file.
3. Run the Replace FPGA command without any options.

Removing the Synopsys Mapping

By default, the FPGA Compiler XNF Writer contains information on how it should map the logic into the CLB and IOBs. The FPGA Compiler uses the FMAP and HMAP symbols to map Boolean logic into F and H function generators, and the BLKNM attribute to group flip-flops and function generators into a CLB.

When the XNF Writer performs the mapping, the estimated timing information is more accurate.

Mapping information from the FPGA Compiler is usually efficient; therefore, Xilinx recommends that you leave the mapping on. Block names, however, can restrict placement and routing. For this reason, Xilinx recommends removing BLKNM attributes.

Note: Using the FPGA Compiler to perform the mapping decreases PPR processing time.

The following section describes how to remove FMAP, HMAP, and BLKNM attributes.

Removing FMAP and HMAP Symbols

To remove the FMAP and HMAP mapping, enter the following at the command line.

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

Removing BLKNM Attributes

To remove the setting of BLKNM attributes, enter the following at the command line.

Note: Xilinx recommends that you set this command.

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

Setting the Design Part Type

Type the following command at the command line to select a specific part for the design. The following example uses a 4005pc84–5 device.

```
set_attribute design "part" -type string
"4005pc84-5"
```

Note: You can also specify the part type when running Syn2XNF or XMake.

Saving the SXNF File

After replacing the design with gates, save the design to an SXNF file. The output file extension is .sxnf to distinguish this file from the output of the translator Syn2XNF, which is an XNF file. In the FPGA Compiler, the file format is XNF; however, the file produced requires some translation to make it a conform to the XNF format.

You can save the design as an SXNF file by either of the following methods.

- Select the design and then select the following from the Design Analyzer menu.

  ```
  File → Save As
  File name: design_name.sxnf
  File Format: xnf
  Save all Designs in Hierarchy: on
  OK
  ```

- Enter the following at the command line. (Make sure the top level of the design is selected.)

  ```
  write -format xnf -hierarchy -output \
  design_name.sxnf
  ```

Translating SXNF Files to XNF Files Using Syn2XNF

The Syn2XNF translator takes a Synopsys SXNF file written by the FPGA Compiler and translates it to an XNF file.

How you run Syn2XNF is determined by your system configuration. Refer to the “FPGA Compiler Design Flow”section at the beginning of this chapter.

Syntax

To use Syn2XNF, enter the following on the command line. Specifying the file name extension is optional.

syn2xnf [options] [*design***.sxnf.|** *design***.sedif.|**
*design***.xnf]**

By default Syn2XNF searches for a design file with an .sxnf or .sedif extension. If both exist, Syn2XNF uses the file with the latest time stamp.

You can run Syn2XNF from the UNIX prompt or from the FPGA Compiler by using the shell command as follows.

sh syn2xnf *design*

In addition, you can run Syn2XNF automatically from XMake; however, you must have the XACT Development System installed on the same network as the XSI software. Refer to the "Before You Begin" section at the beginning of this chapter for more information.

Input Files

Syn2XNF accepts the following file types as input when using the FPGA Compiler.

design.sxnf	This file is the synthesized design generated by the Synopsys synthesis tools.
design.xnf	This file represents the flattened, synthesized design in the Xilinx Netlist Format.
design.sedif	This file is the synthesized design generated by the Synopsys synthesis tools using the EDIF syntax.

Syn2XNF is not case sensitive — you can enter the file name extension in upper- or lower-case letters. However, Xilinx recommends indicating the file name extension to distinguish the FPGA Compiler output from the Syn2XNF output.

Output Files

Syn2XNF creates three output files as follows.

design.xnf	This file represents the flattened, synthesized design in Xilinx Netlist Format.

design.xff	This file represents the flattened, synthesized design in Xilinx Netlist Format.
syn2xnf.log	This file contains error and warning messages that are also displayed onscreen.

Options

This section describes the Syn2XNF options. You can abbreviate all options using the first letter of the option; for example, you can indicate –parttype as –p.

Note: You cannot use the –sub and –map options with the FPGA Compiler XC4000 design flow. You cannot use the –sub option with the FPGA Compiler because the FPGA Compiler writes EXT records in the SXNF file automatically if Insert Pads is executed.

–dir

The –dir option causes Syn2XNF to search *directory_name* for data files as well as the *DS401_dir/*data/synopsys directories and the current working directory.

syn2xnf –d *directory_name*

–force

The –force option forces Syn2XNF to overwrite an XNF file if one already exists.

```
syn2xnf –f
```

–help

The –help option displays onscreen the Syn2XNF help text.

```
syn2xnf –help
```

–l

The –l option lists onscreen all valid part types.

```
syn2xnf –l
```

–out

The –out option specifies the output file name.

syn2xnf –o *new_name design*

By default Syn2XNF creates an XNF and XFF file with the same name as the input design file name. If you use this option, as illustrated by the following example, Syn2XNF reads the file design and outputs newdesign.xff and newdesign.xnf.

```
syn2xnf –o newdesign design
```

–parttype

The –parttype option specifies the Xilinx part and speed grade as follows.

syn2xnf –p *part–speedgrade design*

If you specify no part type, Syn2XNF reads the part type from the SXNF file. If no part type is specified in the SXNF file, Syn2XNF uses the default part type, 4003APC84.

The following example illustrates how to specify the part type for an XC4005–5 device.

syn2xnf –p 4005apc84–5 *design*

To ensure that the XACT tools process your design properly, specify a part and speed grade.

Using the XACT Development System

To translate the design to LCA and BIT files so the XACT tools can program the XC4000 device, use the XMake program.

How you run XMake differs slightly depending on your system configuration. If the XACT Development System is installed on the same network as the Xilinx Synopsys Interface, XMake runs Syn2XNF. If XSI is installed on a machine that cannot access the XACT Development System, run Syn2XNF on the machine with the XSI software and copy the appropriate design files to the machine with the XACT Development System. Refer to the “FPGA Compiler Design Flow” chapter at the beginning of this user guide for more information.

XMake automatically translates X-BLOX modules into gates by running X-BLOX, and maps, places, and routes the design using PPR.

If XSI Is on Same Platform as XACT Software

If XSI is installed on the same platform as the XACT software, invoke XMake.

xmake *design*

XMake runs Syn2XNF and the XACT software tools.

If XSI Is on Different Platform Than XACT Software

If XSI is installed on a different platform than the XACT software, do the following to run the XACT tools.

1. Copy the XNF and XFF files to the platform where the XACT tools reside.

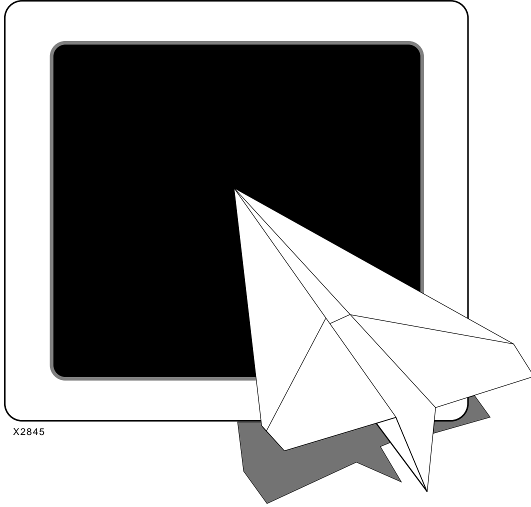
Note: Use the Copy command with the –p option to preserve the files' time stamp.

2. Run XMake with the following option.

xmake –x *design*

The –x option causes XMake to search for an XNF file and any other files not already merged.

For more information on XMake or any of the programs it invokes, refer to the *XACT Reference Guide*.



Xilinx Synopsys Interface FPGA User Guide

Using the Design Compiler

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Chapter 6

Using the Design Compiler

The Design Compiler enables you to synthesize and implement your HDL design for Xilinx FPGA devices. The Design Compiler provides the following features.

- Optimization of flip-flops and latches into the I/O block
- Optimization of 3-state buffers into the I/O block
- Encoding for one-hot state machines
- Automatic usage of CLB Clock Enable pin

Before You Begin

Before beginning a Xilinx design using the Synopsys tools, read the “Getting Started” chapter at the beginning of this manual, which describes how to do the following.

- Verify that the XSI and XACT Development System software exists on your system
- Modify the XSI default Synopsys startup file, if applicable

Design Compiler Design Flow

This section describes the Design Compiler design flow, which varies slightly depending on whether XSI is installed on the same platform or on a different platform than the XACT Development System software.

The difference between the design flow for XC4000 and XC3000/ XC3100 devices is that the X-BLOX DesignWare library is only available for XC4000 designs. However, you can run X-BLOX on XC3000A/L and XC3100/A devices to perform global optimization.

Refer to the *XACT X-BLOX User Guide* for more information on global optimization.

Proceed to the following section that applies to your system configuration.

If XSI Is on Same Platform as XACT Software

Figure 6-1 shows the design flow for the Design Compiler when the XSI software is installed on the same platform or network as the XACT Development System.

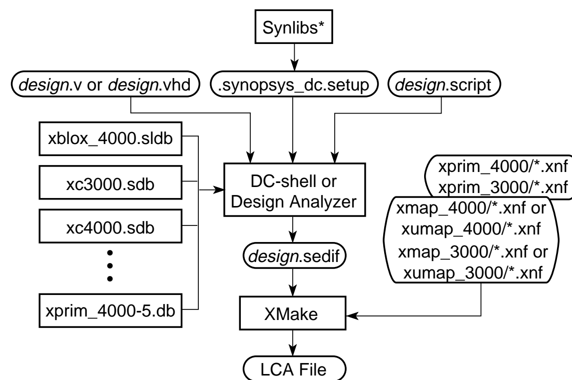


Figure 6-1 Design Flow with XSI Installed on Same Platform

If XSI Is on Different Platform Than XACT Software

If XSI is installed on a machine that does not have access to both the XSI and XACT executable files, you must copy or move the Syn2XNF output XNF and XFF files to the platform where the XACT executable files reside, as illustrated by Figure 6-2.

The basic flow for the different devices are the same. XMake automatically runs the appropriate mapping, placement, and routing tools depending on the specified device.

When running Syn2XNF, you have a choice of using the *mapped* or the *unmapped* cells for the Boolean functions, which allow the XACT Development system to map the Boolean functions.

- The *mapped* libraries retain compatibility between the timing analysis created within the Synopsys environment and the final implementation in the FPGA.
- The *unmapped* libraries for the same Boolean functions do not guarantee timing compatibility between the Synopsys analysis and the actual FPGA devices.

See the “Translating SEDIF Files to XNF Files Using Syn2XNF” section at the end of this chapter for additional information.

Refer to the “Files, Programs, and Libraries” chapter in this user guide for additional library information.

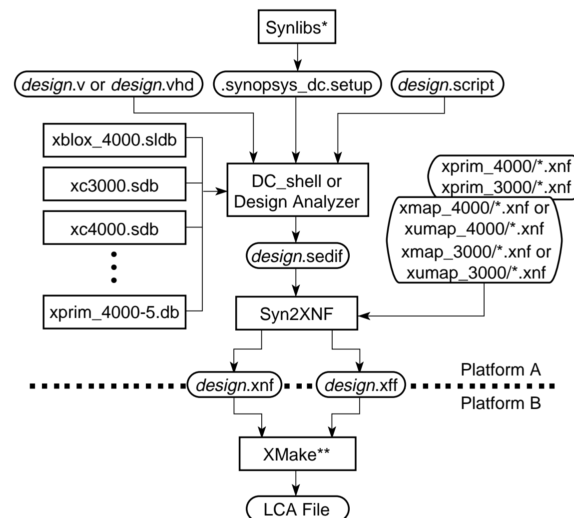


Figure 6-2 Design Flow with XSI Installed on Different Platform

Setting the Wire-Load Model

Each primitive library contains pre-layout and routing-estimated wire-load models that are device and speed-grade specific. The Synopsys tools can use these estimates when optimizing your design for an FPGA. XSI provides two wire-load models per device-speed grade combination — an average model and a worst-case model,

designated by "_avg" and "_wc," respectively. The default wire load is *average*.

Wire-Load Models for Xilinx FPGAs

The following tables list the wire-load models for each Xilinx device. Substitute "_avg" or "_wc" for *a/w*, for example, 4003–4_wc.

Table 6-1 XC4000/A/D/H Wire-Load Models

–4 Speed Grade	–5 Speed Grade	–6 Speed Grade	–10 Speed Grade
	4002–5_*a/w*	4002–6_*a/w*	
4003–4_*a/w*	4003–5_*a/w*	4003–6_*a/w*	
	4004–5_*a/w*	4004–6_*a/w*	
4005–4_*a/w*	4005–5_*a/w*	4005–6_*a/w*	4005–10_*a/w*
4006–4_*a/w*	4006–5_*a/w*	4006–6_*a/w*	
4008–4_*a/w*	4008–5_*a/w*	4008–6_*a/w*	
4010–4_*a/w*	4010–5_*a/w*	4010–6_*a/w*	4010–10_*a/w*
4013–4_*a/w*	4013–5_*a/w*	4013–6_*a/w*	

Table 6-2 XC3000 Wire-Load Models

–50 Speed Grade	–70 Speed Grade	–100 Speed Grade	–125 Speed Grade
3020–50_*a/w*	3020–70_*a/w*	3020–100_*a/w*	3020–125_*a/w*
3030–50_*a/w*	3030–70_*a/w*	3030–100_*a/w*	3030–125_*a/w*
3042–50_*a/w*	3042–70_*a/w*	3042–100_*a/w*	3042–125_*a/w*
3064–50_*a/w*	3064–70_*a/w*	3064–100_*a/w*	3064–125_*a/w*
3090–50_*a/w*	3090–70_*a/w*	3090–100_*a/w*	3090–125_*a/w*

Table 6-3 XC3000A/L Wire-Load Models

–6 Speed Grade	–7 Speed Grade	–8 Speed Grade
3020a–6_*a/w*	3020a–7_*a/w*	3020l–8_*a/w*
3030a–6_*a/w*	3030a–7_*a/w*	3030l–8_*a/w*

–6 Speed Grade	–7 Speed Grade	–8 Speed Grade
3042a–6_*a/w*	3042a–7_*a/w*	3042l–8_*a/w*
3064a–6_*a/w*	3064a–7_*a/w*	3064l–8_*a/w*
3090a–6_*a/w*	3090a–7_*a/w*	3090l–8_*a/w*

Table 6-4 XC3100/A Wire-Load Models

–3 Speed Grade	–4 Speed Grade	–5 Speed Grade
3120–3_*a/w*	3120–4_*a/w*	3120–5_*a/w*
3120a–3_*a/w*	3120a–4_*a/w*	3120a–5_*a/w*
3130–3_*a/w*	3130–4_*a/w*	3130–5_*a/w*
3130a–3_*a/w*	3130a–4_*a/w*	3130a–5_*a/w*
3142–3_*a/w*	3142–4_*a/w*	3142–5_*a/w*
3142a–3_*a/w*	3142a–4_*a/w*	3142a–5_*a/w*
3164–3_*a/w*	3164–4_*a/w*	3164–5_*a/w*
3164a–3_*a/w*	3164a–4_*a/w*	3164a–5_*a/w*
3190–3_*a/w*	3190–4_*a/w*	3190–5_*a/w*
3190a–3_*a/w*	3190a–4_*a/w*	3190a–5_*a/w*
3195–3_*a/w*	3195–4_*a/w*	3195–5_*a/w*
3195a–3_*a/w*	3195a–4_*a/w*	3195a–5_*a/w*

Changing the Wire-Load Model

To change the wire load from average to worst case, use the Set Wire Load command as illustrated by the following example.

set_wire_load "*parttype–s_wc***"**

The speed grade for the wire-load model must match the speed grade of the primitive library as listed in the previous wire-load model tables as illustrated by this example.

```
set_wire_load "4005–5_wc"
```

If you want to evaluate the block delays of the design without the wire load, set the wire load to None by using the Set Wire Load command as follows.

```
set_wire_load none
```

How Wire-Load Models Are Determined

Average and worst-case models are derived from over 6000 designs that were placed and routed on the different Xilinx parts for each of the different speed grades.

The average wire-load model for a given part and speed grade is calculated by collecting all signal nets of a given fanout for all designs using the part type and speed grade. For a given fanout, 50 percent of the nets from the test suite are slower and 50 percent of the nets are faster than the delay number in the average wire-load model.

The worst-case wire-load models add one standard deviation to each average fanout value. For a given fanout, 68 percent of the nets from the test suite are faster and 22 percent are slower than the delay number in the worst-case wire-load model; therefore, the worst-case wire-load models are more conservative than the average wire-load models. You can determine the actual wire-load delays after placing and routing the design.

In all cases, the wire-load delays increase as the die size and fanout of the net increase. The delays decrease with faster device speed grades.

The Report Timing command combines the wire-load delay with the block delay. For more information on the Report Timing command, refer to the “Evaluating Timing Delays” section at the end of this chapter.

Operating Conditions

Only one set of operating condition parameters is available — worst-case commercial (WCCOM) — which is the default in the Xilinx libraries.

Configuring the IOBs

The following sections describe how to configure XC4000/A/D/H, XC3000/A/L, and XC3100/A IOBs. The Design Compiler can infer automatically the following IOB configurations.

- Input buffers, for example, IBUF
- Output buffers, for example, OBUF

- Bidirectional buffers, for example, an input buffer, IBUF, and a 3-state output buffer, OBUFT
- Input flip-flops, for example, IFD
- Latches, for example, ILD_1
- Output flip-flops and 3-state output flip-flops, for example, OFD and OFDT
- Clock buffers, for example, BUFG_F

Use the Set Port Is Pad command to define the specified ports as I/O pads. You can set the Set Port Is Pad command to insert the default pads on all the I/O ports as follows.

```
set_port_is_pad "*"
```

To insert the pads in the design, use the Insert Pads command as follows.

```
insert_pads
```

You can specify directly which I/O cells you want to use by instantiating the I/O cells in your HDL. You need to place a Don't Touch attribute on the I/O cells that are instantiated to avoid compiling errors. See the Synopsys documentation for more information on placing Don't Touch attributes.

The following sections provide a general description of the XC4000/A/D/H, XC3100/A, and XC3000/A/L devices and describes how to implement additional I/O features manually.

XC4000/A/D IOBs

This section describes how to configure the input and output signal, as well as how to set the output slew rate. You can configure XC4000/A/D IOBs as inputs, outputs, or bidirectional signals with or without a pull-up resistor or pull-down resistor, independent of the pin usage.

Inputs

The Design Compiler utilizes registered I/Os if the flip-flop or latch does not use the Clock Enable, Clear Direct, or Preset pin.

The buffered input signal drives the data input of a storage element, which you can configure as a flip-flop or a latch. You can use the buffered signal in conjunction with the input flip-flop or latch.

By default, a delay buffer added to the signal feeding the data input of the input flip-flop or latch avoids a possible hold time violation. Instantiating a flip-flop or latch, such as an IFD_F or ILD_1F, removes this delay because these cells include a NODELAY attribute. Refer to the "XC4000/A/H Primitives and Hard Macros" appendix for a complete list of primitives that include NODELAY attributes.

Outputs

The output signals, which can drive the programmable 3-state output buffer, can be registered or direct. 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, and XC4000A output buffers can sink 24 mA.

XC4000/D Slew Rate

The XC4000 output buffers have a default slow slew rate that alleviates ground-bounce problems or a fast slew rate that reduces the output delay. The SLOW option increases the transition time and reduces the noise level. The FAST option decreases the transition time and increases the noise level.

Warning: Synopsys and Xilinx define slew rate using opposite terms. Synopsys uses *slew control*, whereas Xilinx uses *slew rate*. For example, the Synopsys HIGH slew control is equivalent to the Xilinx SLOW slew rate.

There are two types of output buffers in the XSI libraries. The default output buffer has a FAST attribute assigned to it, that is, OBUF_F (output buffer) and OBUFT_F (3-state output buffer). However, to avoid a possible ground-bounce problem, Xilinx recommends that you select SLOW as the default slew rate. Assign a FAST slew rate only to output buffers that require additional speed.

The Design Compiler V3.1 or later automatically infers a FAST output slew rate. To set the default slew rate to SLOW (high control), use the Set Pad Type command.

```
set_pad_type -slewrate HIGH all_outputs()
```

Set this command before implementing the Insert Pads command.

To change any output port to a FAST slew rate after changing the default to SLOW, use the following command. Replace *port* with the name of the output port.

set_pad_type -slewrate NONE {*port*}

Table 6-5 XC4000 Slew Rate Settings

Xilinx Slewrate	Synopsys Slew Control Attribute	Design Compiler Command
SLOW	HIGH	**set_pad_type -slewrate HIGH** {*port*}
FAST	NONE	**set_pad_type -slewrate NONE** {*port*}

XC4000A Slew Rate

The XC4000A family offers more output slew-rate control options for each individual output drive: 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.

Warning: Synopsys and Xilinx define slew rate using opposite terms. Synopsys uses *slew control*, whereas Xilinx uses *slew rate*. For example, the Synopsys HIGH slew control is equivalent to the Xilinx SLOW slew rate.

The Design Compiler V3.1 or later automatically infers a FAST output slew rate. To set the default slew rate to SLOW, use the Set Pad Type command.

```
set_pad_type -slewrate HIGH all_outputs()
```

Set this command before using the Insert Pads command.

To change an output to a FAST, MEDFAST or MEDSLOW slew rate after setting the default to SLOW, use the slew rate options found in the following table. Replace *port* with the name of the output port.

Table 6-6 XC4000A Slew Rate Settings

Xilinx Slewrate	Synopsys Slew Control Attribute	Design Compiler Command
SLOW	HIGH	**set_pad_type –slewrate HIGH** {*port*}
MEDSLOW	MEDIUM	**set_pad_type –slewrate MEDIUM** {*port*}
MEDFAST	LOW	**set_pad_type –slewrate LOW** {*port*}
FAST	NONE	**set_pad_type –slewrate NONE** {*port*}

The buffers have an _F suffix for FAST slew rate, an _S suffix for a SLOW slew rate, an _MF suffix for MEDFAST, and an _MS suffix for MEDSLOW. Refer to the “XC4000/A/D/H Primitives and Hard Macros” appendix at the end of this user guide for a full listing of all cells that XSI can instantiate into a design.

Warning: The reported IOB timing delays reflect the delays for an XC4000 device, not an XC4000A device. XC4000A delays vary slightly from XC4000 delays. You can find the actual IOB delay numbers for the XC4000A devices in *The Programmable Logic Data Book*. You can use the Report Timing command to generate a timing report after invoking the Insert Pads command to get accurate I/O cell delays. However, the delays for the internal gates are not accurate because no mapping information exists.

XC4000H IOBs

Because the XC4000H family almost doubles the number of input/output pins of XC4000 devices, the output drivers are more powerful and flexible. You can configure the XC4000H IOBs as input, output, or bidirectional signals. You can configure each I/O pad with or without a pull-up or pull-down resistor, independent of the pin usage.

Inputs

XC4000H devices contain no input flip-flops or latches. You can configure each input individually with TTL or CMOS input

thresholds. You must set the threshold level for each input. The buffers have a _CMOS suffix for the CMOS input threshold and a _TTL suffix for the TTL-input threshold. To set the input threshold, you must instantiate an input buffer with a CMOS or TTL threshold, or use the Set Pad Type command with the Exact option. Refer to the Synopsys documentation for more information on the Set Pad Type command.

Refer to the “XC4000/A/D/H Primitives and Hard Macros” appendix at the end of this user guide for a full listing of all cells.

If you do not specify the threshold, Synopsys assigns each input a random input threshold. Use the following commands to set all inputs to CMOS or TTL.

- For the CMOS threshold, enter the following on the command line.

  set_pad_type –vih 3.33 –vil 1.05 all_inputs()

- For the TTL threshold, enter the following on the command line.

  set_pad_type –vih 2.0 –vil 0.8 all_inputs()

Note: You can use the All Inputs command to specify the names of all input ports; refer to your Synopsys documentation for more information.

You must set the input threshold *after* you compile the design. The following is an example set of commands you can use to compile, set the input threshold, set the output threshold, and insert the pads.

```
compile
set_pad_type –vih 3.33 –vil 1.05 all_inputs()
set_pad_type –voh 4.75 –voh 0.6 all_outputs()
set_port_is_pad
insert_pads
```

Figure 6-3 Example Compilation Flow for Setting Input and Output Thresholds

Outputs

XC4000H devices contain no output flip-flops. You can individually configure the outputs as either TTL- or CMOS-compatible. TTL-level outputs are the best choice for systems that use TTL-level input

thresholds. CMOS-level outputs are ideal for systems that use CMOS input thresholds. To change the output threshold, you must instantiate an output or bidirectional buffer with a TTL or CMOS threshold, or use the Set Pad Type command with the Exact option. Refer to the Synopsys documentation for more information on the Set Pad Type command. The output and bidirectional cells are listed in the "XC4000/A/D/H Primitives and Hard Macros" appendix at the end of this user guide.

You must set the threshold level for each output. If you do not specify the threshold, Synopsys assigns each output a random output threshold. Use the following commands to set the output threshold.

- For the CMOS threshold, enter the following on the command line.

  ```
  set_pad_type -voh 4.75 -vol 0.6 all_outputs()
  ```

- For the TTL threshold, enter the following on the command line.

  ```
  set_pad_type -voh 2.4 -vol 0.5 all_outputs()
  ```

Note: Use the All Outputs command to specify all output ports.

You must set the output threshold *after* you compile the design. Figure 6-3 is an example set of commands used to compile, set the input threshold, set the output threshold, and insert the pads.

Warning: XC4000H devices do not have flip-flops in the IOBs. To prevent the Design Compiler from pulling any flip-flops into the IOBs, insert the pads after compiling the design.

Note: The IOB timing delays reported for XC4000H devices are not included. Execute the Report Timing command after running the Insert Pads command to report accurate I/O cell delays. However, the reported internal gate delays are not accurate.

XC4000H Slew Rate

The XC4000H family offers a choice of CMOS- or TTL-level output and input thresholds that you can select per pin. XC4000H devices have a capacitive and a resistive slew rate. The XC4000H outputs sink 24 mA.

You can configure each output for either of two slew-rate options, which affect only the pull-down operation — resistive or capacitive.

The resistive load (RES) has a pull-down transistor that is driven hard, resulting in a practically constant on-resistance of about 10 ohms. Selecting the resistive load results in the fastest High-to-Low transition and the capability to sink 24 mA with a voltage of 500 mV. Many outputs switch High to Low simultaneously, especially when they are discharging a capacitive load, which might result in excessive ground bounce.

When the output is configured for a capacitive load (CAP) or soft edge, the High-to-Low transition starts as described previously, but the drive to the pull-down transistor is reduced as soon as the output voltage reaches a value around 1 V. Selecting a capacitive load results in a higher resistance in the pull-down transistor, slowing down of the falling edge, and significantly reduced ground bounce. Refer to *The Programmable Logic Data Book* for more details.

To change any of the output ports to a capacitive slew rate, use the Set Pad Type command. Replace *port* with the name of the output port.

```
set_pad_type -slewrate HIGH {port}
```

Table 6-7 XC4000H Slew Rates

Xilinx Slew Rates	Synopsys Slew Control Attribute	Design Compiler Command
CAP	HIGH	**set_pad_type -slewrate HIGH** {*port*}
RES	NONE	**set_pad_type -slewrate NONE** {*port*}

Note: Set this command after specifying the Set Port Is Pad command and before using the Insert Pads command.

For bidirectional cells, the input threshold is listed before the output threshold and slew rate. Refer to the “XC4000/A/D/H Primitives and Hard Macros” appendix for a complete listing of all cells that XSI can instantiate into a design.

XC3000/A/L and XC3100/A IOBs

You can configure the XC3000/A/L and XC3100/A IOBs as inputs, outputs, or bidirectional signals.

Inputs

XC3000/A/L and XC3100/A IOBs can have a registered, latched, or direct input.

Outputs

The outputs have registered or direct outputs with 3-state capability. A registered or direct output signal can drive the programmable 3-state output buffer. The register uses a positive-edge-triggered or negative-edge-triggered flip-flop.

You can assign several parameters to the IOBs, which are listed in the following sections.

XC3000/A/L and XC3100/A Slew Rate

The XC3000/A/L and XC3100/A output buffers with a slow slew rate alleviate ground-bounce problems, and those with a fast slew rate reduce the output delay. The SLOW option increases the transition time and reduces the noise level. The FAST option decreases the transition time and increases the noise level.

Warning: Synopsys and Xilinx define slew rate using opposite terms. Synopsys uses *slew control*, whereas Xilinx uses *slew rate*. For example, the Synopsys HIGH slew control is equivalent to the Xilinx SLOW slew rate.

There are two types of output buffers in the XSI libraries. The default output buffer has a FAST attribute assigned to it, that is, OBUF_F (output buffer) and OBUFT_F (3-state output buffer). However, to avoid a possible ground-bounce problem, Xilinx recommends that you select SLOW as the default slew rate. Assign a FAST slew rate only to output buffers that require additional speed.

