

***Getting Started with  
Xilinx EPLDs***

***Designing with EPLDs***

***Compiling Your Design***

***Fitting Your Design***

***Simulating Your  
Design***

***EPLD Architecture***

***Library Component  
Specifications***

***Attributes***

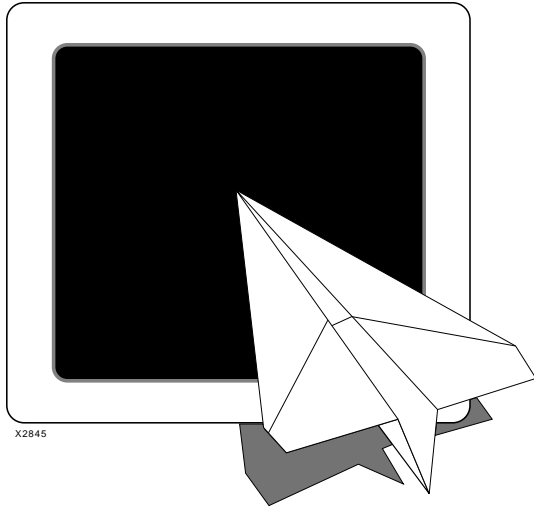
# ***Xilinx Synopsys Interface EPLD User Guide***

**XILINX**<sup>®</sup>, XACT, XC2064, XC3090, XC4005, and XC-DS501 are registered trademarks of Xilinx. All XC-prefix product designations, XACT-Performance, XAPP, X-BLOX, XChecker, XDM, XDS, XEPLD, XFT, XPP, XSI, BITA, Configurable Logic Cell, CLC, Dual Block, FastCLK, HardWire, LCA, Logic Cell, LogicProfessor, MicroVia, PLUSASM, UIM, VectorMaze, and ZERO+ are trademarks of Xilinx. The Programmable Logic Company and The Programmable Gate Array Company are service marks of Xilinx.

IBM is a registered trademark and PC/AT, PC/XT, PS/2 and Micro Channel are trademarks of International Business Machines Corporation. DASH, Data I/O and FutureNet are registered trademarks and ABEL, ABEL-HDL and ABEL-PLA are trademarks of Data I/O Corporation. SimuCad and Silos are registered trademarks and P-Silos and P/C-Silos are trademarks of SimuCad Corporation. Microsoft is a registered trademark and MS-DOS is a trademark of Microsoft Corporation. Centronics is a registered trademark of Centronics Data Computer Corporation. PAL and PALASM are registered trademarks of Advanced Micro Devices, Inc. UNIX is a trademark of AT&T Technologies, Inc. CUPL is a trademark of Logical Devices, Inc. Apollo and AEGIS are registered trademarks of Hewlett-Packard Corporation. Mentor and IDEA are registered trademarks and NETED, Design Architect, QuickSim, QuickSim II, and EXPAND are trademarks of Mentor Graphics, Inc. Sun is a registered trademark of Sun Microsystems, Inc. SCHEMA II+ and SCHEMA III are trademarks of Omaton Corporation. OrCAD is a registered trademark of OrCAD Systems Corporation. Viewlogic, Viewsim, and Viewdraw are registered trademarks of Viewlogic Systems, Inc. CASE Technology is a trademark of CASE Technology, a division of the Teradyne Electronic Design Automation Group. DECstation is a trademark of Digital Equipment Corporation. Synopsys is a registered trademark of Synopsys, Inc. Verilog is a registered trademark of Cadence Design Systems, Inc.

Xilinx does not assume any liability arising out of the application or use of any product described herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. cannot assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx products are protected under at least the following U.S. patent: 5,224,056. Xilinx, Inc. does not represent that Xilinx products are free from patent infringement or from any other third-party right. Xilinx assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx will not be liable for the accuracy or correctness of any engineering or software or assistance provided to a user.

Xilinx products are not intended for use in life support appliances, devices, or systems. Use of a Xilinx product in such applications without the written consent of the appropriate Xilinx officer is prohibited.



***XEPLD Files***

***Design Example***

***Fitter Reports***

***Glossary***

# ***Xilinx Synopsys Interface EPLD User Guide***

*Xilinx Synopsys Interface EPLD User Guide*



# Preface

---

## About This Manual

The Xilinx Synopsys Interface (XSI) allows you to implement Xilinx EPLD designs created with the Synopsys Design Compiler software. The Synopsys HDL synthesis software creates circuit designs from hardware description languages such as VHDL and Verilog HDL, using component libraries supplied by Xilinx for the XC7000 EPLD family.

This manual shows you how to use the XSI software, along with the Synopsys Design Compiler and VSS Simulator, to create efficient designs for Xilinx EPLDs. It is assumed that you are familiar with the Synopsys software. Please refer to the Synopsys manuals for detailed information on how to use the Synopsys software.

## Manual Contents

The following is a brief overview of the contents of each chapter.

- Chapter 1, “Getting Started with Xilinx EPLDs,” presents the basic EPLD design flow along with a brief example. This is a “road map” showing you each step of the process. Each step is explained in more detail in subsequent chapters.
- Chapter 2, “Designing with EPLDs,” shows you how to use the architecture specific features of Xilinx EPLDs to achieve the highest performance and logic density. You should read this chapter before creating your design.
- Chapter 3, “Compiling Your Design,” shows you how to use Design Compiler with the Xilinx EPLD Synopsys Interface software.

- Chapter 4, “Fitting Your Design,” provides a description of fitter operation and shows you how to select a target device, fit your design into the device, create a device programming file, and save your pinouts for design iteration.
- Chapter 5, “Simulating Your Design,” shows you how to perform functional and timing simulation of your design using the Synopsys VSS simulator.
- Appendix A, “EPLD Architecture,” provides an overview of the XC7000 family along with device selection information
- Appendix B, “Library Component Specifications,” provides the specifications of the Xilinx components including instantiated, inferable, and scalable cells.
- Appendix C, “Attributes,” provides the specifications of the Xilinx attributes which are used to control the fitting of your design.
- Appendix D, “XEPLD Files and Programs,” provides a description of each file and program used in the Xilinx software.
- Appendix E, “Design Example,” provides a thorough example demonstrating the key capabilities of the software.
- Appendix F, “Fitter Reports,” provides example reports showing you the type of information available for design analysis.
- Appendix G, “Glossary,” provides the definitions of terms and phrases that may be unfamiliar to you.

## **Xilinx Software Features**

The Xilinx Synopsys Interface (XSI) has the following features for EPLD design development:

- Supports all EPLD devices including the XC7272 and the new XC7336 and XC7318.
- Design synthesis using Synopsys VHDL or Verilog HDL.
- Design compilation using either Synopsys Design Compiler or FPGA Compiler.
- High-level inferencing of +, -, <, <=, =, >=, > operators using the EPLD high-speed arithmetic carry chain for operands of any width.

- VHDL functional simulation of original source designs (including all XC7000-specific library components).
- VHDL full-timing simulation (post-fitting) using device port declarations from the original source design.
- Static Timing Report produced by the XEPLD Fitter (post-fitting).
- Attributes for controlling register initial states.
- Attributes for assigning pin locations.
- Attributes for allocating logic to EPLD Fast Function Blocks and Fast Inputs.
- Attributes for controlling XEPLD optimization of clocks, input pad registers, output enable signals, register initial states, and UIM-AND functions.

## Unsupported Features

XSI currently has the following limitations for EPLD design development:

- No technology-specific optimization (for speed or density) is performed by the Synopsys synthesizer; all such optimization is performed by the Xilinx EPLD Translator Core Tool (XEPLD).
- No timing or area information is contained in the XC7000 library. Therefore, no timing or area estimation is available from the Synopsys Design Compiler. Timing and resource utilization results are available from XEPLD after completion of fitting (fitnet).
- The XEPLD fitter (v5.0) currently does not support timing-constraint-driven optimization; Synopsys timing constraints have no effect on EPLD design processing. Instead, use the `F` attribute to designate EPLD Fast Function Block and Fast Input resources.
- Verilog simulation is not supported by this package.





## 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.
<b>Courier font bold</b>	Literal commands that you must enter in syntax statements are in bold Courier font.
<i>italic font</i>	Variables that you replace in syntax statements are in italic font.
[ ]	Square brackets denote optional items or parameters. However, in bus specifications, such as bus [7:0], they are required.
{ }	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 Getting Started with Xilinx EPLDs

Creating Synopsys Setup Files.....	1-1
The Design Compiler Setup File.....	1-2
The VSS Simulator Setup File (.synopsys_vss.setup) .....	1-2
Verifying Your Installation .....	1-3
Verifying Synopsys Software Access .....	1-3
Verifying Xilinx Software Access .....	1-3
Verifying Your File Structure.....	1-4
Xilinx EPLD Design Flow .....	1-5
Design Example.....	1-5
Design Entry .....	1-9
Step1 — Create a Design Directory .....	1-9
Functional Simulation .....	1-9
Step 2 — Analyze Your Design .....	1-10
Step 3 — Analyze Your Test Bench .....	1-10
Step 4 — Invoke the Simulator.....	1-14
Step 5 — Run the Debugger .....	1-14
Step 6 — Trace Signals.....	1-15
Step 7 — Run the Simulation .....	1-16
Step 8 — Return to UNIX .....	1-16
Synthesizing Your Design (Compiling) .....	1-16
Step 9 — Enter the DC Shell Environment.....	1-16
Step 10 — Analyze Your Source Design.....	1-17
Step 11 — Elaborate Your Design .....	1-17
Step 12 — Synthesize Your Design .....	1-18
Step 13 — Place I/O Buffer Cells .....	1-18
Step 14 — Specify a Target Device .....	1-18
Step 15 — Specify Initial Register States.....	1-18
Step 16 — Output the Netlist.....	1-19
Step 17 — Exit DC Shell .....	1-19

Preparing the Netlist .....	1-19
Step 18 — Create a Flattened Netlist .....	1-19
Fitting Your Design .....	1-20
Step 19 — Fit Your Design .....	1-20
Timing Backannotation .....	1-21
Step 20 — Create a Static Timing Report .....	1-21
Timing Simulation .....	1-21
Step 21 — Analyze Your Original Design .....	1-21
Step 22 — Analyze Your Back-Annotated Design .....	1-21
Step 23 — Analyze Your Test Bench .....	1-22
Step 24 — Invoke the VSS Simulator .....	1-22
Step 25 — Open the Waveform Viewer .....	1-23
Step 26 — Run the Simulation .....	1-24
Step 27 — Return to UNIX .....	1-24
Programming an EPLD .....	1-24
Step 28 — Program an EPLD .....	1-25

## **Chapter 2   Designing with EPLDs**

VHDL Design File Requirements .....	2-1
Using Registers and Latches .....	2-1
Preventing Register/Latch Optimization .....	2-2
Using Input Pad Registers .....	2-2
Using Macrocell Registers .....	2-2
Using Input Pad Latches .....	2-3
Using Macrocell Latches .....	2-3
Specifying Register/Latch Initial States .....	2-3
Specifying the Predefined Initial States .....	2-4
Specifying Initial States for Individual Registers/Latches ..	2-4
Using I/O Ports .....	2-5
Selecting 3-State Control Sources .....	2-6
Assigning Specific Fast Output Enable Signals .....	2-6
Preventing FOE Optimization .....	2-6
Using Special Logic Functions .....	2-6
Binary Up Counters .....	2-6
Binary Down Counters .....	2-7
Binary Up/Down Counters .....	2-7
State Machines .....	2-7
Registered Arithmetic Functions .....	2-8
Comparators .....	2-8
Targeting a Specific Device .....	2-9
Specifying a Device .....	2-10

Using the Synopsys Part Attribute .....	2-11
Using the Xilinx Syn2EPLD Command.....	2-11
Specifying Pin Locations.....	2-11
Controlling Design Performance .....	2-12
Using High-Speed Clocks.....	2-12
Assigning Specific High-Speed Clocks.....	2-12
Preventing FastCLK Optimization .....	2-13
Selecting EPLD Function Block Types .....	2-13
Specifying High-Speed Paths.....	2-13
Specifying High-Density Paths .....	2-13
Using EPLD FastInputs .....	2-14
Selecting Low-Power Operation .....	2-14
The Design Rule Checker.....	2-14
General Design Rule Violations.....	2-14
Pad Component Design Rule Violations.....	2-15
FastCLK, Clock Enable, and Fast Output Enable Violations ..	2-15
 <b>Chapter 3 Compiling Your Design</b>	
Using Synopsys DC Shell .....	3-1
Step 1 — Entering the DC Shell Environment .....	3-2
Step 2 — Analyzing the Design .....	3-2
Step 3 — Elaborating the Design .....	3-2
Step 4 — Compiling Your Design .....	3-3
Step 5 — Defining EPLD I/O Signals.....	3-3
Step 6 — Specifying Attributes .....	3-4
Step 7 — Writing the Netlist.....	3-4
 <b>Chapter 4 Fitting Your Design</b>	
Fitter Overview .....	4-1
Fitter Operation .....	4-2
Step 1 — Create a Flattened XNF Netlist File .....	4-2
Specifying a Target Device.....	4-2
Using a Target Device Specified By a Part Attribute .....	4-3
Step 2 — Fit Your Design .....	4-3
Options .....	4-3
Step 3 — Verify Your Design Timing .....	4-4
Step 4 — Create a Device Programming File.....	4-5
Step 5 — Save Your Pinouts .....	4-5
 <b>Chapter 5 Simulating Your Design</b>	
Recommended EPLD Simulation Strategy .....	5-1

Controlling the Initial States of Registers .....	5-2
Simulating Master Reset.....	5-2
Preparing for Timing Simulation .....	5-3
Preparing for Functional Simulation.....	5-3
Creating a Test Bench File.....	5-4
Initializing Registers .....	5-4
Configuration Declaration .....	5-5
Functional Simulation.....	5-6
Design Implementation .....	5-9
Preparing the Timing Model.....	5-10
Timing Simulation.....	5-11

## **Appendix A EPLD Architecture**

Device Selection .....	A-2
The Universal Interconnect Matrix .....	A-3
High-Density Function Blocks .....	A-3
Shared and Private Product Terms.....	A-4
Arithmetic Logic Unit .....	A-5
Carry Lookahead (7300 Family Only) .....	A-6
Macrocell Flip-Flop.....	A-6
Fast Function Blocks.....	A-7
Product Term Expansion .....	A-9
XC7336 and XC7318 Fast Function Blocks.....	A-10
Input/Output Blocks.....	A-10

## **Appendix B Library Component Specifications**

ACC.....	B-3
Inferencing .....	B-3
Component Instantiation .....	B-3
Truth Table and Logic Symbol .....	B-3
ADD.....	B-4
Inferencing .....	B-4
Component Instantiation .....	B-4
Truth Table and Logic Symbol .....	B-4
ADSU .....	B-5
Inferencing .....	B-5
Component Instantiation .....	B-5
Truth Table and Logic Symbol .....	B-5
ADSUR.....	B-6
Inferencing .....	B-6
Component Instantiation .....	B-6

Truth Table and Logic Symbol .....	B-6
AND2 — AND8 .....	B-7
Inferencing .....	B-7
Component Instantiation .....	B-7
Truth Table and Logic Symbol .....	B-7
BUF .....	B-8
Inferencing .....	B-8
Component Instantiation .....	B-8
Truth Table and Logic Symbol .....	B-8
BUFCE .....	B-9
Inferencing .....	B-9
Component Instantiation .....	B-9
Truth Table and Logic Symbol .....	B-9
BUFE .....	B-10
Inferencing .....	B-10
Component Instantiation .....	B-10
Truth Table and Logic Symbol .....	B-10
BUFFOE .....	B-11
Inferencing .....	B-11
Component Instantiation .....	B-11
Truth Table and Logic Symbol .....	B-11
BUFG .....	B-12
Inferencing .....	B-12
Component Instantiation .....	B-12
Truth Table and Logic Symbol .....	B-12
CBX1 .....	B-13
Inferencing .....	B-13
Component Instantiation .....	B-13
Truth Table and Logic Symbol .....	B-13
CBX2 .....	B-14
Inferencing .....	B-14
Component Instantiation .....	B-14
Truth Table and Logic Symbol .....	B-14
COMPEQ .....	B-15
Inferencing .....	B-15
Component Instantiation .....	B-15
Truth Table and Logic Symbol .....	B-15
COMPLE_TC .....	
COMPLE_US .....	B-16
Inferencing .....	B-16
Component Instantiation .....	B-16

Truth Table and Logic Symbol .....	B-16
COMPLT_TC .....	
COMPLT_US .....	B-17
Inferencing .....	B-17
Component Instantiation .....	B-17
Truth Table and Logic Symbol .....	B-17
COMPNE .....	B-18
Inferencing .....	B-18
Component Instantiation .....	B-18
Truth Table and Logic Symbol .....	B-18
DEC .....	B-19
Inferencing .....	B-19
Component Instantiation .....	B-19
Truth Table and Logic Symbol .....	B-19
FDCP .....	B-20
Inferencing .....	B-20
Component Instantiation .....	B-20
Truth Table and Logic Symbol .....	B-20
FDCPE .....	B-21
Inferencing .....	B-21
Component Instantiation .....	B-21
Truth Table and Logic Symbol .....	B-21
FDPC .....	B-22
Inferencing .....	B-22
Component Instantiation .....	B-22
Truth Table and Logic Symbol .....	B-22
IBUF .....	B-23
Inferencing .....	B-23
Component Instantiation .....	B-23
Truth Table and Logic Symbol .....	B-23
IFD .....	B-24
Inferencing .....	B-24
Component Instantiation .....	B-24
Truth Table and Logic Symbol .....	B-24
IFDX1 .....	B-25
Inferencing .....	B-25
Component Instantiation .....	B-25
Truth Table and Logic Symbol .....	B-25
ILD .....	B-26
Inferencing .....	B-26
Component Instantiation .....	B-26

Truth Table and Logic Symbol .....	B-26
INC .....	B-27
Inferencing .....	B-27
Component Instantiation .....	B-27
Truth Table and Logic Symbol .....	B-27
INV .....	B-28
Inferencing .....	B-28
Component Instantiation .....	B-28
Truth Table and Logic Symbol .....	B-28
IOBUFE .....	B-29
Inferencing .....	B-29
Component Instantiation .....	B-29
Truth Table and Logic Symbol .....	B-29
IOBUFEX1 .....	B-30
Inferencing .....	B-30
Component Instantiation .....	B-30
Truth Table and Logic Symbol .....	B-30
LD .....	B-31
Inferencing .....	B-31
Component Instantiation .....	B-31
Truth Table and Logic Symbol .....	B-31
OBUF .....	B-32
Inferencing .....	B-32
Component Instantiation .....	B-32
Truth Table and Logic Symbol .....	B-32
OBUFE .....	B-33
Inferencing .....	B-33
Component Instantiation .....	B-33
Truth Table and Logic Symbol .....	B-33
OBUFEX1 .....	B-34
Inferencing .....	B-34
Component Instantiation .....	B-34
Truth Table and Logic Symbol .....	B-34
OR2 — OR8 .....	B-35
Inferencing .....	B-35
Component Instantiation .....	B-35
Truth Table and Logic Symbol .....	B-35
SUBT .....	B-36
Inferencing .....	B-36
Component Instantiation .....	B-36
Truth Table and Logic Symbol .....	B-36



XOR2 — XOR8 .....	B-37
Inferencing .....	B-37
Component Instantiation .....	B-37
Truth Table and Logic Symbol .....	B-37

## **Appendix C Attributes**

Global Attributes .....	C-1
LOWPWR .....	C-1
MRINPUT .....	C-1
NO_FOE .....	C-2
NO_FCLK .....	C-2
NO_IFD .....	C-2
PRELOAD .....	C-3
Signal Attributes .....	C-3
F .....	C-3
H .....	C-4
OPT_OFF .....	C-4
OPT_UIM .....	C-4
SLEWRATE .....	C-4
Synopsys Attributes .....	C-5
Part Type .....	C-5
Pin Assignment .....	C-6
Register Initial State .....	C-6

<b>Appendix D XEPLD Files</b> .....	D-1
-------------------------------------	-----

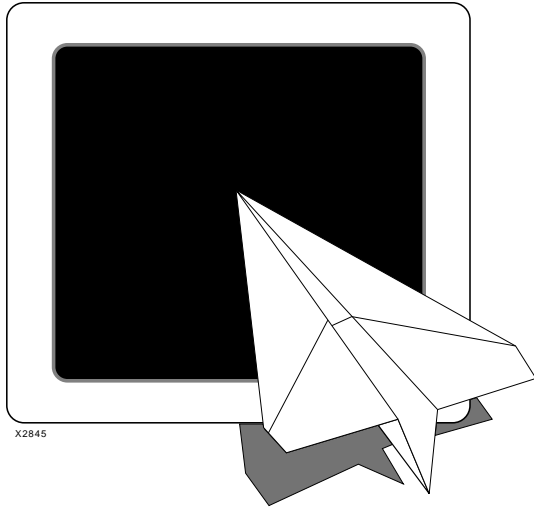
## **Appendix E Design Example**

PCI Bus Interface Design Description .....	E-1
--	-----

## **Appendix F Fitter Reports**

Resource Report .....	F-2
The Static Timing Report .....	F-3
Combinational Pad-to-Pad Delays .....	F-3
Setup-to-Clock Time .....	F-4
Clock-to-Output Delays .....	F-4
Cycle Time .....	F-5
Example Timing Report .....	F-6
Pin-List Report .....	F-13

<b>Appendix G Glossary</b> .....	G-1
----------------------------------	-----



## ***Getting Started with Xilinx EPLDs***

# ***Xilinx Synopsys Interface EPLD User Guide***

# Chapter 1

## Getting Started with Xilinx EPLDs

---

This chapter shows you how to prepare your setup files and verify your installation. It also provides a design walk-through as an overview of the basic steps for implementing Xilinx EPLD designs using Synopsys. The remaining chapters in this manual provide additional detailed information on each step.