The Design Compiler V3.1 or later automatically infers a FAST output slew rate. To set the default slew rate to SLOW (high control) use the Set Pad Type command.

```
set_pad_type –slewrate HIGH all_outputs()
```

Note: Set this command before implementing the Insert Pads commands.

To change any output port to a FAST slew rate after changing the default to SLOW, use the following command. Replace *p* with the name of the output port.

set_pad_type -slewrate NONE {*port*}

Table 6-8 XC3000/A/L and XC3100/A Slew Rate Settings

Xilinx Slew Rate	Synopsys Slew Control Attribute	Design Compiler Command
SLOW	HIGH	**set_pad_type -slewrate HIGH** {*port*}
FAST	NONE	**set_pad_type -slewrate NONE** {*port*}

Assigning and Prohibiting Pad Locations

You can specify pad locations by typing the following in the Command window.

set_attribute *pad* **"pad_location" -type string **
"*pin number***"**

Refer to *The Programmable Logic Data Book* for the locations and name of the pins.

Note: Pin names do not always start with a P.

You can also specify pads locations using a constraints (CST) file. For more information about the constraints file, refer to the *XACT Libraries Guide*.

Implementing 3-State Output

For the Design Compiler to infer the use of 3-state output flip-flops, such as OFDT, two conditions must be met: the flip-flop must directly drive the 3-state signal, and the HDL code of the flip-flop must be in the same process as the 3-state HDL code. The following sections illustrate a flip-flop that does not directly drive the 3-state signal and one that does directly drive the 3-state signal.

Not Directly Driving the 3-State Signal

The flip-flop must directly drive the 3-state signal. If any logic exists between the flip-flop and the 3-state signal connected to the output

flip-flop, the FPGA Compiler does not infer a 3-state output flip-flop, as illustrated by Figure 6-4 and Figure 6-5. Figure 6-6 is a schematic representation.

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

entity three_ex1 is
   port (BUS_IN, EN, CLK: in STD_LOGIC;
         BUS_OUT: out STD_LOGIC);
end three_ex1;

architecture BEHAVIORAL of three_ex1 is

signal BUS_IN_REG, BUS_OUT_REG: STD_LOGIC;

begin
sync: process (CLK)
   begin
        if (CLK'event and CLK='1') then
           BUS_IN_REG <= BUS_IN;
           BUS_OUT_REG <= BUS_IN_REG;
        end if;
   end process;

BUS_OUT <= BUS_OUT_REG when (EN='0') else 'Z';

end BEHAVIORAL;
```

Figure 6-4 Register Not Directly Driving 3-State (VHDL)

```
/*
 * three_ex1 - Behavioral Model
 * Example of 3-state assignment NOT in clock process
 * XSI v3.2
 *      @(#)three_ex1.v 1.2     8/4/94
 */

module three_ex1(BUS_IN, EN, CLK, BUS_OUT) ;
input   BUS_IN ;
input   EN ;
input   CLK ;
output  BUS_OUT ;

reg     BUS_OUT_REG, BUS_IN_REG, BUS_OUT;


//assign BUS_OUT = (EN == 1´b0) ? BUS_OUT_REG : 1´bz ;

always @(posedge CLK)
begin
   BUS_OUT_REG = BUS_IN_REG ;
   BUS_IN_REG = BUS_IN ;
end

always @(EN or BUS_OUT_REG)
begin
   if (!EN)
     BUS_OUT = BUS_OUT_REG;
   else
     BUS_OUT = 1´bz;
end

endmodule
```

Figure 6-5 Register Not Directly Driving 3-State (Verilog HDL)

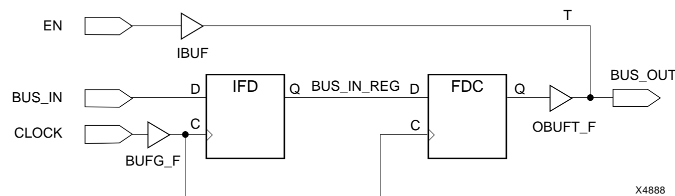


Figure 6-6 No Output Register Inferred

Directly Driving the 3-State Signal

Figure 6-7 and Figure 6-8 illustrate how to behaviorally implement a 3-state output register for VHDL and Verilog HDL, respectively.

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

entity three_ex2 is
   port (BUS_IN,EN,CLK: in STD_LOGIC;
         BUS_OUT: out STD_LOGIC);
end three_ex2;

architecture BEHAVIORAL of three_ex2 is

signal BUS_IN_REG: STD_LOGIC;

begin
sync: process (CLK, EN)
   begin
        if (CLK'event and CLK='1') then
           BUS_IN_REG <= BUS_IN;
           if (EN='0') then
                BUS_OUT <= BUS_IN_REG;
           else
                BUS_OUT <= 'Z';
           end if;
        end if;
   end process;

end BEHAVIORAL;
```

Figure 6-7 Register and 3-State in the Same Process (VHDL)

```
/*
 * three_ex2 - Behavioral Model
 * Example of 3-state assignment in the the clock process
 * XSi v3.2
 *      @(#)three_ex2.v         8/19/94
 */

module three_ex2(BUS_IN, EN, CLK, BUS_OUT) ;
input   BUS_IN ;
input   EN ;
input   CLK ;
output  BUS_OUT ;

reg     BUS_OUT ;
reg     BUS_IN_Q, BUS_IN_REG ;

always @(posedge CLK)
begin
   BUS_IN_REG = BUS_IN_Q ;
   BUS_IN_Q = BUS_IN ;
   if (!EN) BUS_OUT = BUS_IN_REG;
   else  BUS_OUT = 1'bz;
end

endmodule
```

Figure 6-8 Register and 3-State in the Same Process (Verilog HDL)

Figure 6-9 is the schematic representation of a flip-flop driving a 3-state output flip-flop.

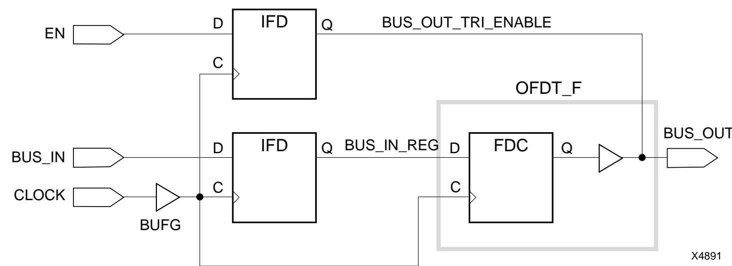


Figure 6-9 Output Register Inferred

Inserting Bidirectional I/Os

The Design Compiler has the ability to insert non-registered bidirectional ports. The 3-state signal that drives the output buffer must be described in the same hierarchy level as the input signal, as shown in Figure 6-7 and Figure 6-8.

Instantiating a Registered Bidirectional I/O

The VHDL design, bidi_reg.vhd, and the Verilog HDL design, bidi_reg.v, are examples of a top-level design that instantiates a core design, reg4. 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 Design Compiler cannot automatically infer the OFDT_F (3-state output buffers with a FAST slew rate) cells in bidirectional I/Os. Therefore, these cells and the IBUF are instantiated into the top-level design.

```
entity bidi_reg is
        port (SIGA: inout BIT_VECTOR (3 downto 0);
              LOADA, CLOCK1, CLOCK2, RST: in BIT);
end bidi_reg;

architecture BEHAV of bidi_reg is
        component reg4
           port (INX: in BIT_VECTOR (3 downto 0);
                 LOAD, CLOCK, RESET: in BIT;
                 OUTX: buffer BIT_VECTOR (3 downto 0));
        end component;

        component OFDT_F
           port (D: in BIT;
                 C: in BIT;
                 T: in BIT;
                 O: out BIT);
        end component;

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

signal INA, OUTA: BIT_VECTOR (3 downto 0);
begin
        U5: reg4 port map (INA, LOADA, CLOCK1, RST, OUTA);
        U0: OFDT_F port map (OUTA(0), CLOCK2, LOADA, SIGA(0));
        U1: OFDT_F port map (OUTA(1), CLOCK2, LOADA, SIGA(1));
        U2: OFDT_F port map (OUTA(2), CLOCK2, LOADA, SIGA(2));
        U3: OFDT_F port map (OUTA(3), CLOCK2, LOADA, SIGA(3));
        U4: IBUF port map (SIGA(0), INA(0));
        U6: IBUF port map (SIGA(1), INA(1));
        U7: IBUF port map (SIGA(2), INA(2));
        U8: IBUF port map (SIGA(3), INA(3));
end BEHAV;
```

Figure 6-10 Bidi_reg.vhd

```
/*
 * bidi_reg - Structural Model
 * Register Bidirectional I/O Example
 * XSI v3.2
 *      @(#)bidi_reg.v  8/19/94
 */

module bidi_reg (SIGA, LOADA, CLOCK1, CLOCK2, RST) ;
inout   [3:0]   SIGA ;
input           LOADA ;
input           CLOCK1 ;
input           CLOCK2 ;
input           RST;

wire    [3:0]   INA, OUTA ;
// Netlist

reg4 U5 (INA, LOADA, CLOCK1, RST, OUTA) ;


OFDT_F  U0 (.D(OUTA[0]), .C(CLOCK2), .T(LOADA), .O(SIGA[0])) ;
OFDT_F  U1 (.D(OUTA[1]), .C(CLOCK2), .T(LOADA), .O(SIGA[1])) ;
OFDT_F  U2 (.D(OUTA[2]), .C(CLOCK2), .T(LOADA), .O(SIGA[2])) ;
OFDT_F  U3 (.D(OUTA[3]), .C(CLOCK2), .T(LOADA), .O(SIGA[3])) ;
IBUF    U4 (.I(SIGA[0]), .O(INA[0])) ;
IBUF    U6 (.I(SIGA[1]), .O(INA[1])) ;
IBUF    U7 (.I(SIGA[2]), .O(INA[2])) ;
IBUF    U8 (.I(SIGA[3]), .O(INA[3])) ;

endmodule
```

Figure 6-11 Bidi_reg.v

Compiling Bidirectional I/O

Do not use the Set Port Is Pad command for the instantiated I/O cells. For example, in the bidi_reg.vhd example, you would only use the following commands to insert the I/Os for the LOADA, RST, CLOCK1, and CLOCK2 signals only.

```
set_port_is_pad {LOADA RST CLOCK1 CLOCK2}

insert_pads
```

Before compiling the design, you must place a Don't Touch attribute on any instantiated I/O cells as follows, so that the I/O cells are not altered.

```
dont_touch {U0 U1 U2 U3 U5 U6 U7 U8}
```

The script files used to compile bidi_reg.vhd and bidi_reg.v are shown in Figure 6-12 and Figure 6-13, respectively.

```
/* ================================================*/
/* Sample Script for Synopsys to Xilinx Using      */
/*           the Design Compiler                   */
/* ================================================*/

/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Read in the design                   */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design    */

   TOP = bidi_reg
   SUB1= reg4

/* Set the Designer and Company name for
   documentation.                                  */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                         */

   analyze -format vhdl SUB1 + ".vhd"
   analyze -format vhdl TOP + ".vhd"
   elaborate SUB1
   elaborate TOP

/* Set the current design to the top level         */

   current_design TOP

/* Add pads to the design. Make sure the current
   design is the top-level module.                */

   set_port_is_pad {LOADA RST CLOCK1 CLOCK2}
   insert_pads
   dont_touch {U0 U1 U2 U3 U4 U6 U7 U8}

/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Compile the design                 */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.           */

   remove_constraint -all

/* Synthesize and optimize the design              */

   compile -map_effort med
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Save the design                     */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                     */

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

/* Set the part type                                */

   set_attribute TOP "part" -type string "3020apc84-6"

/* Write out the design to a DB file                */

   write -format db -hierarchy -output TOP + ".db"

/* Save design in EDIF format as <design>.sedif     */

   write -format edif -hierarchy -output TOP + ".sedif"


/* Exit the Compiler.                               */

   exit
```

Figure 6-12 Bidi_reg.script (VHDL)

```
/* ================================================*/
/* Sample Script for Synopsys to Xilinx Using      */
/*            the Design Compiler                  */
/* ================================================*/

/* +++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Read in the design                  */
/* +++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design    */

   TOP = bidi_reg
   SUB1= reg4

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format verilog SUB1 + ".v"
   analyze -format verilog TOP + ".v"
   elaborate SUB1
   elaborate TOP

/* Set the current design to the top level          */

   current_design TOP

/* Add pads to the design. Make sure the current
   design is the top-level module.                 */

   set_port_is_pad {LOADA RST CLOCK1 CLOCK2}
   insert_pads
   dont_touch {U0 U1 U2 U3 U4 U6 U7 U8}

/* +++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Compile the design                */
/* +++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all

/* Synthesize and optimize the design               */

   compile -map_effort med
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Save the design                        */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                       */

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

/* Set the part type                                  */

   set_attribute TOP "part" -type string "3020apc84-6"

/* Write out the design to a DB file                  */

   write -format db -hierarchy -output TOP + ".db"

/* Save design in EDIF format as <design>.sedif       */

   write -format edif -hierarchy -output TOP + ".sedif"


/* Exit the Compiler.                                 */

   exit
```

Figure 6-13 Bidi_reg.script (Verilog HDL)

Using Unbonded IOBs

Note: This section does not apply to XC4000H devices because they do not have flip-flops in the IOB.

In some package/device pairs, not all pads of the device are bonded to a package pin. You can use these unbonded IOBs and the flip-flops inside them in your design by instantiating unbonded primitives, which are indicated by a _U suffix. Refer to the "XC3000A/L and XC3100/A Primitives" and "XC4000/A/D/H Primitives and Hard Macros" appendixes for a complete listing of all unbonded cells.

Adding Pull-Up and Pull-Down Resistors

XC3000 and XC3100 devices have pull-up resistors that you can use to pull up an unconnected IOB. By default, all unused IOBs are configured as an input with a pull-up resistor. Refer to the "XC3000/A/L and XC3100/A Primitives" appendix for a listing of all cells and their pin names for instantiation.

The XC4000 family has high impedance pull-up and pull-down resistors that you can connect to an input or output buffer. You can instantiate these cells, PULLUP and PULLDOWN, into your HDL

design. Refer to the "XC4000/A/D/H Primitives and Hard Macros" appendix for a listing of all cells and their pin names for instantiation.

Removing the Default Input Delay (XC4000 Only)

The XC4000 input flip-flops and latches have a default delay preceding the data to the input flip-flop or latch. This delay prevents any possible hold-time violations if you have a clock signal that is also coming into the device and clocking the input flip-flop or latch.

You can remove this delay by instantiating a cell that includes the NODELAY attribute if you need additional input speed and have no possibility of a hold-time violation. The "XC4000/A/D/H Primitives and Hard Macros" appendix lists all cells that include a NODELAY attribute. The input flip-flops or latches with an _F suffix have a NODELAY attribute assigned to the cell.

Initializing the IOB Flip-Flop to Preset (XC4000 Only)

You can initialize XC4000 IOB flip-flops to either Clear or Preset. The default is Clear. To initialize an I/O flip-flop or latch to Preset, use the following command to attach an INIT=S attribute to the flip-flop.

```
set_attribute "register_name" xnf_init \
"S" type string
```

Replace *register_name* with the name of the I/O flip-flop.

You can instantiate I/O cells with the INIT=S attribute already assigned to them. Refer to the "XC4000/A/D/H Primitives and Hard Macros" appendix for a list of all cells and their pin names for instantiation.

Inserting Clock Buffers

Xilinx recommends that your design contain global clock buffers to take advantage of the low-skew, high-drive capabilities of the generic clock buffer, BUFG, which is assigned to a specific clock buffer by PPR or APR. The Design Compiler automatically inserts a BUFG_F generic clock buffer whenever an input signal drives a clock signal when you invoke the Insert Pads command.

XC4000/A/D/H Clock Buffers

Each XC4000 device contains four primary and four secondary global buffers that share the same routing resources. Xilinx recommends that you use the generic global buffer, BUFG, for up to four low-skew, high-fanout clock signals. Both the primary and secondary clock buffers can be driven by signals sourced from inside the device; however, the primary global buffer always uses the dedicated I/O pad. You can use the secondary global clock buffer to buffer a high-fanout, low-skew clock signal that is sourced from inside the FPGA.

The Design Compiler assigns a BUFG to any input signals driving a clock pin. XNFPrep converts the generic clock buffers to BUFGS.

You can instantiate clock buffers into the HDL or use the Set Pad Type command with the Exact option to specify exactly which clock buffer should be used. Refer to the Synopsys documentation for more information on the Set Pad Type command.

You can also use a secondary clock buffer to drive a net that is not a clock net or is not sourced from an input pin. Refer to the "XC4000/A/D/H Primitives and Hard Macros" appendix for a list of clock buffer cells.

Note: The _F suffix appended to the clock buffer name indicates that the clock buffer uses the dedicated input pad. This implementation is faster than using a non-dedicated input pad.

Warning: The Design Compiler may assign more clock buffers than are available in the device. Refer to the "Controlling Clock Buffer Insertion" section later in this chapter for more information on how to avoid having clock buffers assigned to input pins that do not directly drive clock pins, and to avoid having more that four generic clock buffers used per design.

XC3000/A/L and XC3100/A Clock Buffers

Each XC3000 and XC3100 device contains one global clock buffer, GCLK, and one alternate clock buffer, ACLK. The Design Compiler automatically assigns a generic clock buffer, BUFG, to any input signal driving a clock pin. APR or PPR assigns a GCLK to the highest fanout clock buffer and a ACLK to the second-highest clock buffer.

You can instantiate clock buffers into the HDL or use the Set Pad Type command with the Exact option to specify exactly which clock buffer. Refer to the “XC3000/A/L and XC3100/A Primitives” or “XC4000/A/D/H Primitives and Hard Macros” appendix in this user guide for a list of clock buffer cells. Refer to the Synopsys documentation for more information on the Set Pad Type command.

Note: The _F suffix appended to the clock buffer name indicates that the clock buffer uses the dedicated input pad. This implementation is faster than using a non-dedicated input pad.

Warning: The Design Compiler may assign more than two clock buffers. (Only two clock buffers are available in the device). To avoid assigning additional clock buffers, refer to the “Controlling Clock Buffer Insertion” section, which follows.

Controlling Clock Buffer Insertion

The Design Compiler assigns a BUFG_F to any input signal that drives a clock signal; however, XC4000 devices have four global buffers and four secondary buffers. XC3000 and XC3100 devices have only two global clock buffers. Xilinx recommends that you use only four generic buffers, BUFG, per XC4000 designs and only two for XC3000 and XC3100 designs.

You should assign the global clock buffer to the top four high-fanout, low-skew clock signals for XC4000 designs and the top two high-fanout, low-skew clock signals for XC3000 and XC3100 designs.

Figure 6-14 and Figure 6-15 illustrate a design with a gated clock using VHDL or Verilog HDL, respectively. By default, the Design Compiler assigns clock buffers to the signals IN1, IN2, IN3, IN4, and CLK because they are considered to be driving a clock pin. However, only use clock buffers for input signals directly driving high-fanout clock nets.

```
entity gate_clock is
   port (IN1,IN2,IN3,IN4,IN5,CLK,LOAD: in BIT;
         OUT1: buffer BIT);
end gate_clock;

architecture BEHAVIORAL of gate_clock is
signal GATECLK: BIT;
begin
GATECLK <= not((((IN1 and IN2) and IN3) and IN4) and CLK);
   process (GATECLK, IN5,LOAD)
   begin
        if (GATECLK'event and GATECLK='1') then
                   if (LOAD='1') then
                        OUT1 <= IN5;
                   else
                        OUT1 <= OUT1;
                   end if;
        end if;
   end process;
end BEHAVIORAL;
```

Figure 6-14 Gated Clock (VHDL)

```
/*
 * Gateclk - Behavioral Model
 * Gated Clock Example
 * XSI v3.2
 *      @(#)gate_clock.v        1.2     8/4/94
 */

module gate_clock(IN1, IN2, IN3, IN4, IN5, CLK, LOAD, OUT1) ;
input   IN1 ;
input   IN2 ;
input   IN3 ;
input   IN4 ;
input   IN5 ;
input   CLK ;
input   LOAD ;
output  OUT1;

reg     OUT1;

wire GATECLK ;

assign GATECLK = ~(IN1 & IN2 & IN3 & IN4 & CLK) ;

always @(posedge GATECLK)
begin
   if (LOAD == 1'b1)
      OUT1 = IN5 ;
end

endmodule
```

Figure 6-15 Gated Clock (Verilog HDL)

In Figure 6-16, a clock buffer should *not* be inserted on the signals IN1 – IN4 and CLK because they are not high-fanout, low-skew clock signals. The Design Compiler inserts global clock buffers for signals IN2 through IN4 and the CLK signal.

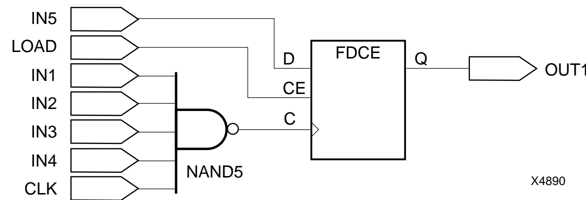


Figure 6-16 Gated Clock Schematic

Warning: The Design Compiler identifies clock ports by tracing back from the clock pins on the flip-flops. If the clock signal is gated, the gated signals also are assigned a clock buffer.

Figure 6-17 shows the gates inserted after executing the Insert Pads and Compile commands.

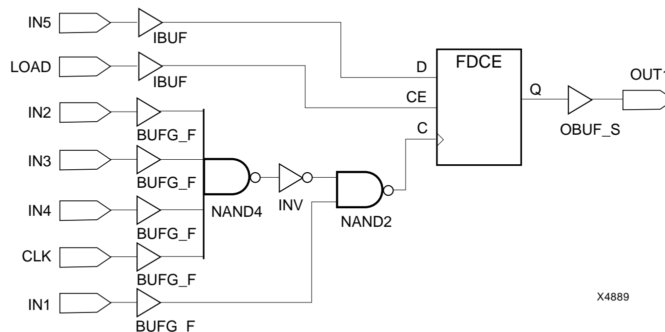


Figure 6-17 Gated Clock After Pad Insertion

If the design contains gated clocks or has more than four input pins that drive the clock pin for XC4000 designs and more than two input pins for XC3000 and XC3100 designs, you must prevent the input pins from having a BUFG inserted.

Determining the Number of Clock Buffers

To determine how many clock buffers the Design Compiler will insert in the design, use the Report Cell command.

report_cell

Figure 6-18 illustrates the report output for the gated clock example.

```
****************************************
Report : cell
Design : gate_clock
Version: v3.1b-20502
Date   : Fri Sep  9 15:15:58 1994
****************************************

Attributes:
    b - black box (unknown)
    h - hierarchical
    n - noncombinational
    r - removable
    u - contains unmapped logic

Cell                    Reference      Library           Area  Attributes
--------------------------------------------------------------------------------
OUT1_reg                FDPEI          xprim_3000a-6     0.50  n
U24                     OBUF           xprim_3000a-6     0.00
U25                     IBUF           xprim_3000a-6     0.00
U26                     BUFG_F         xprim_3020a-6     0.00
U27                     IBUF           xprim_3000a-6     0.00
U28                     BUFG_F         xprim_3020a-6     0.00
U29                     BUFG_F         xprim_3020a-6     0.00
U30                     BUFG_F         xprim_3020a-6     0.00
U31                     BUFG_F         xprim_3020a-6     0.00
U32                     GND            xgen_3000         0.00
U33                     NAND5          xprim_3000a-6     1.00
--------------------------------------------------------------------------------
Total 11 cells                                           1.50
```

Figure 6-18 Clock Buffer Report for Gated Clock Example

Preventing the Insertion of Clock Buffers

To prevent the Design Compiler from inserting the BUFG, specify the Set Pad Type command with the following options after reading the design and before inserting the pads.

set_pad_type –no_clock {*clock_ports*}

Replace *clock_ports* with the name of the input pins on which you do not want a clock buffer inserted. For the gated clock example, you would enter the following.

```
set_pad_type –no_clock {IN1, IN2, IN3, IN4, CLK}
```

Follow the normal procedures for setting the ports as pads and inserting the pads as follows.

```
set_port_is_pad "*"

insert_pads
```

Using Memory

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

XC3000/A/L, XC3100/A, and XC4000/A/D/H devices include on-chip static memory resources. The XC3000 and XC3100 families can efficiently implement ROM using the CLB function generators. The XC4000 family can efficiently implement RAM and ROM using the CLB function generators.

You can implement a ROM by describing it behaviorally as shown in Figure 6-19 and Figure 6-20 for VHDL and Verilog HDL, respectively. The XSI XC4000 libraries contain 16 x 1 (16 deep x 1 wide) RAM and 32 x 1 (32 deep x 1 wide) RAM primitives, and 16 x 1 and 32 x 1 ROM primitives that can be instantiated.

You can also implement memory using the MemGen program, which is included in the XACT Development System. MemGen can create RAM and ROM between 1 to 32 bits wide and 2 to 256 bits deep. This section includes an example of using MemGen with XSI. Refer to the *XACT Reference Guide, Volume 1* for more information about using MemGen.

XC4000 RAMs

You can implement RAMs in your HDL by the following methods.

- You can instantiate 16 x 1 and 32 x 1 RAMs from the XSI primitive libraries.
- You can implement any other RAM size using MemGen.

Warning: Do not behaviorally describe RAMs in VHDL because compiling creates combinatorial loops.

ROMs

You can implement ROMs for XC4000 devices in your HDL by the following methods.

- You can describe ROMs behaviorally.
- You can instantiate 16 x 1 and 32 x 1 ROM primitives.
- You can implement other ROMs using MemGen.

To instantiate the ROM primitives, ROM16X1 and ROM32X1, into your HDL design, use the Set Attribute command to define the ROM value.

set_attribute "*instance_name***" xnf_init "***rom_value***"**
type string

For example, if you gave the ROM16X1 an instance name of "U1" and the value of the ROM is F5A3, you can use this command to set the ROM value as follows.

```
set_attribute "U1" xnf_init "F5A3" type string
```

For a 32x1 ROM, specify an 8-digit hexadecimal (hex) value in place of the 4-digit hex value as shown in the previous example.

Note: Instantiating ROM or RAM does not allow you to functionally simulate the design or easily migrate between FPGA families.

Figure 6-19 and Figure 6-20 illustrate how you can define a ROM in VHDL and Verilog HDL, respectively. The Design Compiler creates ROM from random logic gates that are implemented using function generators.

```
------------------------------------------
--  Behavioral 16x4 ROM Example         --
--          rom16x4_4k.vhd               --
------------------------------------------

entity rom16x4_4k is
        port (  ADDR: in INTEGER range 0 to 15;
                DATA: out BIT_VECTOR (3 downto 0));
end rom16x4_4k;

architecture BEHAV of rom16x4_4k is

        subtype ROM_WORD is BIT_VECTOR (3 downto 0);
        type ROM_TABLE is array (0 to 15) of ROM_WORD;
        constant ROM: ROM_TABLE := ROM_TABLE'(
                ROM_WORD'("0000"),
                ROM_WORD'("0001"),
                ROM_WORD'("0010"),
                ROM_WORD'("0100"),
                ROM_WORD'("1000"),
                ROM_WORD'("1000"),
                ROM_WORD'("1100"),
                ROM_WORD'("1010"),
                ROM_WORD'("1001"),
                ROM_WORD'("1001"),
                ROM_WORD'("1010"),
                ROM_WORD'("1100"),
                ROM_WORD'("1001"),
                ROM_WORD'("1001"),
                ROM_WORD'("1101"),
                ROM_WORD'("1111"));
begin
        DATA <= ROM(ADDR); -- Read from the ROM
end BEHAV;
```

Figure 6-19 Behavioral VHDL for 16 x 4 ROM

```
/*
 * rom16x4_4k - Behavioral Model
 * Behavioral Example of 16x4 ROM
 * XSI v3.2
 *      @(#)rom16x4_4k.v        1.2     8/4/94
 */

module rom16x4_4k(ADDR, DATA) ;
input [3:0] ADDR ;
output [3:0] DATA ;

reg [3:0] DATA ;

// A memory is not created because Synopsys will not synthesize it

always @(ADDR)
begin
   case (ADDR)
      4´b0000 : DATA = 4´b0000 ;
      4´b0001 : DATA = 4´b0001 ;
      4´b0010 : DATA = 4´b0010 ;
      4´b0011 : DATA = 4´b0100 ;
      4´b0100 : DATA = 4´b1000 ;
      4´b0101 : DATA = 4´b1000 ;
      4´b0110 : DATA = 4´b1100 ;
      4´b0111 : DATA = 4´b1010 ;
      4´b1000 : DATA = 4´b1001 ;
      4´b1001 : DATA = 4´b1001 ;
      4´b1010 : DATA = 4´b1010 ;
      4´b1011 : DATA = 4´b1100 ;
      4´b1100 : DATA = 4´b1001 ;
      4´b1101 : DATA = 4´b1001 ;
      4´b1110 : DATA = 4´b1101 ;
      4´b1111 : DATA = 4´b1111 ;
   endcase
end

endmodule
```

Figure 6-20 Behavioral Verilog HDL for 16 x 4 ROM

Alternatively, you can implement ROMs using MemGen as shown in Figure 6-21 and Figure 6-22.