The design walk-through assumes that you have installed and configured the Xilinx software and libraries. For installation instructions, see the *Xilinx Synopsys Interface Release Notes* that accompany this manual.

## Creating Synopsys Setup Files

After you have installed the Xilinx software you must configure the Synopsys Design Compiler and VSS simulator setup files to access the XC7000 libraries. This section shows you how to configure the setup files and verify that your setup is working properly.

The setup files must be located in each design directory where XC7000 designs are processed.

**Note:** You will find sample setup files in the `$DS401/tutorial/synopsys/epld.vhd/scan` directory. However, before using them, you must edit the `.synopsys_dc.setup` file contained in the tutorial directory by typing the actual `$DS401` path into the `search_path` variable.

## The Design Compiler Setup File

Your Design Compiler setup file (`.synopsys_dc.setup`) must contain the following lines:

```
search_path = { . $DS401_path/synopsys/libraries/
                syn}
link_library = {xc7000.db xc7000.sldb}
target_library = {xc7000.db}
symbol_library = {xc7000.sdb}
synthetic_library = {xc7000.sldb}
bus_naming_style = "%s<%d>"
bus_dimension_seperator_style = "><"
bus_interface_style = "%s<%d>"
edifout_netlist_only = true
edifout_power_and_ground_representation = cell
edifout_ground_name = GND
edifout_ground_pin_name = GND
edifout_power_name = VCC
edifout_power_pin_name = VCC
xnfout_library_version = "2.0.0"
```

Where `$DS401_path` is the actual interface directory path specified by the `$DS401` variable.

**Note:** You cannot use environment variables in the `.synopsys_dc.setup` file.

## The VSS Simulator Setup File (`.synopsys_vss.setup`)

Your VSS Simulator setup file, `.synopsys_vss.setup`, must contain the following lines:

```
xc7000: $DS401/synopsys/libraries/dw/lib/epld
TIMEBASE = NS
TIME_RES_FACTOR = 0.1
```

**Note:** You may use either the `$DS401` environment variable or the actual path specification in the `.synopsys_vss.setup` file.

As a final verification that your XEPLD Synopsys Interface is ready to use, we have provided a complete design example for you to run, which is described later in this chapter. To quickly verify VHDL

design entry, you can begin at design example step 9 and run `scan.script` or `scan.dc` as described.

## Verifying Your Installation

Before attempting to compile and fit a design, it is a good idea to verify that you have access to the installed software. A simple verification process is described below.

### Verifying Synopsys Software Access

To verify that your system is correctly configured to access the Synopsys software, enter the following UNIX commands:

```
which dc_shell
which vhdlan    (if you are using the VSS simulator)
```

If you get a negative response for either command, (such as “no vhdlan in ...”) this means that either the software is not installed properly or that your system path is not set properly to include the Synopsys software directories. Refer to the Synopsys documentation for installation instructions.

### Verifying Xilinx Software Access

To verify that your system is correctly configured to access the Xilinx-supplied software, enter the following UNIX commands:

1. **which fitnet**

If **fitnet** cannot be found, the XEPLD Translator Core Tool (DS550) is not installed or is not in your path.

2. **which syn2epld**

If **syn2epld** cannot be found, XSI is not installed or is not in your path.

3. **echo \$DS401**

This variable should point to the XSI directory found in Step 2.

4. **echo \$XACT**

This variable should point to the XACT directory found in Step 1 (unless the XACT data files were installed in a different location)

than the XACT executables). The \$XACT variable should also print to your XSI (DS-401) directory if it was installed to a different location.

## Verifying Your File Structure

To verify that you have the necessary files for EPLD development, use the file structure diagram in Figure 1-1.

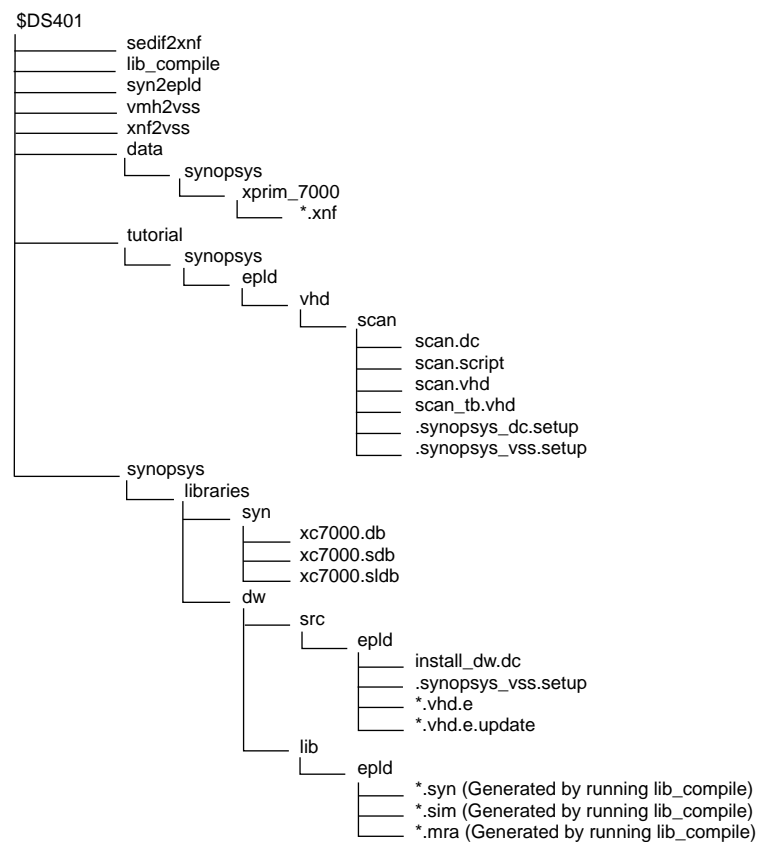


Figure 1-1 XSI File Structure for EPLD Development

## Xilinx EPLD Design Flow

Figure 1-2 shows the basic design flow for creating EPLD designs. Each step is described in the following design example.

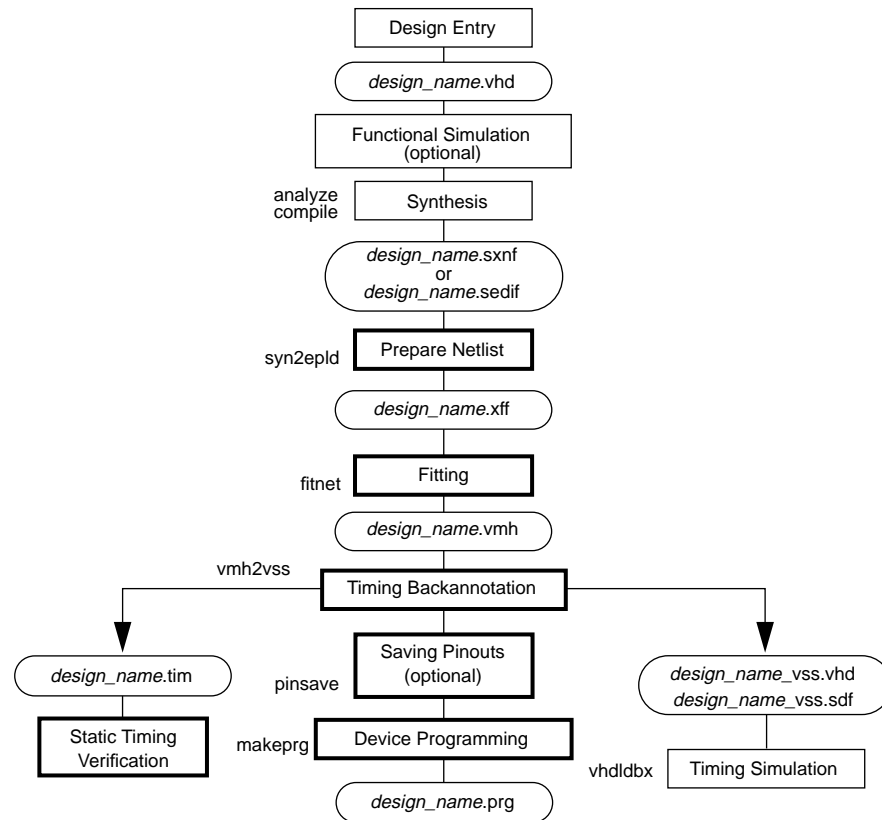


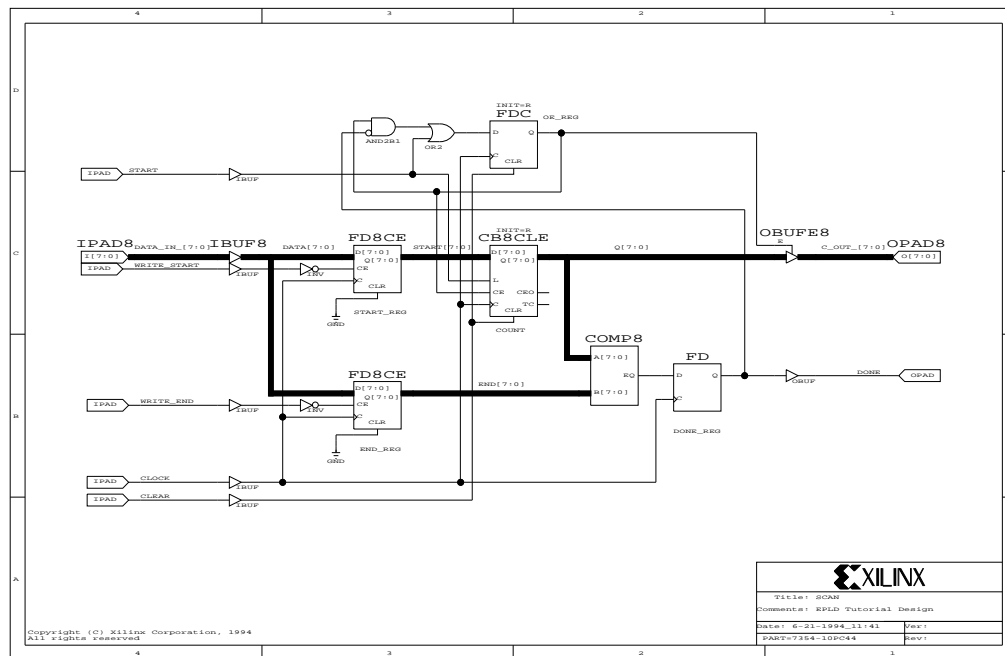
Figure 1-2 Basic EPLD Design Flow

## Design Example

The following design example is used to demonstrate the basic EPLD design flow. This design implements a counter with variable start and stop values which are loaded into registers from a data input bus. When the *START* input is asserted, the start value is loaded into the counter and the counter outputs are enabled. The counter outputs

increment on each clock cycle until the counter value matches the stop value. The counter outputs are disabled on the next clock cycle. The design is implemented in a Xilinx XC7354-10PC44 device.

To help you understand the design, an equivalent schematic is shown in Figure 1-3.



**Figure 1-3 Schematic Representation — SCAN Design**

The VHDL source file (scan.vhd) for the scan example design is shown in Figure 1-4.



```
-----
--
-- Xilinx EPLD Synopsys VHDL Tutorial Design
--
-- File:          scan.vhd
--
-- Target Device:  XC7354-10PC44
--
-- Author:         Xilinx Corporation
--                 Copyright (C) Xilinx Corporation 1994
--                 All rights reserved
--
-- Requirements:   Xilinx Synopsys Interface v3.2
--                 Xilinx XEPLD (DS550) v5.0
--                 Synopsys Design Compiler v3.1
--
-----

-- Standard library configuration --
Library IEEE;
Library xc7000;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use xc7000.components.all;

entity scan is
    port (CLOCK, CLEAR, START, WRITE_START, WRITE_END: in std_logic;
          DATA_IN: in std_logic_vector (7 downto 0);
          C_OUT: out std_logic_vector (7 downto 0);
          DONE: out std_logic;
          MRESET: in std_logic); -- MRESET used for timing simulation only --
end scan;

architecture behavior of scan is
    signal START_REG: std_logic_vector (7 downto 0);
    signal END_REG: std_logic_vector (7 downto 0) := "11111111";
    signal COUNT: std_logic_vector (7 downto 0) := "00000000";
    signal OE_REG, DONE_REG: std_logic := '0'; -- Initial states used by fn'l sim. only --

begin
    -- Registers without asynchronous clear --
    process (CLOCK)
    begin
        if (CLOCK'event and CLOCK='1') then
            if (WRITE_START = '0') then
                START_REG <= DATA_IN;
            end if;
        end if;
    end process;
end behavior;
```

```
        if (WRITE_END = '0') then
            END_REG <= DATA_IN;
        end if;

        -- Registered comparator --
        if (COUNT = END_REG) then
            DONE_REG <= '1';
        else
            DONE_REG <= '0';
        end if;
    end if;
end process;

-- OE_REG register with asynchronous clear --
process (CLEAR, CLOCK)
begin
    if (CLEAR = '1') then
        OE_REG <= '0';
    elsif (CLOCK'event and CLOCK='1') then
        if (START = '1') then
            OE_REG <= '1';
        elsif (DONE_REG = '1') then
            OE_REG <= '0';
        end if;
    end if;
end process;

-- Counter with asynchronous clear and parallel load --
process (CLEAR, CLOCK)
begin
    if (CLEAR = '1') then
        COUNT <= "00000000";
    elsif (CLOCK'event and CLOCK='1') then
        if (START = '1') then
            COUNT <= START_REG; -- Counter parallel load --
        elsif (OE_REG = '1') then
            COUNT <= COUNT + 1; -- Counter increment --
        end if;
    end if;
end process;

-- Three-state counter outputs --
C_OUT <= COUNT when (OE_REG = '1') else
    "ZZZZZZZZ";
DONE <= DONE_REG;
end behavior;
```

**Figure 1-4 Example Design Source File (scan.vhd)**

## Design Entry

Typically you will enter your design in Synopsys VHDL/HDL form by using a text editor. However, all required source, setup, and test bench files for this design example have already been entered for you and are contained in the `$DS401/tutorial/synopsys/epld/vhd/scan` directory (see Figure 1-1).

### Step1 — Create a Design Directory

Create a local copy of the scan tutorial directory as follows:

- Change your current working directory to a local, writable location in which you will place the scan working directory.
- Copy the entire scan directory tree from the XEPLD Synopsys Interface tutorial area into your current directory as follows:

```
cp -r $DS401/tutorial/synopsys/epld/vhd/scan .
```

- Change your current directory to the scan tutorial directory as follows:

```
cd scan
```

- Verify that the `search_path` variable in your `.synopsys_dc.setup` file, in your current working directory, points to the directory path where your Xilinx EPLD Synopsys Interface library is installed, which should be the value of your `$DS401` variable.

**Note:** The `search_path` variable must be explicitly defined; environment variables are not allowed in the `.synopsys_dc.setup` file.

If you need more information on design entry see the Synopsys Design Compiler manuals.

## Functional Simulation

Functional simulation verifies the logic of your design. This will save you time by catching logic errors early in the development cycle. If you are not using the VHDL System Simulator (VSS), skip this section and continue with step 9.

You must analyze your source design file before simulation. If you created a test bench in VHDL/HDL for simulation, you must also analyze it after analyzing your design.

## **Step 2 — Analyze Your Design**

Analyze the scan design by entering the following Synopsys command on the UNIX command line:

```
vhdlan scan.vhd
```

You will see the analyzer version number and a copyright notice. If the analysis works properly you will be returned to the UNIX prompt with no error messages displayed.

## **Step 3 — Analyze Your Test Bench**

For this example a test bench is provided (`scan_tb.vhd`). At the end of this file, a configuration named `CFG_SCAN_TB` is declared. The test bench file is shown in Figure 1-5.

Analyze the test bench for scan by entering the following Synopsys command on the UNIX command line:

```
vhdlan scan_tb.vhd
```

Again, you will see the analyzer version number and a copyright notice. If the analysis works properly you will be returned to the UNIX prompt with no error messages displayed.

```
library IEEE;
library xc7000;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_misc.all;
    use IEEE.std_logic_arith.all;
    use IEEE.std_logic_components.all;
    use STD.Textio.all;
    use xc7000.components.all;

entity scan_tb is
end scan_tb;

architecture test of scan_tb is

    component scan
        port (CLOCK, CLEAR, START, WRITE_START, WRITE_END: in std_logic;
              DATA_IN: in std_logic_vector (7 downto 0);
              C_OUT: out std_logic_vector (7 downto 0);
              DONE: out std_logic;
              MRESET: in std_logic);
    end component;

    signal CLOCK, CLEAR, START, WRITE_START, WRITE_END: std_logic;
    signal DATA_IN: std_logic_vector (7 downto 0);
    signal C_OUT: std_logic_vector (7 downto 0);
    signal DONE: std_logic;
    signal MRESET: std_logic;

begin

    UUT: scan
        port map (CLOCK, CLEAR, START, WRITE_START, WRITE_END,
                  DATA_IN, C_OUT, DONE, MRESET);
```

```
DRIVER: process
begin
    MRESET <= '0';
    CLEAR <= '0';
    START <= '0';
    WRITE_START <= '1';
    WRITE_END <= '1';
    DATA_IN <= "00000000";
    CLOCK <= '0';

    wait for 25 ns;
    MRESET <= '1';
    wait for 25 ns;
    CLOCK <= '1';

    wait for 25 ns;
    CLOCK <= '0';
    DATA_IN <= "01111101";
    WRITE_START <= '0';
    wait for 25 ns;
    CLOCK <= '1';

    wait for 25 ns;
    CLOCK <= '0';
    DATA_IN <= "10000001";
    WRITE_START <= '1';
    WRITE_END <= '0';
    wait for 25 ns;
    CLOCK <= '1';

    wait for 25 ns;
    CLOCK <= '0';
    WRITE_END <= '1';
    START <= '1';
    wait for 25 ns;
    CLOCK <= '1';

    for I in 1 to 6 loop
        wait for 25 ns;
        CLOCK <= '0';
        START <= '0';
        wait for 25 ns;
        CLOCK <= '1';
    end loop;
```

```
        wait for 25 ns;
        CLOCK <= '0';
        START <= '1';
        wait for 25 ns;
        CLOCK <= '1';

        wait for 25 ns;
        CLOCK <= '0';
        START <= '0';
        wait for 25 ns;
        CLOCK <= '1';

        wait for 25 ns;
        CLOCK <= '0';
        CLEAR <= '1';
        wait for 25 ns;
        CLOCK <= '1';

        wait for 25 ns;
        CLOCK <= '0';
        CLEAR <= '0';
        wait for 25 ns;
        wait;

    end process;
end test;

configuration CFG_SCAN_TB of scan_tb is
    for test
    end for;
end CFG_SCAN_TB;
```

**Figure 1-5 Test Bench File (scan\_tb.vhd)**

## Step 4 — Invoke the Simulator

Invoke the simulator by entering the following Synopsys command on the UNIX command line:

```
vhdlldb
```

You will see the following window for selecting the analyzed configurations:

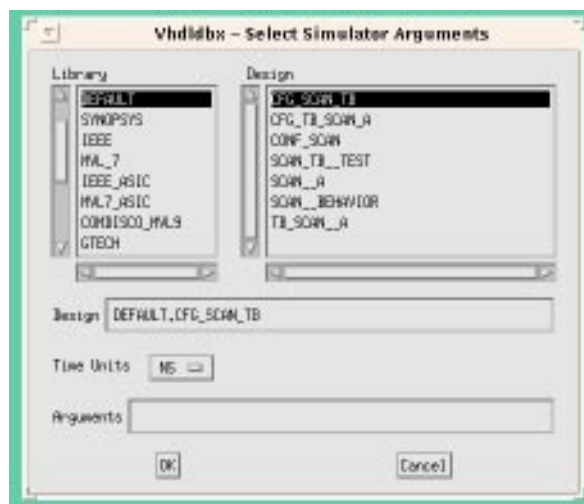


Figure 1-6 VHDLDDBX Window

## Step 5 — Run the Debugger

Select CFG\_SCAN\_TB from the menu. This brings up the Synopsys VHDL Debugger window as shown in Figure 1-7.



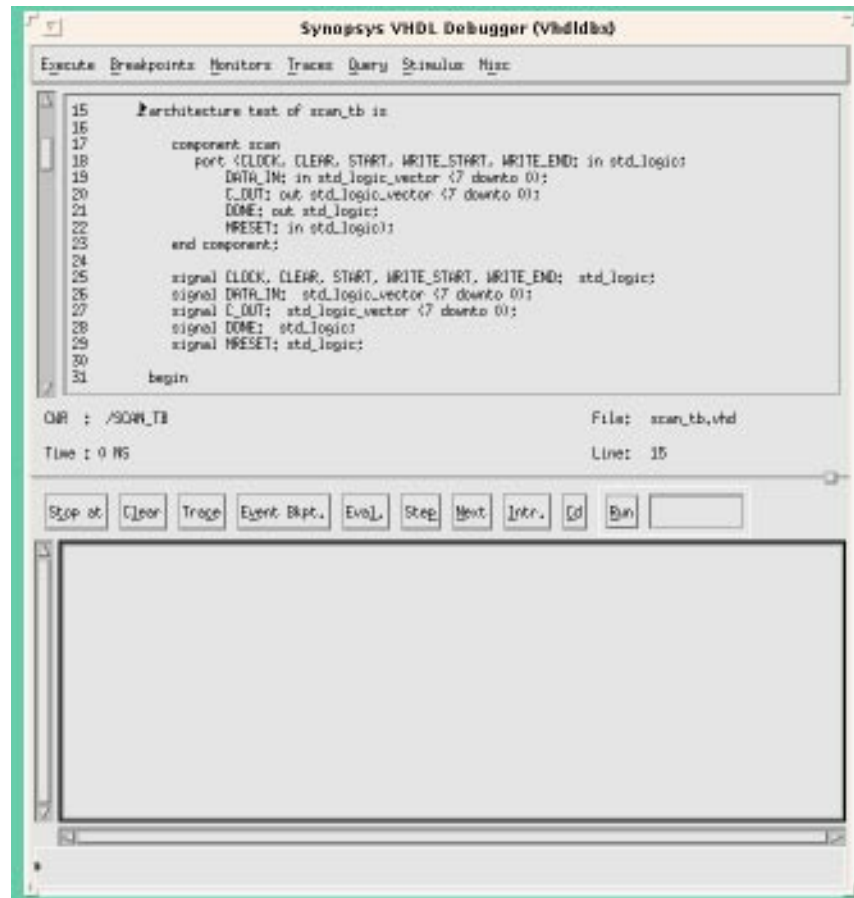


Figure 1-7 Synopsys VHDL Debugger

### Step 6 — Trace Signals

Click in the lower section of the Synopsys VHDL Debugger window and enter the following command:

```
trace *'signal
```

This command selects all signals at the test bench level for display and brings up the Dynamic Waveform Viewer (Waves).

## Step 7 — Run the Simulation

Click the **RUN** button in the Debugger window to run the simulation waveform specified in the test bench. The resulting trace display is shown in Figure 1-8.

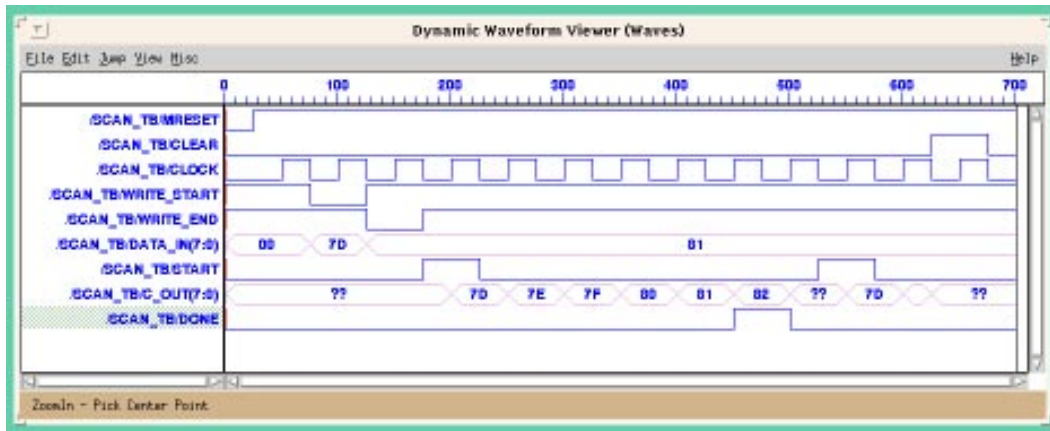


Figure 1-8 Synopsys Dynamic Waveform Viewer (Waves)

## Step 8 — Return to UNIX

Return to the UNIX environment by selecting **EXECUTE-QUIT** from the VHDL Debugger menu.

If you need more information on functional simulation see the “Simulating Your Design” chapter.

## Synthesizing Your Design (Compiling)

Synthesizing your design converts the VHDL source text into a netlist that is composed of logic primitives. The netlist is in a form that can be read by the Xilinx fitter.

## Step 9 — Enter the DC Shell Environment

Enter the Synopsys DC Shell environment by entering the following Synopsys command on the UNIX command line:

```
dc_shell
```

You will see the DC Shell license information and command-line prompt. Verify that the software version is v3.1 or newer.

**Note:** The commands required to compile the scan design example are shown in the following steps 10 through 16. These commands are contained in compiler script files. If you have FPGA compiler, the appropriate commands are contained in `scan.script`, which you can run by entering the following Synopsys command:

```
include scan.script
```

If you have only Design Compiler, the appropriate commands are contained in `scan.dc` which you can run by entering the following Synopsys command:

```
include scan.dc
```

If you choose to use these compiler scripts, go to step 17 when compilation is complete.

Unless otherwise specified, the commands in steps 10-16 are the same for both FPGA Compiler and Design Compiler.

## Step 10 — Analyze Your Source Design

Read and analyze your VHDL source design file by entering the following Synopsys command:

```
analyze -format vhdl scan.vhd
```

The warning messages you see during this step are normal. The source file contains initial signal values that are used only for functional simulation and these values are ignored during synthesis. Actual register initial states are set using attributes as shown in step 15.

## Step 11 — Elaborate Your Design

To build the design based on your analyzed VHDL file, entering the following Synopsys command:

```
elaborate scan
```

This command displays each register and 3-state buffer encountered in your design.

## **Step 12 — Synthesize Your Design**

To synthesize an implementation of your design based on cells in the XC7000 technology library enter the following Synopsys command:

```
compile -map_effort low
```

The mapping effort is set to LOW to save compilation time because the synthesizer does not perform any speed or area optimization for EPLD designs; optimization is performed by the XEPLD fitter.

**Note:** For this design example, the compiler produces a warning about a port not connected to any nets; this warning can safely be ignored. This warning also occurs in step 13.

## **Step 13 — Place I/O Buffer Cells**

To place I/O buffer cells on all top-level ports in the design, enter the following Synopsys commands:

```
set_port_is_pad ""  
insert_pads
```

## **Step 14 — Specify a Target Device**

If you are using FPGA Compiler, enter the following Synopsys command to specify a target EPLD:

```
set_attribute scan part -type string 7354-10pc44
```

## **Step 15 — Specify Initial Register States**

In this design we want the counter and the OE\_REG flip-flop to be initialized to zero. The states of the remaining flip-flops are determined by the fitter to achieve the best logic optimization.

If you have FPGA Compiler, enter the following Synopsys commands to specify the initial states:

```
set_attribute find(cell COUNT*)  
    fpga_xilinx_init_state -type string R  
set_attribute find(cell OE_REG*)  
    fpga_xilinx_init_state -type string R
```

## Step 16 — Output the Netlist

The design database is now complete and ready to be output in netlist form.

If you have FPGA Compiler, write an XNF-formatted netlist by entering the following Synopsys command:

```
write -format xnf -hierarchy -output scan.sxnf
```

If you have Design Compiler, write an EDIF-formatted netlist by entering the following Synopsys command:

```
write -format edif -output scan.sedif
```

## Step 17 — Exit DC Shell

Exit DC Shell by entering the following Synopsys command:

```
exit
```

You are returned to the UNIX prompt.

If you need more information on compiling your design, see the “Compiling Your Design” chapter.

The synthesizer creates a gate-level design with no physical device information; the physical layout of the device is done in the next step. No speed or area estimates are provided by the XC7000 library. Therefore do not attempt to create a timing report or perform estimated timing simulation at this time.

## Preparing the Netlist

The Synopsys compiler produces a file named *design\_name.sxnf* or *design\_name.sedif* which may contain references to macros. This file must be translated into a flattened netlist file (*design\_name.xff*) for the Xilinx fitter.

## Step 18 — Create a Flattened Netlist

If you are using FPGA Compiler (and created a *design\_name.sxnf* file), create a *design\_name.xff* file for the fitter by entering the following on the UNIX command line:

```
syn2epld scan
```

If you are using Design Compiler (and created a *design\_name.sedif* file), create a *design\_name.xff* file for the fitter by entering the following on the UNIX command line:

```
syn2epld scan.sedif -p 7354-10pc44
```

When file translation is finished, you will see the message “Netlist written to file scan.xff.”

If you need more information on preparing the netlist, see the “Fitting Your Design” chapter.

## Fitting Your Design

The XEPLD fitter translates your logical design file (*design\_name.xff*) into a physical device layout.

### Step 19 — Fit Your Design

To fit your design into a target device, enter the following on the UNIX command line:

```
fitnet -n scan
```

The fitter displays a series of progress messages and a resource summary that shows how well your design fits into the target device. During execution, **fitnet** produces a warning message about an AND-gate that “does not drive anything and it is removed”; this can safely be ignored.

When the fitter is finished you will see the message “Design successfully mapped. Examine the following report files:...”. Assuming there are no errors, you need only to examine the resource summary already displayed. The Resource report is also saved in the file *design\_name.res*.

If you need more information on fitting, see the “Fitting Your Design” chapter. If you need more information on interpreting reports, see the “Fitter Reports” appendix.

## Timing Backannotation

After fitting, the XEPLD fitter can produce a static timing report that shows the calculated worst case timing of the logic paths in your design. It also creates a timing simulation file in VHDL for use with the Synopsys VSS simulator.

### Step 20 — Create a Static Timing Report

To create the Static Timing Report and create the backannotated VHDL file for the VSS simulator, enter the following Xilinx command on the UNIX command line:

```
vmh2vss scan
```

When file translation is finished you will see that the following files have been written: `scan.tim`, `scan_vss.vhd`, `scan_vss.sdf`.

The Static Timing Report (`scan.tim`) for this example is provided in the “Fitter Reports” appendix. This report shows that the critical path for this design (clock pad to output pad delay to enable the `C_OUT` pins) is 20ns. The worst case cycle time is 13ns.

## Timing Simulation

Timing simulation uses the actual device delays based on the physical layout of your design after fitting. If you are not using the VSS simulator, skip this section and go to step 28.

### Step 21 — Analyze Your Original Design

Analyze the original scan design to reuse the port declarations contained in the entity by entering the following Synopsys command on the UNIX command line:

```
vhdlan scan.vhd
```

### Step 22 — Analyze Your Back-Annotated Design

Analyze the back-annotated design architecture, produced by the Xilinx `vmh2vss` command, by entering the following Synopsys command on the UNIX command line:

```
vhdlan scan_vss.vhd
```

You will see the analyzer version number and a copyright notice. If the analysis works properly you will be returned to the UNIX prompt with no error messages displayed.

### **Step 23 — Analyze Your Test Bench**

Analyze the simulation test bench by entering the following Synopsys command on the UNIX command line:

```
vhdlan scan_tb.vhd
```

Again, you will see the analyzer version number and a copyright notice. If the analysis works properly you will be returned to the UNIX prompt with no error messages displayed.

### **Step 24 — Invoke the VSS Simulator**

Invoke the Synopsys VSS simulator by entering the following Synopsys command on the UNIX command line:

```
vhdlbxb -sdf scan_vss.sdf -sdf_top /SCAN_TB/UUT  
CFG_SCAN_TB &
```

For your convenience, this command line is contained in a script file, which you can execute by typing the following on the UNIX command line:

```
dbx_scan
```

This will open the simulator window as shown in Figure 1-9. The **-sdf** parameter specifies the timing back-annotation file produced by **vmh2vss**. The **-sdf\_top** parameter specifies the level in the test bench hierarchy at which the back annotation information will be applied.



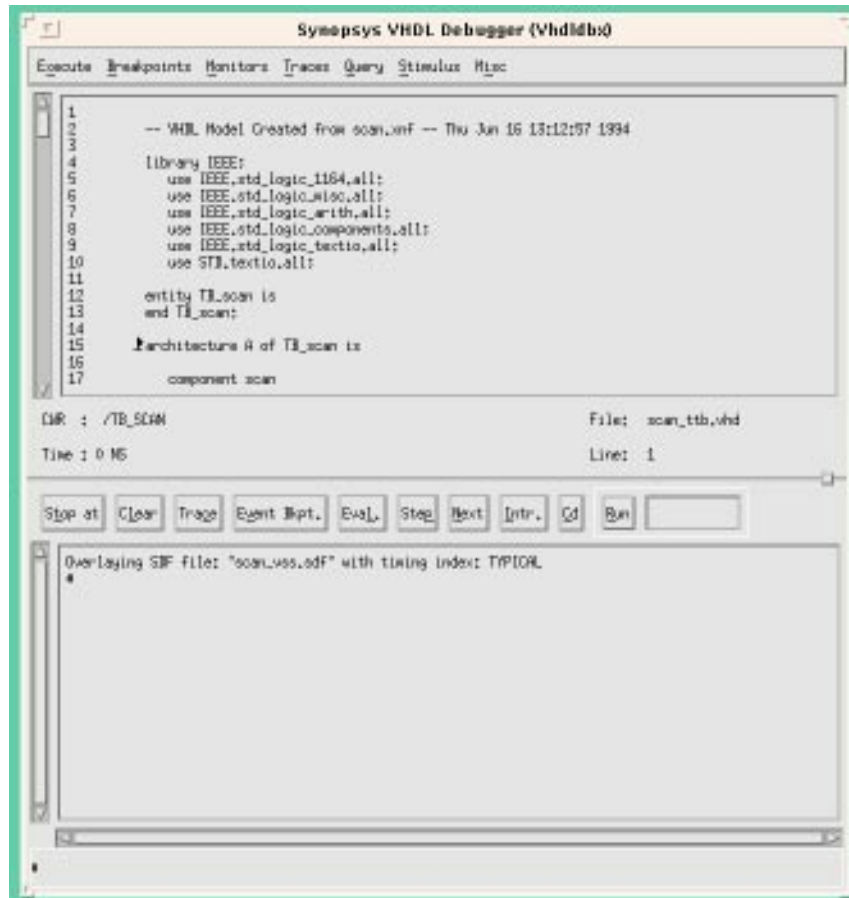


Figure 1-9 Synopsys VHDL Debugger

### Step 25 — Open the Waveform Viewer

Use the **TRACE** command to specify the same signals used during functional simulation in step 6. Enter the following command on the VHDL Debugger command line:

```
trace *'signal
```

This opens the Dynamic Waveform Viewer window.

## Step 26 — Run the Simulation

Run the simulation by clicking the RUN button in the lower section of the Synopsys VHDL Debugger window.

This will run the timing simulation test bench and display the simulation trace of your design as shown below in Figure 1-10.

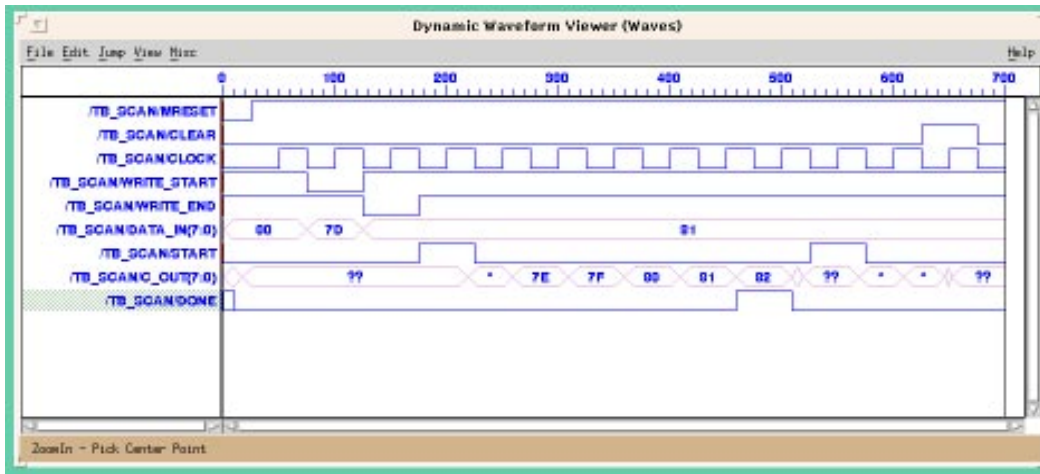


Figure 1-10 Synopsys Dynamic Waveform Viewer (Waves)

## Step 27 — Return to UNIX

Return to the UNIX environment by selecting **EXECUTE-QUIT** from the simulator menu.

If you need more information on timing simulation, see the "Simulating Your Design" chapter.

## Programming an EPLD

After you have verified your design you are ready to program an EPLD.

## **Step 28 — Program an EPLD**

Create an EPLD programming file by entering the following on the UNIX command line:

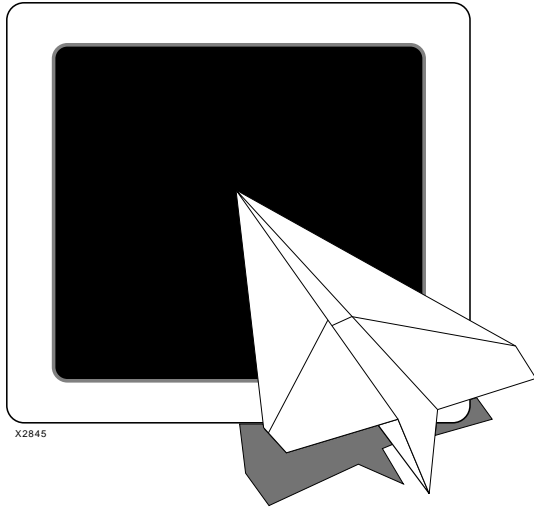
```
makeprg scan -s scan01
```

This creates a bit map file that can be downloaded to a device programmer.

The **-s** parameter specifies a user signature string “scan01” that is programmed into special EPROM cells in the device that you can read for identification.

If you need more information on device programming, see the documentation that accompanies your device programmer.





***Designing with EPLDs***

# ***Xilinx Synopsys Interface EPLD User Guide***

## Chapter 2

### Designing with EPLDs

---

This chapter discusses how to use design techniques, library components, and attributes to get the best performance from Xilinx EPLDs. For more information on library components, see the “Library Component Specifications” appendix. For more information on attributes, see the “Attributes” appendix.

#### VHDL Design File Requirements

If you plan to instantiate any components from the XC7000 library or perform any simulation you will need to declare the Xilinx `XC7000.components` package in your design source file. It is generally a good idea to always declare this package in all EPLD designs.

To declare the `XC7000.components` package, insert the following two lines at the top of your VHDL source file:

```
library xc7000;  
use xc7000.components.all;
```

#### Using Registers and Latches

The Xilinx EPLD architecture allows you to implement both registers and latches within function block macrocells and within input pads. This section shows you how to assign logic to specific registers and latches, and how to control their initial states after power is applied.

The Xilinx fitter uses input pad registers and latches to implement functions whenever possible to reduce the device macrocell resource requirements. Register functions using any control inputs, such as clear, preset, or clock enable, will only be implemented in macrocell registers; only simple D-type flip-flops can be optimized into input pads.

To be eligible for optimization into an input pad, a register's **D** and **C** inputs must come directly from input ports or I/O ports. The **C** (clock) input signal must not be used for anything other than register clocking, and the **D** (data) input signal must not be used for any other input.

## Preventing Register/Latch Optimization

To prevent the fitter from automatically assigning any registered or latched functions to the input pads, instantiate the **NO\_IFD** global attribute cell in your source design as follows:

```
U1: NO_IFD;
```

where U1 is any instance name.

**Note:** The **NO\_IFD** attribute does not prevent you from instantiating specific input pad register and latch components.

## Using Input Pad Registers

If you want to assign a specific register in your design to an input pad, instantiate the **IFDX1** component. The Clock input must be driven by a **BUFG** component (global FastClk), and the Clock Enable input (if used) must be driven by a **BUFCE** component (Global Clock Enable). Except for signals declared as FastInputs, the **D** input signal must not be used for any other input.

## Using Macrocell Registers

Inferred registered functions will be placed either into macrocells or input pads at the discretion of the fitter unless register optimization is turned off. If register optimization is turned off (using the **NO\_IFD** attribute) then all inferred registers will be placed into macrocells.

The techniques used to infer registers in EPLD designs is no different than for any other Synopsys design. For example, the following behavioral VHDL process implements a D-type flip-flop with asynchronous clear and clock-enable:

```
process (CLEAR, CLOCK)
begin
  if (CLEAR = '1') then
    Q <= '0';
  elsif (CLOCK'event and CLOCK='1') then
    if (CE = '1') then
      Q <= 'D';
    end if;
  end if;
end process;
```

You can also instantiate the `FDCP`, `FDPC`, or `FDCPE` register components. If none of the control inputs are used, the software will attempt to optimize these registers into input pads, provided optimization is enabled.

## Using Input Pad Latches

If you want to assign a specific latch in your design to an input pad, instantiate the `ILD` component. The `G` input must be driven by a `BUFG` component (global `FastClk`). Except for signals declared as `FastInputs`, the `D` input signal must not be used for any other input.

## Using Macrocell Latches

Inferred latched functions will be placed into macrocells only. You can also instantiate the `LD` (latch) component.