Using MemGen

The following steps illustrate how to use MemGen.

1. Create the memory description file, for example, promdata.mem.

   You can use any file name. See Figure 6-21 for a sample memory description file.

2. Run MemGen on the memory description file to create the promdata.xnf file as follows.

   ```
   memgen promdata
   ```

3. Instantiate the memory submodule into the HDL design, as shown in Figure 6-22 or Figure 6-23.

   The name of the address lines *must* be called A0 – A3 and the output data lines O0 – O3. When the design is compiled in the Design Compiler, the following warning occurs.

   ```
   Warning: Unable to resolve reference 'promdata'
   in 'ROM_INT' (LINK-5)
   ```

   You can ignore this warning message.

4. Save the design to an SEDIF file, for example, rom_memgen.sedif.

5. Translate the output file, rom_memgen.sedif, into an XNF file using Syn2XNF.

   The translator, Syn2XNF, automatically merges in the XNF file for the memory, for example, promdata.xnf. Refer to the "Translating SEDIF Files to XNF Files Using Syn2XNF" section at the end of this chapter for more information.

The following figure illustrates a memory description file, promdata.mem.

```
; ==========================================
; Memory file for PROM symbol called 'PROM_SYM' driving 'D_OUT_BUS'
; ==========================================
TYPE     ROM
WIDTH    4
DEPTH    16
DEFAULT  0  ; <== default value here
DATA
        2#0000#,
        2#0001#,
        2#0010#,
        2#0100#,
        2#1000#,
        2#1000#,
        2#1100#,
        2#1010#,
        2#1001#,
        2#1001#,
        2#1010#,
        2#1100#,
        2#1001#,
        2#1001#,
        2#1101#,
        2#1111#; <== END of ROM data
```

Figure 6-21 Memory Description File

Figure 6-22 and Figure 6-23 illustrate instantiating a ROM submodule using VHDL and Verilog HDL, respectively.

```
-----------------------------------------
-- Example of Instantiating a MemGen    --
-- Created Memory File                  --
--          rom_memgen.vhd              --
-----------------------------------------

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

architecture BEHAV of rom_memgen is

component promdata
        port (  A3,A2,A1,A0: in BIT;
                O3,O2,O1,O0: out BIT);
end component;

begin
        u1: promdata port map (A3=>ADDR(3),A2=>ADDR(2),A1=>ADDR(1),A0=>ADDR(0),
                O3=>DATA(3),O2=>DATA(2),O1=>DATA(1),O0=>DATA(0));

end BEHAV;
```

Figure 6-22 Instantiating 16 x 4 ROM Submodule (VHDL)

```
/*
 * rom_memgen - Structural Model
 * Example of Using MEMGEN to create a ROM
 * XSI v3.2
 *      @(#)rom_memgen.v        1.2     9/9/94
 */

module rom_memgen (ADDR,
                   DATA) ;
input [3:0] ADDR ;
output [3:0] DATA ;

promdata u1(.A3(ADDR[3]), .A2(ADDR[2]), .A1(ADDR[1]), .A0(ADDR[0]),
            .O3(DATA[3]), .O2(DATA[2]), .O1(DATA[1]), .O0(DATA[0])) ;

endmodule

module promdata(A3, A2, A1, A0, O3, O2, O1, O0);
input A3, A2, A1, A0;
output O3, O2, O1, O0;

endmodule
```

Figure 6-23 Instantiating 16 x 4 ROM Submodule (Verilog HDL)

Performing Boundary Scan for XC4000 Devices

The XC4000 FPGA devices contain boundary-scan facilities that are compatible with IEEE Standard 1149.1. Refer to the *XACT User Guide* for a detailed description of the XC4000 boundary scan capabilities.

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

Full access to the built-in boundary-scan logic is always available between power-up and the start of configuration. During configuration, you can use the Sample/Preload and Bypass instructions only. Optionally, the built-in logic is fully available after configuration if you specified boundary scan in the design.

In a configured FPGA device, the boundary-scan logic might not be active depending on the configuration data loaded into the part. Activation of the boundary-scan logic, if desired, is part of the design process. For HDL designs, you must instantiate the boundary-scan symbol, BSCAN, and the boundary scan I/O pins, TDI, TMS, TCK, and TDO to access the boundary scan logic after configuration. After configuration, you cannot activate or deactivate boundary scan without changing the entire chip configuration.

Warning: *Do not* use these boundary scan commands: Set JTAG Implementation, Set JTAG Instruction, and Set JTAG Port because they do not work with FPGA devices.

Figure 6-24 illustrates the BSCAN symbol instantiated into an HDL design.

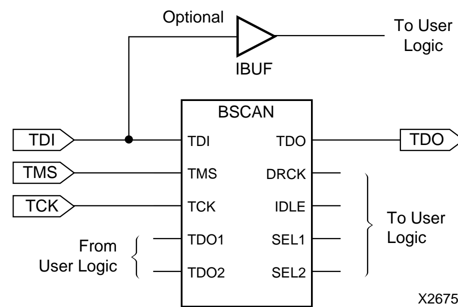


Figure 6-24 Boundary Scan Symbol

Using the Global Set/Reset Net

XC4000, XC3000, and XC3100 devices have a dedicated Global Set/Reset (GSR) net that initializes all CLBs and IOBs. The function of the Global Set/Reset net is separate from the individual Preset (PRE) and Direct Clear (CLR) pin.

How you use the GSR net depends on which Xilinx device you use. Refer to the appropriate subsection.

XC4000 Devices

If the design has a Preset or Direct Clear signal, using the Global Set/Reset net increases the design's performance by reducing the overall routing congestion. You can remove the Preset or Direct Clear signal from the synthesized design and implement it using the dedicated Global Set/Reset net.

Startup State

The STARTUP symbol's Global Set/Reset pin drives the Set/Reset net and connects to each flip-flop's Preset and Direct Clear pin. When

you connect a signal to the STARTUP symbol's GSR pin, the Global Set/Reset net is activated.

The Global Set/Reset net does not appear in the pre-placed and pre-routed XNF file. When the GSR signal is asserted High (the default), every flip-flop and latch is set to the same state it had at the end of configuration as illustrated by Table 6-9. For XC3000 devices, all flip-flops and latches reset to 0 after configuration. When you simulate the routed design, the gate simulator's translation program correctly models the GSR function.

Table 6-9 Initialization State After Configuration (XC4000 Only)

Initializes to 0		Initializes to 1	
FDC	OFD_MF	FDP	OFDI_MF
FDCE	OFD_MS	FDPE	OFDI_MS
IFD	OFD_S	IFDI	OFDI_S
IFD_F	OFD_U	IFDI_F	OFDI_U
IFD_U	OFDT	IFDI_U	OFDTI
ILD_1	OFDT_F	IFDI_1	OFDTI_F
ILD_1F	OFDT_MF	IFDI_1F	OFDTI_MF
ILD_1U	OFDT_MS	IFDI_1U	OFDTI_MS
OFD	OFDT_S	OFDI	OFDTI_S
OFD_F	OFDT_U	OFDI_F	OFDTI_U
OFD_FU			

Note: PPR implements inverters in the XC4000 devices without using additional CLB resources. You can connect any signal to drive the STARTUP symbol GSR pin.

Preset Versus Direct Clear

You can program each flip-flop and latch to be Preset or Direct Clear but *not* both. The flip-flops and latches can be Preset or Direct Clear upon completion of configuration by asserting the Global Set/Reset net and the individual Preset (PRE) and Direct Clear (CLR) pins of the flip-flop or latch. The value of the flip-flop (Preset or Direct Clear) is

the same for all cases. The value of the flip-flop is determined by whether you used the PRE or CLR pin.

If the CLR or PRE pin on a non-I/O flip-flop cell is tied to an active signal, the state of that signal controls the startup state of those flip-flop cells; for example, if you use the PRE pin, the flip-flop starts up in Preset state. If you do not use the CLR and PRE pin, the default is to startup in a Clear state.

You can use the Report Cell command to determine the instance names and the type of flip-flop used as follows.

- FDPE or FDP — Preset upon power-up
- FDCE or FDC — Direct Clear upon power-up

Changing States

If you are not using the Clear pin of a FDCE or FDC cell (grounded), you can override the initial state by issuing the Set Attribute command.

```
set_attribute "cell" fpga_xilinx_init_state \
-type string "S"
```

For IOBs, the default is to start up in a Direct Clear state. You can instantiate an I/O flip-flop and a latch with an INIT=S parameter to have the flip-flop start up in a Preset state.

The following illustrates using the Set Attribute command to change a flip-flop with the cell name of OUTX_reg<0> from a Reset-upon-powerup to a Set-upon-powerup as follows.

Warning: You can use the following command to change the initial state only if you are inferring a flip-flop without Clear and Preset pins. You cannot change the initialization state of instantiated flip-flops.

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

Refer to the Synopsys documentation on the Design Compiler for more information.

Increasing Performance with the GSR Net

Many designs have a net that initializes the majority of the design's flip-flops. If this signal can initialize *all* the flip-flops, you can use the Global Set/Reset net.

To have your HDL simulation match that of the resulting design, you should modify the HDL code so that every flip-flop and latch is Preset or Clear when the Global Set/Reset signal is asserted. However, you must disconnect this signal with the Disconnect Net command after compiling the design and before saving it.

The Design Compiler cannot infer the usage of the Global Set/Reset net from the HDL code.

Note: X-BLOX has the ability to use the Global Set/Reset net automatically if every flip-flop and latch in the design has a common signal driving the Set Direct or Reset Direct pin, that is, the CLR or PRE pin. You can run X-BLOX on any XNF, XTG, or XTF file.

Figure 6-25 and Figure 6-26 are examples of a design that can use the Global Set/Reset net for VHDL and Verilog HDL, respectively. The design contains two flip-flops. One flip-flop is reset and one is set when the signal "RST" is High.

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

entity gsr_ex is
        port (  CLK,RST : in STD_LOGIC;
                ST: buffer std_logic_vector (1 downto 0));
end gsr_ex;

architecture EXAMPLE of gsr_ex is

begin
        process (CLK, RST)
        begin
                if RST= ´1´ then
                        ST <= "01";
                elsif (CLK´event and CLK=´1´) then
                        ST <= ST + "01";
                end if;
        end process;

end EXAMPLE;
```

Figure 6-25 Before Using the GSR Net (VHDL)

```
/*
 * gsr_ex - Behavioral Model
 * XC4000 Global Set/Reset Example
 * XSI v3.2
 *      @(#)gsr_ex.v    1.2     8/22/94
 */

module gsr_ex (CLK, RST, ST) ;
input          CLK ;
input          RST ;
output  [1:0]  ST ;

reg     [1:0]  ST ;

always @(posedge CLK or posedge RST)
begin
   if (RST == 1´b1)
      ST = 2´b01 ;
   else
      ST = ST + 1´b1 ;
end

endmodule
```

Figure 6-26 Before Using the GSR Net (Verilog HDL)

To utilize the Global Set/Reset net, create a level of hierarchy that instantiates the STARTUP symbols and the core design as illustrated in Figure 6-27 and Figure 6-28 for VHDL and Verilog HDL, respectively. Use another signal name, such as GSR in the following design example, and route this signal to the STARTUP symbol's GSR pin.

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

entity top_gsr is
        port(   CLK,GSR,RST: in STD_LOGIC;
                ST : buffer STD_LOGIC_VECTOR (1 downto 0));
end top_gsr;

architecture EXAMPLE of top_gsr is
        component STARTUP
                port (  GSR: in STD_LOGIC);
        end component;

        component gsr_ex
                port (  CLK,RST: in STD_LOGIC;
                        ST : buffer STD_LOGIC_VECTOR (1 downto 0));
        end component;

        begin

         U1 : STARTUP port map (GSR=>GSR);
         U2 : gsr_ex port map (CLK=>CLK,RST=>RST,ST=>ST);
end EXAMPLE;
```

Figure 6-27 Top_gsr.vhd

```
/*
 * top_gsr - Structural Model
 * Example of using the Global Set/Reset net
 * XSI v3.2
 *      @(#)top_gsr.v   1.2     8/4/94
 */

module top_gsr (CLK, GSR, RST, ST) ;
input           CLK ;
input           GSR ;
input           RST ;
output  [1:0]   ST ;

STARTUP U1 (.GSR(GSR)) ;
gsr_ex  U2 (.CLK(CLK), .RST(RST), .ST(ST)) ;

endmodule
```

Figure 6-28 Top_gsr.v

Figure 6-29 and Figure 6-30 contain the procedures for executing the top_gsr.vhd and top_gsr.v designs, respectively.

```
/* ================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*           the Design Compiler                    */
/* ================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Read in the design                    */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design    */

   TOP = top_gsr
   SUB1 = gsr_ex

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format vhdl TOP + ".vhd"
   analyze -format vhdl SUB1 + ".vhd"
   elaborate TOP
   elaborate SUB1

/* Set the current design to the top level          */

   current_design TOP

/* Since the STARTUP block does not have any outputs
   that are beng used in this example, use the dont_
   touch command so the compiler does not remove the
   STARTUP block.                                   */

   dont_touch "U1"


/* Add pads to all ports except RST. Make sure the
   current design is the top-level module.
   Change the default slew rate to SLOW (HIGH slew
   control).                                       */

   set_port_is_pad {CLK GSR ST}
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Compile the design                 */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all

/* Synthesize and optimize the design               */

   compile -map_effort med
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Save the design                     */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                      */

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

/* Set the part type                                 */

   set_attribute TOP "part" -type string "4005pc84-5"

/* Remove the RST signal                             */

   disconnect_net RST -all

/* Write out the design to a DB file                 */

   write -format db -hierarchy -output TOP + ".db"

/* Save design in EDIF format as <design>.sedif      */

   write -format edif -hierarchy -output TOP + ".sedif"


/* Exit the Compiler.                                */

   exit
```

Figure 6-29 Top_gsr Script File (VHDL)

```
/* ==================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*            the Design Compiler                   */
/* ==================================================*/

/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Read in the design                   */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design    */

   TOP = top_gsr
   SUB1 = gsr_ex

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format verilog TOP + ".v"
   analyze -format verilog SUB1 + ".v"
   elaborate TOP
   elaborate SUB1

/* Set the current design to the top level          */

   current_design TOP

/* Since the STARTUP block does not have any outputs
   that are beng used in this example, use the dont_
   touch command so the compiler does not remove the
   STARTUP block.                                   */

   dont_touch "U1"


/* Add pads to all ports except RST. Make sure the
   current design is the top-level module.
   Change the default slew rate to SLOW (HIGH slew
   control).                                       */

   set_port_is_pad {CLK GSR ST}
   set_pad_type -slewrate HIGH all_outputs()
   insert_pads

/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Compile the design                 */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all

/* Synthesize and optimize the design               */

   compile -map_effort med
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Save the design                     */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                     */

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

/* Set the part type                                */

   set_attribute TOP "part" -type string "4005pc84-5"

/* Remove the RST signal                            */

   disconnect_net RST -all

/* Write out the design to a DB file                */

   write -format db -hierarchy -output TOP + ".db"

/* Save design in EDIF format as <design>.sedif     */

   write -format edif -hierarchy -output TOP + ".sedif"


/* Exit the Compiler.                               */

   exit
```

Figure 6-30 Top_gsr Script File (Verilog HDL)

Read the top level (top_gsr) and the core design (gsr_ex) into the Design Compiler. Since the STARTUP block does not use any outputs, the Design Compiler removes the STARTUP block unless you specify the Don't Touch attribute for U1. You must issue this command before inserting the I/O pads.

Before saving the design to an SEDIF file, you must remove the RST signal from the design using the Disconnect Net command. PPR removes any unconnected gates from the design.

When the RST net is disconnected from the circuit, the PRE and CLR pin is no longer used. The flip-flop ST<0> is mapped to an FDPE that initializes to a Preset state (INIT=S), and the flip-flop ST<1> is mapped to an FDCE that initializes to a Direct Clear state (INIT=R).

XC3000 and XC3100 Devices

Xilinx XC3000 and XC3100 devices have a dedicated Global Reset input package pin called $\overline{\text{RESET}}$. You can find the $\overline{\text{RESET}}$ pin's location in *The Programmable Logic Data Book*. This active Low pin has three functions as follows.

- Prior to the start of configuration, a Low input delays the start of the configuration process. An internal circuit senses the application of power and begins a minimal time-out cycle. When the time-out and $\overline{\text{RESET}}$ are completed, the levels of the mode pins (M0, M1, and M2) are sampled and configuration begins.
- If $\overline{\text{RESET}}$ is asserted during a configuration, the FPGA device is reinitialized and restarts the configuration at the termination of $\overline{\text{RESET}}$.
- If $\overline{\text{RESET}}$ is asserted after configuration is complete, it provides a global asynchronous reset of all IOB and CLB storage elements of the FPGA device.

When the $\overline{\text{RESET}}$ pin is asserted Low, *all* flip-flops and I/O latches are asynchronously reset. To use the Global Reset pin, the design's Reset net must be sourced by an input signal and must tolerate having all flip-flops and I/O latches asynchronously reset when asserted. To have your HDL simulation match that of the resulting design, you should modify the HDL code so that every flip-flop and latch is asynchronously reset when the Reset signal is asserted.

You should keep the Reset signal in the HDL code so that your HDL simulation results match the placed and routed simulation results. However, you must disconnect this net using the Disconnect Net command after compiling but before saving the design.

To implement the Global Reset net, connect the Reset net to the $\overline{\text{RESET}}$ package pin. The Reset signal does not appear in your design since it is part of your board layout.

Figure 6-31 and Figure 6-32 illustrate how to implement the Global Reset package pin in VHDL and Verilog HDL, respectively. This example contains two flip-flops. When the input signal "RST" is Low, both flip-flops are asynchronously reset.

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

entity greset is
        port (  clk,rst : in std_logic;
                st: buffer std_logic_vector (0 to 1));
end greset;

architecture example of greset is

begin
        process (clk, rst)
        begin
                if rst= ´0´ then
                        st <= "00";
                elsif (clk´event and clk=´1´) then
                        st <= st + "01";
                end if;
        end process;

end example;
```

Figure 6-31 Implementing Global Reset Pin (VHDL)

```
/*
 * greset - Behavioral Model
 * Example of Design with a global reset
 * XSI v3.2
 *      @(#)greset.v    1.2     8/4/94
 */

module greset(CLK, RST, ST) ;
input           CLK ;
input           RST ;
output  [1:0]   ST ;

reg     [1:0]   ST ;

always @(posedge CLK or negedge RST)
begin
   if (RST == 1´b0)
      ST = 2´b00 ;
   else
      ST = ST + 1´b1 ;
end

endmodule
```

Figure 6-32 Implementing Global Reset Pin (Verilog HDL)

Figure 6-33 and Figure 6-34 are script files that contain the procedures for executing the greset.vhd and greset.v designs, respectively.

```
get_license {Designware-Basic}

/* ==================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*            the Design Compiler                   */
/* ==================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Read in the design                   */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design     */

   TOP = greset

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format vhdl TOP + ".vhd"
   elaborate TOP

/* Set the current design to the top level          */

   current_design TOP

/* Add pads to the design. Make sure the current
   design is the top-level module.
   Insert pads on all ports except ´rst´.             */

   set_port_is_pad {clk, st}
   insert_pads

/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Compile the design                 */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all


/* Synthesize and optimize the design               */

   compile -map_effort med
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Save the design                     */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */

/* Disconnect the RST signal                        */

   disconnect_net rst -all

/* Write the design report file                     */

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

/* Set the part type                                */

   set_attribute TOP "part" -type string "3020apc84-6"

/* Write out the design to a DB file                */

   write -format db -hierarchy -output TOP + ".db"

/* Save design in EDIF format as <design>.sedif     */

   write -format edif -hierarchy -output TOP + ".sedif"


/* Exit the Compiler.                               */

   exit
```

Figure 6-33 Global Reset Pin Script File (VHDL)

```
get_license {Designware-Basic}
/* ================================================*/
/* Sample Script for Synopsys to Xilinx Using      */
/*            the Design Compiler                  */
/* ================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Read in the design                  */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design    */

   TOP = greset

/* Set the Designer and Company name for
   documentation.                                  */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format verilog TOP + ".v"
   elaborate TOP

/* Set the current design to the top level         */

   current_design TOP

/* Add pads to the design. Make sure the current
   design is the top-level module.
   Insert pads on all ports except ´rst´.            */

   set_port_is_pad {CLK, ST}
   insert_pads

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Compile the design                */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.           */

   remove_constraint -all


/* Synthesize and optimize the design              */

   compile -map_effort med
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Save the design                        */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */

/* Disconnect the RST signal                          */

   disconnect_net RST -all

/* Write the design report file                       */

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

/* Set the part type                                  */

   set_attribute TOP "part" -type string "3020apc84-6"

/* Write out the design to a DB file                  */

   write -format db -hierarchy -output TOP + ".db"

/* Save design in EDIF format as <design>.sedif       */

   write -format edif -hierarchy -output TOP + ".sedif"


/* Exit the Compiler.                                 */

   exit
```

Figure 6-34 Global Reset Pin Script File (Verilog HDL)

Using the X-BLOX DesignWare Library

The XC4000 family DesignWare library describes adders, subtracters, comparators, incrementers, and decrementers that map to X-BLOX modules. Refer to “Getting Started” at the beginning of this user guide to ensure that you have X-BLOX installed on your system.

Warning: You can only use the X-BLOX DesignWare library with the version of the Synopsys compiler for which it was analyzed. See your system administrator or refer to the release notes for more information.

HDL Operators Using X-BLOX Modules

For XC4000 designs using the VHDL or Verilog arithmetic operators, Xilinx highly recommends that you use X-BLOX to take advantage of the X-BLOX DesignWare library. This DesignWare library contains the arithmetic functions that utilize the XC4000 dedicated carry logic to improve both the area and speed of the design.

The following is a list of the VHDL and Verilog arithmetic operators and the X-BLOX modules to which they map.

Table 6-10 Arithmetic Operators for X-BLOX Modules

Operators	X-BLOX Module
+	ADD_SUB
–	ADD_SUB
<, <=, >, >=	COMPARE
+ 1	INC_DEC
– 1	INC_DEC

X-BLOX is run on the output from the Synopsys-to-Xilinx translator, Syn2XNF. X-BLOX translates these modules into XNF primitives and performs the necessary optimization and implementation.

The X-BLOX DesignWare library contains twos complement and unsigned binary modules of width 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, and 48. Sixty-four-bits are available for the COMPARE module only. For operands falling between bit ranges, Synopsys maps them to the next higher bit-width module. X-BLOX removes any unused logic if implementing a smaller bit width. X-BLOX removes any unused logic if adding, subtracting, or comparing with a constant value.

Improving the Timing of X-BLOX Modules

The X-BLOX DesignWare modules contain path timing. The timing of the module depends on how many columns the module uses: the larger the device, the more CLBs per column. The fastest implementation of an X-BLOX module is implemented in the fewest columns. Table 6-11 shows the maximum bits that can be implemented in one column per device size.

To improve the timing of the X-BLOX module, choose a device type that requires the fewest columns. For example, if you wanted the fastest implementation of a 33-bit twos complement adder (without carry out), you should select a XC4008 or larger part type. Since the XC4008 can implement a 34-bit twos complement adder in one column, using a XC4008 or larger device gives you the fastest

implementation since the adder does not have to wrap into the next column.

In Table 6-11, replace _# with the number of bits, for example, add_sub_co_two_comp_14.

Table 6-11 Maximum Size of X-BLOX Module Before Wrapping

Device Type	4002	4003	4005	4006	4008	4010	4013
CLB Array Size	**8x8**	**10x10**	**12x12**	**16x16**	**18x18**	**20x20**	**24x24**
Add_Sub							
Twos Complement add_sub_two_comp_#	14	18	22	30	34	38	46
Unsigned Binary add_sub_ubin_#	14	18	22	30	34	38	46
Compare							
Greater Than or Equal To, Twos Complement comp_ge_two_comp_#	11	14	21	28	30	34	38
Greater Than or Equal To, Unsigned Binary comp_ge_ubin_#	13	17	21	29	33	37	45
Greater Than, Twos Complement comp_gt_two_comp_#	11	14	21	28	30	34	38
Greater Than, Unsigned Binary comp_gt_ubin_#	13	17	21	29	33	37	45
Less Than or Equal To, Twos Complement comp_le_two_comp_#	11	14	21	28	30	34	38
Less Than or Equal To, Unsigned Binary comp_le_ubin_#	13	17	21	29	33	37	45

Device Type	4002	4003	4005	4006	4008	4010	4013
CLB Array Size	**8x8**	**10x10**	**12x12**	**16x16**	**18x18**	**20x20**	**24x24**
Compare (Cont'd)							
Less Than, Twos Complement comp_lt_two_comp_#	11	14	21	28	30	34	38
Less Than, Unsigned Binary comp_lt_ubin_#	13	17	21	29	33	37	45
Not Equal, Twos Complement comp_ne_two_comp_#	32	38	48	64	N/A	N/A	N/A
Not Equal, Unsigned Binary comp_ne_ubin_#	32	38	48	64	N/A	N/A	N/A
Inc_Dec							
Twos Complement inc_dec_two_comp_#	16	20	24	32	36	38	46
Unsigned Binary inc_dec_ubin_#	16	20	24	32	36	38	46

Compiling the Design

Once you insert the I/O pads, you can optimize the design for area and/or speed. To get the most effective results from the Design Compiler, the constraints applied must be accurate and achievable. For example, if you set a timing goal of 0 ns on all ports, the Design Compiler attempts to meet this goal by duplicating logic to reduce critical paths, which can result in a significant and possibly unwarranted increase in CLB usage.

This following sections describe the commands you use to compile and optimize your HDL design using the Synopsys Compile command. Refer to the Synopsys documentation for more information on the Compile command.

Compiling a Design That Contains Feedthroughs

You must set the Compile Fix Mulltiple Port Nets command to True before you compile to prevent PPR from deleting logic if the design contains feedthroughs, or if the same net is connected to more than one port as follows.

```
compile_fix_multiple_port_nets = true
```

Compiling XC3000 and XC4000 Designs

Figure 6-35 illustrates a script file that demonstrates how to compile an XC3000 design.

```
/* ================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*           the Design Compiler                    */
/* ================================================*/

/* Define the set-up variables either in the script  */
/*        or in the .synopsys_dc.setup file          */

/* Set the search path in your script or in your     */
/* .synopsys_dc.setup file                           */
/* Replace <DS401-XACT-Dir> with the directory       */
/* path where the DS-401 was installed and replace   */
/* $SYNOPSYS with the Synopsys installation directory*/

/* search_path = {. \
   <DS401-XACT-Dir>/synopsys/libraries/syn \
   $SYNOPSYS/libraries/syn}                         */

/* Set the link, target and synthetic library variable
   either in the script or in the .synopsys_dc.setup
   Use synlibs to determine the link and target
   libraries                                         */

   link_library = {xprim_3020a-6.db xprim_3000a-6.db \
        xgen_3000.db xdc_3000a-6.db}

   target_library = {xprim_3020a-6.db xprim_3000a-6.db \
        xgen_3000.db xdc_3000a-6.db}

   symbol_library = xc3000.sdb

   define_design_lib WORK -path ./WORK

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Read in the design                     */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design     */

   TOP = <design_name>

/* Set the Designer and Company name for
   documentation.                                    */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                           */

   analyze -format vhdl TOP + ".vhd"
   elaborate TOP

/* Set the current design to the top level           */

   current_design TOP

/* Add pads to the design. Make sure the current
   design is the top-level module.                  */

   set_port_is_pad "*"
   insert_pads
```

```
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Compile the design                   */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.             */

   remove_constraint -all

/* If setting timing constraints, do it here.
   For example:                                      */

/*
   create_clock <clock_pad_name> -period 50
*/

/* Synthesize and optimize the design                */

   compile -map_effort med

/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Save the design                       */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                      */

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

/* Set the part type                                 */

   set_attribute TOP "part" -type string "3020apc84-6"

/* Write out the design to a DB file                 */

   write -format db -hierarchy -output TOP + ".db"

/* Save design in EDIF format as <design>.sedif      */

   write -format edif -hierarchy -output TOP + ".sedif"

/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Implement the Design                   */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Run xmake to process the design through the XACT  */
/* tools.                                            */

/*
   sh xmake TOP
*/

/* Exit the Compiler.                                */

   exit
```

Figure 6-35 Sample Script File for Compiling XC3000 Design

Compiling a XC4000H Design

Figure 6-36 and Figure 6-37 illustrate a script file that demonstrates how to compile an XC4000H design for VHDL and Verilog HDL, respectively.

The design is compiled before the pads are inserted to avoid pulling the flip-flops into the IOB. When compiling an XC4000H design, you must specify the input and output voltage levels. Refer to the "XC4000H IOBs" section for more information.