**Note:** The EPLD architecture does not support transparent latches with asynchronous clear or preset using a single macrocell.

## Specifying Register/Latch Initial States

When the EPLD is powered on, or when the Master Reset pin is activated, all registers and latches are forced into an initial state. You can control the initial state of any register or latch, or you can allow the fitter to choose the initial state based on the most efficient usage of resources. You can see the initial states of the registers in your design, after fitting, by performing timing simulation.

The fitter will determine the initial states of all registers in the device, by default, based on optimal design performance, unless you specify initial states as described below.



## Specifying the Predefined Initial States

Each registered and latched component in the library has a defined initial state. If you want to use these predefined states, you must prevent the fitter from optimizing the registers/latches in any way that would change their initial states.

To use the predefined initial states, as specified in the “Library Component Specifications” appendix, instantiate the `preload` attribute cell in your source design as follows:

```
U1: preload;
```

where U1 is any instance name.

**Note:** When you specify `preload` you may inhibit the fitter from using certain resources in the EPLD to implement the registers in your design. This may require more device resources and can decrease performance.

## Specifying Initial States for Individual Registers/Latches

If you want to define the initial state of selected registers/latches, set the initial state attribute by using the following DC Shell commands.

If you are using FPGA Compiler, enter the following:

```
set_attribute "inst_name"  
fpga_xilinx_init_state -type string state
```

where:

- *inst\_name* is the name of a cell instance and may be evaluated by means of the DC Shell “find” function.
- *state* is either R (reset to 0) or S (set to 1).

For example, to specify an initial state of “1” for the register named QOUT\_reg<2> using FPGA Compiler, enter the following:

```
set_attribute "QOUT_reg<2>"  
fpga_xilinx_init_state -type string S
```

or, for all QOUT registers:

```
set_attribute find (cell QOUT_reg*)  
fpga_xilinx_init_state -type string S
```

**Note:** This attribute overrides all other methods for specifying initial states.

**Note:** You cannot change the initial state of input pad registers or latches (IFDX1 or ILD components); their initial states are always implemented as indicated in the “Library Component Specifications” appendix.

For more information on using attributes, see the “Attributes” appendix.

## Using I/O Ports

Unless otherwise specified, the compiler will automatically infer IBUF, OBUF, and IOBUFE cells for all top-level input, output, and I/O ports. However, the Xilinx component library also includes special-purpose I/O buffer cells that allow you to explicitly instantiate specific I/O functions.

You will want to explicitly assign I/O buffer cells for the following reasons:

- *There are more clocks or OE signals in your design than there are FastClock or FOE pins available on the device.* The Xilinx fitter automatically assigns the most frequently used clock signals to FastClock pins and the most frequently used 3-state control inputs to FOE pins. You can force specific clocks onto the global FastClock pins by instantiating the BUFG cell. You can force specific output enable signals onto the global FOE pins by instantiating the BUFFOE cell.
- *You do not want some clocks, output enable signals, or registers to be optimized automatically.* You can globally inhibit optimization of these resources by instantiating the NO\_FCLK, NO\_FOE, and NO\_IFD attribute cells. In this case you can manually assign selected clock or OE inputs to the global FastClock or FOE inputs by instantiating the BUFG or BUFFOE component. Instantiate the IFDX1 or ILD components to explicitly implement registers and latches in input pads.
- *You are generating global clock or FOE signals from within your design.* If you want to drive the global FastClock or FOE inputs from signals within your design, you must first drive those signals onto the corresponding device I/O pad through an output buffer and

then back into the chip through either the `BUFG` component (for FastClocks) or through the `BUFFOE` component (for FOE inputs).

## Selecting 3-State Control Sources

Xilinx EPLDs have dedicated high-speed routing that can be used for fast output enable signals (FOE). Any unused FOE routing is automatically assigned by the fitter to the most used output enable signals in your design unless you turn off optimization by using the `NO_FOE` attribute.

To be eligible for optimization, an output enable signal must come directly from an input or I/O port and not be used for any other logic function.

## Assigning Specific Fast Output Enable Signals

If you want to assign a specific output enable signal in your design to an FOE net, instantiate the `BUFFOE` input buffer to drive the Enable input of an `OBUFEX1` or `IOBUFEX1` component. The signal produced by the `BUFFOE` component cannot be used by any other logic, including the `OE` input of ordinary `OBUFE` or `IOBUFE` components.

## Preventing FOE Optimization

To prevent the fitter from automatically assigning output enable signals to any unused FOE nets, instantiate the `NO_FOE` cell in your source design as follows:

```
U1: NO_FOE;
```

where `U1` is any instance name.

## Using Special Logic Functions

### Binary Up Counters

You can infer binary up counters by using the “+1” operation and achieve optimal performance in the EPLD. You can also instantiate either the `CBX1` or `CBX2` components and use only the up count mode.

## Binary Down Counters

You can infer binary down counters by using the “-1” operation and achieve optimal performance. You can also instantiate either the CBX1 or CBX2 components and use only the down count mode.

## Binary Up/Down Counters

For best results, when creating up/down counters, instantiate the CBX1 component (up/down counter with asynchronous clear) or the CBX2 component (up/down counter with synchronous reset). These counters are scalable for any width and they are optimized for the Xilinx EPLD architecture.

If you infer up/down counters, your design will require more device resources to implement and will run slower.

**Note:** Inferred counters will automatically wrap from all 1’s to all 0’s. You do not need to write conditional expressions to detect terminal count; this would create unnecessary additional logic and may cause your counter to run slower. For example, do not write the following:

```
if (count = "11111111") then
    count <= "00000000";
else
    count <= count + 1;
```

## State Machines

When you initially compile a state machine, use the binary encoding option (the default). If the logic complexity of a binary encoded state machine results in poor device resource utilization, you can try less fully encoded state assignments explicitly in your VHDL design. In general you can use a few more registers to represent state vectors to reduce the amount of combinational logic required for each state flip-flop.

One-hot-encoding is rarely the most efficient way to create state machines for EPLDs (unlike Xilinx FPGA designs). Other schemes such as Gray coding do not help in EPLD designs because the EPLD architecture is primarily composed of D-type flip-flops.

## Registered Arithmetic Functions

When creating simple pipelined arithmetic functions (where no register control logic is required), you can use the “+” and “-” operators in an ordinary clocked process and achieve good results.

For example:

```
process (clock)
begin
    if (clock'event and clock = '1') then
        Q <= A + B;
    end if;
```

When creating registered arithmetic functions with any register control logic, instantiate the ACC component (Adder/Subtractor/Accumulator) or the ADSUR component (Adder/Subtractor with registered output) for best results. These components are scalable for any width and they are optimized for the Xilinx EPLD architecture.

If you infer these functions, your design will require more device resources to implement and will run slower.

**Note:** Register control logic includes functions such as synchronous or asynchronous reset, clock enable, and parallel load.

## Comparators

Magnitude comparators can be expressed either behaviorally, using the “<” or “>” operators, or by instantiating the COMPLT or COMPLE components. They are implemented essentially the same as a subtracter, with the carry-out serving as the comparator output. 3-bit look-ahead logic at the low-order end of the comparator saves about 2 macrocells in the EPLD over the straight subtracter solution. The EPLD high-speed arithmetic carry chain is used for all magnitude comparators larger than 4 bits.

Equality comparators are implemented combinatorially using XOR gates for each operand bit. The EPLD HDFFB can accommodate up to an 8-bit equality compare in a single macrocell. Equality comparators up to 8 bits can be expressed either behaviorally, using the “=” operator, or by instantiating the COMPEQ or COMPNE components. For comparators larger than 8 bits, which require more than one macrocell, you should use the COMPEQ or COMPNE components so

that the gate logic combining the macrocells' intermediate results are implemented in the UIM if possible (without extra delay).

Comparator outputs in high-speed applications are often pipelined before driving other logic or passing off-chip. By breaking larger comparators into 8-bit slices and pipelining each slice, the gate logic combining the slices can still be implemented in the UIM (for on-chip logic).

In the following example, a pipelined 16-bit comparator (with Boolean-type output *Q*) cannot be run at the maximum frequency of the EPLD because the logic preceding the register cannot fit a single macrocell:

```
process (clock)
begin
    if (clock'event and clock = '1') then
        Q <= A(0 to 15) = B(0 to 15);
    end if;
end process;
```

In the following example, the 16-bit comparator is broken into two 8-bit registered comparators, joined by a UIM-based AND-gate. This solution can be clocked at the maximum frequency of the EPLD if it drives on-chip logic:

```
process (clock)
begin
    if (clock'event and clock = '1') then
        Q0 <= A(0 to 7) = B(0 to 7);
        Q8 <= A(8 to 15) = B(8 to 15);
    end if;
end process;
Q <= Q0 and Q8;
```

## Targeting a Specific Device

Before fitting your design you must select a target device. You have three key questions to consider when selecting an EPLD:

- How many signal pins are required?
- How much Logic resources are required?
- How much performance (speed) is required?

The answers to these questions determine which device you will choose to contain your design.

Device selection can be an iterative process, as shown in the following steps:

1. Use the Xilinx EPLD data book to make a preliminary choice. This choice is usually based on the number of required signal pins because this is often the easiest question to answer. It is easiest to begin with the largest device (XC73108); this gives you the best chance for a successful fit. Otherwise, you can get a very rough estimate of the number of required macrocells as follows:

```
[(the number of output ports)
+ (the number of internal registers not driving
output ports)]
+ [20%]
```

2. Run the fitter on your design using the selected device. After fitting, the Resource Report indicates how much device resources were required. This will help you determine the best device size. If your design does not fit you will need to choose a larger device or partition your design among multiple devices. If you have unused logic resources, you may want to try a smaller device.
3. Once an optimal device size has been determined, you can create a Static Timing Report that will indicate the calculated timing of your design based on the device layout. You can also simulate the timing of your design using the Synopsys simulator. This timing information will help you select the optimal target device speed.

The “EPLD Architecture” appendix shows you a device selection chart. The “Library Component Specifications” appendix shows you which library components can be used with specific target devices. See the device data sheets for more information.

## **Specifying a Device**

There are two ways to specify a target EPLD in which to implement your design:

- By setting the `part` attribute in DC Shell (using FPGA Compiler).
- By specifying a part parameter in the Xilinx `syn2ep1d` command.

## Using the Synopsys Part Attribute

If you are using FPGA Compiler, set the following Synopsys attribute at the DC Shell prompt to specify a target device:

```
set_attribute design_name part -type string  
part_number
```

For example:

```
set_attribute scan part -type string 7354-10PC44
```

This attribute is optional if you specify a part number in the Xilinx **syn2ep1d** command, as shown below. If you are using Design Compiler, you should always specify the part number parameter in the **syn2ep1d** command.

## Using the Xilinx Syn2EPLD Command

When flattening the netlist file for the fitter after compiling, you can also specify a target device by entering the following Xilinx command on the UNIX command line:

```
syn2ep1d -p part_number design_name [.sedif]
```

For example, at the UNIX prompt enter:

```
syn2ep1d -p 7354-10PC44 scan
```

**Note:** A valid part number specified in the **syn2ep1d** command will override any part number specified as a DC Shell attribute.

See the “Attributes” appendix for more information on attributes. See the “Fitting Your Design” chapter for more information on fitter commands.

## Specifying Pin Locations

Specify the device pins on which to place signals by using one of the following Synopsys attributes in DC Shell.

If you are using FPGA Compiler, enter the following:

```
set_attribute port_name pad_location -type string  
pin_number
```



For example, to place the “start” signal on pin 23, using FPGA Compiler:

```
set_attribute start pad_location -type string P23
```

You can use the location attribute to explicitly override any saved pinout made with the fitter **pinsave** command. See the “Fitting Your Design” chapter for more information on the **pinsave** command.

**Note:** The method of specifying pin numbers depends on the target device package type. See the “Attributes” appendix for more information.

## Controlling Design Performance

Devices in the Xilinx EPLD family include Fast Function Blocks (FFBs) and/or High Density Function Blocks (HDFBs). Fast Function Blocks provide the shortest delay paths while High Density Function Blocks provide the most logic resources. EPLDs also contain special high speed routing for clocks, output enable signals, clock enable signals, and logic inputs to FFBs.

You can control your design performance by using attributes to assign specific signals in your design to the appropriate physical EPLD resources.

### Using High-Speed Clocks

Xilinx EPLDs have dedicated high-speed (FastCLK) routing that can be used for global clock signals. Any unused FastCLK routing is automatically assigned by the fitter to the most used clock signals in your design (if eligible) unless you turn off optimization. To be eligible for FastCLK optimization, an input port signal must be used only for register clocking using the positive clock edge.

**Note:** EPLD Fast Function Blocks, input pad registers, and input pad latches must use FastCLK routing; they cannot use normal signal routing for clocks.

### Assigning Specific High-Speed Clocks

If you want to assign a specific clock in your design to a FastCLK net instantiate the **BUFG** buffer cell in your design.

**Note:** Signals driven by the `BUFG` buffer always use FastCLK routing independent of the `NO_FCLK` attribute.

### Preventing FastCLK Optimization

To prevent the fitter from automatically assigning your clock signals to any unused high-speed FastCLK nets, instantiate the `NO_FCLK` cell in your source design as follows:

```
U1: NO_FCLK;
```

where U1 is any instance name.

### Selecting EPLD Function Block Types

By assigning logic signals to specific EPLD Function Block resources, you can control the performance of logic paths in your design.

#### Specifying High-Speed Paths

To assign a logical signal to a Fast Function Block (shortest delay paths), instantiate the `F` attribute cell in your source design and connect it to the intended FFB output signal as follows:

```
instance_name: F port map (signal_name);
```

For example:

```
U1: F port map (OE_REG);
```

#### Specifying High-Density Paths

To assign a logic signal to a High Density Function Block (normal delay paths), instantiate the `H` attribute cell in your source design and attach it to the intended HDFB output signal as follows:

```
instance_name: H port map (signal_name);
```

For example:

```
U1: H port map (DONE_REG);
```

See the “Attributes” appendix for more information.

## Using EPLD FastInputs

Some of the inputs to FFBs can be taken directly from input pins using a high-speed FastInput path which bypasses the Universal Interconnection Matrix. To assign input port signals to the EPLD FastInputs, instantiate the `F` attribute cell in your source design and connect it to the intended FastInput signal as follows:

```
instance_name: F port map (signal_name);
```

For example, to assign the `fast_in1` signal to an EPLD Fast Input:

```
U1: F port map (fast_in1);
```

**Note:** An input signal declared as a FastInput can also be used as the `D` input to an input pad register or latch (`IFDX1` or `ILD`).

## Selecting Low-Power Operation

Macrocells in most EPLD devices can be configured to operate in either high-speed (default) or low-power mode. To specify that all macrocells in the device are to operate in low-power mode, instantiate the global `LOWPWR` attribute cell in your source design as follows:

```
U1: LOWPWR;
```

where `U1` is any instance name.

## The Design Rule Checker

The Design Rule Checker (DRC) reads the design from the database and checks to see if any of the design rules have been violated. The following is a partial list of rules that are checked.

### General Design Rule Violations

The DRC displays an error or warning if:

- Open (hanging) inputs are found. Unless otherwise specified, all inputs of a library component must be connected or tied to VCC or GND.
- Some library components can only be used for a particular target EPLD. The DRC will generate an error if you attempt to use these

components for other EPLDs. Restrictions on the use of components can be found in the library data sheets.

## **Pad Component Design Rule Violations**

Pad component correct usage and applications are illustrated in the Library data sheets. The DRC displays an error if:

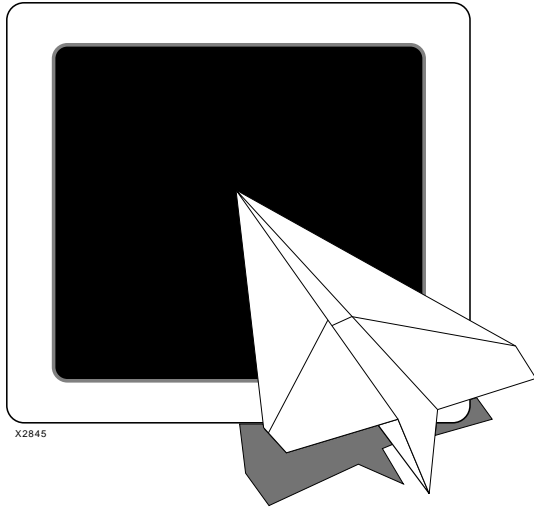
- Two component outputs are connected to the same pad.
- One component output is connected to two pads.
- An input pad is connected directly to an output or I/O pad.
- Pad pins are driven by VCC or GND.
- Pad clocks are driven by VCC or GND.
- Multiple input buffers are connected to the same pad (the exception is when an `IBUF` is used with an `IFD`, `IFDX1`, or `ILD` to receive a FastInput signal).
- A pad is connected to a component other than an I/O buffer, or to another pad.
- An `IPAD` is connected to an `OBUF`-type component.
- An `OPAD` is connected to an input or control-input buffer (such as `IBUF`, `BUFG`, or `IFD`).

## **FastCLK, Clock Enable, and Fast Output Enable Violations**

The DRC displays an error if:

- There are more FastCLK, CE, or FOE pins in the design than the target EPLD can support.
- A FastCLK, CE, or FOE signal drives a component pin that is not a clock, CE, or FOE input.
- A combination of fast clocks for logic components and I/O pads cannot be supported by the target EPLD.
- The clocking requirement of a component is not met. Some component clock inputs can only be driven by a fast clock and others only by a logic clock. Component clocking requirements are listed in the library data sheets.





***Compiling Your Design***

# ***Xilinx Synopsys Interface EPLD User Guide***

## Chapter 3

### Compiling Your Design

---

XSI supports both VHDL and Verilog HDL design synthesis. Either the Synopsys FPGA Compiler or Design Compiler can be used to compile EPLD designs; there are no differences between the two compilers in either features supported or in mapping efficiency. In the following discussion, the term "compiler" refers to either FPGA Compiler or Design Compiler.

This chapter describes how to compile your design using the Synopsys Design Compiler shell (DC Shell). You can also use the Synopsys graphical user interface, Design Analyzer, to process your designs.

Before compiling you will need to develop your VHDL or Verilog HDL source file (*design\_name.vhd* or *design\_name.v*). Usually it is a good idea to perform a functional simulation of your VHDL source design before trying to synthesize it. See the "Simulating Your Design" chapter for information on functional simulation.

### Using Synopsys DC Shell

The Synopsys compiler synthesizes your source design and creates a netlist file composed of logic primitives that is used by the Xilinx fitter (XEPLD) to implement your design in an EPLD. All compiler commands are executed from within the Synopsys DC Shell environment

## Step 1 — Entering the DC Shell Environment

Enter the Synopsys DC Shell environment by entering the following Synopsys command on the UNIX command line:

```
dc_shell
```

You will see the DC Shell prompt.

## Step 2 — Analyzing the Design

To interpret your design and verify that it is free of errors, enter the following Synopsys command for VHDL designs:

```
analyze -format vhd1 design_name.vhd
```

or, for Verilog HDL designs:

```
analyze -format verilog design_name.v
```

For example, the command used in the scan example in the “Getting Started with Xilinx EPLDs” chapter:

```
analyze -format vhd1 scan.vhd
```

If your source file contains initial signal values (which are used only for functional simulation) they will cause warnings that can be safely ignored; these initial signal values are not used during synthesis. Actual register initial states are set using attributes as shown in the “Attributes” appendix.

If the **analyze** command finds errors, you will need to make the necessary corrections to your source file before continuing with synthesis.

If the **analyze** command is successful, you can continue to the next step which builds your logic design.

## Step 3 — Elaborating the Design

To derive a logical design, based on your VHDL/HDL description, enter the following Synopsys command:

```
elaborate design_name
```



For example, the command used in the scan example in the “Getting Started with Xilinx EPLDs” chapter:

```
elaborate scan
```

During this step, the compiler displays information about all registers and 3-state buffers encountered in your design.

You are now ready to compile your design using the XC7000 target library.

## Step 4 — Compiling Your Design

When you compile your design, the Synopsys synthesizer uses the components in the Xilinx XC7000 technology library to create an actual implementation of your design.

To synthesize your design based on the XC7000 technology library enter the following Synopsys command:

```
compile [-map_effort low]
```

The mapping effort parameter is optional. However, it is recommended that you set it to LOW to save compilation time. The synthesizer does not perform any speed or area optimization for EPLD designs; this optimization is performed after compilation by the XEPLD fitter.

## Step 5 — Defining EPLD I/O Signals

Now you must define which signals are connected to the physical I/O pins of the EPLD.

Use the following command to identify all ports in your design for which the synthesizer needs to infer an I/O buffer:

```
set_port_is_pad port_name
```

Do not use this command for any ports that are already instantiated using I/O buffer cells from the library.

To automatically place I/O buffer cells on all top-level ports in the design, enter the following Synopsys commands:

```
set_port_is_pad ""
```

For the ports that were specified by `set_port_is_pad`, the following command adds the appropriate I/O buffer cells to your design:

```
insert_pads
```

## Step 6 — Specifying Attributes

Attributes are used to control the physical implementation of your design; all attributes are optional. If you are using FPGA Compiler, the attributes that you may want to set at this time are:

- Part type (you can also set part type from the fitter command line).
- Register initial states.
- Pin assignments.

For example, the attributes used in the scan example in the “Getting Started with Xilinx EPLDs” chapter:

```
set_attribute scan part -type string 7354-10pc44
set_attribute find(cell COUNT*)
    fpga_xilinx_init_state -type string R
set_attribute find(cell OE_REG*)
    fpga_xilinx_init_state -type string R
```

See the “Attributes” appendix for complete details on all supported attributes.

The design database is now complete and you are ready to output a netlist file for the Xilinx fitter.

## Step 7 — Writing the Netlist

If you are using FPGA Compiler, write your synthesized design file in XNF netlist format by entering the following Synopsys command:

```
write -format xnf -hierarchy -output
    design_name.sxnf
```

where:

- `-format xnf` specifies the XNF file format.
- `-hierarchy` specifies that all levels of the design hierarchy are to be written.

- **-output** *design\_name.sxnf* specifies your output file name, which should be the same as your source file name, with the extension: *.sxnf*.

For example, the command used in the scan example in the “Getting Started with Xilinx EPLDs” chapter:

```
write -format xnf -hierarchy -output scan.sxnf
```

If you are using Design Compiler, you must write your synthesized design file in EDIF netlist format by entering the following Synopsys command:

```
write -format edif -output design_name.sedif
```

where:

- **-format** *edif* specifies the EDIF file format.
- **-output** *design\_name.sedif* specifies your output file name, which should be the same as your source file name, with the extension: *.sedif*.

For example, if you have only Design Compiler, you would write the scan design from the “Getting Started with Xilinx EPLDs” chapter using the following command:

```
write -format edif -output scan.sedif
```

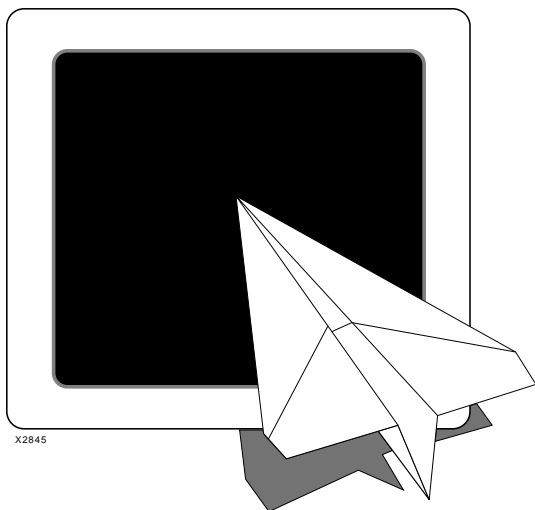
This is the end of the process in DC Shell. You usually exit DC Shell before fitting. Before exiting you may wish to save the design database in Synopsys *db* format by executing the **write** command. You can exit DC Shell by entering the following Synopsys command:

```
exit
```

None of the Synopsys timing or area analysis reports are useful at this time because the XC7000 technology library does not contain timing or area estimation data. The Xilinx fitter provides a Static Timing Report which shows the calculated worst case timing for each logic path in your design.

You are now ready to begin the fitting process as described in the next chapter.





***Fitting Your Design***

# ***Xilinx Synopsys Interface EPLD User Guide***

## Chapter 4

### Fitting Your Design

---

After you have compiled your logic design using the Synopsys compiler you are ready to implement your design in an EPLD. This chapter shows you how to fit your design, create a device programming file, and save your pinouts for design iteration.

#### Fitter Overview

XEPLD is the Xilinx EPLD fitter software. XEPLD uses the logical design produced by the Synopsys compiler to create a physical layout for a target EPLD.

XEPLD performs the following functions:

- Reads the netlist file (*design\_name.sxnf* or *design\_name.sedif*) produced by the Synopsys compiler and reports any rule violations to the error log file (*design\_name.err*).
- Minimizes the combinational logic of your design so that it requires the least number of product term resources.
- Optimizes, partitions, and maps your design to fit within the architecture of the target device.
- Creates a pin-save file (optional) that is used to lock signal names to device pins, allowing you to keep the device pinouts during design iterations.
- Creates a Static Timing Report that shows the calculated worst-case timing for all signal paths in your design.
- Creates a timing simulation files that can be used by the Synopsys VSS simulator.
- Creates a device programming file (*design\_name.prp*).

- Creates detailed reports that show you information such as the type and quantity of device resources used and device pinouts.

## Fitter Operation

The following steps show you how to fit your logical design into a target device using the XEPLD fitter.

### Step 1 — Create a Flattened XNF Netlist File

The Synopsys FPGA Compiler produces a top-level XNF-formatted file named *design\_name.sxnf*. The Synopsys Design Compiler produces a top-level EDIF-formatted file named *design\_name.sedif*. Either of these files may contain macros and must be translated into a flattened netlist file for the Xilinx fitter by using the **syn2ep1d** command.

If you did not specify a target device part number to the Synopsys DC Shell (by using a **part** attribute) you must specify a target device type at this time.

#### Specifying a Target Device

To create the flattened netlist, and specify a target device, enter the following Xilinx command on the UNIX command line:

```
syn2ep1d -p part_number design_name [.sedif]
```

If you omit the **.sedif** extension, **syn2ep1d** looks for a *design\_name.sxnf* file (produced by FPGA Compiler). If you specify the **.sedif** extension, **syn2ep1d** reads the EDIF netlist produced by Design Compiler.

For example, at the UNIX prompt enter:

```
syn2ep1d -p 7354-10PC44 scan
```

This command creates a flattened netlist which is saved as *design\_name.xff*.

**Note:** A valid part number specified in the **syn2ep1d** command will override any part number specified in a DC Shell attribute.

A complete list of device types is shown in the “EPLD Architecture” appendix.

## Using a Target Device Specified By a Part Attribute

If you are using FPGA Compiler and you specified a part attribute in DC Shell, you can enter the `syn2ep1d` command without the `-p` parameter on the UNIX command line as follows:

```
syn2ep1d design_name
```

## Step 2 — Fit Your Design

To invoke the fitter, enter the following Xilinx command on the UNIX command line:

```
fitnet -n design_name [options]
```

### Options

**-i**

Ignore the pin assignments specified by `pad_location` attributes.

**-u**

Drive all unused I/O pads to GND.

**-f**

Use the previously saved (frozen) pinout.

For example, to invoke the fitter for the scan design, driving all unused EPLD I/O signals to GND, enter the following command:

```
fitnet -n scan -u
```

The `fitnet` command produces various reports:

- The Resource Report (*design\_name.res*) indicates how well your design fits in the target device. This report shows the utilization of macrocells, Function Blocks and each type of device pin, and indicates the amount of remaining logic and I/O resources in the device. The Resource summary is also displayed near the end of the `fitnet` process.
- The Pinlist Report (*design\_name.pin*) shows the signals assigned to each pin of the target device.



- The Partitioner Report (*design\_name.par*) and the Mapping Report (*design\_name.map*) show the detailed physical layout of your design within the EPLD.
- The Equation File (*design\_name.eqn*) contains boolean equations representing the final implementation of your design after minimization and optimization, and is expressed in Xilinx PLUSASM syntax.

## Step 3 — Verify Your Design Timing

Generate a Static Timing Report and timing simulation files by entering the following Xilinx command on the UNIX command line:

```
vmh2vss [-b | -t] design_name
```

### Options

#### -b

Allows you to use the same test bench file for timing simulation and for functional simulation. It generates the *design\_name\_vss.vhd* file as architecture only, using the original bus indexes matching the port declarations in your original source design file (*design\_name.vhd*). The **-b** option is the default if you do not specify an option.

#### -t

Generates a new test bench file for timing simulation (*tb\_design\_name\_vss.vhd*). It also generates the *design\_name\_vss.vhd* file with both entity and architecture.

The Static Timing Report is saved as *design\_name.tim*. This report shows the calculated worst case timing for logic paths in your design. See the “Fitter Reports” chapter for a complete description of the Static Timing report.

The timing simulation model produced by **vmh2vss** is composed of two files:

- *design\_name\_vss.vhd* — A structural VHDL design file (for simulation only).
- *design\_name\_vss.sdf* — A Verilog-style timing back-annotation file.

If you select the `-t` option you will also get the following file:

- `tb_design_name_vss.vhd` — A VHDL test bench for timing simulation.

## Step 4 — Create a Device Programming File

After you are satisfied that your design is functioning properly you can create a device programming file. This file is used by an EPLD programmer to implement your design. To create the device programming file, enter the following Xilinx command on the UNIX command line:

```
makeprg design_name [-s signature]
```

Where *signature* is an optional user-defined signature string that is programmed into the device for identification.

This command creates a file named `design_name.prp`.

EPLD programmers are available from Xilinx and from third-party developers. See your device programmer documentation for instructions on how to download the programming file.

## Step 5 — Save Your Pinouts

Modifying EPLD designs after PC board layout requires the ability to save and re-use device pin location information.

To save your pinouts for use in future design iterations, enter the following Xilinx command on the UNIX command line:

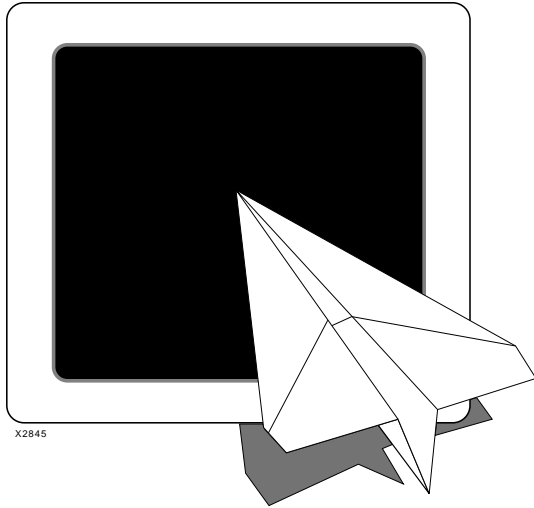
```
pinsave design_name
```

This command creates a pin-save file named `design_name.vmf`. During a subsequent invocation of the `fitnet` command, you can specify the `-f` option to restore pinouts saved in the `.vmf` file.

**Note:** Making major changes to your design may prevent the fitter from achieving a successful mapping if you use the `fitnet -f` option. If the fitter fails, try running without the `-f` option to see if a fit is still possible. To fit a modified design into the selected device, you may need to delete some of the pin assignments in the `.vmf` file, allowing those pins to move to new locations.

**Note:** You cannot use a previously saved pinout if you change the size or package type of the target device. See the EPLD Data Book to determine which devices in the EPLD family have compatible pinouts across similar packages.

After a successful fit of your design you are ready to perform timing simulation as described in the next chapter.



# ***Xilinx Synopsys Interface EPLD User Guide***

***Simulating Your  
Design***

## Chapter 5

### Simulating Your Design

---

The Xilinx EPLD Synopsys Interface supports both functional and timing simulation of designs using the VSS simulator. This chapter shows you how to prepare designs for simulation and how to use a test bench.

#### Recommended EPLD Simulation Strategy

Because of the flexibility of the simulation environment, there are many ways in which you can verify your design. The following steps, which are explained in subsequent sections, show you one recommended flow for EPLD simulation.

1. Specify The initial states of your registers. If you use attributes to control the initial states of the registers in your actual design implementation, you must also re-specify those initial states in your source design file for functional simulation.
2. Create a test bench file. By following the guidelines described in this chapter, the same test bench can be used for both functional and timing simulation.
3. Perform functional simulation. This allows you to debug the logic in your source design before implementing an EPLD.
4. Implement the design in an EPLD. This provides the necessary physical resource information necessary for timing simulation.
5. Prepare the timing model. The `vmh2vss` software prepares the timing model of your design for simulation and provides a static timing report.

6. Perform timing simulation. By re-using the functional simulation test bench file, you can easily compare results and prevent errors that can be caused by accidental differences between separate test bench files.

All of these preparation and simulation steps are demonstrated in the design example shown in the “Getting Started with Xilinx EPLDs” chapter.

## Controlling the 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, then you can skip this section and go to the next section, “Creating a Test Bench File”.

The actual initial states of your registers are determined by the initial state attributes specified in DC Shell during compilation or by the default initial states which are specified for each registered cell in the Xilinx component library.

**Note:** If preload optimization is turned OFF, the default initial states as defined in the library, will not be changed by the Xilinx software. If preload optimization is turned ON, then the initial states may be changed by the fitter during optimization. See the “Attributes” appendix for more information on preload optimization control.

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 fitter.

## Simulating Master Reset

Xilinx EPLDs have a Master Reset function that initializes the device registers either when power is applied or when the MR input pin is pulsed. The reset signal is not physically available to your logic. However, you must pulse the reset signal at the beginning of timing simulation for proper register initialization.

The following sections show you how to simulate the Master Reset function for both functional and timing simulation.

## Preparing for Timing Simulation

When you generate your timing simulation model, `vmh2vss` automatically creates a new input port named `MRESET`. When simulating, you must first pulse `MRESET` low, prior to exercising the logic, to get all the registers into their initial states. If you use a test bench to stimulate your design, you must include the `MRESET` signal as one of the input ports of the EPLD in the test bench as described in the next section “Creating A Test Bench File”.

The `MRESET` signal is used for timing simulation only; it is not used for functional simulation and it cannot be used in your design. However, if you include it in your functional simulation test bench, that test bench can also be used later for timing simulation without modification.

If you are using the same test bench file for both functional and timing simulation, you must also include the `MRESET` port declaration in your source design file as follows:

```
port (... MRESET : in std_logic ...);
```

`MRESET` is not used anywhere else in your design. During synthesis you will get warnings about the unconnected `MRESET` port (during the **Compile** and **Insert Pads** operations). The Xilinx fitter software will also ignore the unconnected `MRESET` port during implementation.

See the scan tutorial source file listing for an example of how the `MRESET` input port is declared in a VHDL design.

## Preparing for Functional Simulation

Simulate register initialization (Master Reset) by defining, in your source design file, the initial values for registered signals. 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:

```
port Nreg5 out std_logic := '0';
signal Qreg6: std_logic := '0';
variable Qreg: std_logic_vector := "00000001";
```

These initial values are used only for functional simulation; they are not used during synthesis and the synthesizer will give you a warning that these values are being ignored. Also, these initial values are not used by the Xilinx software for device implementation.

**Note:** The fitter can change the initial states of registers during optimization (assuming that preload optimization remains enabled). Therefore, for functional simulation, you should declare only the initial states that will actually be implemented by the Xilinx fitter, based on your specifications. These states are specified in your source design file by using initial state attributes in DC Shell.

You are now ready to create a test bench file.

## Creating a Test Bench File

This section shows you how to create a test bench file that can be used for both functional and timing simulation. The example test bench presented here consists of a VHDL file containing one instance of an EPLD design being tested and a procedure that applies simulation input waveforms to the EPLD.

### 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 `MRESET` input port in the EPLD component declaration and in its instance port map as shown in Figure 5-1. At the beginning of the simulation sequence, applying an active-low pulse to `MRESET` initializes the registers. This pulse is ignored during functional simulation because the `MRESET` signal is not used anywhere in the source design.

During `vmh2vss` (after `fitnet`) the `MRESET` port is automatically generated in the timing simulation model. Then, during timing simulation when the test bench applies the `MRESET` pulse, the timing simulation model will initialize all registers as they are actually implemented in the EPLD.



**Note:** In designs targeted for the XC7318, XC7336, or XC7354, if you specify the MRINPUT attribute, the device will not have a Master Reset pin. However, the timing model will still respond to the reset pulse on the port in order to simulate a power-on reset function, which is always performed by the EPLD when power is applied.

## Configuration Declaration

For any design or test bench you wish to simulate, you must declare a configuration which 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 5-1 shows a typical configuration declaration in a test bench file. If the test bench is always used to simulate the design source file, it does not need its own configuration declaration.

```
library xc7000
use xc7000.components.all;...                               --and other packages--

entity scan_tb is
end scan_tb;                                                --test bench has no ports--

architecture test of scan_tb is
  component scan
    port (CLOCK, CLEAR, ...                                --same as in scan.vhd--
          MRESET : in std_logic);
  end component;
  signal CLOCK, CLEAR, ...MRESET;                          --same as ports of scan.vhd--
begin
  UUT: scan port map (CLOCK, CLEAR, ... MRESET);           --connect local signals to ports--
  driver: process begin
    MRESET <= '0';CLEAR <='0';...                          --assert initial values on all inp ports
    wait for 25ns;                                          --wait, repeat--
    MRESET <= '1';...                                      --release MRESET before applying other
                                                         input transitions--
    wait;                                                  --after all inputs, suspend process--
  end process;
end test;

configuration CFG_SCAN_TB of scan_tb is
  for test
  end for;
end CFG_SCAN_TB;
```

**Figure 5-1 Simulation Test Bench — SCAN Design**

After you have created a test bench file, you are ready to begin using a VSS simulator (such as vhdldb) for functional simulation.

## Functional Simulation

Functional simulation is used to debug your logic before fitting your design into an EPLD. The Xilinx EPLD Synopsys Interface fully supports functional simulation of all cells in the XC7000 library (including all DesignWare operators).

To prepare a test bench configuration for simulation, you must analyze each of the design and test bench source files in the proper bottom-up sequence.

The following procedure uses the stand-alone VHDL Analyzer (**vhdlan**) and the VHDL Debugger Simulator (**vhdlldb**).

1. Analyze your source EPLD design file. Enter the following UNIX command:

```
vhdlan design_name.vhd
```

For example:

```
vhdlan scan.vhd
```

2. Analyze the test bench file. Enter the following UNIX command:

```
vhdlan test_bench_name.vhd
```

For example:

```
vhdlan scan_tb.vhd
```

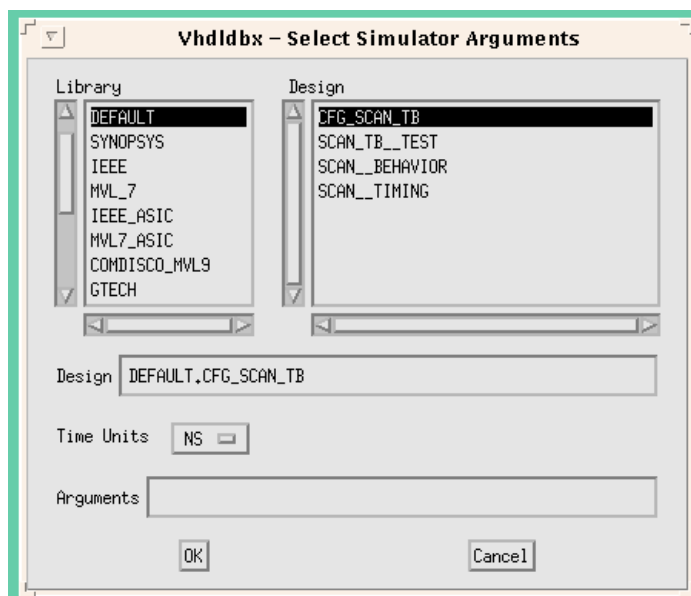
3. Invoke the Synopsys VSS Simulator. Enter the following UNIX command to invoke the VHDL debugger:

```
vhdlldb
```

You are then prompted for a configuration name. Select the name of the configuration declared in the *test\_bench\_name.vhd* file. For example, for the scan design, select the following:

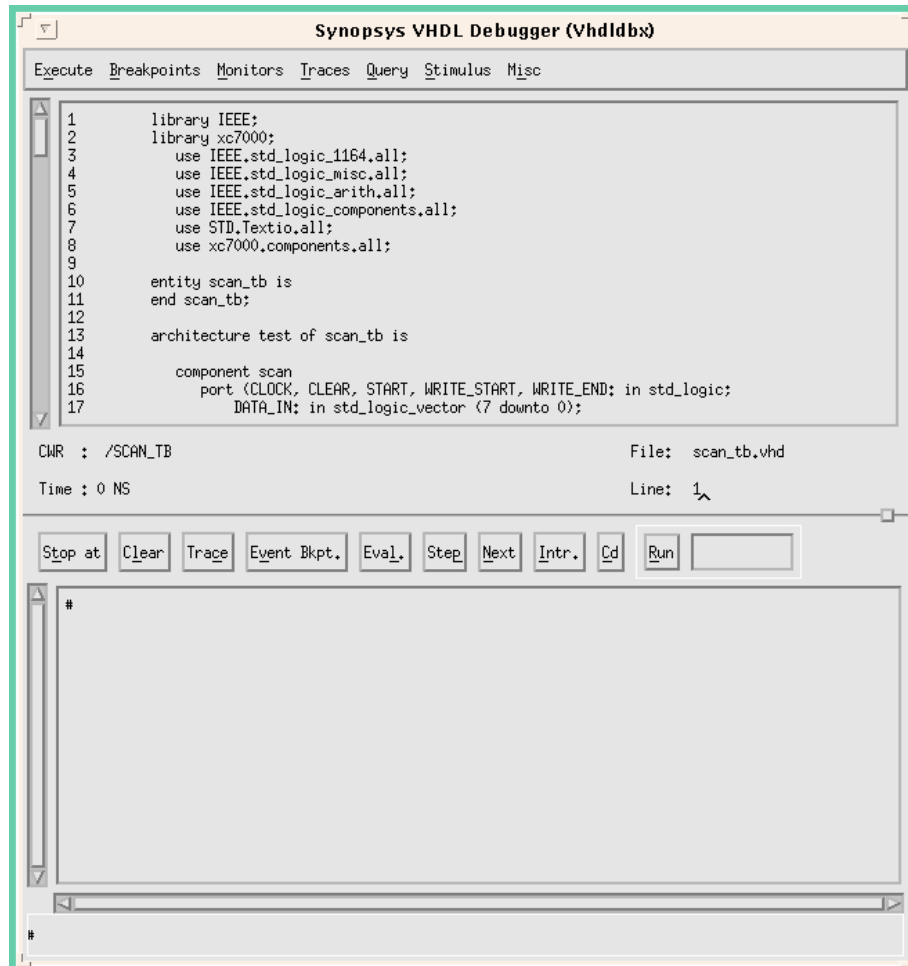
```
CFG_SCAN_TB
```

The **vhdlldb** selector window appears as shown in Figure 5-2.