```
/* =================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*           the Design Compiler                    */
/* =================================================*/

/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*            Read in the design                    */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design     */

   TOP = three_ex2

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format vhdl TOP + ".vhd"
   elaborate TOP

/* Set the current design to the top level          */

   current_design TOP

/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Compile the design                  */
/* +++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all

/* Synthesize and optimize the design               */

   compile -map_effort med

/* Add the pads to the design. Make sure the current
   design is the top-level module. Change the default
   slewrate to CAP (HIGH slew control).
   The pads are inserted AFTER the design is compiled
   for XC4000H devices to prevent the registers from
   being implemented in the IOBs.                   */

   set_port_is_pad "*"
   set_pad_type -slewrate HIGH all_outputs()

/* FOR CMOS levels:                                */
   set_pad_type -vih 3.33 -vil 1.05 all_inputs()

/* FOR TTL levels:                                 */
   set_pad_type -voh 2.4 -vil 0.5  all_outputs()

insert_pads
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Save the design                       */
/* ++++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                       */

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

/* Set the part type                                  */

   set_attribute TOP "part" -type string "4005hpq240-5"

/* Write out the design to a DB file                  */

   write -format db -hierarchy -output TOP + ".db"

/* Save design in EDIF format as <design>.sedif       */

   write -format edif -hierarchy -output TOP + ".sedif"


/* Exit the Compiler.                                 */

   exit
```

Figure 6-36 Sample Script File for Compiling XC4000H Designs (VHDL)

```
/* ==================================================*/
/* Sample Script for Synopsys to Xilinx Using       */
/*            the Design Compiler                   */
/* ==================================================*/

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*             Read in the design                   */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the top-level modules name for the design    */

   TOP = three_ex2

/* Set the Designer and Company name for
   documentation.                                   */

   designer = "XSI Team"
   company  = "Xilinx, Inc"

/* Analyze and Elaborate  the design file and specify
   the design file format                          */

   analyze -format verilog TOP + ".v"
   elaborate TOP

/* Set the current design to the top level          */

   current_design TOP

/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*               Compile the design                 */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Set the synthesis design constraints.            */

   remove_constraint -all

/* Synthesize and optimize the design               */

   compile -map_effort med

/* Add the pads to the design. Make sure the current
   design is the top-level module. Change the default
   slewrate to CAP (HIGH slew control).
   The pads are inserted AFTER the design is compiled
   for XC4000H devices to prevent the registers from
   being implemented in the IOBs.                   */

   set_port_is_pad "*"
   set_pad_type -slewrate HIGH all_outputs()

/* FOR CMOS levels:                                */
   set_pad_type -vih 3.33 -vil 1.05 all_inputs()

/* FOR TTL levels:                                 */
   set_pad_type -voh 2.4 -vil 0.5  all_outputs()

insert_pads
```

```
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/*              Save the design                     */
/* ++++++++++++++++++++++++++++++++++++++++++++++++ */
/* Write the design report file                     */

   report_area > TOP + ".area"
   report_timing > TOP + ".timing"

/* Set the part type                                */

   set_attribute TOP "part" -type string "4005hpq240-5"

/* Write out the design to a DB file                */

   write -format db -hierarchy -output TOP + ".db"

/* Save design in EDIF format as <design>.sedif     */

   write -format edif -hierarchy -output TOP + ".sedif"


/* Exit the Compiler.                               */

   exit
```

Figure 6-37 Sample Script File for Compiling XC4000H Designs (Verilog HDL)

Creating the Area Report

The Design Compiler reports area with the Report Area command.

```
report_area
```

The Design Compiler reports area in three parts: combinatorial, non-combinatorial, and total. Synopsys reports the area as the number of CLBs used.

Each XC4000 CLB contains two 4-input function generators, one 3-input function generator, and two flip-flops. Each XC3000 and XC3100 CLB contains one 5-input function generator and two flip-flops. You can also configure the 5-input function generator as two 4-input function generators whose input pins are shared between the two function generators.

The reported combinatorial area is the maximum number of function generators required. The reported non-combinatorial area is the maximum number of flip-flops required. The total area reported is the sum of the combinatorial and non-combinatorial area. The number of CLBs required is usually less than the total area reported

because function generators and flip-flops can often share the same CLBs.

Generally, for the Xilinx mapped libraries, the minimum number of CLBs required is the larger of the combinatorial and non-combinatorial areas reported. The maximum number of CLBs required is the total area reported.

When running Syn2XNF, you can choose to map the combinatorial logic into function generators or leave the design unmapped, which allows the Xilinx placement and routing tools to perform the mapping as follows.

- Selecting the mapped option (–map) in Syn2XNF forces the combinatorial logic combinations to be retained, which makes the area report more accurate.
- Using the unmapped option in Syn2XNF, which is the default, allows PPR or APR to determine the mapping. This method is preferred because PPR and APR provide the most efficient mapping, placement, and routing for each design/device combination. Only PPR or APR can compute accurately the number of CLBs actually required.

Refer to the *XACT Reference Guide, Volume 2* for more information on PPR or APR.

Evaluating Timing Delays

The Synopsys tools report all delays in nanoseconds. The delays reported are pre-placement and routing estimates. You can obtain accurate timing only after running PPR. You can use either average or worst-case wire-load models. The Design Compiler cannot determine the actual wire-load delays until after the design is placed and routed.

To evaluate the timing results, use the Report Timing command as follows.

```
report_timing
```

Refer to the online *Synopsys Command Reference Manual* for information on other report options.

In most cases, flip-flops and combinatorial logic are combined into one CLB. In these cases, the Design Compiler adds an extra wire-load delay that you must delete manually from the delay path, as

illustrated by Figure 6-38. For an approximate estimate of the wire load, you can subtract the block delay from the path delay.

In some cases, multiple primitive functions may be combined into a single CLB, which causes the post-fanout timing results to be one or more block delays faster than the timing reported by the Design Compiler.

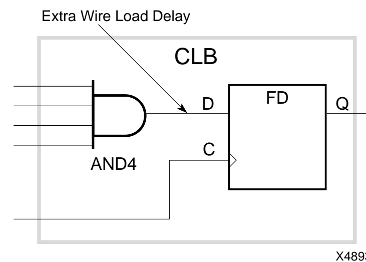


Figure 6-38 Combinatorial Logic Mapped with a Flip-Flop

Writing and Saving the Design

Once the design meets your timing and area requirements, you can save the design as a DB file, set the design part type, and save the file as an SEDIF file.

Before saving the design, set the appropriate variables to define the I/O pad locations, slew rates, and so on. Refer to the “Configuring the IOBs” section at the beginning of this chapter for more information.

Saving the DB File

To save the DB file, you can choose one of the following methods.

- Enter the following from the Design Analyzer menu.

  File → Save As
  File name: *design_name***.db**
  File Format: db
  Save all Designs in Hierarchy: on
  OK

- Type the following at the command line. (Make sure the top level of the design is selected.)

  write -format db -hierarchy -output *design_name***.db**

Setting the Design Part Type

Enter the Set Attribute command at the command line to select a specific part for the design. The following example uses a 4005pc84–5 device.

```
set_attribute design "part" -type string
"4005pc84-5"
```

Note: You can also specify the part type when running Syn2XNF or XMake.

Saving the SEDIF File

After compiling the design, save the design file as an SEDIF file, so you can translate it to an XNF format for use with the Xilinx tools.

You can save the design as an SEDIF file by either of the following methods.

- Select the design and then select the following from the Design Analyzer menu.

  File → Save As
  File name: *design_name***.sedif**
  File Format: edif
  Save all Designs in Hierarchy: on
  OK

- Enter the following at the command line. (Make sure the top level of the design is selected.)

  **write -format edif -hierarchy -output **
  *design_name***.sedif**

Translating SEDIF Files to XNF Files Using Syn2XNF

The Syn2XNF translator takes a Synopsys SEDIF file written by the Design Compiler and translates it to an XNF file.

How you run Syn2XNF is determined by your system configuration. Refer to the “Design Compiler Design Flow”section at the beginning of this chapter.

Syntax

To use Syn2XNF, enter the following on the command line.

syn2xnf [options] [*design***.sxnf.|** *design***.sedif.|** *design***.xnf]**

Specifying the file name extension is optional. By default Syn2XNF searches for a design file with an .sxnf or .sedif extension. If both exist, Syn2XNF uses the file with the latest time stamp.

You can run Syn2XNF from the UNIX prompt or from the Command window by using the shell command as follows.

sh syn2xnf *design*

In addition, you can run Syn2XNF automatically from XMake; however, you must have the XACT Development System installed on the same network as the XSI software. Refer to the “Before You Begin” section at the beginning of this chapter for more information.

Input Files

Syn2XNF accepts the following file types as input.

design.sxnf	This file is the synthesized design generated by the Synopsys synthesis tools.
design.xnf	This file represents the flattened, synthesized design in Xilinx Netlist Format.
design.sedif	This file is the synthesized design generated by the Synopsys synthesis tools using the EDIF syntax.

Syn2XNF is not case sensitive — you can enter the file name extension in upper- or lower-case letters. However, Xilinx recommends indicating the file name extension to distinguish the Design Compiler output from the Syn2XNF output.

Output Files

Syn2XNF creates three output files as follows.

design.xnf	This file represents the flattened, synthesized design in Xilinx Netlist Format.
design.xff	This file represents the flattened, synthesized design in Xilinx Netlist Format.
syn2xnf.log	This file contains error and warning messages that are also displayed onscreen.

Options

This section describes the Syn2XNF options. You can abbreviate all options using the first letter of the option; for example, you can indicate –parttype as –p.

–dir

The –dir option causes Syn2XNF to search *directory_name* for data files as well as the *DS401_dir/* data/synopsys directories and the current working directory.

syn2xnf –d *directory_name*

–force

The –force option forces Syn2XNF to overwrite an XNF file if one already exists.

```
syn2xnf –f
```

–help

The –help option displays onscreen the Syn2XNF help text.

```
syn2xnf –help
```

–l

The –l option lists onscreen all valid part types.

```
syn2xnf –l
```

–map

The –map option causes the Boolean functions, for example, fxx cells, from the xdc_*family–speedgrade*.db libraries to be mapped into function generators, allowing the number of cells reported in the Design Compiler to be more accurate.

```
syn2xnf –map design
```

If you do not specify this option, the Boolean functions are unmapped, which allows PPR or APR to map the design.

–out

The –out option specifies the output file name.

```
syn2xnf –o new_name design
```

By default Syn2XNF creates an XNF and XFF file with the same name as the input design file name. If you use this option, as illustrated by the following example, Syn2XNF reads the file design and outputs newdesign.xff and newdesign.xnf.

```
syn2xnf –o newdesign design
```

–parttype

The –parttype option specifies the Xilinx part and speed grade.

```
syn2xnf –p part–speedgrade design
```

If you specify no part type, Syn2XNF reads the part type from the SEDIF file. If no part type is specified in the SEDIF file, Syn2XNF uses the default part type, 4003APC84.

The following example illustrates how to specify the part type for an XC4005–5 device.

```
syn2xnf –p 4005apc84–5 design
```

To ensure that the XACT tools process your design properly, specify a part and speed grade.

–sub

The –sub option saves a hierarchical design as a separate XNF file.

```
syn2xnf –sub design
```

You must use this option on all subdesign SEDIF files, which causes Syn2XNF to create XNF files without I/O pads (EXT records) for each subdesign.

Running Syn2XNF on the top level automatically merges the submodule XNF files. The XSI tools do not have to create the submodules; for example, XNF files from MemGen can be merged in. You can merge any valid XNF file into the design. The XNF file must have the same name and I/Os as the “black box” or “instantiated component” that represents it in the Synopsys tools.

Using the XACT Development System

To translate the design to LCA and BIT files so the XACT tools can program the FPGA device, use the XMake program.

How you run XMake differs slightly depending on your system configuration. If the XACT Development System is installed on the same network as the Xilinx Synopsys Interface, XMake runs Syn2XNF. If XSI is installed on a machine that cannot access the XACT Development System, run Syn2XNF on the machine with the XSI software and copy the appropriate design files to the machine with the XACT Development System. Refer to the“Design Compiler Design Flow” chapter at the beginning of this user guide for more information.

XMake automatically translates the X-BLOX modules into gates by running X-BLOX, and maps, places, and routes the design using PPR.

If XSI Is on Same Platform as XACT Software

If XSI is installed on the same platform as the XACT software, invoke XMake.

xmake *design*

XMake runs Syn2XNF and the XACT software tools.

If XSI Is on Different Platform Than XACT Software

If XSI is installed on a different platform than the XACT software, do the following to run the XACT tools.

1. Copy *both* the XNF and XFF files to the platform where the XACT tools reside.

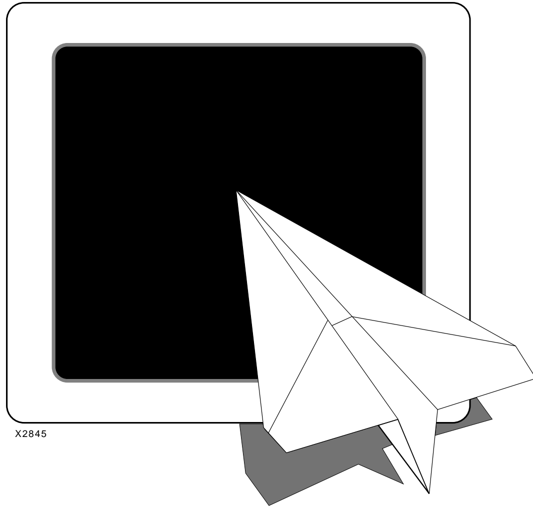
Note: Use the Copy command with the –p option to preserve the files' time stamp.

2. Run XMake with the following option.

   xmake –x *design*

The –x option causes XMake to search for an XNF file and any other files not already merged. (XMake reruns the XNFMerge program.)

For more information on XMake or any of the programs it invokes, refer to the *XACT Reference Guide*.



Xilinx Synopsys Interface FPGA User Guide

Simulating Your FPGA Design

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Chapter 7

Simulating Your FPGA Design

XSI supports both functional and timing simulation. This chapter shows you how to prepare XC3000 and XC4000 FPGA designs for simulation and how to use a test bench.

Recommended FPGA Simulation Strategy

Because of the flexibility of the simulation environment, you can verify your design using various methods. The following steps, which are explained in subsequent sections, show you one recommended flow for FPGA simulation.

- Edit your .synopsys_vss.setup file — Before you can begin simulation, you must edit your simulation setup file.
- Check the source file — If you use the same test bench file for both functional and timing simulation, you need library definitions for the appropriate Xilinx FTGS models in your original top-level VHDL source file.
- Specify the initial states of your registers — If you use attributes to control the initial states of the registers in the actual design implementation, you must also specify those initial states in your source design file for functional simulation.
- Create a test bench file — By following the guidelines described in this section, you can use the same test bench for both functional and timing simulation.
- Perform functional simulation — This step allows you to debug the logic in your source design before implementing an FPGA.
- Implement the design in an FPGA — This step provides the necessary physical resource information necessary for timing simulation.

- Prepare the timing model — The XNF2VSS program prepares the timing model of your design for simulation and provides a static timing report.
- Perform timing simulation — By reusing the functional simulation test bench file, you can easily compare results and prevent errors caused by accidental differences between separate test bench files.

Editing the VSS Setup File

To properly analyze and simulate Xilinx XC3000 and XC4000 designs using VSS, you must edit your Synopsys VSS setup file, .synopsys_vss.setup.

You can either use the standard VSS setup file that is included with the Synopsys software, or you can create one for yourself. In either case, the following items must be included.

- Timebase and resolution factors — You must set the simulator timebase to nanoseconds and the timebase resolution to 0.1 ns. Use the following two commands.

  ```
  TIMEBASE = NS
  TIME_RES_FACTOR = 0.1
  ```

- Hazard messages — Because FPGAs contain signal paths with varying delays, outputs may switch through several states before settling. To avoid generating hazard warnings on each short output transition, turn hazard messages off with the following command. The system still reports setup, hold, and pulse-width violations.

  ```
  NO_HAZARD_MESG = TRUE
  ```

- Library paths to FTGS models — For VSS to find the Xilinx FTGS models, create the following library path definitions.

  ```
  XC4000 : $DS401/synopsys/libraries/vss/lib/xc4000
  XC3000 : $DS401/synopsys/libraries/vss/lib/xc3000
  XC7000 : $DS401/synopsys/libraries/dw/lib/epld
  ```

- Work library definition — You must select a working library for the VHDL analyzer (vhdlan). The standard Synopsys VSS setup file maps the WORK library to the current directory. In the Xilinx

tutorial example design, count8, the WORK library is mapped to the subdirectory ./WORK.

Check Your Source File

If you use the same test bench file for both functional and timing simulation, you need library definitions for the appropriate Xilinx FTGS models in your original top-level VHDL source file. For example, for a XC4000 design, you would use the following.

```
library XC4000;
use XC4000.components.all;
```

These statements allow the VHDL analyzer to link the timing architecture you produce later to your original entity definition. If these definitions are not included, the system generates several error messages, including those regarding unbound components.

Controlling Initial States of Registers

This section shows you how to declare the initial states of registers in your design for simulation. If your design does not depend on the initial states of any registers, you can skip this section and go to the “Creating a Test Bench File” section.

The initial state attributes specified in DC Shell during compilation or the default initial states specified for each registered cell in the Xilinx component library determines the registers’ actual initial states.

The timing simulation model produced by the Xilinx software reflects the actual register initial states that are implemented in the device, regardless of whether they are explicitly specified or automatically assigned by the XACT software.

Simulating Global Set/Reset

Xilinx FPGAs have a Global Set/Reset (GSR) function that initializes the device registers either when power is applied or when the GSR input pin is pulsed; however, the GSR Reset signal is not available to connect to any other logic in the device. You must pulse the GSR Reset signal at the beginning of timing simulation for proper register initialization.

The following sections show you how to simulate the GSR function for both functional and timing simulation.

Preparing for Timing Simulation

When you generate your timing simulation model, XNF2VSS automatically creates a new input port named GSR. When simulating, you must first pulse GSR High prior to exercising the logic to get all the registers into their initial states. If you use a test bench to simulate your design, include the GSR signal as one of the input ports of the FPGA in the test bench as described in the "Creating a Test Bench File" section.

You can use the GSR signal for timing simulation only; you cannot use it for functional simulation or in your design. However, if you include it in your functional simulation test bench, you can use that test bench later for timing simulation without modification.

If you are using the same test bench file for both functional and timing simulation, you must include the GSR port declaration in your source design file as follows.

```
port (... GSR : in std_logic ...);
```

Because the GSR signal is not used anywhere else in your design, you receive warning messages about the unconnected GSR port during the Compile and Insert Pads operations, which do not cause any problems during synthesis. The XACT software ignores the unconnected GSR port during implementation.

See the source file listing in Figure 7-1 for an example of how the GSR input port is declared in a VHDL design.

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
library XC4000;
use XC4000.components.all;

entity count8 is
        port (GSR : in std_logic;
              CLOCK, CLEAR, ENABLE: in std_logic;
              COUT: out std_logic_vector (7 downto 0) );
end count8;

architecture BEHAVIORAL of count8 is
signal QOUT: std_logic_vector (7 downto 0) := "00000000";
begin
    process (CLEAR, CLOCK, ENABLE)
        begin
        if (CLEAR = '1') then
            QOUT <= "00000000";
        elsif (CLOCK'event and CLOCK='1') then
            if (ENABLE = '1') then
                QOUT <= QOUT + "00000001";
            end if;
        end if;
    end process;
COUT <= QOUT;
end BEHAVIORAL;

configuration conf_cnt8 of count8 is
for BEHAVIORAL
end for;
end conf_cnt8;
```

Figure 7-1 Count8 Example VHDL Source File

Preparing for Functional Simulation

Simulate register initialization (GSR) by defining the initial values for registered signals in your source design file. Use signal declarations such as the following.

> **port** *signal_name*: *port_direction signal_type* **:=** *initial_value*;
> **signal** *signal_name*: *signal_type* **:=** *initial_value*;
> **variable** *signal_name*: *signal_type* **:=** *initial_value*;

For example, you could enter the following.

```
port Nreg5 out std_logic := '0';
signal Qreg6: std_logic := '0';
variable Qreg: std_logic_vector := "00000001";
```

Use these initial values for functional simulation only, not during synthesis. The synthesizer sends a warning that these values are

being ignored. Also, the XACT software does not use these initial values for device implementation.

Note: You should assign initial states for simulation based on your behavioral coding. The synthesis tools select either Preset or Reset flip-flops based on your behavioral code, not on the initial state definitions. See the “Using the FPGA Compiler” or “Using the Design Compiler” chapter for detailed information on Preset and Direct Clear.

Creating a Test Bench File

This section shows you how to create a test bench file that you can use for both functional and timing simulation. The example test bench consists of a VHDL file containing one instance of an FPGA design being tested and a procedure that applies simulation input waveforms to the FPGA.

Initializing Registers

For functional simulation, all registers are initialized before the first simulation cycle (at time zero) by the initial values declared in your source design file.

For timing simulation in the test bench, include the GSR input port in the FPGA component declaration and in its instance port map as shown in Figure 7-2. At the beginning of the simulation sequence, applying an active-High pulse to GSR initializes the registers. The VSS simulator ignores the pulse during functional simulation because the GSR signal is not used anywhere in the source design.

```
-- VHDL Model Created from count8.xnf -- Mon May  9 15:59:34 1994

library IEEE;
library Synopsys;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_unsigned.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_components.all;
   use synopsys.attributes.all;
   use STD.Textio.all;

entity count8_tb is
end count8_tb;

architecture TEST of count8_tb is

   component count8
      Port(
         GSR    : in    std_logic;
         CLOCK  : in    std_logic;
         CLEAR  : in    std_logic;
         ENABLE : in    std_logic;
         COUT   : out   std_logic_vector (7 downto 0));
   end component;

   signal GSR, ENABLE, CLOCK, CLEAR : std_logic;
   signal COUT : std_logic_vector (7 downto 0);
signal CHECK_COUNT : std_logic_vector (7 downto 0);

begin

   UUT : count8
        Port Map (GSR, CLOCK, CLEAR, ENABLE, COUT);

DRIVER: process
begin
        CHECK_COUNT <= "00000000";

        GSR <= ´0´;
        CLEAR <= ´0´;
        ENABLE <= ´1´;
        CLOCK <= ´0´;
        wait for 50 ns;
        GSR <= ´1´;
        wait for 100 ns;
        GSR <= ´0´;
        wait for 100 ns;

        CLEAR <= ´1´;
        wait for 100 ns;
        CLEAR <= ´0´;
        wait for 50 ns;

        CLOCK <= ´1´;
        CHECK_COUNT <= CHECK_COUNT + conv_std_logic_vector(1,8);
        wait for 50 ns;
        assert COUT = CHECK_COUNT
          report "Counter output does not match" severity error;
```

```
        for I in 1 to 20 loop
                CLOCK <= ´0´;
                wait for 50 ns;
                CLOCK <= ´1´;
                CHECK_COUNT <= CHECK_COUNT + "00000001";
                wait for 50 ns;
                assert COUT = CHECK_COUNT
                    report "Counter output does not match" severity error;
        end loop;

        ENABLE <= ´0´;
        CLOCK <= ´0´;
        wait for 50 ns;
        CLOCK <= ´1´;
        wait for 50 ns;
        assert COUT = CHECK_COUNT
          report "Counter output does not match" severity error;

        CLOCK <= ´0´;
        CLEAR <= ´1´;
        CHECK_COUNT <= "00000000";
        wait for 50 ns;
        assert COUT = CHECK_COUNT
          report "Counter output does not match" severity error;
        CLEAR <= ´0´;
        wait for 100 ns;

        wait;
end process;

end TEST;

configuration CFG_count8_tb of count8_tb is
for TEST
end for;
end CFG_count8_tb;
```

Figure 7-2 Count8_tb.vhd

XNF2VSS automatically generates the GSR port in the timing simulation model. During timing simulation when the test bench applies the GSR pulse, the timing simulation model initializes all registers as they are actually implemented in the FPGA.

Configuration Declaration

For any design or test bench you wish to simulate, you must declare a configuration that identifies the specific architecture you are applying to a design. When you invoke the VSS simulator, you must select the name of a configuration that has been previously analyzed.

Figure 7-2 shows a typical configuration declaration in a test bench file. If you always use the test bench to simulate the design source file, it does not need its own configuration declaration.

After you have created a test bench file, you are ready to begin using the VSS simulator for functional simulation.

Functional Simulation

Functional simulation is used to debug your logic before fitting your design into an FPGA.

To prepare a test bench configuration for simulation, you must analyze each design and test bench source file in the proper bottom-up sequence.

The following procedure uses the stand-alone Synopsys VHDL Analyzer (Vhdlan) and the Synopsys VHDL Debugger (Vhdldbx).

Note: Use the second syntax option in the following steps if you are using the count8 design to practice functional simulation.

1. Analyze your source FPGA design file.

   vhdlan *design_name***.vhd**

   If using the count8 design, enter the following.

   ```
   vhdlan count8.vhd
   ```

2. Analyze the test bench file.

   vhdlan *test_bench_name***.vhd**

   If using the count8 design, enter the following.

   ```
   vhdlan count8_tb.vhd
   ```

3. Invoke the VHDL debugger as follows.

   vhdldbx

   The system prompts you for a configuration name. The Vhdldbx selector window appears as shown in Figure 7-3.

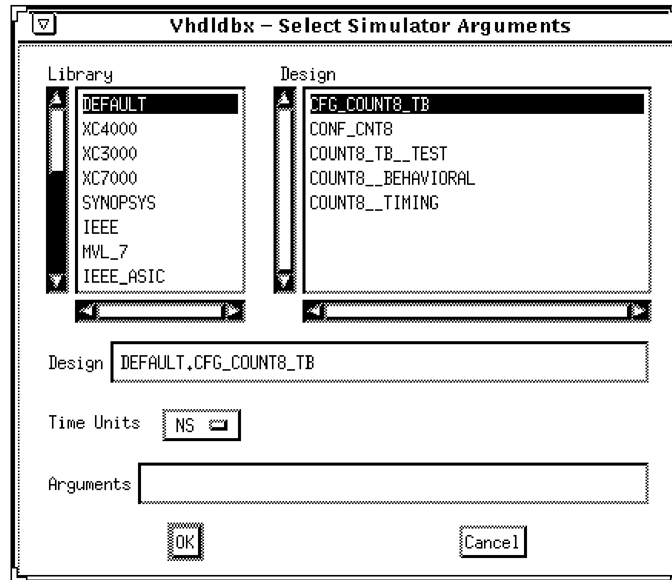


Figure 7-3 Vhdldbx Select Simulator Arguments Window

4. Select the name of the configuration declared in the *test_bench_name*.vhd file. For the count8 design, select the following.

   `CFG_COUNT8_TB`

5. Click **OK**.

   The Vhdldbx user interface window appears as shown in Figure 7-4.

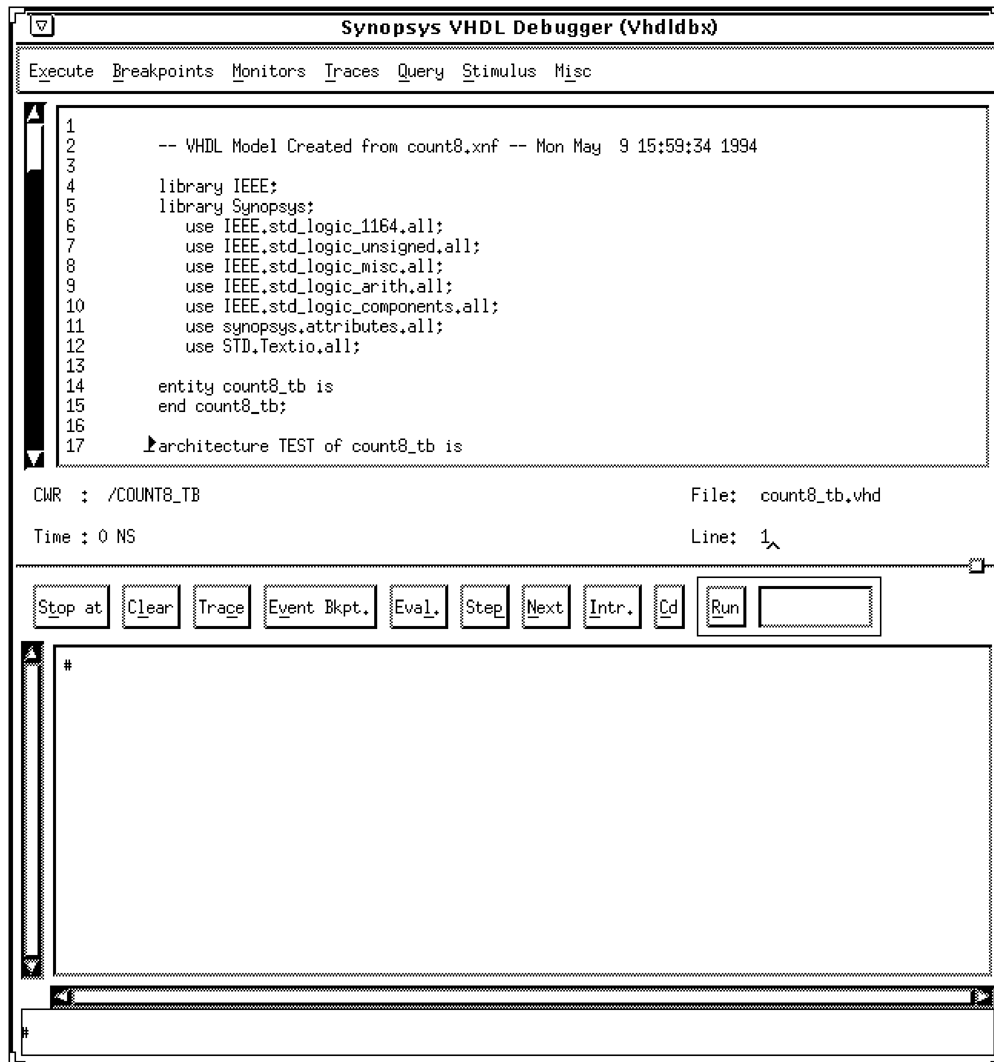


Figure 7-4 Vhdldbx User Interface Window

6. To run your simulation, declare the signals you want to display in a trace window. For example, to display all signals appearing on the FPGA pins, you can enter the following Vhdldbx command.