**Figure 5-2 Selector Window (vhdlDbx)**

After you click OK, the vhdlDbx user interface window appears as shown in Figure 5-3.

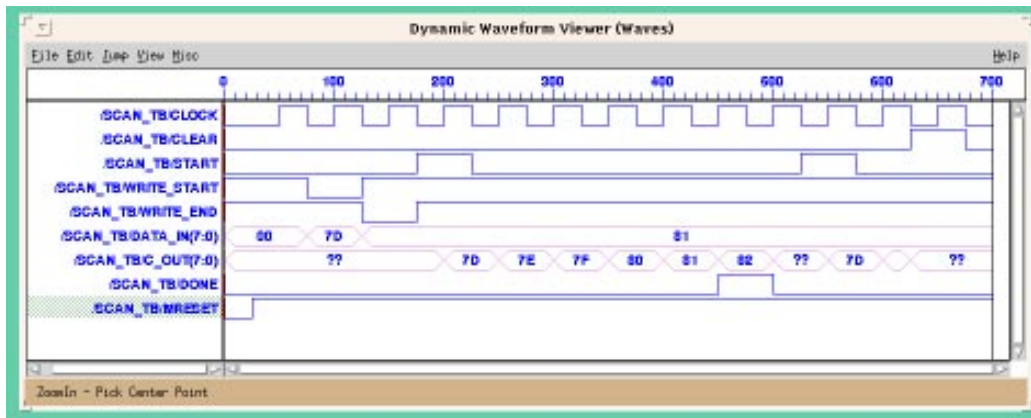


**Figure 5-3 User Interface Window (vhldbx)**

To run your simulation, typically you first declare the signals you want to display in a trace window. For example, to display all signals appearing on the EPLD pins, you can enter the following vhldbx command:

```
trace *'signal.
```

To run all the simulation vectors in your test bench, select the **RUN** command. The resulting trace window will look similar to Figure 5-4



**Figure 5-4 Functional Simulation Waveforms — SCAN Design**

After functional simulation is successful, you are ready to implement your design and create the physical layout information required for timing simulation.

## Design Implementation

After you have debugged your design using functional simulation, you can compile it using synthesis and implement it in an EPLD using the Xilinx fitter. Design implementation is a prerequisite for performing timing simulation.

You can use DC Shell or you can use the Synopsys graphic interface (Design Analyzer) to create the XNF or EDIF netlist file required by the Xilinx fitter. This gate-level netlist file consists of cells from the XC7000 library but does not contain timing information. The Xilinx fitter processes the netlist file and places the logical design into the physical architecture of a target EPLD.

After the design is implemented by the Xilinx fitter, the actual target device timing information is available for timing simulation.

The following steps show you an overview of the EPLD implementation procedure.

1. Analyze the source design file. This must be repeated in the synthesis environment (DC Shell); the results of `vhdlan` cannot be used for synthesis.
2. Compile the design, targeting the XC7000 library, and create a netlist.
3. Run the Xilinx fitter, using the `fitnet` command to process the netlist.

Usually, simulation is not repeated until after fitting when all actual timing results have been applied.

To verify that the fitting process is completed, review the error file (`design_name.err`) which shows all errors and warnings that occurred during implementation. These errors are also displayed on screen as the process is running. You can also examine the Resource Report (`design_name.res`) which is also displayed to the screen as the process is running. The Resource Report tells you how well your design fits into the target device. You may wish to target a smaller device or add more functions to your design if there are remaining unused resources.

After design implementation, you are ready to prepare the timing model for timing simulation.

## Preparing the Timing Model

When you synthesize your design, and create an XNF or EDIF netlist file for the Xilinx fitter, all busses (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 fitting process.

The `vmh2vss` 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 `vmh2vss` with the `-b` option** (the default) generates the timing model as an architecture only, without the entity. The external signals appearing in the design, that were originally defined as bus ports, will then be represented within the model architecture using subscript notation compatible with bus port declarations. By re-using the entity from your source design with the architecture of the timing model (produced by `vmh2vss -b`),

you can perform timing simulation using the same test bench and chip interface as used for functional simulation. For example:

```
vmh2vss scan
```

- **Using vmh2vss with the -t option** generates the timing model as a complete VHDL design. The entity of that design will list the individual signals that comprise the busses in the original design. However, the original bus structure is not preserved. This normally forces you to modify the chip interface in your functional simulation test bench before using it for timing simulation. This is because your original test bench interfaces to the EPLD using bus ports and cannot interface to the timing model. For example:

```
vmh2vss -t scan
```

The **vmh2vss** software also creates a static timing report which shows you the calculated timing for the logic paths in your design. You should review this report (*design\_name.tim*) for satisfactory timing, before simulation. At this point you may need to rerun the fitter, specifying a different EPLD speed grade. Also, to achieve the required timing, you may need to modify your design or apply attributes to control the mapping of speed-critical paths.

When you are satisfied with your static timing results, you can proceed to timing simulation.

## Timing Simulation

Timing simulation is performed after implementing your design (using the Xilinx **fitnet** command), creating the timing model (using **vmh2vss**), and reviewing the static timing report.

If you prepared your test bench properly, and used the **-b** option (default) of **vmh2vss**, you can use 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 re-use the port declarations in its entity. Enter the following UNIX command:

```
vhdlan design_name.vhd
```

For example:

```
vhdlan scan.vhd
```

2. Replace the architecture of your source design with the timing architecture produced by **vmh2vss -b**:

```
vhdlan design_name_vss.vhd
```

For example:

```
vhdlan scan_vss.vhd
```

The architecture is replaced in the Synopsys data base 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. Enter the following UNIX command:

```
vhdlan test_bench_name.vhd
```

For example:

```
vhdlan scan_tb.vhd
```

The simulation data base now contains the test bench design which 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. Enter the following UNIX command:

```
vhdl1dbx
```

You are then prompted for the configuration named in the *test\_bench\_name.vhd* file. For example, for the scan design, select the following:

```
CFG_SCAN_TB
```

Before clicking "OK" you must specify the timing backannotation file information in the Arguments box.



All backannotated timing in the `.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 EPLD design instance to which you want to apply the back-annotated timing. It can then find all the referenced instances.

If you are using **vhdlbxb** you need to specify two parameters:

- The file name of the `.sdf` backannotation timing file:

**-sdf design\_name\_vss.sdf**

For example:

**-sdf scan\_vss.sdf**

- The `sdf_top` instance in the test bench configuration to which the backannotated timing is applied:

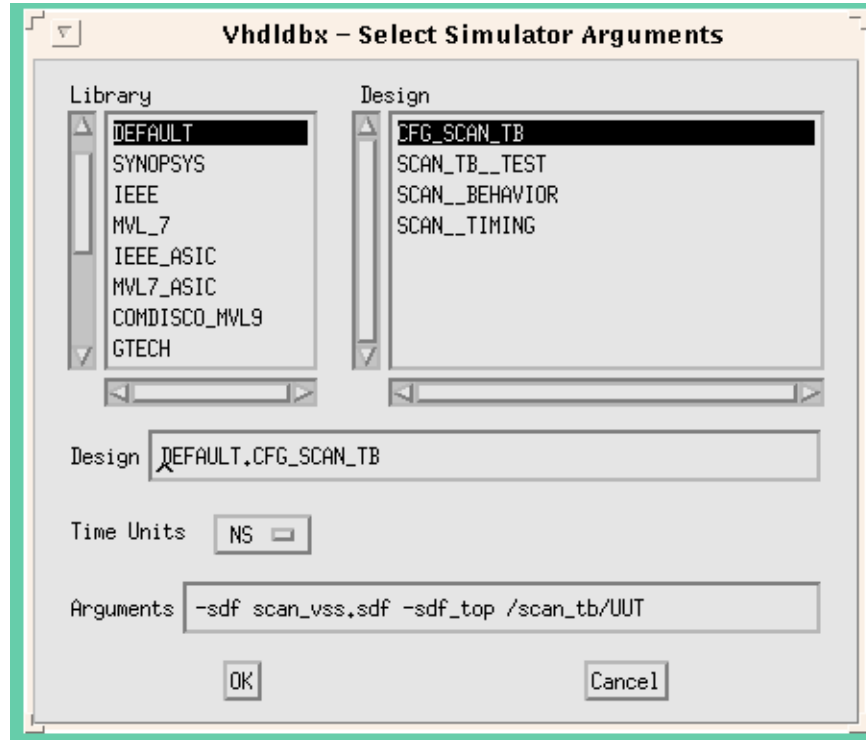
**-sdf\_top chip\_instance\_name**

For example:

**-sdf\_top /scan\_tb/UUT**

All backannotated timing parameters in the `.sdf` file are applied relative to the chip instance.

You can specify these parameters either in the dialog box which appears after invoking **vhdlbxb** (as shown in Figure 5-5), or on the UNIX command line as you invoke **vhdlbxb**.



**Figure 5-5 Selector Window with Timing Backannotation Parameters Entered (vhdlldb)**

For convenience, you can put all parameters into a command script file. The command line for the scan design is provided in the dbx\_scan script file in the tutorial directory.

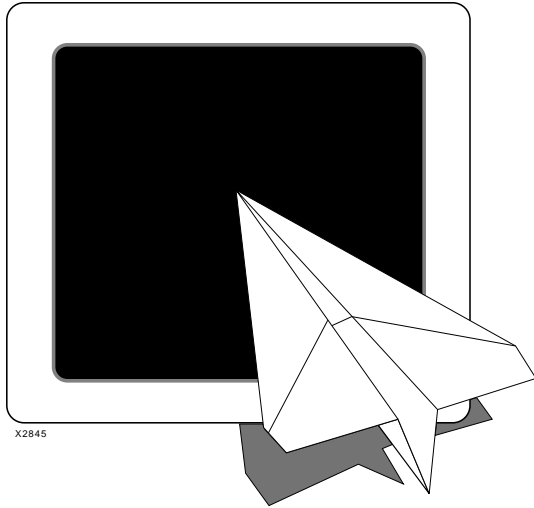
The command line invocation format is:

```
vhdlldb -sdf design_name_vss.sdf -sdf_top
        chip_instance_name configuration_name
```

For the scan design example, you should enter the following:

```
vhdlldb -sdf scan_vss.sdf -sdf_top /scan_tb/UUT
        CFG_SCAN_TB
```





# ***Xilinx Synopsys Interface EPLD User Guide***

***EPLD Architecture***

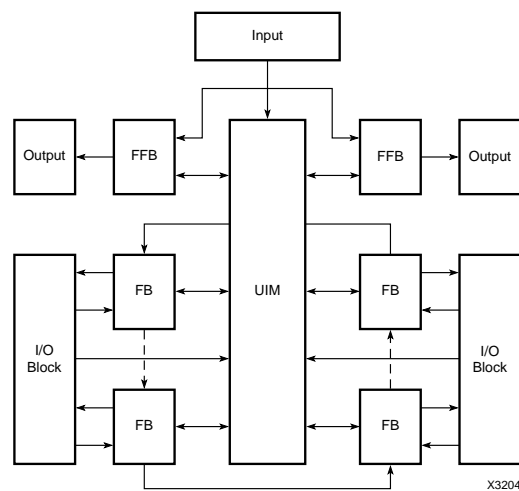
## Appendix A

### EPLD Architecture

---

The Xilinx EPLD family uses a simple PAL-like architecture to provide both high speed and high density in a variety of packages and configurations. Through a unique Dual-Block architecture, High Density Function Blocks (FBs) provide high speed and maximum logic density for implementing complex functions while Fast Function Blocks (FFBs) provide even higher speed for critical decoding and ultra-fast state machine applications. For more information see *The Programmable Logic Data Book*.

The EPLD architecture consists of multiple Function Blocks and I/O blocks interconnected by the UIM as shown in Figure A-1.



**Figure A-1 EPLD Architecture Block Diagram**

## Device Selection

The following table shows the Xilinx EPLD family, grouped by user application. Use this table to select the best target device for your design.

Package options and speed grades are always being updated. Check the latest device data sheets for the most up to date information.

**Table A-1 Device Selection Chart**

Features	Fast Functions		Dual Block Arch. Fast and High Density			High Density Functions	
	7318	7336	7354	7372	73108	7236A	7272A
22VIO Equivalent	2	4	6	8	12	4	8
Macrocells	18	36	54	72	108	36	72
FFBs	2	4	2	2	2	0	0
FBs	0	0	4	6	10	4	8
Flip-Flops	18	36	108	126	198	68	126
Fast Inputs Supported	12	12	12	12	12	0	0
Fast Clock Inputs	2	2	3	3	3	3	2
Fast Output Enab.	2	2	2	2	2	1	0
Fast Clock Enab.	0	0	2	2	2	0	0
1 Pin-to-Pin delay (ns.)	5	5	7	7	7	25	25
1 Clock Frequency (Mhz)	167	167	100	100	80	60	60
2 Signal Pins (max)	38	38	58	74	120	36	72
Speed Grades	-5 -7	-5 -7 -10 -12 -15	-7 -10 -12 -15	-7 -10 -12 -15	-7 -10 -12 -15	-16 -20	-16 -20 -25
44 Pin PLCC	X	X	X			X	
44 Pin CLCC	X	X	X			X	
44 Pin PQFP	X	X					
68 Pin PLCC			X	X			X
68 Pin CLCC			X	X			X
84 Pin PLCC				X	X		X
84 Pin CLCC				X	X		X
84 Pin PGA				X	X		X
100 Pin PQFP					X		
144 Pin PGA					X		
160 Pin PQFP					X		
225 Pin BGA					X		

## The Universal Interconnect Matrix

The Universal Interconnect Matrix™ (UIM) functions as an unrestricted crossbar switch. It guarantees complete interconnection of all internal functions and provides constant, short interconnect delays. It receives inputs from Macrocells, bidirectional I/O pins, and dedicated input pins and provides 21 outputs to each High-Density Function Block and 24 outputs to each Fast Function Block. Any UIM input can drive one or more UIM outputs with the interconnect delay remaining constant.

When multiple inputs are connected to the same output, this output produces the logical AND of the input signals. By choosing the appropriate signal inversions, this AND logic can also implement wide NAND, OR or NOR functions. This provides an additional level of logic with no additional delay.

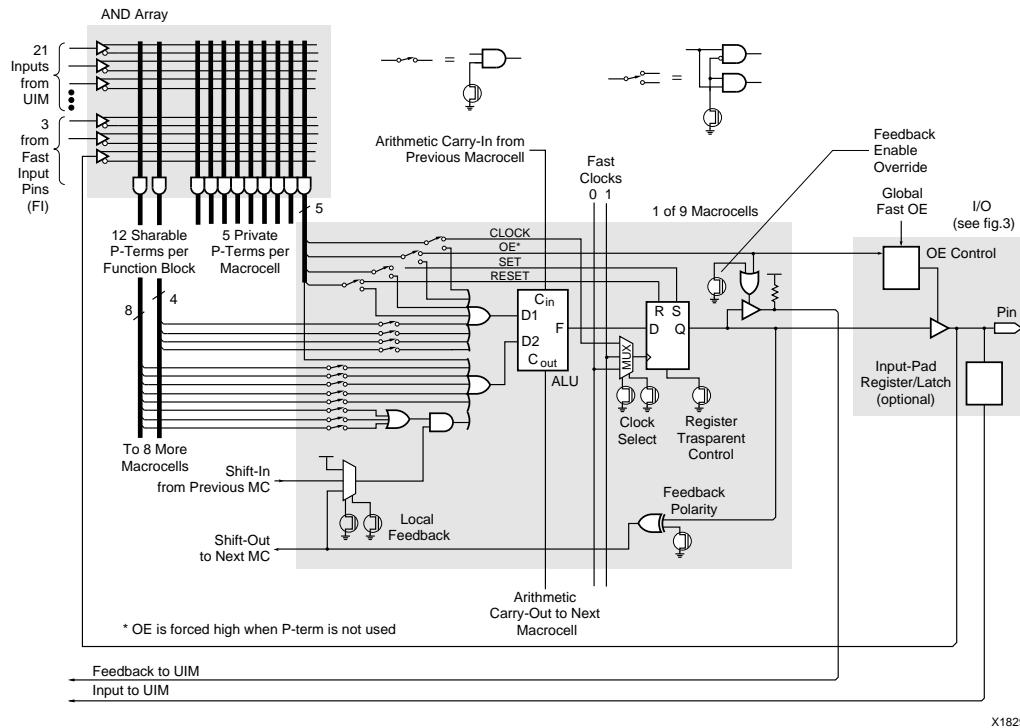
A Macrocell feedback signal that is disabled by the output enable product term represents a High input to the UIM. Programming several such Macrocell outputs onto the same UIM output thus emulates a 3-state bus line. When one of the Macrocell outputs is enabled, the UIM output assumes its level.

## High-Density Function Blocks

Each High Density Function Block contains nine Macrocells which can be configured for either registered or combinatorial logic. A detailed Function Block diagram is shown in Figure A-2.

Each FB receives 21 signals and their complements from the UIM and an additional three inputs from the FastInput (FI) pins.

**Note:** The XC7272A FB architecture, including the ALU, is slightly different. See the data sheet for details.



**Figure A-2 Function Block Schematic**

## Shared and Private Product Terms

Each Macrocell contains five private product terms that can be used as the primary inputs for combinatorial functions implemented in the Arithmetic Logic Unit (ALU), or as individual Reset, Set, Output-Enable, and Clock logic functions for the flip-flop. Each Function Block also provides an additional 12 shared product terms, which are uncommitted product terms available for any of the nine Macrocells within the FB.

Four private product terms can be ORed together with up to four shared product terms to drive the D1 input to the ALU. The D2 input is driven by the OR of the fifth private product term and up to eight of the remaining shared product terms. The shared product terms



add no logic delay and each shared product term can be connected to one or all nine Macrocells in the Function Block.

## Arithmetic Logic Unit

The versatility of each Macrocell is enhanced through additional gating and control functions available in the ALU. A detailed block diagram of the XC7300 and XC7236A ALU is shown in Figure A-3.

The ALU has a logic mode and an arithmetic mode. In logic mode, the ALU functions as a 2-input function generator using a 4-bit look-up table that can be programmed to generate any Boolean function from the D1 and D2 inputs.

The function generator can OR its inputs, widening the OR function to a maximum of 17 inputs. It can AND them, which means that one sum-of-products can be used to mask the other. It can also XOR them, toggling the flip-flop or comparing the two sums of products. Either or both of the sum-of-product inputs to the ALU can be inverted, and either or both can be ignored. Therefore, the ALU can implement one additional layer of logic with no speed penalty.

In arithmetic mode, the ALU can generate the arithmetic sum or difference of the D1 and D2 inputs. Combined with the carry input from the next lower Macrocell, the ALU operates as a 1-bit full adder generating a carry output to the next higher Macrocell. The dedicated carry chain propagates between adjacent Macrocells and crosses the boundaries between Function Blocks providing very fast arithmetic operation with no additional resource requirements.

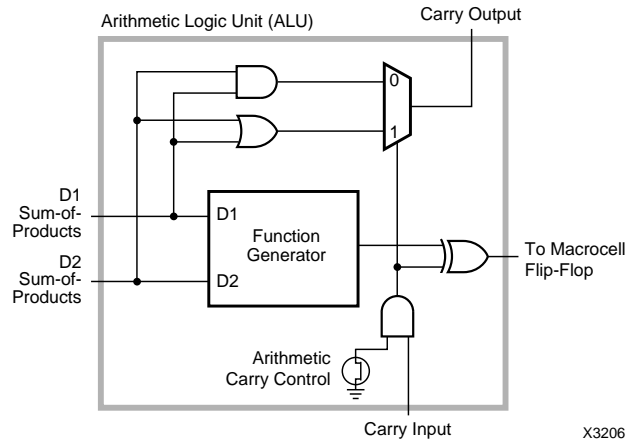


Figure A-3 ALU Schematic

## Carry Lookahead (7300 Family Only)

Each Function Block provides a carry lookahead generator capable of anticipating the carry across all nine Macrocells. This reduces the ripple-carry delay of wide arithmetic functions such as add, subtract, and magnitude compare to that of the first nine bits, plus the carry lookahead delay of the higher-order Function Blocks.

## Macrocell Flip-Flop

The ALU output drives the input of a programmable D-type flip-flop. The flip-flop is triggered by the rising edge of the clock input, and it can be configured as transparent, making the Q output identical to the D input, independent of the clock.