   ```
   trace *'signal
   ```

7. To run all the simulation vectors in your test bench, select the Run button in the VHDL Debugger dialog box or enter **run** at the command-line prompt.

   The system displays the functional simulation waveforms in the Dynamic Waveform Viewer as illustrated in Figure 7-5.

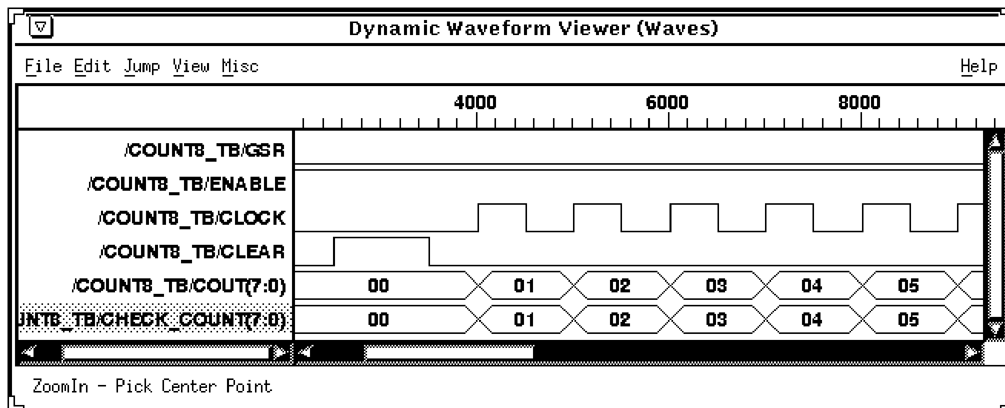


Figure 7-5 Functional Simulation Waveforms for Count8 Design

8. After functional simulation is successful, you can exit the VHDL Debugger window.

You are now ready to implement your design and create the physical layout information required for timing simulation as described in the "Design Implementation" section, which follows.

Design Implementation

After you have debugged your design using functional simulation, you can compile it using synthesis and implement it in an FPGA using the Xilinx XACT software. Design implementation is a prerequisite for performing timing simulation.

You can use DC shell commands as described in the “Using the FPGA Compiler” or “Using the Design Compiler” chapter, or the Synopsys graphical interface (Design Analyzer) to create the XNF or EDIF netlist file required by the XACT software. This gate-level netlist file contains cells from the XC3000 or XC4000 library but not timing information. The XACT software processes the netlist file and places the logical design into the physical architecture of a target FPGA.

After the design is implemented by the XACT software, the actual target device timing information is available for timing simulation.

Using the count8 design as an example, the following steps show you an overview of the FPGA implementation procedure described in the “Using the FPGA Compiler” or “Using the Design Compiler” chapter.

1. Compile the design, targeting the appropriate libraries, and create an XNF or EDIF netlist by executing the following command at the command line.

   ```
   dc_shell –f count8.script
   ```

   During processing, the system displays informational messages onscreen.

Note: The results of the Vhdlan command cannot be used for synthesis.

2. Run the XACT software, using the XMake command to process the netlist.

   ```
   xmake count8
   ```

3. Create a post-layout XNF file with timing information using the LCA2XNF command.

   ```
   lca2xnf –w count8
   ```

   The –w option suppresses a warning from LCA2XNF that indicates it is overwriting the XNF file produced by Syn2XNF. The original synthesis XNF file still exists with an .xff extension.

You can now prepare a gate-level VHDL model for timing simulation.

Preparing the Timing Model

When you synthesize your design and create an XNF or EDIF netlist file for the XACT software, all buses (such as those declared as std_logic_vector) are decomposed into individual nets. The original definition of your bus ports in the design entity are not retained through the place and route process.

The XNF2VSS software cannot regenerate a timing model complete with your original bus port declarations, but it does provide two options for preparing the timing model.

- Using XNF2VSS without any options generates the timing model as an architecture only, without the entity.

  xnf2vss *design_name*

  If using the count8 design, enter the following.

  ```
  xnf2vss count8
  ```

  The external signals appearing in the design that were originally defined as bus ports are represented within the model architecture using subscript notation compatible with bus port declarations. By reusing the entity from your source design with the architecture of the timing model, you can perform timing simulation using the same test bench and chip interface as used for functional simulation.

- Using XNF2VSS with the –t option generates the timing model as a complete VHDL design and a test bench template.

  xnf2vss –t *design_name*

  This optional mode produces a test bench template and a new gate-level VHDL file that contains both entity and architecture definitions, although it does not preserve original bus notation. You determine the stimulus input for the test bench.

When the static timing results are satisfactory, as reported by the XDelay program, you can proceed to timing simulation.

Timing Simulation

Perform timing simulation after implementing your design, creating the timing model, and reviewing the static timing report.

Once you prepare your test bench and run XNF2VSS, you can use the same test bench for timing simulation as used for functional simulation. By using the same test bench, you can easily verify that the functionality of the device after mapping matches the functionality of your source design. You also eliminate any risk of errors from accidental differences between the test bench files.

1. Analyze your source design file to reuse the port declarations in its entity as follows.

   vhdlan ***design_name*****.vhd**

   `vhdlan count8`

2. Replace the architecture of your source design with the timing architecture produced by XNF2VSS.

   vhdlan ***design_name*****_vss.vhd**

   `vhdlan count8_vss.vhd`

   The architecture is replaced in the Synopsys database by analyzing the timing model file; you do not need to modify your design source file.

3. Analyze the test bench file name as used for functional simulation as follows.

   vhdlan ***test_bench_name*****.vhd**

   `vhdlan count8_tb.vhd`

   The simulation database now contains the test bench design that interfaces to the chip through your source design entity read in step 1 but it contains the timing model architecture read in step 2.

4. Invoke the Synopsys VSS Simulator.

   vhdldbx

5. Indicate the configuration named in the *test_bench_name*.vhd file. For example, for the count8 design, select the following.

   `CFG_COUNT8_TB`

Warning: Before clicking **OK**, you must specify the timing back-annotation file information in the Vhdldbx — Select Simulator Arguments dialog box.

All back-annotated timing in the standard delay format (SDF) file is applied to various instances within the *design_name*_vss.vhd file. However, if you are simulating with a test bench, you must specify to the simulator the FPGA design instance to which you want to apply the back-annotated timing. The simulator can then find all the referenced instances.

6. In the Arguments field, indicate the file name of the SDF back-annotation timing file.

   -sdf *design_name*_**vss.sdf**

   For the count8 example, enter the following.

   ```
   -sdf count8_vss.sdf
   ```

7. Specify the sdf_top instance in the test bench configuration to which the back-annotated timing is applied.

   -sdf_top *chip_instance_name*

   For the count8 example, enter the following.

   ```
   -sdf_top /count8_tb/UUT
   ```

   All back-annotated timing parameters in the SDF file are applied to the selected chip instance. Figure 7-6 illustrates the Select Simulator Arguments dialog box with the back-annotation timing parameters specified in the Arguments field.

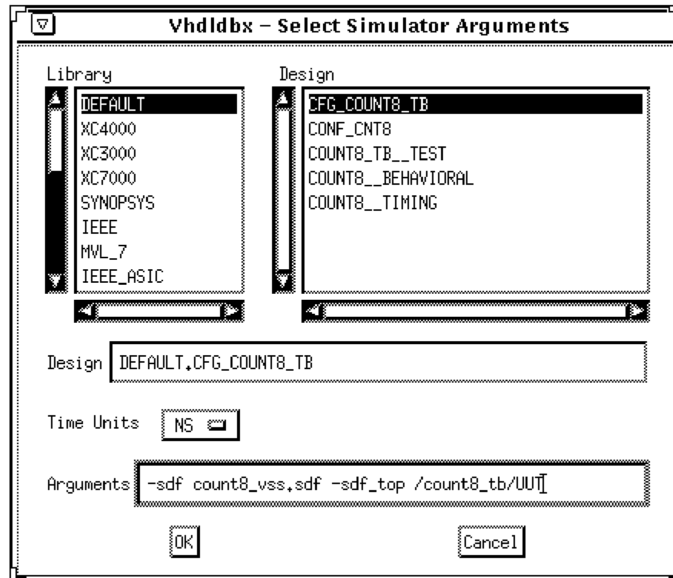


Figure 7-6 Vhdldbx Selector Window with Timing Back-Annotation Parameters

8. Click **OK**.

Note: You can also enter the back-annotation parameters on the command line as follows.

vhdldbx -sdf *design_name*_**vss.sdf -sdf_top **
/*design_tb***.vhd/***chip_instance_name configuration_name*

For the count8 design example, you should enter the following.

```
vhdldbx -sdf count8_vss.sdf -sdf_top \
/count8_tb/UUT CFG_COUNT8_TB
```

For convenience, you can put all parameters into a script file. The command line for the count8 design is provided in the dbx_count8 script file in the VSS examples directory.

Now you can run the same simulation vectors for timing simulation as you ran for functional simulation. However, in timing simulation, the registers are set to their initial states in response to the active-High pulse on the GSR input.

To set up a trace window for all FPGA ports and run the simulator, use the Trace and Run commands, as illustrated in the “Functional Simulation” section. After executing these commands, you can view the waveforms, as illustrated by Figure 7-7.

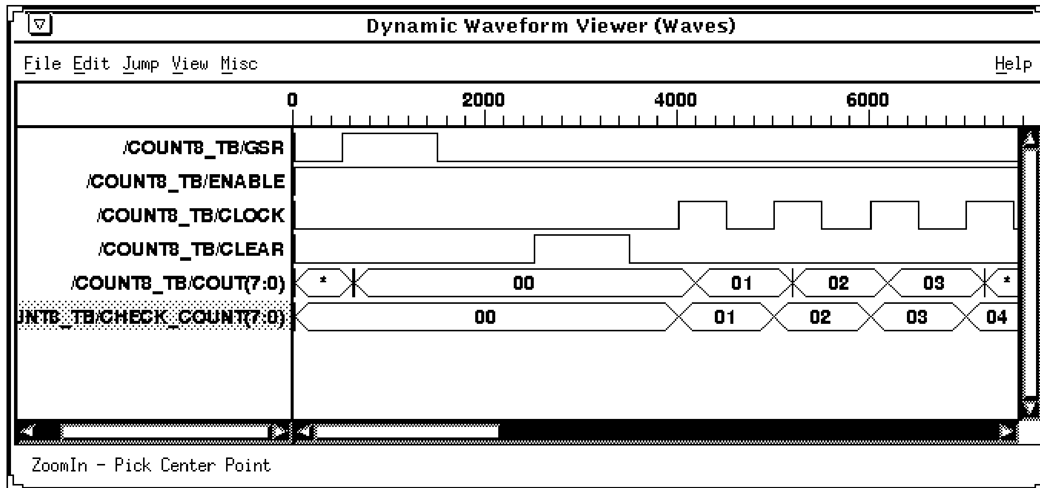
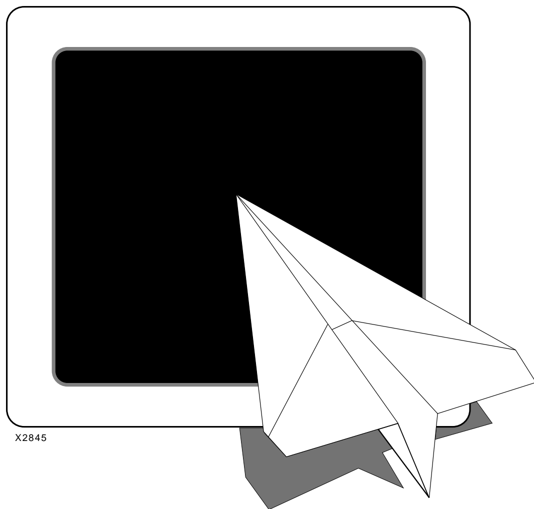


Figure 7-7 Timing Simulation Waveforms for the COUNT8 Design



Xilinx Synopsys Interface FPGA User Guide

Files, Programs, and Libraries

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Chapter 8

Files, Programs, and Libraries

This chapter describes the files, programs, and Xilinx-supplied libraries you need to translate your HDL design using the Synopsys FPGA Compiler or Design Compiler.

Directory Structure for XSI

This section describes the XSI directory tree, so you can easily find XSI files, programs, and libraries.

Figure 8-1 illustrates the XSI directory tree. The directory tree contains the following variables.

DS401_dir	The directory where XSI is installed.
platform	The *platform* can be one of the following: sparc, apollo, hppa, or rs6000.
family	The *family* refers to the family of Xilinx devices, for example, 4000, 3000, 3000a, 3000l, 3100, or 3100a.
parttype	The *parttype* is the specific Xilinx device, for example, 4003, 4005, or 3020.

```
DS401_dir/
   bin/platform/
       synlibs
       syn2xnf
       sedif2xnf
       speedcheck
       xnfmerge
   data/synopsys/
       parttype.spd  <~44 .spd files>
       partlist.xct
       text.spd
       xmap_3000/ <~250 .xnf files>
       xmap_4000/ <~250 .xnf files>
       xprim_3000/ <~60 .xnf files>
       xprim_4000/ <~160 .xnf files>
       xunmap_3000/ <~250 .xnf files>
       xunmap_4000/ <~250 .xnf files>
   synopsys/
       libraries/dw/lib/fpga
           <xblox_dw_modules>.o
           <xblox_dw_modules>.syn
           <xblox_dw_modules>.sim
           <xblox_dw_modules>.mra
       libraries/dw/src/fpga
           README
           install_dw.dc
           <xblox_dw_modules>.vhd.e
           <xblox_dw_modules>.vhd.e.update
       libraries/syn/
           xgen_family.db
           xfpga_family-speedgrade.db
           xprim_parttype-speedgrade.db
           xprim_family-speedgrade.db
           xio_parttype-speedgrade.db
           xdc_family-speedgrade.db
           xc3000.sdb
           xc4000.sdb
           xblox_4000.sldb
       libraries/sim/src/
           xc4000
               README
               install_xc4000.dc
               xc4000_FTGS.vhd.e
               xc4000_FTGS.vhd
           xc3000
               README
               install_xc3000.dc
               xc3000_FTGS.vhd.e
               xc3000_FTGS.vhd
```

```
    libraries/sim/lib/
        xc4000
            <vss4k_FTGS>.syn
            <vss4k_FTGS>.sim
            <vss4k_FTGS>.mra
        xc3000
            <vss3k_FTGS>.syn
            <vss3k_FTGS>.sim
            <vss3k_FTGS>.mra
tutorial/synopsys/
    fpga/x4000/
        vhd
        verilog
    dc/x3000a/
        vhd
        verilog
    vss/xc4000
examples/synopsys/
    fc4k.synopsys_dc.setup
    dc4k.synopsys_dc.setup
    dc3k.synopsys_dc.setup
    fpga.script
    dc.script
    fpga/
        xc4000/
            vhd/<design-directory>
            verilog/<design-directory>
        xc4000a/
            vhd/<design-directory>
            verilog/<design-directory>
        xc4000h/
            vhd/<design-directory>
            verilog/<design-directory>
    dc/
        xc3000/
            vhd/<design-directory>
            verilog/<design-directory>
        xc4000/
            vhd/<design-directory>
            verilog/<design-directory>
        xc4000a/
            vhd/<design-directory>
            verilog/<design-directory>
        xc4000h/
            vhd/<design-directory>
            verilog/<design-directory>
```

Figure 8-1 XSI Directory Tree Structure

File Descriptions

This section describes the files you need to translate, map, place, and route your design using the XSI and Synopsys tools.

Table 8-1 File Descriptions

File	Description	FPGA Compiler or Design Compiler
design.script	The *design*.script file is user-created and contains the commands for the Synopsys FPGA Compiler or Design Compiler. These commands specify the operating conditions, the name and format of the design file, and synthesis directives. **Note:** Script files can have extensions other than .script.	Both
design.v	The .v extension indicates the Verilog HDL format.	Both
design.vhd	The .vhd extension indicates the VHDL format.	Both
.synopsys_dc.setup	The .synopsys_dc.setup file is the startup file for the Synopsys synthesis tools. It must be located in your home directory or working directory.	Both
XC4000.sdb	The XC4000.sdb file contains XC4000 schematic symbols for Synopsys.	Both
XC3000.sdb	The XC3000.sdb file contains XC3000/A/L and XC3100/A schematic symbols for Synopsys.	Both
design.sxnf	The *design*.sxnf file is the synthesized design generated by the Synopsys synthesis tools, which becomes the input to the Syn2XNF program.	FPGA Compiler

File	Description	FPGA Compiler or Design Compiler
design.sedif	The *design*.sedif file is the synthesized design generated by the Synopsys synthesis tools using the EDIF syntax. This file is the input to the Syn2XNF program.	Design Compiler
design.xff	The *design*.xff file, generated by the Syn2XNF program, represents the flattened, synthesized design in Xilinx Netlist Format (XNF).	Both
design.xnf	The design.xnf file, generated by the Syn2XNF program, represents the flattened, synthesized design in Xilinx Netlist Format (XNF).	Both
design.lca	The LCA file, generated by PPR, is used to configure the specified FPGA.	Both
.syn	SYN files define synthetic library elements for the Synopsys DesignWare software. These files only support XC4000 devices.	Both
.sim	SIM files are used for VSS simulation.	Both
.mra	MRA files are used for VSS simulation.	Both
*design*_vss.vhd	This file is the VHDL timing simulation model created by the VMH2VSS program.	Both
*design*_vss.sdf	This file is the timing back-annotation file created by the VMH2VSS program.	Both

Program Descriptions

This section describes the programs you use when translating, mapping, placing, and routing your design using the XSI and Synopsys tools. You can use the following programs with both the Design Compiler and FPGA Compiler.

Table 8-2 Program Descriptions

Program	Description
Design Analyzer	The Design Analyzer is the Synopsys graphic interface to the Synopsys synthesis tools.
DC Shell	The DC Shell is the Synopsys UNIX command-line interface for entering commands, arguments, and options to the Synopsys synthesis tools.
Synlibs	This program displays onscreen the target and link libraries for the specified part type and speed grade. You can append the output of the Synlibs command to the .synopsys_dc.setup file. **Warning:** You must list the libraries in your setup file in the order that they appear in the Synlibs output.
SYN2XNF	This program translates a Synopsys SXNF or SEDIF file into XNF and XFF files.
XMake	XMake supports the automatic translation of design files. XMake invokes the necessary tools to flatten the XNF file and map, place, and route the design for the specified Xilinx FPGA device.
Vhdlan	The Vhdlan program analyzes a VHD source file for simulation.
Vhdldbx	The Vhdldbx program is the VHDL Debugger, a graphical interface to the VHDL simulator. Through its dialog box, you can issue simulator commands, view command output, and view source code.
XNF2VSS	The XNF2VSS program creates a timing model for timing simulation.

Library Descriptions

This section describes the Xilinx-supplied libraries and supported part types and speed grades. Table 8-3 contains the following variables.

family — The *family* refers to the family of Xilinx devices, for example, 4000, 3000, 3000a, 3000l, 3100, or 3100a.

parttype — The *parttype* is the specific Xilinx device, for example, 4003, 4005, or 3020.

4kparttype — The *4kparttype* is the specific Xilinx XC4000 device, for example, 4003, 4005, or 4000a.

–s — The *–s* indicates the part type's speed grade, for example, –5. Not all speed grades are available for all part types. Run Synlibs with the –h option to get a listing of all available part type/speed grade combinations.

Table 8-3 Library Descriptions

Library	Description	FPGA Compiler or Design Compiler
xgen_4000.db	The xgen_4000.db library describes the XC4000 cells that do not contain timing information, for example, FMAP, PULLUP, and VCC.	Both
xgen_3000.db	The xgen_3000.db library describes the XC3000 and XC3100 cells that do not contain timing information, for example, CLBMAP, PULLUP, net flags, and VCC.	Both
xprim_*family-s*.db	The xprim_*family*-s.db libraries describe the Xilinx XC4000/XC3100/XC3000 gates, flip-flops, input/output buffers, and other simple circuit elements whose delays do not vary with the density of the part. These files contain worst-case commercial (WCCOM) timing information.	Both
xprim_*parttype–s*.db	The xprim_*parttype–s*.db libraries describe the Xilinx XC4000/XC3100/XC3000 3-state buffers, clock buffers, I/O decoders, and other simple circuit elements whose delays vary with the density of the part. These files contain WCCOM timing information.	Both

Library	Description	FPGA Compiler or Design Compiler
xio_*4kparttype–s*.db	The xio_*4kparttype–s*.db libraries describe the Xilinx XC4000/A/H input/output buffers whose delays vary with the device type. XC4000D devices use the XC4000 library. These files contain WCCOM timing information.	Both
xfpga_*family–s*.db	The xfpga_*family–s*.db libraries describe the Xilinx XC4000 CLB and IOB primitives, which allow the FPGA Compiler to directly map to CLBs and IOBs. These files contain WCCOM timing information.	FPGA Compiler
xdc_*family–s*.db	The xdc_*family–s*.db libraries contain Boolean functions to which the Synopsys tools map.	Design Compiler
xblox_4000.sldb	The xblox_4000.sldb library contains the DesignWare macros that allow adders, subtracters, incrementers, decrementers, and comparators to map directly to X-BLOX modules.	Both
xprim_*family/**.xnf	The xprim_*family/* directory contains the XNF files for the following Xilinx primitives: xgen_*family–s*.db, xprim_*family-s*.db, and xprim_*device*.db.	Both
xmap_*family/**.xnf	The xmap_*family/* directory contains the XNF files for the xdc_*family*-s.db Boolean functions that are already mapped to CLB function generators.	Design Compiler
xunmap_*family/* *.xnf	The xunmap_*family/* directory contains the XNF files for the xdc_*family–s*.db Boolean functions that are not mapped into CLB function generators until APR or PPR is run.	Design Compiler

Supported Part Types and Speed Grades

This section describes the supported part types and speed grades for the Xilinx-supplied libraries described in Table 8-3. The following table lists the speed grades available for each Xilinx family.

Table 8-4 Available Speed Grades

Family	Available Speed Grades
XC4000	–4, –5, –6, or –10
XC3000	–50, –70, –100, or –125
XC3000A	–6 or –7
XC3000L	–8
XC3100	–3, –4, or –5

xprim_*family*–s.db and xprim_*parttype–s*.db

These libraries contain cells specific to Xilinx FPGAs, including flip-flops, 3-state buffers, and other performance-improving features. The following tables summarize the Xilinx-specific primitive libraries.

Table 8-5 XC4000 Primitive Libraries

–4 Speed Grade	–5 Speed Grade	–6 Speed Grade	–10 Speed Grade
xprim_4000–4	xprim_4000–5	xprim_4000–6	xprim_4000–10
	xprim_4002–5	xprim_4002–6	
xprim_4003–4	xprim_4003–5	xprim_4003–6	
xprim_4004–4	xprim_4004–5	xprim_4004–6	
xprim_4005–4	xprim_4005–5	xprim_4005–6	xprim_4005–10
xprim_4006–4	xprim_4006–5	xprim_4006–6	
xprim_4008–4	xprim_4008–5	xprim_4008–6	
xprim_4010–4	xprim_4010–5	xprim_4010–6	xprim_4010–10
xprim_4013–4	xprim_4013–5	xprim_4013–6	

Table 8-6 XC3000 Primitive Libraries

–50 Speed Grade	–70 Speed Grade	–100 Speed Grade	–125 Speed Grade
xprim_3000–50	xprim_3000–70	xprim_3000–100	xprim_3000–125
xprim_3020–50	xprim_3020–70	xprim_3020–100	xprim_3020–125
xprim_3030–50	xprim_3030–70	xprim_3030–100	xprim_3030–125
xprim_3042–50	xprim_3042–70	xprim_3042–100	xprim_3042–125
xprim_3064–50	xprim_3064–70	xprim_3064–100	xprim_3064–125
xprim_3000–50	xprim_3000–70	xprim_3000–100	xprim_3000–125

Table 8-7 XC3000A/L Primitive Libraries

–6 Speed Grade	–7 Speed Grade	–8 Speed Grade
xprim_3000a–6	xprim_3000a–7	xprim_3000l–8
xprim_3020a–6	xprim_3020a–7	xprim_3020l–8
xprim_3030a–6	xprim_3030a–7	xprim_3030l–8
xprim_3042a–6	xprim_3042a–7	xprim_3042l–8
xprim_3064a–6	xprim_3064a–7	xprim_3064l–8
xprim_3090a–6	xprim_3090a–7	xprim_3090l–8

Table 8-8 XC3100/A Primitive Libraries

–3 Speed Grade	–4 Speed Grade	–5 Speed Grade
xprim_3100–3	xprim_3100–4	xprim_3100–5
xprim_3100a–3	xprim_3100a–4	xprim_3100a–5
xprim_3120–3	xprim_3120–4	xprim_3120–5
xprim_3120a–3	xprim_3120a–4	xprim_3120a–5
xprim_3130–3	xprim_3130–4	xprim_3130–5
xprim_3130a–3	xprim_3130a–4	xprim_3130a–5
xprim_3142–3	xprim_3142–4	xprim_3142–5
xprim_3142a–3	xprim_3142a–4	xprim_3142a–5
xprim_3164–3	xprim_3164–4	xprim_3164–5
xprim_3164a–3	xprim_3164a–4	xprim_3164a–5

–3 Speed Grade	–4 Speed Grade	–5 Speed Grade
xprim_3190–3	xprim_3190–4	xprim_3190–5
xprim_3190a–3	xprim_3190a–4	xprim_3190a–5
xprim_3195–3	xprim_3195–4	xprim_3195–5
xprim_3195a–3	xprim_3195a–4	xprim_3195a–5

xio_*4kparttype–s*.db

These libraries contain the I/O cells specific to the XC4000 Xilinx devices. Synopsys Release V3.1 or later cannot automatically synthesize some cells in the I/O libraries. You must instantiate these cells in your HDL description. The following table summarizes the Xilinx-specific I/O libraries. Refer to the "Configuring IOBs" section in the "Using the FPGA Compiler" chapter.

Table 8-9 XC4000 I/O Libraries

–4 Speed Grade	–5 Speed Grade	–6 Speed Grade	–10 Speed Grade
xio_4000–4	xio_4000–5	xio_4000–6	xio_4000–10
	xio_4000a–5	xio_4000a–6	
	xio_4000h–5	xio_4000h–6	

xfpga_*family*–s.db

The FPGA Compiler-specific libraries contain CLBs and IOBs, so that the FPGA Compiler maps directly into CLBs and IOBs. The cells in these libraries are synthesized automatically from the equations or circuitry in your design. You never have to instantiate any of these cells in your design. The following table summarizes the FPGA Compiler-specific libraries.

Table 8-10 XC4000 FPGA Compiler-Specific Libraries

–4 and –6 Speed Grades	–5 and –10 Speed Grades
xfpga_4000–4	xfpga_4000–5
xfpga_4000–6	xfpga_4000–10

xdc_*family*–s.db

The cells in the Design Compiler-specific libraries are also synthesized automatically from the equations or circuitry in your design. These cells are 2- to 4-input combinatorial functions. You never have to instantiate any of these cells in your design. The following table summarizes the Design Compiler-specific libraries.

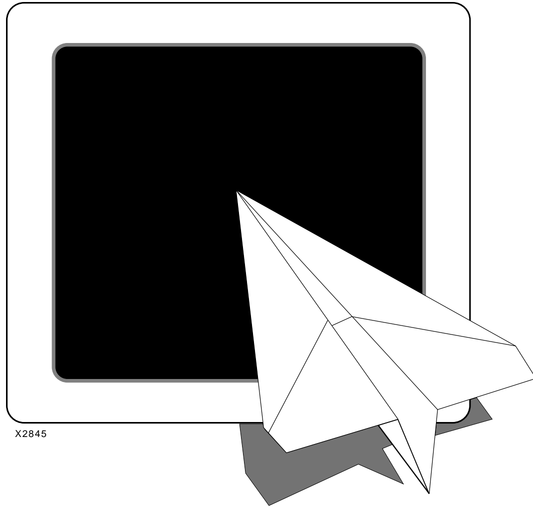
Table 8-11 Design Compiler-Specific Libraries

XC4000	XC3000	XC3000A	XC3000L	XC3100
xdc_4000–4	xdc_3000–50	xdc_3000a–6	xdc_3000l–8	xdc_3100–3
xdc_4000–5	xdc_3000–70	xdc_3000a–7		xdc_3100–4
xdc_4000–6	xdc_3000–100			xdc_3100–5
xdc_4000–10	xdc_3000–124			

Unsupported Part Types and Speed Grades

If you are designing for a part type and/or speed grade for which no libraries are available, use the libraries for the closest part type or speed grade in the same family, and indicate the part type or speed grade you are actually using when you run APR or PPR.

Note: For more information on specifying the part type, refer to the *XACT Reference Guide*.



XC3000/A/L and XC3100/A Primitives

Xilinx Synopsys Interface FPGA User Guide

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Appendix A

XC3000/A/L and XC3100/A Primitives

All primitives and macros available are located in the XSI-supplied libraries and can be instantiated in a VHDL or Verilog HDL file. Use the Synlibs program to list the appropriate libraries for the specific part type. Refer to the “Getting Started” chapter for information on how to use Synlibs. In the following listings, the primitive names are followed by the names of the inputs and outputs and timing notations where appropriate.

The name of a primitive or macro is used to instantiate it, and you must identify the signals connected to the input and output pins when instantiating a primitive or macro. You can connect signals to the primitives and macros in two ways.

- You must connect signals to all the pins and list them in the order given in the following tables.

Note: If using this first method to connect signals, be sure to follow exactly the order of input and output pins given in the primitive tables.

- You must connect signals to explicitly named pins.

In general, pins are organized with data pins before control pins. When several pins are part of a bus, they are listed with the MSB first. Buses of four or more bits follow bus notation, for example, A<7:0>. Buses with fewer bits are kept as separate signals.

The following table lists the cell name suffixes and their corresponding descriptions.

Table A-1 Cell Name Suffixes

Suffix	Description
I	Inverted global reset (INIT=S)
_F	Fast slew rate or fast implementation of clock buffer (using dedicated input clock pad)
_S	Slow slew rate
_U	Unbonded pad
_FLAG	Net/pin constraints
_N	No meaning; used as a placeholder

Although Synopsys cannot synthesize some primitives (primitives with Don't Touch attribute), you can instantiate them into your HDL. An asterisk (*) next to the primitive name indicates that you can instantiate it. Refer to the Synopsys documentation for more information on the Don't Touch attribute.

All cells in the libraries contain timing parameters. The column labeled "Notes" includes specific timing details on Xilinx primitives as well as additional functional information. See *The Programmable Logic Data Book* and the *XACT Libraries Guide* for additional timing information.

XC3000 Primitives

This section lists the XC3000 primitives, which include basic gates, flip-flops, latches, clock buffers, special input and output pads, I/O primitives, and special functions.

Basic Gates

This section lists the basic gates, which include AND/OR gates, inverters, buffers, 3-state buffers, and pull-up resistors.

Table A-2 AND/OR Gates

Name	Outputs	Inputs
AND2	O	I1, I0
AND3	O	I2, I1, I0
AND4	O	I3, I2, I1, I0
AND5	O	I4, I3, I2, I1, I0
NAND2	O	I1, I0
NAND3	O	I2, I1, I0
NAND4	O	I3, I2, I1, I0
NAND5	O	I4, I3, I2, I1, I0
OR2	O	I1, I0
OR3	O	I2, I1, I0
OR4	O	I3, I2, I1, I0
OR5	O	I4, I3, I2, I1, I0
NOR2	O	I1, I0
NOR3	O	I2, I1, I0
NOR4	O	I3, I2, I1, I0
NOR5	O	I4, I3, I2, I1, I0
XOR2	O	I1, I0
XOR3	O	I2, I1, I0
XOR4	O	I3, I2, I1, I0
XOR5	O	I4, I3, I2, I1, I0
XNOR2	O	I1, I0
XNOR3	O	I2, I1, I0
XNOR4	O	I3, I2, I1, I0
XNOR5	O	I4, I3, I2, I1, I0

Table A-3 Inverter

Name	Outputs	Inputs	Notes
INV	O	I	No delay

Table A-4 Buffer

Name	Outputs	Inputs	Notes
BUF	O	I	No delay

Table A-5 3-State Buffer

Name	Outputs	Inputs	Notes
BUFT	O	I, T	Delay value with one pull-up resistor

Synopsys tools synthesize an internal 3-state condition using BUFTs. A high-impedance state is floating unless you instantiate a pull-up resistor.

Table A-6 Pull-Up Resistor to V_{CC}

Name	Outputs	Inputs	Notes
PULLUP	O		No delay; used for IOBs or BUFTs

Flip-Flops and Latches

This section lists flip-flop and latches, which include D flip-flops and 1-bit transparent-High latches.

Table A-7 D Flip-Flops

Name	Outputs	Inputs	Notes
FDC	Q	D, C, CLR	With Clear Direct; initial startup value is 0
FDCE	Q	D, C, CE, CLR	Clock Enable with Clear Direct; initial startup value is 0
FDPI	Q	D, C, PRE	With Preset Direct; initial startup value is 1
FDPEI	Q	D, C, CE, PRE	Clock Enable with Preset Direct; initial startup value is 1

Table A-8 1-Bit Transparent-High Latches

Name	Outputs	Inputs	Notes
LD	Q	D, G	
LDC	Q	D, G, CLR	With Clear Reset
LDP	Q	D, G, PRE	With Preset Direct

These cells are not recommended since they are built from gates. The delay depends on how the cell is routed. Use D flip-flops instead.

Clocks

This section lists the input and output pins for the clock buffer primitives.

Table A-9 Clock Buffers

Name	Outputs	Inputs	Notes
GCLK*	O	I	Global
GCLK_F	O	I	Global; using dedicated pad
BUFG*	O	I	Generic
BUFG_F	O	I	Generic; using dedicated pad
ACLK*	O	I	Alternate
ACLK_F	O	I	Alternate; using dedicated pad

* Indicates that you must instantiate this primitive.

Oscillators

This section lists the input and output pins for the oscillator primitives.

Table A-10 Oscillators

Name	Outputs	Inputs	Notes
OSC*	O		No delay
GXTL*	O		Crystal; no delay

* Indicates that you must instantiate this primitive.

I/O Primitives

This section lists the I/O primitives, which include single input buffers, input buffers with D flip-flop, input buffers with D latch, output buffers, output buffers with D flip-flop, 3-state output buffers, and bidirectional buffers.

Table A-11 Single Input Buffers

Name	Outputs	Inputs	Notes
IBUF	O	I	
IBUF_U*	O	I	Unbonded pad

* Indicates that you must instantiate this primitive.

Table A-12 Input Buffers with D Flip-Flop

Name	Outputs	Inputs	Notes
IFD	Q	D, C	
IFD_U*	Q	D, C	Unbonded pad

* Indicates that you must instantiate this primitive.

Table A-13 Input Buffer with D Latch from One Input Pad

Name	Outputs	Inputs	Notes
ILD	Q	D, G	

Table A-14 Output Buffers

Name	Outputs	Inputs	Notes
OBUF	O	I	Slow slew rate
OBUF_F	O	I	Fast slew rate
OBUF_U*	O	I	Unbonded pad

* Indicates that you must instantiate this primitive.

Table A-15 Output Buffers with D Flip-Flop

Name	Outputs	Inputs	Notes
OFD	Q	D, C	Slow slew rate
OFD_F	Q	D, C	Fast slew rate
OFD_FU*	Q	D, C	Fast slew rate and unbonded pad
OFD_U*	Q	D, C	Unbonded pad

* Indicates that you must instantiate this primitive.

Table A-16 Output Buffers with D Flip-Flop and 3-State

Name	Outputs	Inputs	Notes
OFDT	O	D, C, T	Slow slew rate
OFDT_F	O	D, C, T	Fast slew rate

Table A-17 3-State Output Buffers

Name	Outputs	Inputs	Notes
OBUFT	O	I, T	Slow slew rate
OBUFT_F	O	I, T	Fast slew rate

Table A-18 Bidirectional Buffers

Name	Outputs	Inputs	Inputs/ Outputs	Notes
IOBUF	O	I, T	IO	Slow slew rate
IOBUF_N_F	O	I, T	IO	Fast output slew rate
IOBUF_N_S	O	I, T	IO	Slow output slew rate

Special Functions

This section lists the special function primitives, which include CLBMAPs, flag cells, power, and ground.

Table A-19 CLBMAPs

Name	Inputs	Notes
CLBMAP_PUC*	A, B, C, D, E, K, EC, DI, RD, X, Y	Pins unlocked from signals; function generator closed to additional logic
CLBMAP_PLC*	A, B, C, D, E, K, EC, DI, RD, X, Y	Pins locked to external signals; function generator closed to additional logic
CLBMAP_PUO*	A, B, C, D, E, K, EC, DI, RD, X, Y	Pins unlocked from signals; function generator open to additional logic
CLBMAP_PLO*	A, B, C, D, E, K, EC, DI, RD, X, Y	Pins locked to external signals; function generator open to additional logic

* Indicates that you must instantiate this primitive.

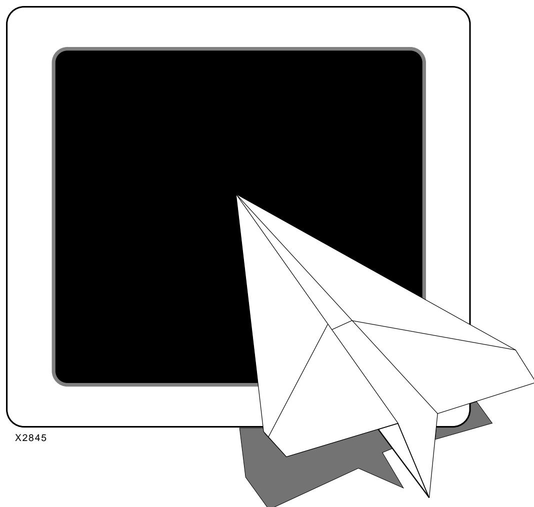
Table A-20 Flag Cells

Cell	Inputs	Description
C_FLAG*	I	Signal is on a critical path.
L_FLAG*	I	Signal should be routed along a longline.
N_FLAG*	I	Signal timing is not critical.
S_FLAG*	I	Save signal; treat it as external connection.
X_FLAG*	I	Signal is an explicit LCA net.

* Indicates that you must instantiate this primitive.

Table A-21 Power/Ground

Name	Outputs
VCC	VCC
GND	GROUND



XC4000/A/D/H
Primitives and Macros

Xilinx Synopsys Interface FPGA User Guide

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Appendix B

XC4000/A/D/H Primitives and Hard Macros

All primitives and macros available are located in the XSI-supplied libraries and can be instantiated in a VHDL or Verilog HDL file. Use the Synlibs program to list the appropriate libraries for the specific part type. Refer to the "Getting Started" chapter for information on how to use Synlibs. In the following listings, the primitive names are followed by the names of the inputs and outputs and timing notations where appropriate.

Hard macros are obsolete; however, new relationally placed module (RPM) versions of the hard macros are supported in XSI V3.1 or later and XACT V5.0 or later to help you make the design transition to RPMs. Xilinx-supplied hard macros have been converted to RPMs and are located in the $XACT/data/hmlib directory.

If you have Xilinx-supplied hard macros in an existing design, you must copy the appropriate XNF file from the $XACT/data/hmlib directory to your design directory. You must convert user-generated hard macros to RPMs with the HM2RPM program. Refer to the *XACT Reference Guide, Volume 1* for detailed information on hard macro conversion.

The name of a primitive or macro is used to instantiate it, and you must identify the signals connected to the input and output pins when instantiating a primitive or macro. You can connect signals to the primitives and macros in two ways.

- You must connect signals to all the pins and list them in the order given in the following tables.

Note: If using this first method to connect signals, be sure to follow exactly the order of input and output pins given in the primitive tables.

- You must connect signals to explicitly named pins.

In general, pins are organized with data pins before control pins. When several pins are part of a bus, they are listed with the MSB first. Buses of four or more bits follow bus notation, for example, A<7:0>. Buses with fewer bits are kept as separate signals.

The following table lists the cell name suffixes and their corresponding descriptions.

Table B-1 Cell Name Suffixes

Suffix	Description
I	Inverted global reset (INIT=S)
_F	Fast slew rate or fast implementation of clock buffer (using dedicated input clock pad) for output buffers; NODELAY attribute added for input registers
_S	Slow slew rate
_MF	Medium-fast slew rate (XC4000A only)
_MS	Medium-slow slew rate (XC4000A only)
_U	Unbonded pad
_1	Inverted clock or gate on flip-flop or latch
_FLAG	Net/pin constraints
_TTL	TTL-compatible level (XC4000H only)
_CMOS	CMOS-compatible level (XC4000H only)
CAP	Capacitive slew rate (XC4000H only)
RES	Resistive slew rate (XC4000H only)
_N	No meaning; used as a placeholder

Although Synopsys cannot synthesize some primitives (primitives with Don’t Touch attribute) and macros, they can be instantiated. An asterisk (*) next to the primitive name indicates that you can instantiate it. Refer to the Synopsys documentation for more information on the Don’t Touch attribute.

All cells in the libraries contain timing parameters. The column labeled "Notes" includes specific timing details on Xilinx primitives as well as additional functional information. See *The Programmable Logic Data Book* and the *XACT Libraries Guide* for additional timing information.

XC4000 Primitives

This section lists the XC4000 primitives, which include basic gates, flip-flops, latches, clock buffers, special input and output pads, I/O primitives, and special functions.

Basic Gates

This section lists the basic gates, which include AND/OR gates, inverters, buffers, 3-state buffers, wired-AND, wired OR-AND, wide I/O decoders, pull-up resistors, pull-down resistors, and RAM/ROM primitives.

Table B-2 AND/OR Gates

Name	Outputs	Inputs
AND2	O	I1, I0
AND3	O	I2, I1, I0
AND4	O	I3, I2, I1, I0
AND5	O	I4, I3, I2, I1, I0
NAND2	O	I1, I0
NAND3	O	I2, I1, I0
NAND4	O	I3, I2, I1, I0
NAND5	O	I4, I3, I2, I1, I0
OR2	O	I1, I0
OR3	O	I2, I1, I0
OR4	O	I3, I2, I1, I0
OR5	O	I4, I3, I2, I1, I0
NOR2	O	I1, I0
NOR3	O	I2, I1, I0
NOR4	O	I3, I2, I1, I0

Name	Outputs	Inputs
NOR5	O	I4, I3, I2, I1, I0
XOR2	O	I1, I0
XOR3	O	I2, I1, I0
XOR4	O	I3, I2, I1, I0
XOR5	O	I4, I3, I2, I1, I0
XNOR2	O	I1, I0
XNOR3	O	I2, I1, I0
XNOR4	O	I3, I2, I1, I0
XNOR5	O	I4, I3, I2, I1, I0

Table B-3 Inverter

Name	Outputs	Inputs	Notes
INV	O	I	No delay

Table B-4 Buffer

Name	Outputs	Inputs	Notes
BUF	O	I	No delay

Table B-5 3-State Buffer

Name	Outputs	Inputs	Notes
BUFT	O	I, T	Synopsys tools synthesize an internal 3-state condition using BUFTs. A high-impedance state is floating unless a pull-up resistor is instantiated.

Table B-6 Wired-AND

Name	Outputs	Inputs	Notes
WAND1*	O	I	No pull-up resistor

* Indicates that you must instantiate this primitive.

Table B-7 Wired OR-AND

Name	Outputs	Inputs	Notes
WOR2AND*	O	I1, I0	No pull-up resistor

* Indicates that you must instantiate this primitive.

Table B-8 Wide I/O Decoders

Name	Outputs	Inputs	Notes
DECODE1_IO*	O	I	1-bit I/O edge decoder; no pull-up resistor
DECODE4*	O	A<3:0>	4-bit I/O edge decoder; no pull-up resistor
DECODE8*	O	A<7:0>	8-bit I/O edge decoder; no pull-up resistor
DECODE16*	O	A<15:0>	16-bit I/O edge decoder; no pull-up resistor
DECODE1_INT*	O	I	1-bit edge decoder; no pull-up resistor; input from internal logic

* Indicates that you must instantiate this primitive.

Table B-9 Resistor to V_{CC} for Inputs, Open-Drain and 3-State Outputs

Name	Outputs	Inputs	Notes
PULLUP*	O		No delay; used for IOBs or BUFTs

* Indicates that you must instantiate this primitive.

Table B-10 Resistor to Ground for Inputs

Name	Outputs	Inputs	Notes
PULLDOWN*	O		No delay; used for IOB or BUFTs

* Indicates that you must instantiate this primitive.

Table B-11 RAM/ROM Primitives

Name	Outputs	Inputs	Notes
RAM16X1	O	D, A3, A2, A1, A0, WE	
RAM32X1	O	D, A4, A3, A2, A1, A0, WE	
ROM16X1	O	A3, A2, A1, A0	Must add ROM value*
ROM32X1	O	A4, A3, A2, A1, A0	Must add ROM value*

* Refer to the “Using the Design Compiler” or the “Using the FPGA Compiler” chapter.

Flip-Flops and Latches

This section lists flip-flops and latches, which include D flip-flops and 1-bit transparent-High latches.

Table B-12 D Flip-Flops

Name	Outputs	Inputs	Notes
FDC	Q	D, C, CLR	With Clear Direct; initial startup value is 0
FDCE	Q	D, C, CE, CLR	Clock Enable with Clear Direct; initial startup value is 0
FDP	Q	D, C, PRE	With Preset Direct; initial startup value is 1
FDPE	Q	D, C, CE, PRE	Clock Enable with Preset Direct; initial startup value is 1

Table B-13 1-bit Transparent-High Latches

Name	Outputs	Inputs	Notes
LD_1*	Q	D, G	
LDC_1*	Q	D, G, CLR	With Clear Reset
LDP_1*	Q	D, G, PRE	With Preset Direct

* These cells are not recommended because they are built from gates. The delay depends on how the cell is routed. Use D flip-flops instead.

Clocks

This section lists the clock buffer primitives.

Table B-14 Clock Buffers

Names	Outputs	Inputs	Notes
BUFG*	O	I	No pad delay included
BUFG_F	O	I	Fast implementation of clock; using dedicated pad
BUFGP_F	O	I	Fast implementation of clock; using dedicated pad
BUFGS*	O	I	No pad delay included
BUFGS_F	O	I	Fast implementation of clock; using dedicated pad

* Indicates that you must instantiate this primitive.

I/O Primitives

This section lists the I/O primitives, which include input buffers, input buffers with D flip-flop, input buffers with inverted latch, output buffers, 3-state output buffers, 3-state output buffers with D flip-flop, output buffers with D flip-flop, and bidirectional buffers.

Note: I/O buffers with flip-flops or latches are not available for the XC4000H libraries.

Table B-15 Input Buffers

Name	Outputs	Inputs	Notes
IBUF	O	I	
IBUF_U*	O	I	Unbonded pad

* Indicates that you must instantiate this primitive.

Table B-16 Input Buffers — XC4000H Only

Name	Outputs	Inputs	Notes
IBUF_CMOS	O	I	CMOS-compatible level
IBUF_TTL	O	I	TTL-compatible level

Table B-17 Input Buffers and D Flip-Flop

Name	Outputs	Inputs	Notes
IFD	Q	D, C	
IFD_F	Q	D, C	Includes NODELAY attribute
IFD_U*	Q	D, C	Unbonded pad
IFDI*	Q	D, C	INIT=S; inverted Global Reset
IFDI_F*	Q	D, C	Includes NODELAY attribute; INIT=S; inverted Global Reset
IFDI_U*	Q	D, C	Unbonded pad; INIT=S; inverted Global Reset

* Indicates that you must instantiate this primitive.

Table B-18 Input Buffers and D Latch with Inverted Latch

Name	Outputs	Inputs	Notes
ILD_1	Q	D, G	
ILD_1F	Q	D, G	NODELAY attribute added
ILD_1U*	Q	D, G	Unbonded pad
ILDI_1	Q	D, G	Inverted Global Reset
ILDI_1F	Q	D, G	NODELAY attribute added; inverted Global Reset
ILDI_1U*	Q	D, G	Unbonded pad; inverted Global Reset

* Indicates that you must instantiate this primitive.

Table B-19 Output Buffers

Name	Outputs	Inputs	Notes
OBUF	O	I	
OBUF_F	O	I	Fast slew rate
OBUF_S	O	I	Slow slew rate
OBUF_U*	O	I	Unbonded pad

* Indicates that you must instantiate this primitive.

Table B-20 Output Buffers — XC4000A Only

Name	Outputs	Inputs	Notes
OBUF_MF	O	I	Medium-fast slew rate
OBUF_MS	O	I	Medium-slow slew rate

Table B-21 Output Buffers — XC4000H Only

Name	Outputs	Inputs	Notes
OBUF_CMOSCAP	O	I	CMOS-compatible, capacitive slew rate
OBUF_CMOSRES	O	I	CMOS-compatible, resistive slew rate
OBUF_TTLCAP	O	I	TTL-compatible, capacitive slew rate
OBUF_TTLRES	O	I	TTL-compatible, resistive slew rate

Table B-22 3-State Output Buffers

Name	Outputs	Inputs	Notes
OBUFT	O	I, T	
OBUFT_F	O	I, T	Fast slew rate
OBUFT_S	O	I, T	Slow slew rate
OBUFT_U*	O	I, T	Unbonded pad

* Indicates that you must instantiate this primitive.

Table B-23 3-State Output Buffers — XC4000A Only

Name	Outputs	Inputs	Notes
OBUFT_MF	O	I, T	Medium-fast slew rate
OBUFT_MS	O	I, T	Medium-slow slew rate

Table B-24 3-State Output Buffers — XC4000H Only

Name	Outputs	Inputs	Notes
OBUFT_CMOSCAP	O	I, T	CMOS-compatible, capacitive slew rate
OBUFT_CMOSRES	O	I, T	CMOS-compatible, resistive slew rate
OBUFT_TTLCAP	O	I, T	TTL-compatible, capacitive slew rate
OBUFT_TTLRES	O	I, T	TTL-compatible, resistive slew rate

Table B-25 3-State Output Buffers with D Flip-Flop

Name	Outputs	Inputs	Notes
OFDT	O	D, C, T	
OFDT_F	O	D, C, T	Fast slew rate
OFDT_S	O	D, C, T	Slow slew rate
OFDT_U*	O	D, C, T	Unbonded pad

* Indicates that you must instantiate this primitive.

Table B-26 3-State Output Buffers with D Flip-Flop — XC4000A Only

Name	Outputs	Inputs	Notes
OFDT_MF	O	D, C, T	Medium-fast slew rate
OFDT_MS	O	D, C, T	Medium-slow slew rate

Table B-27 3-State Output Buffers with D Flip-Flop and Inverted Global Reset

Name	Outputs	Inputs	Notes
OFDTI*	O	D, C, T	
OFDTI_F*	O	D, C, T	Fast slew rate
OFDTI_S*	O	D, C, T	Slow slew rate
OFDTI_U*	O	D, C, T	Unbonded pad

* Indicates that you must instantiate this primitive.

Table B-28 3-State Output Buffers with D Flip-Flop and Inverted Global Reset — XC4000A Only

Name	Outputs	Inputs	Notes
OFDTI_MF*	O	D, C, T	Medium-fast slew rate
OFDTI_MS*	O	D, C, T	Medium-slow slew rate

* Indicates that you must instantiate this primitive.

Table B-29 Output Buffers with D Flip-Flop

Name	Outputs	Inputs	Notes
OFD	Q	D, C	
OFD_F	Q	D, C	Fast slew rate
OFD_FU*	Q	D, C	Fast slew rate; unbonded pad
OFD_S	Q	D, C	Slow slew rate
OFD_U*	Q	D, C	Unbonded pad

* Indicates that you must instantiate this primitive.

Table B-30 Output Buffers with D Flip-Flop — XC4000A Only

Name	Outputs	Inputs	Notes
OFD_MF	Q	D, C	Medium-fast slew rate
OFD_MS	Q	D, C	Medium-slow slew rate

Table B-31 Output Buffers with D Flip-Flop and Inverted Global Reset

Name	Outputs	Inputs	Notes
OFDI*	Q	D, C	
OFDI_F*	Q	D, C	Fast slew rate
OFDI_S*	Q	D, C	Slow slew rate
OFDI_U*	Q	D, C	Unbonded pad

* Indicates that you must instantiate this primitive.

Table B-32 Output Buffers with D Flip-Flop and Inverted Global Reset — XC4000A Only

Name	Outputs	Inputs	Notes
OFDI_MF*	Q	D, C	Medium-fast slew rate
OFDI_MS*	Q	D, C	Medium-slow slew rate

* Indicates that you must instantiate this primitive.

Table B-33 Bidirectional Buffers

Name	Outputs	Inputs/ Outputs	Inputs	Notes
IOBUF	O	IO	I, T	Slow slew rate

Name	Outputs	Inputs/ Outputs	Inputs	Notes
IOBUF_N_F	O	IO	I, T	Fast output slew rate
IOBUF_N_S	O	IO	I, T	Slow output slew rate

Table B-34 Bidirectional Buffers — XC4000A Only

Name	Outputs	Inputs/ Outputs	Inputs	Notes
IOBUF_N_MF	O	IO	I, T	Medium-fast output slew rate
IOBUF_N_MS	O	IO	I, T	Medium-slow output slew rate

Table B-35 Bidirectional Buffers — XC4000H Only

Name	Outputs	Inputs/ Outputs	Inputs	Notes
IOBUF_CMOS_CMOSCAP	O	IO	I, T	CMOS input threshold; CMOS output level; capacitive slew rate
IOBUF_CMOS_CMOSRES	O	IO	I, T	CMOS input threshold; CMOS output level; resistive slew rate
IOBUF_TTL_CMOSCAP	O	IO	I, T	TTL input threshold; CMOS output level; capacitive slew rate
IOBUF_TTL_CMOSRES	O	IO	I, T	TTL input threshold; CMOS output level; resistive slew rate

Name	Outputs	Inputs/ Outputs	Inputs	Notes
IOBUF_TTL_TTLCAP	O	IO	I, T	TTL input threshold; TTL output level; capacitive slew rate
IOBUF_TTL_TTLRES	O	IO	I, T	TTL input threshold; TTL output level; resistive slew rate

Special Functions

This section lists special functions, which include the boundary scan, readback, startup, mapping, flag cells, power, and ground primitives.

Table B-36 Boundary-Scan Logic Controller

Name	Outputs	Inputs	Notes
BSCAN	TDO, DRCK, IDLE, SEL1, SEL2,	TDI, TMS, TCK, TDO1, TDO2,	No delay
MD1		O	Output pad for BSCAN
TDO		O	Output pad for BSCAN
MD0	I		Input pad for BSCAN
MD2	I		Input pad for BSCAN
TCK	I		Input pad for BSCAN
TDI	I		Input pad for BSCAN
TMS	I		Input pad for BSCAN

Note: Do not connect an IBUF to the TCK, TDI, or TMS input pads. Similarly, do not connect an OBUF to the TDO output. You must connect MD0 and MD2 to an IBUF symbol. Similarly, you must connect an MD1 pad to an OBUF symbol.

Table B-37 LCA Bitstream Readback Boundary-Scan Logic Controller (for Readback Function)

Name	Outputs	Inputs	Notes
RDBK	DATA, RIP	TRIG	No delay

Table B-38 Readback Controller (for Readback Function)

Name	Outputs	Inputs	Notes
RDCLK		I	No delay

Table B-39 Readback Function

Name	Outputs	Inputs	Notes
READBACK	DATA, RIP	CLK, TRIG	No delay

Table B-40 Startup and Configuration Controller

Name	Outputs	Inputs
STARTUP	Q2, Q3, Q1Q4, DONEIN	GSR, GTS, CLK

Table B-41 Internal 5-Frequency Clock Signal Generator

Name	Outputs	Notes
OSC4	F8M, F500K, F16K, F490, F15	No delay

Table B-42 Mapping Primitives

Name	Inputs	Notes
FMAP_PUC	I4, I3, I2, I1, O	Pins unlocked from signals; function generator closed to additional logic
FMAP_PLC	I4, I3, I2, I1, O	Pins locked to external signals; function generator closed to additional logic
FMAP_PUO	I4, I3, I2, I1, O	Pins unlocked from signals; function generator open to additional logic
FMAP_PLO	I4, I3, I2, I1, O	Pins locked to external signals; function generator open to additional logic
HMAP_PUC	I3, I2, I1, O	Pins unlocked from signals; function generator closed to additional logic

Table B-43 Flag Cells

Cell	Inputs	Description
C_FLAG	I	Signal is on a critical path.
N_FLAG	I	Signal timing is not critical.
S_FLAG	I	Save signal; treat it as an external connection.
X_FLAG	I	Signal is an explicit LCA net.