The Macrocell clock source is programmable and can be one of the private product terms or one of the global FastCLK signals. The FastCLK signals are distributed to every Macrocell flip-flop with short delay and minimal skew. The asynchronous Set and Reset product terms override the clocked operation. If both asynchronous inputs are active simultaneously, Reset overrides Set.

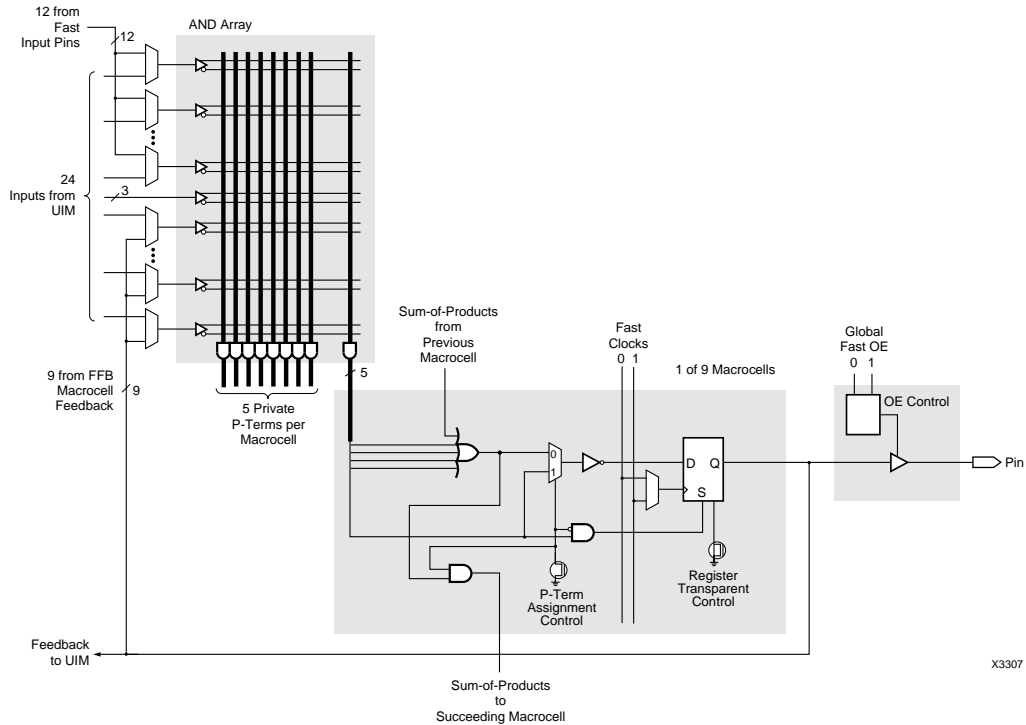
In addition to driving the chip output buffer, the Macrocell output is routed back to the UIM. One private product term can be configured to control the Output Enable of the output buffer and the feedback to

the UIM. If it is configured to control UIM feedback, the Output Enable product term forces the UIM feedback control input High when the Macrocell output is disabled.

## **Fast Function Blocks**

Each Fast Function Block receives 24 signals and their complements from the UIM. The 24 inputs can be individually selected from the UIM, the 12 FastInput pins, or the nine Macrocell feedbacks from the FFB. The programmable AND array in each FFB generates 45 product terms to drive the nine Macrocells, which can be configured for registered or combinatorial logic. The FFB logic is shown in Figure A-4.

Five product terms from the programmable AND array are allocated to each Macrocell. Four of these product terms are ORed together and drive the input of a programmable D-type flip-flop. The fifth product term drives the asynchronous active-high Set Input to the Macrocell flip-flop. The flip-flop can be configured as transparent to produce a combinatorial output.



**Figure A-4 Fast Function Block Schematic for 7354, 7372, 73108, and 73144**

The programmable clock source is one of two global FastCLK signals (FCLK0 or FCLK1) that are distributed with short delay and minimal skew over the entire chip.

The FFB Macrocells drive chip outputs directly through 3-state buffers. Each output buffer can be permanently enabled, permanently disabled, or controlled by one of two dedicated Fast Output Enable inputs. The Macrocell output is also routed back to the FFB and to the UIM. The XC7300 family provides a product term expansion feature that increases product-term flexibility without disabling Macrocell outputs. Product term expansion transfers product terms in increments of four product terms from one Macrocell to the next.

## Product Term Expansion

Complex logic functions requiring up to 36 product terms can be implemented using this method. When product terms are assigned to adjacent Macrocells, the product term normally dedicated to the Set function becomes the D-input to the Macrocell register. Thus, the Macrocell is still usable while product terms are transferred to adjacent Macrocells.

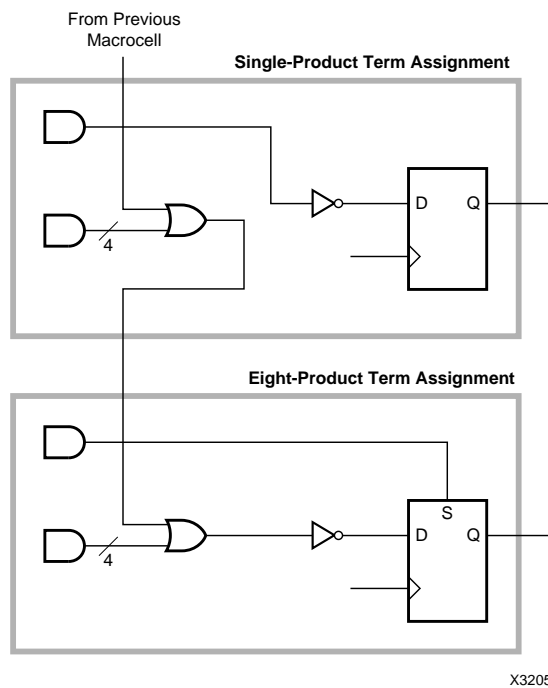


Figure A-5 FFB Product Term Expansion

## XC7336 and XC7318 Fast Function Blocks

The Fast Function Blocks within the XC7318 and XC7336 are slightly different from those in the rest of the Xilinx EPLD family as shown in Figure A-6.

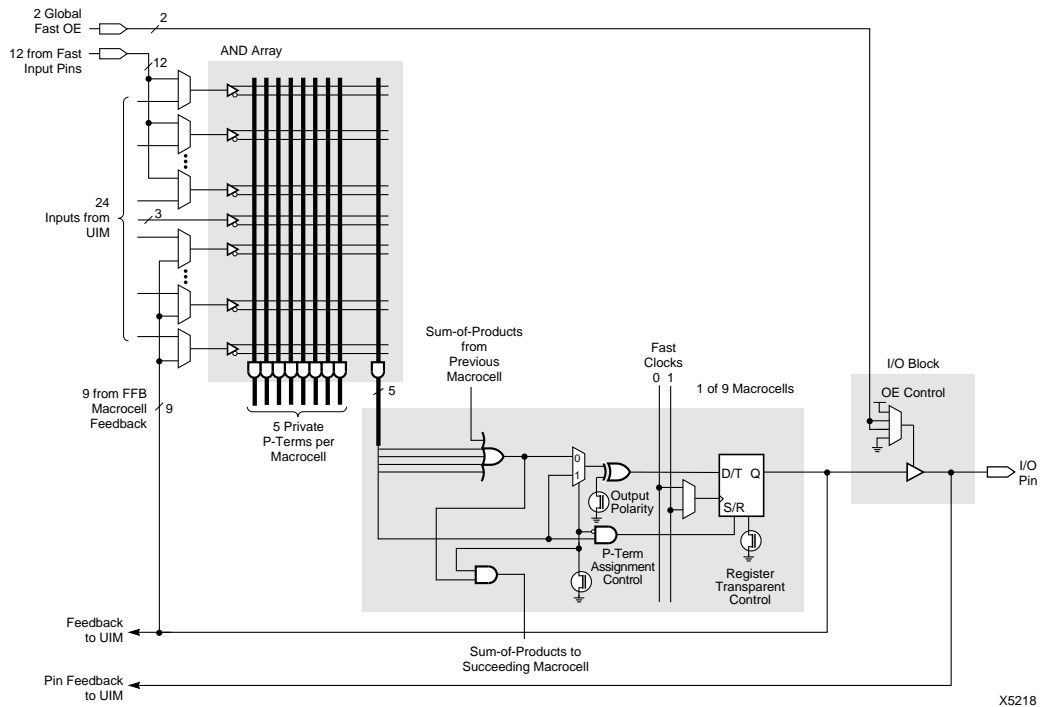


Figure A-6 Fast Function Block Schematic for 7318 and 7336

## Input/Output Blocks

I/O blocks provide 3-state outputs and registered, latched, or direct inputs. The I/O block registers can also implement logic equations and therefore decrease macrocell resource requirements.

Macrocells drive chip outputs directly through 3-state output buffers, each individually controlled by the Output Enable product term. An additional configuration option allows the output to be disabled permanently. Two dedicated Fast Output Enable inputs can also be

configured to control any of the chip outputs instead of, or in conjunction with, the individual Output Enable product term. See Figure A-7 for the I/O block schematic diagram.

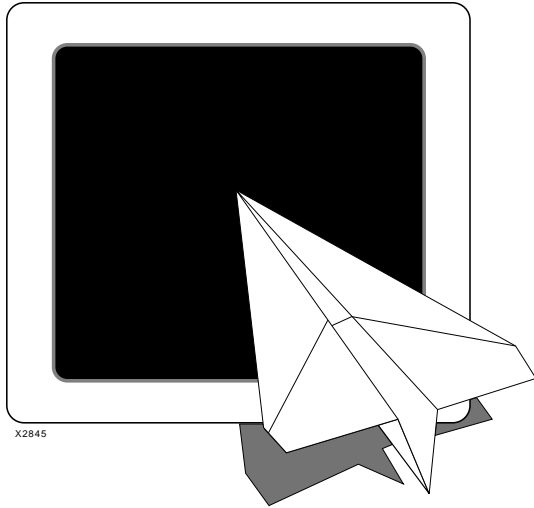
Each signal input to the chip is connected to a programmable input structure that can be configured as direct, latched, or registered. The latch and flip-flop can use the FastCLK signals as latch enable or clock. Latches are transparent when FastCLK is High, and flip-flops clock on the rising edge of FastCLK.

The flip-flop includes an active-low clock enable, which when High, holds the present state of the flip-flop and inhibits response to the input signal. The clock enable source is one of two global Clock Enable signals ( $\overline{\text{CKEN0}}$  and  $\overline{\text{CKEN1}}$ ). An additional configuration option is polarity inversion for each input signal.



The `CKEN0` and `CKEN1` inputs are only available in XC7300 family devices. Also, the programmable input polarity feature is not available in the XC7272A.





# ***Xilinx Synopsys Interface EPLD User Guide***

***Library Component  
Specifications***

## Appendix B

### Library Component Specifications

This appendix describes each of the Xilinx library components which are summarized in the following table.

Component Name	Component Description	Scalable	Inferable	Used with These Devices			
				7272	7236	7318 7336	7354 7372 73108
ACC	Adder/Subtractor/Accumulator	X			X		X
ADD	Adder	X	X	X	X		X
ADSU	Adder/Subtractor	X	X	X	X		X
ADSUR	Adder/Subtractor with Registered Outputs	X			X		X
AND2-AND8	AND Gates		X	X	X	X	X
BUF	Buffer			X	X	X	X
BUFCE	Clock Enable Inp. Buff. for Input Pad Reg.						X
BUFE	3-State Buffer		X	X	X	X	X
BUFFOE	Fast Output Enable Input Buffer				X	X	X
BUFG	FastCLK Input Buffer			X	X	X	X
CBX1	Up/Down Counter with Asynchronous Clear	X		X	X	X	X
CBX2	Up/Down Counter with Asynchronous Reset	X		X	X	X	X
COMPEQ	Equal-To Comparator	X		X	X	X	X
COMPLE_TC	Less-Than-Or-Equal Comparator, 2's Comp.	X	X	X	X		X
COMPLE_US	Less-Than-Or-Equal Comparator, Unsigned	X	X	X	X		X
COMPLT_TC	Less-Than Comparator, 2's Complement	X	X	X	X		X
COMPLT_US	Less-Than Comparator, Unsigned	X	X	X	X		X
COMPNE	Not-Equal Comparator	X		X	X	X	X
DEC	Decrementor	X	X	X	X	X	X
FDCP	Edge-Triggered D-Type Flip-Flop with Asynchronous Clear and Preset		X	X	X	X	X
FDCPE	Edge-Triggered D-Type Flip-Flop with Clock Enable, Async. Clear and Preset			X	X		X
FDPC	Edge-Triggered D-Type Flip-Flop with Asynchronous Clear and Preset		X	X	X	X	X
IBUF	Input Buffer		X	X	X	X	X
IFD	Input Pad Register			X	X		X

Component Name	Component Description	Scalable	Inferable	Used with These Devices			
				7272	7236	7318 7336	7354 7372 73108
IFDX1	Input Pad Register with Clock Enable						X
ILD	Input Pad Latch			X	X		X
INC	Incrementer	X	X	X	X	X	X
INV	Inverter		X	X	X	X	X
IOBUFE	Bi-Directional I/O Buffer		X		X	X	X
IOBUFEX1	Bi-Directional I/O Buffer				X	X	X
LD	D-Type Latch			X	X		X
OBUF	Output Buffer		X	X	X	X	X
OBUFE	3-State Output Buffer		X	X	X	X	X
OBUFEX1	3-State Output Buffer with FOE Enable				X	X	X
OR2-OR8	OR Gates		X	X	X	X	X
SUBT	Subtractor	X	X	X	X		X
XOR2-XOR8	XOR Gates		X	X	X	X	X

## ACC

ACC is an adder/subtractor/accumulator.

### Inferencing

The synthesizer does not use this component by inference.

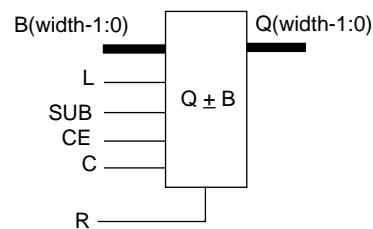
### Component Instantiation

```
U1: ACC generic map (WIDTH => wordlength)
  port map (Q=>output, B=>in_operand,
    C=>clock, CE=>clock_en, R=>sync_reset,
    L=>load_en, SUB=>add_sub_ctl);
```

### Truth Table and Logic Symbol

R	L	CE	C	SUB	Q*
1	X	X	↑	X	0
0	0	0	X	X	Q
0	0	1	↑	0	Q+B
0	0	1	↑	1	Q-B
0	1	X	↑	X	B

\* The initial state is "0".



## ADD

ADD is an adder and is bound to the “+” operator.

### Inferencing

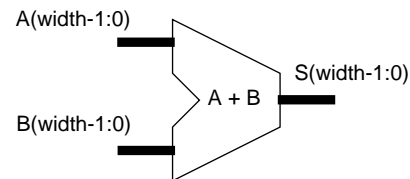
```
sum_signed <= in1_signed + in2_signed;
```

### Component Instantiation

```
U1: ADD generic map (WIDTH => wordlength)
      port map (S=>sum, A=>in1, B=>in2);
```

### Truth Table and Logic Symbol

A	B	S
A	B	A+B



## ADSU

ADSU is an adder/subtractor. ADSU is bound to the “+” and “-” operators.

### Inferencing

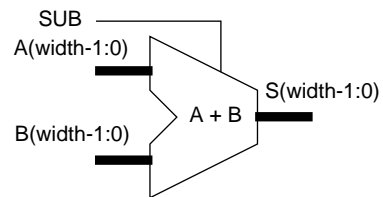
```
if (sub_ctl = '0') then
    sum_signed <= in1_signed + in2_signed;
else
    sum_signed <= in1_signed - in2_signed;
end if;
```

### Component Instantiation

```
U1: ADSU generic map (WIDTH => wordlength)
    port map (S=>output, A=>in1, B=>in2,
        SUB=>sub_ctl);
```

### Truth Table and Logic Symbol

SUB	S
0	A+B
1	A-B



## ADSUR

ADSUR is a registered adder/subtractor.

### Inferencing

The synthesizer does not use this component by inference.

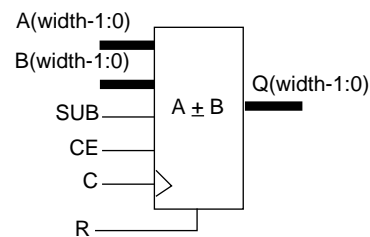
### Component Instantiation

```
U1: ADSUR generic map (WIDTH => wordlength)
  port map (Q=>output, A=>in1, B=>in2,
    C=>clock, CE=>clock_en, R=>sync_reset,
    SUB=>add_sub_ctl);
```

### Truth Table and Logic Symbol

R	CE	C	SUB	Q*
1	X	↑	X	0
0	0	x	X	Q
0	1	↑	0	A+B
0	1	↑	1	A-B

\* The initial state is "0".



## AND2 — AND8

AND2 through AND8 are AND gates with 2 to 8 inputs.

### Inferencing

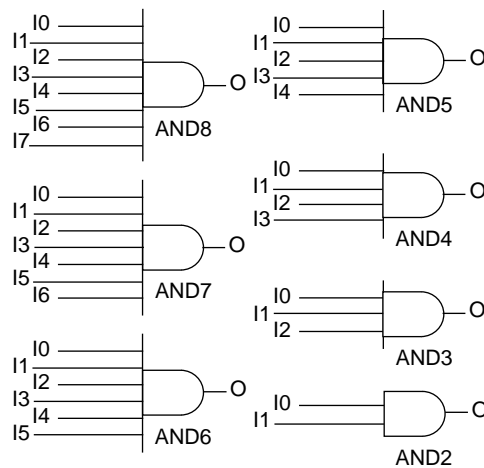
The synthesizer uses these components when creating functions that require AND gates.

### Component Instantiation

```
U1: AND2 port map (O=>out,I1=>in2,I0=>in1);
```

### Truth Table and Logic Symbol

I0	I1	O
0	0	0
0	1	0
1	0	0
1	1	1





## BUF

BUF is a buffer.

### Inferencing

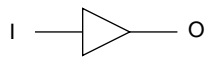
The synthesizer does not use this component by inference.

### Component Instantiation

```
U1: BUF port map (O=>out_port, I=>in_port);
```

### Truth Table and Logic Symbol

I	O
0	0
1	1



## BUFCE

BUFCE is an input buffer used to drive the global CE signal (Chip Enable) for EPLD input pad registers. BUFCE may only be used to drive the CE input of IFDX1 components.

### Inferencing

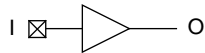
The synthesizer does not use this component by inference.

### Component Instantiation

```
U1: BUFCE port map (O=>global_ce, I=>in_port);
```

### Truth Table and Logic Symbol

I	O
0	0
1	1



## BUFE

BUFE is a non-inverting 3-state buffer.

### Inferencing

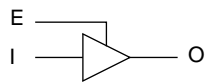
The synthesizer uses these components when creating functions that require 3-state buffers that drive internal signals.

### Component Instantiation

```
U1: BUFE port map (O=>ts_out, I=>inp, E=>enable);
```

### Truth Table and Logic Symbol

I	E	O
X	0	Z
0	1	0
1	1	1



## BUFFOE

BUFFOE is a an input buffer used to drive the global FOE signal (Fast Output Enable). BUFFOE may only be used to drive the E input of OBUFEX1 and IOBUFEX1 components

### Inferencing

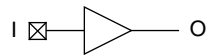
The synthesizer does not use this component by inference.

### Component Instantiation

```
U1: BUFFOE port map (O=>global_foe, I=>in_port);
```

### Truth Table and Logic Symbol

I	O
0	0
1	1



## BUFG

BUFG is an input buffer used to drive the Global FastCLK signal.

**Note:** BUFG can only drive register clock inputs (including IFDX1) and the G input of ILD components. It cannot drive the LD component.

### Inferencing

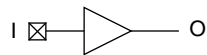
The synthesizer does not use this component by inference.

### Component Instantiation

```
U1: BUFG port map (O=>global_clk, I=>in_port);
```

### Truth Table and Logic Symbol

I	O
0	0
1	1



## CBX1

CBX1 is a loadable up/down counter with asynchronous clear.

### Inferencing

The synthesizer does not use this component by inference.

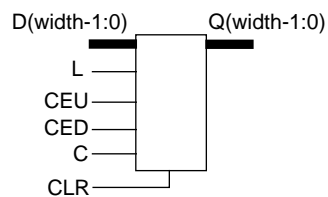
### Component Instantiation

```
U1: CBX1 generic map (WIDTH => wordlength)
  port map (Q=>output, TCU => all_ones, TCD
=>    all_zeros, D=>load_data, C=>clock,
  CLR=>async_clr, L=>load_ctl,
  CEU=>count_up_ctl, CED=>count_down_ctl);
```

### Truth Table and Logic Symbol

CLR	L	CEU	CED	C	TCU	TCD	Q*
1	X	X	X	X	0	1	0
0	1	X	X	↑	D=111...	D=000...	D
0	0	0	0	X	Q=111...	Q=000...	Q
0	0	1	0	↑	Q=111...	Q=000...	Q+1
0	0	0	1	↑	Q=111...	Q=000...	Q-1
0	0	1	1	↑	ILLEGAL CONDITION		

\* The initial state is "0".