Table B-44 Power/Ground

Name	Outputs
VCC	VCC
GND	GROUND

X-BLOX Modules

The DesignWare module naming conventions are illustrated in Figure B-44. The example provided is for a comparator module and contains the four possible components used in naming the modules. Other module names may not contain all four components. Refer to Table B-45 for a description of each component.

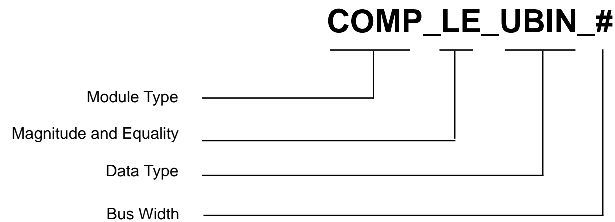


Figure B-44 DesignWare Module Naming Conventions

Table B-45 DesignWare Module Naming Conventions

Module Type	ADD_SUB	Adder/Subtracter
	COMP	Comparator
	INC_DEC	Incrementer/Decrementer
Magnitude and Equality	GE	Greater than or equal to
	GT	Greater than
	LE	Less than or equal to
	LT	Less than
Data Type	TWO_COMP	Twos complement
	UBIN	Unsigned binary
Bus Width	#	Bus width can be 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, or 48 (and 64 for COMP only). Use <(#–1):0> to translate bus width to bus notation. For example, if Bus A has a bus width of 6, then the correct bus notation is A<(6–1):0> or A<5:0>.

Table B-46 maps DesignWare modules to X-BLOX Modules and provides inputs and outputs.

Table B-46 DesignWare Modules

DesignWare Module	X-BLOX Module	Inputs	Outputs
ADD_SUB_TWO_COMP_#	ADD_SUB	C_IN, ADD_SUB, B<(#–1):0>, A<(#–1):0>	FUNC<(#–1):0>
ADD_SUB_UBIN_#		C_IN, ADD_SUB, B<(#–1):0>, A<(#–1):0>	FUNC<(#–1):0>
COMP_GE_TWO_COMP_#	COMPARE	B<(#–1):0>, A<(#–1):0>	Z
COMP_GE_UBIN_#		B<(#–1):0>, A<(#–1):0>	Z
COMP_GT_TWO_COMP_#		B<(#–1):0>, A<(#–1):0>	Z
COMP_GT_UBIN_#		B<(#–1):0>, A<(#–1):0>	Z
COMP_LE_TWO_COMP_#		B<(#–1):0>, A<(#–1):0>	Z
COMP_LE_UBIN_#		B<(#–1):0>, A<(#–1):0>	Z
COMP_LT_TWO_COMP_#		B<(#–1):0>, A<(#–1):0>	Z
COMP_LT_UBIN_#		B<(#–1):0>, A<(#–1):0>	Z
INC_DEC_TWO_COMP_#	INC_DEC	INC_DEC, A<(#–1):0>	FUNC<(#–1):0>
INC_DEC_UBIN_#		INC_DEC, A<(#–1):0>	FUNC<(#–1):0>

XC4000 Hard Macros

Synopsys cannot synthesize the following hard macros. You must instantiate them into your HDL code. However, hard macros are obsolete. Refer to the introductory material at the beginning of this chapter for more information on converting hard macros to RPMs, which are supported by XACT V5.0 or later.

Table B-47 8-Bit Accumulator with Overflow

Name	Inputs	Outputs	CLB Usage
ACC8H*	A<7:0>, R, C	Q<7:0>, OFL	6

* Indicates that you must instantiate this primitive.

Table B-48 16-Bit Accumulator with Overflow

Name	Inputs	Outputs	CLB Usage
ACC16H*	A<15:0>, R, C	Q<15:0>, OFL	10

Table B-49 8-Bit Adder/Subtracter with Overflow

Name	Inputs	Outputs	CLB Usage
ADSU8H*	A<7:0>, B<7:0>, ADD	S<7:0>, OFL	6

Table B-50 16-Bit Adder/Subtracter with Overflow

Name	Inputs	Outputs	CLB Usage
ADSU16H*	A<15:0>, B<15:0>, ADD	S<15:0>, OFL	10

Table B-51 8-Bit Identity Comparator

Name	Inputs	Outputs	CLB Usage
COMP8H*	A<7:0>, B<7:0>	EQ	2

Table B-52 16-Bit Identity Comparator

Name	Inputs	Outputs	CLB Usage
COMP16H*	A<15:0>, B<15:0>	EQ	5

* Indicates that you must instantiate this primitive.

Table B-53 8-Bit Magnitude Comparator

Name	Inputs	Outputs	CLB Usage
COMPM8H*	A<7:0>, B<7:0>	GE, LT	5

Table B-54 16-Bit Magnitude Comparator

Name	Inputs	Outputs	CLB Usage
COMPM16H*	A<15:0>, B<15:0>	GE, LT	9

Table B-55 8-Bit Parallel Up Counter

Name	Inputs	Outputs	CLB Usage
CUP8H*	D<7:0>, PE, CE, RD, C	Q<7:0>, TC	5

Table B-56 16-Bit Parallel Up Counter

Name	Inputs	Outputs	CLB Usage
CUP16H*	D<15:0>, PE, CE, RD, C	Q<15:0>, TC	9

Table B-57 Hex-to-7-Segment Decoder

Name	Inputs	Outputs	CLB Usage
D7SEGH*	A3, A2, A1, A0, RBI	A, B, C, D, E, F, G, RBO	6

* Indicates that you must instantiate this primitive.

Table B-58 Hex-to-7-Segment Decoder Military

Name	Inputs	Outputs	CLB Usage
D7SEGMH*	A3, A2, A1, A0, RBI	A, B, C, D, E, F, G, RBO	6

Table B-59 2-to-4 Decoder with Enable

Name	Inputs	Outputs	CLB Usage
DEC2_4EH*	A1, A0, EN	O3, O2, O1, O0	2

Table B-60 3-to-8 Decoder with Enable

Name	Inputs	Outputs	CLB Usage
DEC3_8EH*	A2, A1, A0, EN	O<7:0>	4

Table B-61 8-to-3 Priority Encoder

Name	Inputs	Outputs	CLB Usage
ENCPR8H*	I<7:0>, EI	A2, A1, A0, EO	4

Table B-62 4-to-1 Multiplexer

Name	Inputs	Outputs	CLB Usage
MUX4_1H*	D3, D2, D1, D0, S1, S0	O	1

* Indicates that you must instantiate this primitive.

Table B-63 8-to-1 Multiplexer

Name	Inputs	Outputs	CLB Usage
MUX8_1H*	D<7:0>, S2, S1, S0	O	3

Table B-64 16-to-1 Multiplexer

Name	Inputs	Outputs	CLB Usage
MUX16_1H*	D<15:0>, S3, S2, S1, S0	O	5

Table B-65 9-Bit Even Parity Generator

Name	Inputs	Outputs	CLB Usage
PARE9H*	I<9:1>	EVE	1

Table B-66 9-Bit Odd Parity Generator

Name	Inputs	Outputs	CLB Usage
PARO9H*	I<9:1>	ODD	1

Table B-67 Divide by 8/9 Prescaler

Name	Inputs	Outputs	CLB Usage
PRSC8_9H*	DIV9, CKI	CKO	2

Table B-68 8-Bit Register with Clock Enable and Reset Direct

Name	Inputs	Outputs	CLB Usage
RD8H*	D<7:0>, CE, RD, C	Q<7:0>	4

* Indicates that you must instantiate this primitive.

Table B-69 16-Bit Register with Clock Enable and Reset Direct

Name	Inputs	Outputs	CLB Usage
RD16H*	D<15:0>, CE, RD, C	Q	8

Table B-70 16 x 2 RAM (16 deep x 2 wide)

Name	Inputs	Outputs	CLB Usage
RM16X2H*	D1, D0, A3, A2, A1, A0, WE	O1, O0	1

Table B-71 16 x 4 RAM (16 deep x 4 wide)

Name	Inputs	Outputs	CLB Usage
RM16X4H*	D3, D2, D1, D0, A3, A2, A1, A0, WE	O3, O2, O1, O0	2

Table B-72 16 x 8 RAM (16 deep x 8 wide)

Name	Inputs	Outputs	CLB Usage
RM16X8H*	D<7:0>, A3, A2, A1, A0, WE	O<7:0>	4

Table B-73 32 x 4 RAM (32 deep x 4 wide)

Name	Inputs	Outputs	CLB Usage
RM32X4H*	D3, D2, D1, D0, A4, A3, A2, A1, A0, WE	O3, O2, O1, O0	4

* Indicates that you must instantiate this primitive.

Table B-74 32 x 8 RAM (32 deep x 8 wide)

Name	Inputs	Outputs	CLB Usage
RM32X8H*	D<7:0>, A4, A3, A2, A1, A0, WE	O<7:0>	8

Table B-75 64 x 4 RAM (64 deep x 4 wide)

Name	Inputs	Outputs	CLB Usage
RM64X4H*	D3, D2, D1, D0, A5, A4, A3, A2, A1, A0, WE	O3, O2, O1, O0	11

Table B-76 64 x 8 RAM (64 deep x 8 wide)

Name	Inputs	Outputs	CLB Usage
RM64X8H*	D<7:0>, A5, A4, A3, A2, A1, A0, WE	O<7:0>	21

Table B-77 128 x 4 RAM (128 deep x 4 wide)

Name	Inputs	Outputs	CLB Usage
RM128X4H*	D3, D2, D1, D0, A6, A5, A4, A3, A2, A1, A0, WE	O3, O2, O1, O0	22

* Indicates that you must instantiate this primitive.

Table B-78 128 x 8 RAM (128 deep x 8 wide)

Name	Inputs	Outputs	CLB Usage
RM128X8H*	D<7:0>, A6, A5, A4, A3, A2, A1, A0, WE	O<7:0>	42

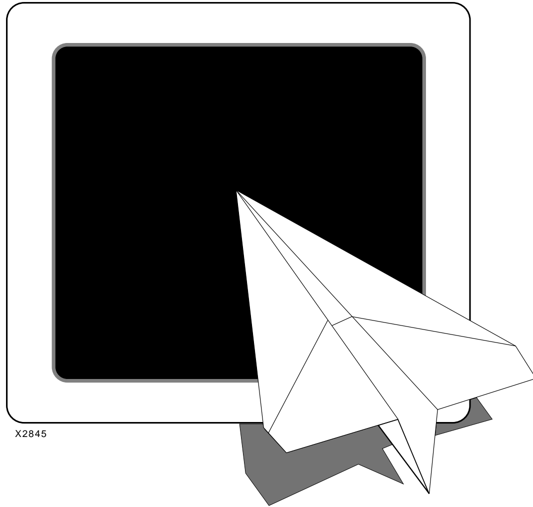
Table B-79 8-Bit Parallel Loadable Serial Shift Register

Name	Inputs	Outputs	CLB Usage
RS8PH*	D<7:0>, SERIN, PE, CE, RD, C	Q<7:0>	5

Table B-80 16-Bit Parallel Loadable Serial Shift Register

Name	Inputs	Outputs	CLB Usage
RS16PH*	D<15:0>, SERIN, PE, CE, RD, C	Q	9

* Indicates that you must instantiate this primitive.



Selection Guide

Xilinx Synopsys Interface FPGA User Guide

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Appendix C

Selection Guide

Use the tables in this section to determine if the primitives and macros used to create pre-XACT 5.0 designs have changed or become obsolete. The following tables list the exact or closest Unified Libraries replacement primitive where applicable. Additionally, the tables indicate if the primitive is obsolete and if the pin description has changed. A complete pin description for each primitive is provided in either Appendix A, “XC3000/A/L and XC3100/A Primitives, or Appendix B, “XC4000/A/D/H Primitives and Hard Macros.”

XC3000/A/L and XC3100/A Primitives

This section lists XC3000/A/L and XC3100/A primitives.

Note: The footnote explanations appear at the end of Table C-1.

Table C-1 XC3000/XC3100 Primitives/Macros

XC3000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
ACLK	ACLK_F			No
ACLK_NP	ACLK			No
AND2				Yes
AND3				Yes
AND4				Yes
AND5				Yes
BUF				No
C_FLAG				No

XC3000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
CLBMAP_PLC				No
CLBMAP_PLO				No
CLBMAP_PUC				No
CLBMAP_PUO				No
DFF	FDC			Yes
FDC		FDCE		Yes
FDCRD	FDCE			Yes
FDRD	FDC			Yes
GCLK	GCLK_F			No
GCLK_NP	GCLK			No
GND				Yes
GXTL				No
IBUF				No
IBUF4[2]		IBUF		Yes
IBUF8[2]		IBUF		Yes
IBUF16[2]		IBUF		Yes
IBUF32[2]		IBUF		Yes
INFF		IFD		Yes
INFF4[2]		IFD		Yes
INFF8[2]		IFD		Yes
INFF16[2]		IFD		Yes
INFF32[2]		IFD		Yes
INFF_U		IFD_U		Yes
INLAT		ILD		Yes
INLAT4[2]		ILD		Yes
INLAT8[2]		ILD		Yes
INLAT16[2]		ILD		Yes
INLAT32[2]		ILD		Yes
INV				No
LD				Yes
LDRD	LDC			Yes
LDSD	LDP			Yes

XC3000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
L_FLAG				No
NAND2				Yes
NAND3				Yes
NAND4				Yes
NAND5				Yes
NDFF	FDCE			Yes
NOR2				Yes
NOR3				Yes
NOR4				Yes
NOR5				Yes
N_FLAG				No
OBUF				No
OBUF_F				No
OBUF4_F[2]		OBUF_F		Yes
OBUF8_F[2]		OBUF_F		Yes
OBUF16_F[2]		OBUF_F		Yes
OBUF32_F[2]		OBUF_F		Yes
OBUFT				No
OBUFT_F				No
OBUFT4_F		OBUFT_F		Yes
OBUFT8_F[2]		OBUFT_F		Yes
OBUFT16_F[2]		OBUFT_F		Yes
OBUFT32_F[2]		OBUFT_F		Yes
OR2				Yes
OR3				Yes
OR4				Yes
OR5				Yes
OSC				No
OUTFF	OFD			No
OUTFF4_F[2]		OFD_F	Obsolete	No
OUTFF8_F[2]		OFD_F		No
OUTFF16_F[2]		OFD_F		No

XC3000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
OUTFF32_F[2]		OFD_F		No
OUTFFT	OFDT			No
OUTFFT4_F[2]		OFDT_F		No
OUTFFT8_F[2]		OFDT_F		No
OUTFFT16_F[2]		OFDT_F		No
OUTFFT32_F[2]		OFDT_F		No
OUTFFT_F	OFDT_F			No
OUTFF_F	OFD_F			No
OUTFF_U	OFD_U			No
OUTFF_UF	OFD_FU			No
PULLUP	For I/O, use PULLUP; for others, use BUS_PULLUP			
PWR	VCC			Yes
S_FLAG				No
TBUF	BUFT			No
TBUF_F		BUFT	Obsolete	No
TBUF_P	BUFT			No
XNOR2				Yes
XNOR3				Yes
XNOR4				Yes
XNOR5				Yes
XOR2				Yes
XOR3				Yes
XOR4				Yes
XOR5				Yes
X_FLAG				No

1. A pin change is a pin name change and/or a pin order change. Refer to Appendix A for the pin description.
2. These primitives were provided for instantiation in the previous release of Synopsys; in the current version, Synopsys can automatically infer these primitives. The bused versions of I/O primitives have been removed from the libraries because manual instantiation is not recommended. For more information, see the “Configuring the IOBs” section in the “Using the Design Compiler” chapter or the “Using the FPGA Compiler” chapter.

XC4000/A/D/H Primitives

This section lists XC4000/A/D/H primitives.

Note: The footnote explanations appear at the end of Table C-2.

Table C-2 XC4000 Primitives/Macros

XC4000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
ACC8H				No
ACC16H				No
ADD_SUB_CO_TWO_COMP_#[2]			Obsolete	
ADD_SUB_CO_UBIN_#[2]			Obsolete	
ADD_SUB_TWO_COMP_#[2]			Obsolete	
ADD_SUB_UBIN_#[2]			Obsolete	
ADSU8H				No
ADSU16H				No
AND2				Yes
AND3				Yes
AND4				Yes
AND5				Yes
BSCAN				No
BUF			Obsolete	
BUFGP	BUFGP_F			No
BUFGS	BUFGS_F			No
BUFGS_NP	BUFGS			No
COMP8H				No
COMP16H				No
COMPM8H				No
COMPM16H				No
COMP_GE_TWO_COMP_#[3]			Obsolete	
COMP_GE_UBIN_#[3]			Obsolete	
COMP_GT_TWO_COMP_#[3]			Obsolete	
COMP_GT_UBIN_#[3]			Obsolete	
COMP_LE_TWO_COMP_#[3]			Obsolete	

XC4000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
COMP_LE_UBIN_#[3]			Obsolete	
COMP_LT_TWO_COMP_#[3]			Obsolete	
COMP_LT_UBIN_#[3]			Obsolete	
CUP8H				No
CUP16H				No
C_FLAG				No
D7SEGH				No
D7SEGMH				No
DEC2_4EH				No
DEC3_8EH				No
ENCPR8H				No
FDRD	FDCE			Yes
FDRDE	FDC			Yes
FDSD	FDPE			Yes
FDSDE	FDP			Yes
FMAP_PLC				No
FMAP_PLO				No
FMAP_PUC				No
FMAP_PUO				No
GND				Yes
HMAP_PUC				No
IBUF				No
IBUF4[4]		IBUF		Yes
IBUF8[4]		IBUF	Obsolete	
IBUF16[4]		IBUF		
IBUF32[4]		IBUF		
IBUFT4_F[4]			Obsolete	
IBUFT8_F[4]			Obsolete	
IBUFT16_F[4]			Obsolete	
IBUFT32_F[4]			Obsolete	
INFF		IFD		Yes
INFF4[4]		IFD		Yes

XC4000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
INFF8[4]		IFD		Yes
INFF16[4]		IFD		Yes
INFF32[4]		IFD		Yes
INFF_F		IFD_F		Yes
INFF_FS		IFDI_F		Yes
INFF_S		IFDI		Yes
INFF_U		IFD_U		Yes
INLAT		ILD_1		Yes
INLAT4[4]		ILD_1		
INLAT8[4]		ILD_1		
INLAT16[4]		ILD_1		
INLAT32[4]		ILD_1		
INLAT_F		ILD_1F		Yes
INLAT_FS		ILDI_1F		Yes
INLAT_S		ILDI_1		Yes
INREG			Obsolete	
INREG4			Obsolete	
INREG8			Obsolete	
INREG16			Obsolete	
INREG32			Obsolete	
INREG_F			Obsolete	
INREG_FS			Obsolete	
INREG_S			Obsolete	
INV				No
LD[5]		LD_1		Yes
LDE[5]		LD_1		
LDRD[5]		LDC_1		Yes
LDSD[5]		LDP_1		Yes
MD0				No
MD1				No
MD2				No
MUX4_1H				No

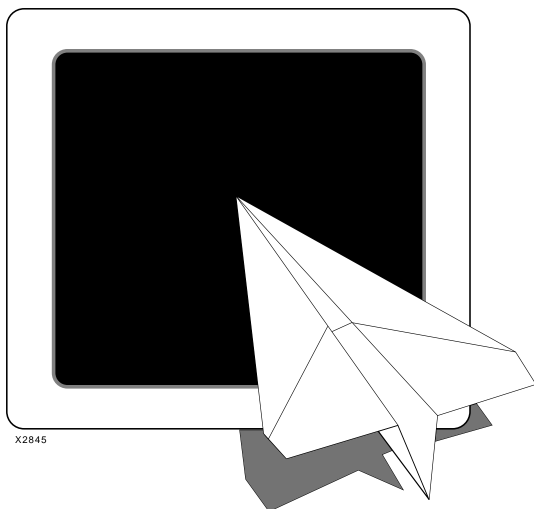
XC4000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
MUX8_1H				No
MUX16_1H				No
NAND2				Yes
NAND3				Yes
NAND4				Yes
NAND5				Yes
NOR2				Yes
NOR3				Yes
NOR4				Yes
NOR5				Yes
N_FLAG				No
OBUF				No
OBUF8[4]		OBUF		
OBUF16[4]		OBUF		
OBUF32[4]		OBUF		
OBUF_F				No
OBUFT				No
OBUFT_F				No
OR2				Yes
OR3				Yes
OR4				Yes
OR5				Yes
OSC4				No
OUTFF	OFD			No
OUTFF_F	OFD_F			No
OUTFF4_F[4]		OFD_F		
OUTFF8_F[4]		OFD_F		
OUTFF16_F[4]		OFD_F		
OUTFF32_F[4]				
OUTFF_S	OFDI			No
OUTFF_U	OFD_U			No
OUTFF_FS	OFDI_F			No

XC4000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
OUTFF_UF	OFD_FU			No
OUTFFT	OFDT			No
OUTFFT_F	OFDT_F			No
OUTFFT4_F[4]		OFDT_F		
OUTFFT8_F[4]		OFDT_F		
OUTFFT16_F[4]		OFDT_F		
OUTFFT32_F[4]		OFDT_F		
OUTFFT_S	OFDTI			No
OUTFFT_FS	OFDTI_F			No
PARE9H				No
PARO9H				No
PRSC8_9H				No
PULLDOWN				Yes
PULLUP				Yes
PWR	VCC			Yes
RAM16X1				No
RAM32X1				No
RD8H				No
RD16H				No
RDBK				No
RDCLK				No
READBACK				No
RM16X2H				No
RM16X4H				No
RM16X8H				No
RM32X4H				No
RM32X8H				No
RM64X4H				No
RM64X8H				No
RM128X4H				No
RM128X8H				No
RS8PH				No

XC4000 Primitive Name	Exact Unified Replacement	Closest Unified Replacement	Obsolete	Pin Change[1]
RS16PH				No
STARTUP				No
S_FLAG				No
TBUF	BUFT			No
TBUF_F		BUFT	Obsolete	
TBUF_P	BUFT			
TCK				No
TDI				No
TDO				No
TMS				No
WAND1				No
WAND1_D			Obsolete	
WAND1_F			Obsolete	
WAND1_FD			Obsolete	
WAND1_P			Obsolete	
WAND1_PD			Obsolete	
WOR2AND				No
WOR2AND_F			Obsolete	
WOR2AND_P			Obsolete	
XNOR2				Yes
XNOR3				Yes
XNOR4				Yes
XNOR5				Yes
XOR2				Yes
XOR3				Yes
XOR4				Yes
XOR5				Yes
X_FLAG				No

1. A pin change is a pin name change and/or a pin order change. Refer to Appendix B for the pin description.
2. The # symbol represents a bus width equal to 9, 15, 17, 21, 25, 29, 33, 37, 45, 49, 57, or 64.
3. The # symbol represents a bus width equal to 9, 15, 17, 21, 25, 29, 33, 37, 45, 49, or 57.

4. These primitives were provided for instantiation in the previous release of Synopsys; in the current version, Synopsys can automatically infer these primitives. The bused versions of I/O primitives have been removed from the libraries because manual instantiation is not recommended. For more information, see the "Configuring the IOBs" section in the "Using the Design Compiler" chapter or the "Using the FPGA Compiler" chapter.
5. The XC4000 libraries contain transparent low latches (LD_1, LDC_1, and LDP_1). Synopsys requires that the CLB latch match the transparent low IOB latch to support pad mapping. To obtain a transparent high latch, invert the G pin.



Index

Xilinx Synopsys Interface FPGA User Guide

Xilinx Synopsys Interface FPGA User Guide — December, 1994 (0401291 01) Printed in U.S.A.

Index

Symbols

.synopsys_dc.setup file *see* startup file
.synopsys_vss.setup, 7-2

Numerics

3-state buffers
  XC3000 devices, A-4
  XC4000 devices, B-4
3-state output
  Design Compiler, 6-15
  FPGA Compiler, 5-15
3-state output buffers
  XC4000 devices, B-11
  XC4000A devices, B-11
  XC4000H devices, B-12

A

ADD_SUB, 5-47, 6-55
All Inputs command, 5-12, 6-11
All Outputs command, 5-13, 6-12
Analyze File dialog box, 3-9, 4-8
Analyze window, 3-10, 4-9
AND gates
  XC3000 devices, A-3
  XC4000 devices, B-3
APR *see XACT Reference Guide*
area utilization report, 5-65
  creating, 3-24, 4-22
  output, 3-26, 4-24

B

bidi_reg.script
  Verilog, 5-25, 6-25
  VHDL, 5-23, 6-23
bidi_reg.v, 5-21, 6-21
bidi_reg.vhd, 5-20, 6-20
bidirectional buffers
  XC3000 devices, A-8
  XC4000 devices, B-14
  XC4000A devices, B-15
  XC4000H devices, B-15
bidirectional I/Os, inserting, 5-19, 6-19
BLKNM attributes
  purpose, 5-73
  removing, 5-73
boundary scan
  Design Compiler, 6-38
  FPGA Compiler, 5-37
BSCAN primitive, B-16
BSCAN symbol, 5-37, 6-39
buffers
  3-state
    XC3000 devices, A-8
    XC4000 devices, B-11
    XC4000A devices, B-11
    XC4000H devices, B-12
  bidirectional
    XC3000 devices, A-8
    XC4000 devices, B-14
    XC4000A devices, B-15
    XC4000H devices, B-15
  generic *see also* clock buffers, A-4, B-4
  input
    with D flip-flop, B-9
    with inverted latch, B-10
    XC3000 devices, A-7
    XC4000 devices, B-9
    XC4000H devices, B-9

output
XC3000 devices, A-7
XC4000 devices, B-10, B-13
XC4000A devices, B-10, B-14
XC4000H devices, B-11
Bus Selector dialog box, 3-17, 4-15

C

capacitive load, 5-14, 6-13
cells *see* primitives
CLBMAP primitives, A-9
CLBs
area report, 6-65
configuration report, 5-66
designs without hierarchy, 5-56
flattening the design, 5-54
generating schematic, 5-69
hierarchical designs, 5-55
implementing memory, 5-31, 6-32
mapped, 6-66
optimizing, 5-54, 6-55
removing mapping, 5-73
replacing with gates, 5-72
reset, global
using GSR net, 5-38, 6-39
using RESET pin, 6-48
timing report, 5-65, 6-66
unmapped, 6-66
utilization report, 5-64
clock buffers
alternate, A-6
controlling insertion, 5-27, 6-28
description, 5-26
determining how many, 5-30, 6-31
disabling insertion, 5-30, 6-31
global, A-6, B-8
instantiating, 5-27
primitives for instantiation, A-5, B-8
report, 6-31
XC3000/A/L devices, 6-27
XC3100/A devices, 6-27
XC4000/A/D/H devices, 6-27
clock constraint, setting, 3-20, 4-18
CLR pin, 5-39, 6-41
CMOS thresholds
setting input, 5-12, 6-10
setting output, 5-13, 6-12
Command window, 3-18, 4-16
command.log file, 3-6
COMPARE, 5-47, 6-55
Compile command, 5-54, 6-57
designs with hierarchy, 5-56
designs without hierarchy, 5-56
compiling a design, 5-53, 6-57
converting to gates, 3-29
evaluating the results, 3-22, 4-20
using Design Compiler, 4-19
using FPGA Compiler, 3-21
with
feedthroughs, 5-57, 6-58
hierarchy, 5-56
instantiated I/O cells, 5-57
without hierarchy, 5-56
XC3000 devices, 6-58
XC4000 devices, 5-57, 6-58
XC4000H devices, 5-60, 6-61
configuration, software, 2-1
constraints file *see XACT Libraries Guide*
count8.dly, 3-46, 4-44
count8.rpt, 3-39, 3-44, 4-42
count8.script
Verilog HDL, 3-34, 4-33
VHDL, 3-33, 4-32
count8.timing, 3-28, 4-27
count8.v, 3-6, 4-5
count8.vhd, 3-5, 4-4, 7-5
count8_tb.vhd, 7-8
Create Clock command, 5-50
CST file *see XACT Libraries Guide*

D

D flip-flops
XC3000 devices, A-5
XC4000 devices, B-7

DB file, 5-71, 6-67
DC Shell, 8-6
dc.script, 6-60
dc3k.synopsys_dc.setup, 2-12
dc4k.synopsys_dc.setup, 2-8
debugging, 5-66
decoders, B-5
Defaults dialog box, 3-7, 4-6
delays
    removing default, 5-26, 6-26
    timing, 5-65, 6-66
Derive Clocks command, 5-52
Design Analyzer
    description, 8-6
    Designs view, 3-24, 4-22
    executing script file, 3-31, 4-30
    exiting, 3-31, 4-29
    invoking, 3-6, 4-5
    Schematic view, 3-23, 4-21
    Symbol view, 3-15, 4-13
Design Compiler
    before you begin, 6-1
    clock buffer insertion, 6-26
    compiling a design, 6-57
        with feedthroughs, 6-58
        XC3000 devices, 6-58
        XC4000 devices, 6-58
        XC4000H devices, 6-61
    design flow
        on different platforms, 6-2
        on same platform, 6-2
    DesignWare directory, 2-10
    DesignWare library, 6-54
    EDIF parameters, 2-10, 2-14
    features, 1-1
    Global Set/Reset (GSR) net, 6-39
    intermediate files for VSS, 2-10, 2-14
    introduction, 6-1
    IOB configuration, 6-6
        3-state output, 6-15
        bidirectional I/Os, 6-19
        primitives inferred automatically, 6-6
        XC4000H, 6-10
    libraries, 6-2, 8-12
    part types, setting, 6-68
    READ parameters, 2-10, 2-14
    reports
        area, 6-65
        timing delays, 6-66
    saving design
        as DB file, 6-67
        as SEDIF file, 6-68
    search path, 2-9, 2-13
    slew rate
        XC3000/A/L devices, 6-14
        XC3100/A devices, 6-14
        XC4000/D devices, 6-8
        XC4000A devices, 6-9
        XC4000H devices, 6-12
    startup file (.synopsys_dc.setup), 2-8, 2-12
    symbol libraries, 2-10, 2-14
    Syn2XNF, running, 6-69
    synthetic library, 2-10
    tutorial, 4-1
    use with
        XACT software, 6-72
        XC3000 designs, 2-12
        XC4000 designs, 2-8
        XMake, 6-72
    wire-load models
        changing, 6-5
        setting, 6-4
    writing the design, 6-67
design flow
    Design Compiler, 4-3, 6-1
    FPGA Compiler, 3-4, 5-2
design implementation for VSS, 7-12
design optimization, 5-53
Design Optimization dialog box, 3-22, 4-20

designs
- analyzing
  - using Design Compiler, 4-7
  - using FPGA Compiler, 3-8
- compiling
  - using Design Compiler, 6-57
  - using FPGA Compiler, 5-53
- elaborating
  - using Design Compiler, 4-9
  - using FPGA Compiler, 3-10
- flattening, 5-54
- optimizing
  - using Design Compiler, 4-17
  - using FPGA Compiler, 3-19
- saving
  - as DB file, 5-71, 6-67
  - as SEDIF file, 6-68
  - as SXNF file, 5-74
- setting part type, 5-74, 6-68
- simulation, 7-1
- translating
  - using Syn2XNF, 5-74, 6-68
  - using XMake, 5-77, 6-72
- writing, 5-71

DesignWare library
- HDL operators, 5-47, 6-54
- installation, 2-2
- search path, 2-7, 2-11
- use with
  - Design Compiler, 6-54
  - FPGA Compiler, 5-47

directory tree, 8-1

Disconnect Net command, 5-46, 6-48

documentation
- XACT Development System, 1-3
- XSI, 1-2

Don’t Touch attribute
- on instantiated I/O buffers, 5-57
- on instantiated I/O cells, 5-21, 6-21
- use with Synopsys, B-2

Dynamic Waveform Viewer, 7-12, 7-18

E

EDIF parameters, 2-10, 2-14

Elaborate Design dialog box, 3-11, 4-10

Elaborate window, 3-12, 4-11

examples
- area utilization report, 5-65
- bidi_reg.script
  - Verilog, 5-25, 6-25
  - VHDL, 5-23, 6-23
- bidi_reg.v, 5-21, 6-21
- bidi_reg.vhd, 5-20, 6-20
- bidirectional I/O insertion
  - Verilog HDL, 5-25, 6-25
  - VHDL, 5-23, 6-23
- boundary scan symbol, 5-37, 6-39
- combinatorial design, 5-55
- count8.dly, 3-46, 4-44
- count8.rpt, 3-39, 3-44, 4-42
- count8.script
  - Verilog HDL, 3-34, 4-33
  - VHDL, 3-33, 4-32
- count8.timing, 3-28, 4-27
- count8.v, 3-6, 4-5
- count8.vhd, 3-5, 4-4, 7-5
- count8_tb.vhd, 7-8
- dc.script, 6-60
- fpga.script, 5-60
- gate_clock
  - Verilog, 5-28, 6-29
  - VHDL, 5-28, 6-29
- gated clock
  - after pad insertion, 5-29, 6-30
  - schematic, 5-29, 6-30
  - Verilog, 5-28, 6-29
  - VHDL, 5-28, 6-29
- greset
  - Verilog, 6-50
  - VHDL, 6-50
- gsr_ex
  - Verilog, 5-41, 6-43
  - VHDL, 5-41, 6-42

hierarchy, merging, 5-55
implementation flow
XC3000A, 4-3
XC4000, 3-4
implementing Global Reset
Verilog, 6-50, 6-54
VHDL, 6-50, 6-52
memory description file, 5-35, 6-37
no output register inferred, 5-17, 6-17
output register inferred, 5-19, 6-19
PPR report file, 3-44, 4-42
register not driving 3-state
Verilog, 5-17, 6-17
VHDL, 5-16, 6-16
Report Cell output, 5-69
ROM
behavioral Verilog, 5-34, 6-35
behavioral VHDL, 5-33, 6-34
rom_memgen.v, 5-36, 6-38
rom_memgen.vhd, 5-36, 6-37
rom16x4_4k
Verilog, 5-34, 6-35
VHDL, 5-33, 6-34
sample script file
XC3000A design, 6-60
XC4000 design, 5-60
XC4000H design (Verilog), 5-64, 6-65
XC4000H design (VHDL), 5-62, 6-63
sequential design, 5-55
test bench file, 7-8
three_ex1
Verilog, 5-17, 6-17
VHDL, 5-16, 6-16
three_ex2
Verilog, 5-19, 6-18
VHDL, 5-18, 6-18
three_ex2.script
Verilog, 5-64
VHDL, 5-62
thresholds, setting, 5-12
timing report, 3-46, 4-44
top_gsr script file
Verilog, 5-46, 6-48
VHDL, 5-44, 6-46
top_gsr.v, 5-42, 6-44
top_gsr.vhd, 5-42, 6-44
XDelay report, 3-46, 4-44
Execute File dialog box, 3-32, 4-31
exiting
Design Analyzer, 3-31, 4-29
Design Compiler tutorial, 4-2
FPGA Compiler tutorial, 3-2

F

fc4k.synopsys_dc.setup, 2-4
feedthroughs, 5-57, 6-58
file descriptions
.lca, 8-5
.mra, 8-5
.script, 8-4
.sedif, 8-5
.sim, 8-5
.sxnf, 8-4
.syn, 8-5
.v, 8-4
.vhd, 8-4
.xff, 8-5
.xnf, 8-5
directory structure, 8-1
XC3000.sdb, 8-4
XC4000.sdb, 8-4
flag cells
XC3000 devices, A-9
XC4000 devices, B-18
flip-flops
XC3000 devices, A-5
XC4000 devices, B-7
FMAP primitives, B-18
FMAP symbols, removing, 5-73
FPGA Compiler
before you begin, 5-2

clock buffer insertion, 5-26
compiling a design
purpose, 5-53
with feedthroughs, 5-57
with instantiated I/O cells, 5-57
XC4000 devices, 5-57
XC4000H devices, 5-60
design flow
on different platforms, 5-3
on same platform, 5-2
DesignWare directory, 2-5
DesignWare library
purpose, 5-47
search path, 2-11
features, 1-2
Global Set/Reset (GSR) net, 5-38
hierarchical schematic, 5-69
intermediate files for VSS, 2-5
introduction, 5-1
invoking, 3-21
IOB configuration, 5-8
3-state output, 5-15
bidirectional I/Os, 5-19
XC4000/A/D, 5-8
XC4000H, 5-11
libraries, 8-11
mapping, 5-73
operating conditions, 5-7
part types, setting, 5-74
READ parameters, 2-6
reports
area, 5-64
configuration, 5-66
debugging, 5-66
timing delays, 5-65
saving design
as DB file, 5-71
as SXNF file, 5-74
search path, 2-5
slew rate
XC4000/D devices, 5-10
XC4000A devices, 5-10
XC4000H devices, 5-14
startup file, 2-4
symbol libraries, 2-5
Syn2XNF, running, 5-74
synthetic library, 2-5
timing delays, 5-65
timing specifications, 5-50
tutorial, 3-1
use with
XACT software, 5-77
XC3000 devices, 5-4
XMake, 5-77
wire-load models
changing, 5-6
setting, 5-5
writing the design, 5-71
XNF Writer, 5-73
FPGA Compiler dialog box, 3-21
fpga.script, 5-60
functional simulation
analyzing files, 7-9
declaring
configuration, 7-10
signals, 7-12
displaying waveforms, 7-12
invoking VHDL debugger, 7-9
preparing for, 7-5

G

gate_clock
Verilog, 5-28, 6-29
VHDL, 5-28, 6-29
gated clock
after pad insertion, 5-29, 6-30
example, 5-27
report, 5-30, 6-31
schematic, 5-29, 6-30
gates
3-state buffers, A-4, B-4
AND/OR, A-3, B-3
buffers, A-4, B-4

inverters, A-4, B-4
pull-down resistor, B-6
pull-up resistors, A-4, B-6
wide I/O decoders, B-5
wired OR-AND, B-5
wired-AND, B-5
global clock buffers *see* clock buffers
Global Reset script file
Verilog, 6-54
VHDL, 6-52
Global Set/Reset net *see* GSR net
GND primitive
XC3000 devices, A-10
XC4000 devices, B-18
greset
Verilog, 6-50
VHDL, 6-50
greset.script
Verilog, 6-54
VHDL, 6-52
ground *see* GND primitive
GSR net
Design Compiler
changing states, 6-41
function, 6-39
GSR pin, 6-43
increasing performance, 6-42
initial state after configuration, 6-40
Preset vs. Direct Clear, 6-40
RESET pin, using, 6-48
startup state, 6-39
XC3000 devices, 6-48
XC3100 devices, 6-48
XC4000 devices, 6-39
FPGA Compiler
changing states, 5-40
function, 5-38
GSR pin, 5-42
increasing performance, 5-40
initial state after configuration, 5-38
Preset vs. Direct Clear, 5-39
startup state, 5-38
functional simulation, use in, 7-5
timing simulation, use in, 7-3
GSR pin, 5-42, 6-43
gsr_ex
Verilog, 5-41, 6-43
VHDL, 5-41, 6-42

H

hard macros
accumulators, B-20
adders/subtracters, B-21
comparators, B-21
converting to RPMs, B-1
counters, B-22
decoders, B-22
encoders, B-23
multiplexers, B-23
parity generators, B-24
prescalers, B-24
RAM, B-25
registers, B-24
shift registers, B-27
hazard warnings, turning off for VSS, 7-2
HDL operators, 5-47, 6-54
hierarchical designs
compiling, 5-56
creating unique instance names, 2-6, 5-57
flattening, 5-54
generating a schematic, 5-69
for CLBs and IOBs, 5-70
for function generators, 5-70
merging into single level, 5-55
optimizing logic, 5-54
saving and writing, 5-72
HMAP primitives, B-18
HMAP symbols, removing, 5-73

I

I/O buffers
- defining input ports as pads, 3-14, 4-13
- defining output ports as pads, 3-16, 4-15
- inserting, 3-13, 4-12
- using Insert Pads, 3-17, 4-16

I/O primitives
- for instantiation, A-7, B-8
- libraries, 8-11

implementation flow
- Design Compiler, 4-3
- FPGA Compiler, 3-4

INC_DEC, 5-47, 6-55

initialization state
- after configuration, 5-39, 6-40
- changing, 5-40, 6-41

input buffers
- with D flip-flop, B-9
- with inverted latch, B-10
- XC3000 devices, A-7
- XC4000 devices, B-9
- XC4000H devices, B-9

Input Port Attributes dialog box, 3-16, 4-14

Insert Pads command
- to insert clock buffers, 5-26
- to insert I/O buffers, 3-17, 4-16, 5-8, 6-7

installation
- DesignWare library, 2-2
- verifying, 2-1
- XACT software, 2-1
- X-BLOX, 2-2
- XSI *see also* release notes, 2-2

inverters
- XC3000 devices, A-4
- XC4000 devices, B-4

IOBs
- 3-state output, 5-15, 6-15
- bidirectional, 5-19, 6-19
- configuring, 5-7, 6-6
- initializing flip-flop to Preset, 5-26, 6-26
- pad locations, 5-15, 6-15
- removing default delay, 5-26, 6-26
- unbonded, 5-25, 6-25
- XC3000/A/L
  - description, 6-13
  - inputs, 6-14
  - outputs, 6-14
- XC3100/A
  - description, 6-13
  - inputs, 6-14
  - outputs, 6-14
- XC4000/A/D
  - description, 5-8, 6-7
  - inputs, 5-8, 6-7
  - outputs, 5-9, 6-8
- XC4000H
  - description, 5-11, 6-10
  - inputs, 5-12, 6-10
  - outputs, 5-13, 6-11

L

latches
- XC3000 devices, A-5
- XC4000 devices, B-7

LCA file, 8-5

libraries
- descriptions
  - xblox_4000.sldb, 8-8
  - xdc_*family-s*.db, 8-8
  - xfpga_*family-s*.db, 8-8
  - xgen_3000.db, 8-7
  - xgen_4000.db, 8-7
  - xio_*4kparttype-s*.db, 8-8
  - xprim_*family-s*.db, 8-7
  - xprim_*parttype-s*.db, 8-7
- I/O primitives, 8-11
- mapped, 6-3, 6-66
- unmapped, 6-3, 6-66
- xdc, 8-12
- xfpga, 8-11
- xprim, 8-9

library paths to FTGS models, 7-2

link libraries, default, 2-5, 2-9, 2-13

M

mapping
- by XNF Writer, 5-73
- obsolete primitives to new, C-1
- removing
  - BLKNM attributes, 5-73
  - FMAP and HMAP symbols, 5-73

Max Period command, 5-51
MemGen
- implementing memory, 5-31
- instantiating ROM submodule, 5-34, 6-35
- memory description file, 5-35, 6-36

memory
- functional simulation, 5-32, 6-33
- migrating between FPGAs, 5-32, 6-33
- RAMs, 5-31, 6-32
- ROMs, 6-33
  - implementing, 5-32
  - instantiating, 6-33
  - Verilog HDL example, 5-34, 6-35
  - VHDL example, 5-33, 6-34
- using, 5-31, 6-32

memory description file, 5-35, 6-36
MRA file, 8-5

N

NODELAY attribute, 5-9, 5-26, 6-8, 6-26

O

operating conditions, 5-7, 6-6
optimizing the design, 5-53, 6-57
- using Design Compiler, 4-17
- using FPGA Compiler, 3-19

OR gates
- XC3000 devices, A-3
- XC4000 devices, B-3

oscillator primitives, A-6
OUT file, 3-37, 4-36
output buffers
- 3-state
  - XC3000 devices, A-8
  - XC4000 devices, B-11
  - XC4000A devices, B-11
  - XC4000H devices, B-12
- XC3000 devices, A-7
- XC4000 devices, B-10, B-13
- XC4000A devices, B-10, B-14
- XC4000H devices, B-11

Output Port Attributes dialog box, 3-17, 4-15

P

pad locations, 5-15, 6-15
part types
- changing, 5-77
- displaying, 5-76
- setting, 5-74, 6-68
- supported, 8-9
- unsupported, 8-12

pin order
- XC3000 primitives, A-3

power *see* VCC primitive
PPR *see XACT Reference Guide*
PPR constraints file *see XACT Libraries Guide*
PRE pin, 5-39, 6-41
pre-layout timing, estimating, 3-18, 4-16
primary clock buffer *see* clock buffers
primitive libraries, 8-9
primitives
- basic gates
  - 3-state buffer, A-4, B-4
  - AND/OR, A-3, B-3
  - buffers, A-4, B-4
  - inverters, A-4, B-4
  - pull-down resistor, B-6
  - pull-up resistors, A-4, B-6
  - wide I/O decoders, B-5
  - wired OR-AND, B-5
  - wired-AND, B-5
- bidirectional buffers
  - XC3000 devices, A-8

XC4000 devices, B-14
XC4000A devices, B-15
XC4000H devices, B-15
boundary scan (BSCAN), B-16
cell name suffixes, A-2, B-2
CLBMAPs, A-9
clock buffers
XC3000 devices, A-5
XC4000 devices, B-8
DesignWare modules, B-19
flag cells
XC3000 devices, A-9
XC4000 devices, B-18
flip-flops
XC3000 devices, A-5
XC4000 devices, B-7
FMAP and HMAP, B-18
ground
XC3000 devices, A-10
XC4000 devices, B-18
input/output, A-7, B-8
instantiation, A-1, B-1
latches
XC3000 devices, A-5
XC4000 devices, B-7
mapping to Unified Libraries, C-1
naming conventions
XC3000 devices, A-2
XC4000 devices, B-2
obsolete, C-1
oscillators, A-6
power
XC3000 devices, A-10
XC4000 devices, B-18
RAM/ROM, B-6
readback
RDBK, B-17
RDCLK, B-17
READBACK, B-17
registers *see* flip-flops or latches
special functions, A-9, B-16
startup, B-17
unbonded, 5-25
program descriptions
DC shell, 8-6
Design Analyzer, 8-6
Syn2XNF, 8-6
Synlibs, 8-6
Vhdlan, 8-6
Vhdldbx, 8-6
XMake, 8-6
XNF2VSS, 8-6
PRP file, 3-37, 4-36
pull-down resistors, 5-25, 6-25, B-6
pull-up resistors, 5-25, 6-25
XC3000 primitive, A-4
XC4000 primitive, B-6

R

RAM primitives, B-6
RAM *see* memory
READ parameters
Design Compiler, 2-10, 2-14
FPGA Compiler, 2-6
readback primitive, B-17
README file, 2-2
registers, initial states *see also* flip-flops or latches, 7-3
Replace FPGA command
hierarchy level
for CLBs and IOBs, 5-70
for function generators, 5-71
replacing CLBs and IOBs with gates, 5-72
Report Area command, 6-65
Report Cell command
for determining
clock buffers, 6-31
types of flip-flops, 5-39
generating configuration report, 5-67
Report dialog box, 3-25, 4-23
Report FPGA command
area report, 5-64

determining clock buffers, 5-30
Report Timing command, 5-65, 6-66
reports
area, 5-64, 6-65
configuration, 5-66
debugging, 5-66
saving, 3-27, 4-25
timing, 5-65, 6-66
RESET pin, 6-48
resistive load, 5-14, 6-13
ROM primitives, B-6
ROM *see* memory
rom_memgen.v, 5-36, 6-38
rom_memgen.vhd, 5-36, 6-37
rom16x4_4k
Verilog, 5-34, 6-35
VHDL, 5-33, 6-34
RPT file, 3-39, 4-37
RST net, 5-46

S

Save File dialog box, 3-30, 4-29
saving your design
as DB file, 3-29, 4-28
as SEDIF file, 4-28
as SXNF file, 3-30
SCRIPT file, 8-4
script files, executing, 3-31, 4-30
search paths
Design Compiler, 2-11, 2-14
FPGA Compiler, 2-6
secondary clock buffer *see* clock buffers
SEDIF file, 6-68, 8-5
Set Attribute command
changing initial states, 5-40, 6-41
instantiating ROM primitives, 5-32, 6-33
removing FMAP and HMAP symbols, 5-73
setting part type, 5-74, 6-68
specifying pad locations, 5-15, 6-15
Set False Path command, 5-52
Set Input Delay command, 5-51
Set Max Delay command, 5-51
Set Output Delay command, 5-51
Set Pad Type command
disabling clock buffers, 5-30, 6-31
setting slew rate
XC3000/A/L devices, 6-14
XC3100/A devices, 6-14
XC4000/D devices, 5-10, 6-9
XC4000A devices, 5-10, 6-9
XC4000H devices, 5-14, 6-13
specifying clock buffers, 6-28
threshold settings
input, 5-12, 6-11
output, 5-13, 6-12
Set Port Is Pad command, 5-8
defining ports as pads, 6-7
use on instantiated I/Os, 5-21, 6-21
Set Wire Load command, 5-6, 6-5
setup file *see* startup file
SIM files
description, 8-5
locating, 2-2
simulation, 7-1
.synopsys_vss.setup, 7-2
compiling the design, 7-13
design implementation, 7-12
functional, 7-9
Global Set/Reset (GSR) function, 7-3
functional simulation, 7-5
timing simulation, 7-4
hazard warnings, turning off, 7-2
initial states of registers, 7-3
introduction, 7-1
library path to FTGS models, 7-2
post-layout XNF file, creating, 7-13
preparing timing model, 7-14
recommended strategy, 7-1
setting timebase and resolution factors, 7-2
source file considerations, 7-3

test bench file, creating, 7-6
timing, 7-14
Vhdlan command, 7-9
Vhdldbx command, 7-9
VSS setup file, 7-2
warnings, turning off, 7-2
WORK library, 7-3
Xilinx netlist, creating, 7-13
XNF2VSS command, 7-14
slew rate
XC3000/A/L
attributes, 6-15
setting, 6-14
XC3100/A
attributes, 6-15
setting, 6-14
XC4000/D
attributes, 5-10, 6-9
setting, 5-9, 6-9
XC4000A
attributes, 5-11, 6-10
setting, 5-10, 6-9
XC4000H
attributes, 5-14, 6-13
setting, 5-14, 6-12
software configuration, 2-1
Specify Clock dialog box, 3-20, 4-18
speed grades
supported, 8-9
unsupported, 8-12
startup file
defaults
Design Compiler, 2-3
FPGA Compiler, 2-3
renaming, 2-3
description, 8-4
Design Compiler
contents, 2-9, 2-13
DesignWare directory, 2-10
DesignWare library search path, 2-11
EDIF parameters, 2-10, 2-14
examples, 2-8, 2-12
intermediate files for VSS, 2-10, 2-14
READ parameters, 2-10, 2-14
search path, 2-9, 2-11, 2-13, 2-14
symbol libraries, 2-10, 2-14
synthetic library, 2-10
FPGA Compiler
contents, 2-4
default libraries, 2-13
DesignWare directory, 2-5
DesignWare library search path, 2-7
examples, 2-4
intermediate files for VSS, 2-5
READ parameters, 2-6
search path, 2-5, 2-6
symbol libraries, 2-5
synthetic library, 2-5
unique instance names, 2-6
STARTUP primitive, B-17
startup state, 5-38
STARTUP symbol, 5-38, 5-42, 6-39, 6-43
suffixes, primitive names, A-2, B-2
SXNF file, 5-74, 8-4
symbol libraries
Design Compiler, 2-10, 2-14
FPGA Compiler, 2-5
SYN files
description, 8-5
locating, 2-2
Syn2XNF
help, 5-76, 6-70
input files, 5-75, 6-69
invoking
by XMake, 5-78, 6-72
via command line, 5-75, 6-69
mapped libraries, using, 6-71
options
abbreviations, 5-76, 6-70

–dir, 5-76, 6-70
–force, 5-76, 6-70
–help, 5-76, 6-70
–l, 5-76, 6-70
–map, 6-71
–out, 5-77, 6-71
–parttype, 5-77, 6-71
–sub, 6-71
output file name, specifying, 5-77, 6-71
output files, 5-75, 6-70
overwrite existing XNF file, 5-76, 6-70
part types
changing, 5-77, 6-71
displaying valid, 5-76, 6-70
purpose, 5-74, 6-68
syntax, 5-75, 6-69
Synlibs
description, 8-6
Design Compiler
output, 2-15
syntax, 2-11, 2-15
target and link libraries, 2-10, 2-14
use with startup file, 2-12, 2-15
FPGA Compiler
output, 2-7
syntax, 2-7
target and link libraries, 2-5
use with startup file, 2-8
Synopsys startup file *see* startup file
synopsys_dc.setup file *see* startup file
synthetic library
Design Compiler, 2-10
FPGA Compiler, 2-5
system configuration, 2-1

T

target libraries, default, 2-5, 2-9, 2-13
test bench file
configuration declaration, 7-8
initializing registers, 7-6
purpose, 7-6
three_ex1
Verilog, 5-17, 6-17
VHDL, 5-16, 6-16
three_ex2
Verilog, 5-19, 6-18
VHDL, 5-18, 6-18
three_ex2.script
Verilog, 5-64
VHDL, 5-62
Tikpid, 5-66
timebase, setting for VSS, 7-2
timing constraints *see* timing specifications
timing delays, 5-65, 6-66
creating report, 3-26, 4-25
estimating, 3-18, 4-16
report output, 3-28, 4-27
timing model, 7-14
timing simulation, 7-14
back-annotation, 7-15
invoking VSS simulator, 7-15
preparing for, 7-4
source file analysis, 7-15
test bench analysis, 7-15
viewing waveforms, 7-18
timing specifications
clock constraint, 3-19
controlling how written, 5-52
Create Clock command, 5-50
creating defaults, 5-53
Derive Clocks command, 5-52
Max Period command, 5-51
overwriting using CST file, 5-50
path types, 5-50
purpose, 5-50
reporting timing delays, 5-65
Set False Path command, 5-52
Set Input Delay command, 5-51
Set Max Delay command, 5-51
Set Output Delay command, 5-51
setting, 5-50
XNFout Constraints Per Endpoint, 5-52

XNFout Default Time Constraints, 5-53
top_gsr script file
Verilog, 5-46, 6-48
VHDL, 5-44, 6-46
top_gsr.v, 5-42, 6-44
top_gsr.vhd, 5-42, 6-44
Tpickd, 5-66
tri-state buffers *see* 3-state buffers
tri-state output buffers *See* 3-state output buffers
tri-state output *see* 3-state output
TTL thresholds
setting input, 5-12, 6-10
setting output, 5-13, 6-12
tutorials
Design Compiler, 4-1
FPGA Compiler, 3-1

U

unbonded IOBs, 5-25, 6-25
Ungroup command, 5-56

V

V file, 8-4
VCC primitive
XC3000 devices, A-10
XC4000 devices, B-18
VHD file, 8-4
Vhdlan program
description, 8-6
how to use, 7-9
Vhdldbx program
description, 8-6
how to use, 7-9
voltage levels
setting input, 5-12, 6-11
setting output, 5-13, 6-12
VSS *see* simulation
VSS setup file, 7-2

W

wide I/O decoders, B-5
wired OR-AND primitive, B-5
wired-AND primitive, B-5
wire-load models
changing, 5-6, 6-5
default, 5-5, 6-3
description, 5-7, 6-6
determining block delay, 5-7, 6-6
setting, 5-5, 6-3
XC3000 devices, 5-5, 6-4
XC3000A/L devices, 5-6, 6-4
XC3100/A devices, 5-6, 6-5
XC4000/A/H devices, 5-5, 6-4
work library for VSS, 7-2
Write command, 5-72, 6-68

X

XACT software
use with
Design Compiler, 6-72
FPGA Compiler, 5-77
verifying installation, 2-1
XACT-Performance *see* timing specifications
X-BLOX, 1-1
DesignWare library
search path, 2-7
using, 5-47, 6-54
disabling, 2-7
GSR net, use with, 5-40
installation, verifying, 2-2
X-BLOX modules
compiling, 5-56
HDL operators, 5-47, 6-54
implementation, 5-48
maximum size before wrapping, 5-48, 6-56
naming conventions, B-19
timing, 5-48, 6-55
xblox_4000.sldb library, 8-8
XChecker, 3-47, 4-45
xdc libraries, 8-12
xdc_*family-s*.db libraries, 8-8
XDelay

invoking, 3-46, 4-44
purpose, 3-45, 4-43
report file, 3-46, 4-44
XFF file, 8-5
xfpga libraries, 8-11
xfpga_*family-s*.db libraries, 8-8
xgen_3000.db library, 8-7
xgen_4000.db library, 8-7
xio libraries, 8-11
xio_*4kparttype-s*.db libraries, 8-8
xlnx_hier_blknm=1, 5-57
XMake
description, 8-6
output files, 3-37, 4-35
report file, 3-39, 4-37
using
on different platforms, 5-78, 6-72
on same platform, 5-78, 6-72
XNF file, 8-5
XNF Writer, 5-73
XNF2VSS, 7-8, 7-14
XNF2VSS program
description, 8-6
how to use, 7-14
XNFout Constraints Per Endpoint, 5-52
XNFout Default Time Constraints, 5-53
xprim libraries, 8-9
xprim_*family-s*.db libraries, 8-7
xprim_*parttype-s*.db libraries, 8-7
XSI
installation, verifying *see also* release notes, 2-2
introduction, 1-1