## CBX2

CBX2 is a loadable up/down counter with synchronous reset.

### Inferencing

The synthesizer does not use this component by inference.

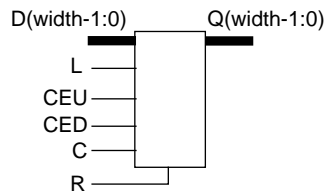
### Component Instantiation

```
U1: CBX2 generic map (WIDTH => wordlength)
  port map (Q=>output, TCU => all_ones, TCD =>
    all_zeros, D=>load_data, C=>clock,
    R=>sync_reset, L=>load_ctl,
    CEU=>count_up_ctl, CED=>count_down_ctl);
```

### Truth Table and Logic Symbol

R	L	CEU	CED	C	TCU	TCD	Q*
1	X	X	X	↑	0	1	0
0	1	X	X	↑	D=11...	D=00...	D
0	0	0	0	X	Q=11...	Q=00...	Q
0	0	1	0	↑	Q=11...	Q=00...	Q+1
0	0	0	1	↑	Q=11...	Q=00...	Q-1
0	0	1	1	↑	ILLEGAL CONDITION		

\* The initial state is "0".



## COMPEQ

COMPEQ is an equal-to comparator.

### Inferencing

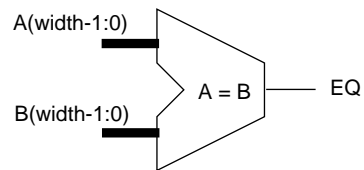
The synthesizer does not use this component by inference.

### Component Instantiation

```
U1: COMPEQ generic map (WIDTH => wordlength)
  port map (EQ=>comparison, A=>in1, B=>in2);
```

### Truth Table and Logic Symbol

Condition	EQ
$A < B$	0
$A = B$	1
$A > B$	0





## COMPLE\_TC COMPLE\_US

COMPLE\_US is an unsigned binary less-than-or-equal-to comparator.  
COMPLE\_TC is a two's complement less-than-or-equal-to comparator.  
These components are bound to the "<=" and ">=" operators.

### Inferencing

```
comparison <= (in1_signed <= in2_signed);
```

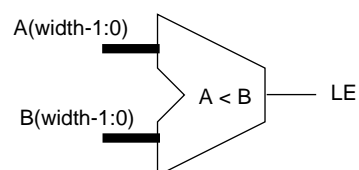
### Component Instantiation

```
U1: COMPLE_US generic map (WIDTH => wordlength)
  port map (LE=>comparison, A=>in1, B=>in2);

U1: COMPLE_TC generic map (WIDTH => wordlength)
  port map (LE=>comparison, A=>in1, B=>in2);
```

### Truth Table and Logic Symbol

Condition	LE
A<B	1
A=B	1
A>B	0



## COMPLT\_TC COMPLT\_US

COMPLT\_US is an unsigned binary less-than comparator. COMPLT\_TC is a two's complement less-than comparator. These components are bound to the "<" and ">" operators.

### Inferencing

```
comparison <= (in1_unsigned < in2_unsigned);
```

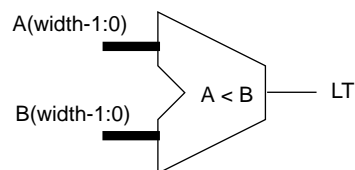
### Component Instantiation

```
U1: COMPLT_US generic map (WIDTH => wordlength)
  port map (LT=>comparison, A=>in1, B=>in2);
```

```
U1: COMPLT_TC generic map (WIDTH => wordlength)
  port map (LT=>comparison, A=>in1, B=>in2);
```

### Truth Table and Logic Symbol

Condition	LT
A<B	1
A=B	0
A>B	0



## COMPNE

COMPNE is a not-equal-to comparator.

### Inferencing

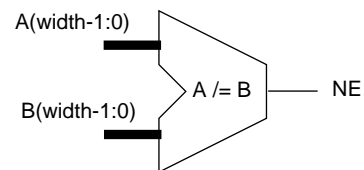
The synthesizer does not use this component by inference.

### Component Instantiation

```
U1: COMPNE generic map (WIDTH => wordlength)
  port map (NE=>comparison, A=>in1, B=>in2);
```

### Truth Table and Logic Symbol

Condition	NE
$A < B$	1
$A = B$	0
$A > B$	1



## DEC

DEC is an decrementor. It is bound to the “-1” operation.

### Inferencing

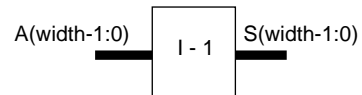
```
sum_signed <= in_signed - 1;
```

### Component Instantiation

```
U1: DEC generic map (WIDTH => wordlength)
      port map (S=>sum, A=>in);
```

### Truth Table and Logic Symbol

A	S
A	A-1



## FDCP

FDCP is an edge-triggered D-type flip-flop with preset and clear.

### Inferencing

The synthesizer uses this component for all functions that require D-type registers or latches.

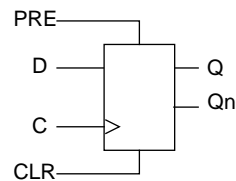
### Component Instantiation

```
U1: FDCP port map (Q=>out, QN=>out_inv, D=>data,
                  C=>clock, CLR=>async_clr, PRE=>async_set);
```

### Truth Table and Logic Symbol

CLR	PRE	C	Q*	Qn
1	X	X	0	1
0	1	X	1	0
0	0	↑	D	$\overline{D}$

\* The initial state is "0".



## FDCPE

FDCPE is an edge-triggered D-type flip-flop with preset, clear, and clock enable.

### Inferencing

The synthesizer does not use this component by inference.

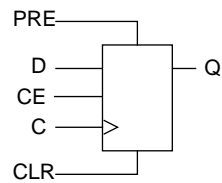
### Component Instantiation

```
U1: FDCPE port map (Q=>out, D=>in, C=>clock,
                    CE=>clock_enab, CLR=>async_clr,
                    PRE=>async_set);
```

### Truth Table and Logic Symbol

CLR	PRE	CE	C	Q*
1	X	X	X	0
0	1	X	X	1
0	0	0	X	Q
0	0	1	↑	D

\* The initial state is "0".



## FDPC

FDPC is an edge-triggered D-type flip-flop with preset and clear.

### Inferencing

The synthesizer uses this component for all functions that require D-type registers.

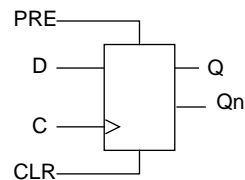
### Component Instantiation

```
U1: FDPC port map (Q=>out, QN=>out_inv, D=>data,
                  C=>clock, CLR=>async_clr, PRE=>async_set);
```

### Truth Table and Logic Symbol

CLR	PRE	C	Q*	Qn
X	1	X	1	0
1	0	X	0	1
0	0	↑	D	$\overline{D}$

\* The initial state is "0".



## IBUF

IBUF is an input buffer.

### Inferencing

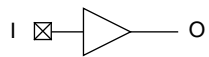
The synthesizer uses these components to receive inputs from device pins.

### Component Instantiation

```
U1: IBUF port map (O=>received_signal,  
I=>in_port);
```

### Truth Table and Logic Symbol

I	O
0	0
1	1





## IFD

IFD is an edge-triggered D-type flip-flop. The C input must be driven by a BUFG component. IFD is only available for use in EPLD Input Blocks.

### Inferencing

The synthesizer does not use this component by inference.

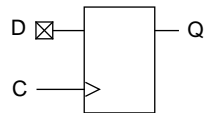
### Component Instantiation

```
U1: IFD port map (Q=>output, D=>in_port,
                  C=>global_clock);
```

### Truth Table and Logic Symbol

C	Q*
X	Q
↑	D

\* The initial state is "0".



## IFDX1

IFDX1 is an edge-triggered D-type flip-flop with active-low clock enable. The C input must be driven by a BUFG component. The CE input, if used, must be driven by a BUFCE component. IFDX1 is only available for use in EPLD Input Blocks.

### Inferencing

The synthesizer does not use this component by inference.

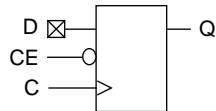
### Component Instantiation

```
U1: IFDX1 port map (Q=>output, D=>in_port,
                   C=>global_clock, CE=>global_ce);
```

### Truth Table and Logic Symbol

CE	C	Q*
1	X	Q
0	↑	D

\* The initial state is "0".



## ILD

ILD is a D-type flip-flop available in the EPLD Input Block. The G input must be driven by a BUFG buffer.

### Inferencing

The synthesizer does not use this component by inference.

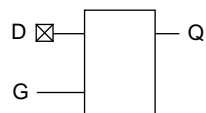
### Component Instantiation

```
U1: ILD port map (Q=>output, D=>in_port,
                  G=>global_clock);
```

### Truth Table and Logic Symbol

G	Q*
0	Q
1	D

\* The initial state is "0".



## INC

INC is an Incrementer. It is bound to the “+1” operator.

### Inferencing

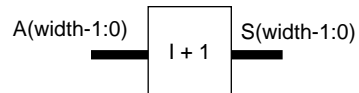
```
sum_signed <= in_signed + 1;
```

### Component Instantiation

```
U1: INC generic map (WIDTH => wordlength)  
  port map (S=>sum, A=>in);
```

### Truth Table and Logic Symbol

A	S
A	A+1



## INV

INV is an inverter.

### Inferencing

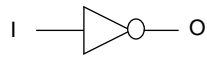
The synthesizer uses this component for signal inversion.

### Component Instantiation

```
U1: INV port map (O=>not_in1, I=>in1);
```

### Truth Table and Logic Symbol

I	O
0	1
1	0



## IOBUFE

IOBUFE is a non-inverting 3-state I/O buffer.

### Inferencing

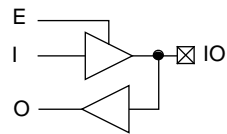
The synthesizer uses these components to create bidirectional I/O.

### Component Instantiation

```
U1: IOBUFE port map (O=>received_signal,
                     IO=>inout_port, I=>driving_signal,
                     E=>output_enable);
```

### Truth Table and Logic Symbol

I	E	IO
X	0	Z
0	1	0
1	1	1



## IOBUFEX1

IOBUFEX1 is a bidirectional I/O buffer that uses the EPLD FOE enable signal. The E input must be driven by a BUFFOE buffer.

### Inferencing

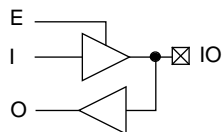
The synthesizer does not use this component by inference.

### Component Instantiation

```
U1: IOBUFEX1 port map (O=>received_signal,
                       IO=>inout_port, I=>driving_signal,
                       E=>global_enable);
```

### Truth Table and Logic Symbol

I	E	IO	O
X	0	Z	Z
0	1	0	0
1	1	1	1



## LD

LD is a D-type latch. The G input of LD cannot be driven by a BUFG buffer.

### Inferencing

The synthesizer does not use this component by inference. Instead, it will infer FDCP cells to implement transparent latches

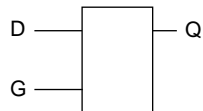
### Component Instantiation

```
U1: LD port map (Q=>out, D=>data,  
                 G=>latch_enable);
```

### Truth Table and Logic Symbol

G	Q*
0	Q
1	D

\* The initial state is "0".





## OBUF

OBUF is an output buffer.

### Inferencing

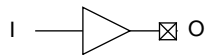
The synthesizer uses this component when creating external outputs to device pins.

### Component Instantiation

```
U1: OBUF port map (O=>out_port,
                  I=>driving_signal);
```

### Truth Table and Logic Symbol

I	O
0	0
1	1
Z	Z



## OBUFE

OBUFE is a 3-state output buffer. (macro of BUFE and OBUF)

### Inferencing

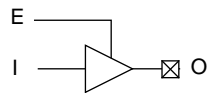
The synthesizer uses this component when creating 3-state external outputs which connect to device pins.

### Component Instantiation

```
U1: OBUF port map (O=>out_port,  
                  I=>driving_signal, E=enable);
```

### Truth Table and Logic Symbol

I	E	O
X	0	Z
0	1	0
1	1	1



## OBUFEX1

OBUFEX1 is a 3-state output buffer that uses the EPLD FOE enable signal. The **E** input must be driven by a BUFFOE buffer.

### Inferencing

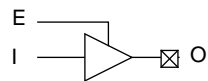
The synthesizer does not use this component by inference.

### Component Instantiation

```
U1: OBUFEX1 port map (O=>out_port,
                      I=>driving_signal, E=>global_foe);
```

### Truth Table and Logic Symbol

I	E	O
X	0	Z
0	1	0
1	1	1



## OR2 — OR8

OR2 through OR8 are OR gates with 2 to 8 inputs.

### Inferencing

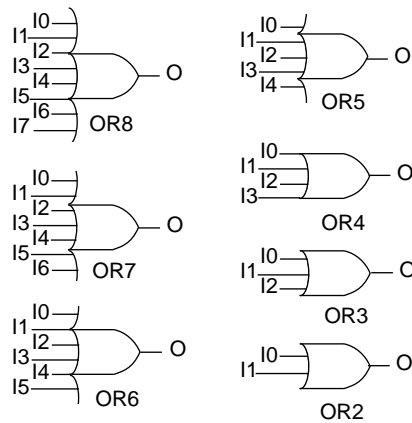
The synthesizer uses these components when creating functions that require OR gates.

### Component Instantiation

```
U1: OR2 port map (O=>out, I1=>in2, I0=>in1);
```

### Truth Table and Logic Symbol

I0	I1	O
0	0	0
0	1	1
1	0	1
1	1	1



## SUBT

SUBT is a subtracter and is bound to the “-” operator.

### Inferencing

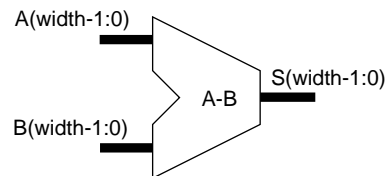
```
diff_signed <= in1_signed - in2_signed;
```

### Component Instantiation

```
U1: SUBT generic map (WIDTH => wordlength)
  port map (S=>diff, A=>in1, B=>in2);
```

### Truth Table and Logic Symbol

A	B	S
A	B	A-B



## XOR2 — XOR8

XOR2 through XOR8 are XOR gates with 2 to 8 inputs.

### Inferencing

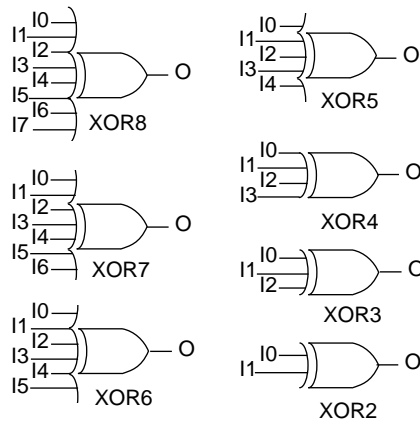
The synthesizer uses these components when creating functions that require XOR gates.

### Component Instantiation

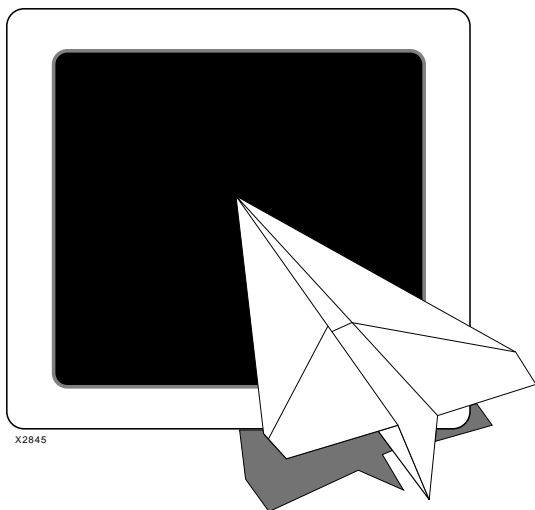
```
U1: XOR2 port map (O=>out, I1=>in2, I0=>in1);
```

### Truth Table and Logic Symbol

I0	I1	O
0	0	0
0	1	1
1	0	1
1	1	0







# ***Xilinx Synopsys Interface EPLD User Guide***

***Attributes***



## Appendix C

### Attributes

---

Attributes are used to control how the software uses the architecture specific features of the XC7000 EPLDs. See the device data sheets for more information about these device features.

#### Global Attributes

Global attributes are applied by instantiating the following components. These components have no ports.

##### LOWPWR

This attribute controls the macrocell power usage. If the `LOWPWR` attribute is specified it indicates that all macrocells have low power operation. If `LOWPWR` is not specified, the macrocells have standard power operation.

To specify low power operation for all macrocells, instantiate the `LOWPWR` component as follows:

```
U1: LOWPWR;
```

where U1 is any instance name.

##### MRINPUT

This attribute controls the use of the Master Reset pin on the XC7354 and XC7336 devices. If the `MRINPUT` attribute is specified it indicates that the pin is used as a logic input. If `MRINPUT` is not specified, the pin is used as the Master Reset input.

To specify that the Master Reset pin is used as a logic input, instantiate the `MRINPUT` component as follows:

```
U1: MRINPUT;
```

where U1 is any instance name.

## **NO\_FOE**

This attribute controls the automatic use of global Fast Output Enable inputs. If the `NO_FOE` attribute is specified, it indicates that the software will *not* automatically assign 3-state control inputs to the global FOE inputs. If `NO_FOE` is not specified, the software will assign the 3-state control signals in your design to the global FOE inputs of the device, if possible.

To specify that no 3-state control signals will automatically be assigned to the global FOE inputs, instantiate the `NO_FOE` component as follows:

```
U1: NO_FOE;
```

where U1 is any instance name.

## **NO\_FCLK**

This attribute controls the automatic use of global FastClock inputs. If the `NO_FCLK` attribute is specified, it indicates that the software will not automatically assign clock signals to the global FastClock inputs. If `NO_FCLK` is not specified, the software will assign clock signals in your design to the dedicated FastCLK inputs of the device, if possible.

To specify that no clock signals will automatically be assigned to the global FastClock nets, instantiate the `NO_FCLK` component as follows:

```
U1: NO_FCLK;
```

where U1 is any instance name.

## **NO\_IFD**

This attribute controls the automatic usage of input pad registers. If the `NO_IFD` attribute is specified, it indicates that the software will not automatically use the registers in the input pads. If `NO_IFD` is not specified, the software will assign registers in your design to the

input pads whenever possible, to reduce macrocell resource requirements.

To specify that registers will not be automatically placed into the input pads, instantiate the `NO_IFD` component as follows:

```
U1: NO_IFD;
```

where U1 is any instance name.

## PRELOAD

This attribute controls the use of default initial state values for registers in your design. If `PRELOAD` is specified, the software will use the default initial states for each register as shown in the library specifications. If the `PRELOAD` attribute is not specified, it indicates that the software may change the initial states of registers (unless explicitly specified) if the change allows a more efficient implementation.

To prevent the software from changing the initial state of registers in your design, instantiate the `PRELOAD` component as follows:

```
U1: PRELOAD;
```

where U1 is any instance name.

## Signal Attributes

Signal attributes are applied by instantiating the following components. Each of these components has one port (input) which you connect to the signal that receives the attribute.

### F

This attribute indicates either a Fast Function Block output signal or a Fast Input signal. Use this attribute to assign specific functions to EPLD Fast Function Blocks, which provide the highest performance.

To specify that a signal is driven from a Fast Function Block, use:

```
U1: F port map (signal_name);
```

## H

This attribute indicates a High Density Function Block output signal. Use this attribute to assign specific functions to High Density Function Blocks, which provide the most macrocell resources.

To specify that a signal is driven from a High Density Function Block, use:

```
U1: H port map (signal_name);
```

## OPT\_OFF

This attribute inhibits the software from optimizing the cell that drives the connected signal.

To specify that a signal is to remain as a macrocell output, use:

```
U1: OPT_OFF port map (signal_name);
```

## OPT\_UIM

This attribute forces the software to place the specified AND gate into the UIM. It can only be connected to a signal that originates from an AND gate.

To specify that an AND gate is to be implemented in the UIM, instantiate the OPT\_UIM component as follows:

```
U1: UIM_OPT port map (signal_name);
```

**Note:** The optimization of AND gates into the UIM is done automatically by the fitter whenever possible.

## SLEWRATE

This attribute controls the output buffer slew rate. The options are:

- **HIGH** — Slows the output signal transition time and thus reduces internal ground-bounce noise.
- **NONE** — Speeds up the output signal transition time. (The default is for fast transition times.)

The output buffers (IBUF, OBUF, and IOBUFE) default to fast transition outputs. However, in order to reduce possible ground-bounce problems, it is recommended that you use the fast transition

default only for those output signals that require maximum speed.

To set all outputs for slow slew rate, use the following command:

```
set_pad_type -slewrates HIGH all_outputs ( )
```

Use this command after specifying the `set_port_is_pad` command and before implementing the `insert_pads` command.

After you have globally changed all outputs to the HIGH option (for slow signal transition) you can set any individual output for fast signal transition by using the following command:

```
set_pad_type -slewrates NONE port_name
```

If you need fast transition time on most of your outputs, you can leave the default slew rate as NONE and slow only the selected outputs, by using the following command:

```
set_pad_type -slewrates HIGH port_name
```

**Note:** Synopsys and Xilinx define the slew rate using opposite terms. The Synopsys HIGH attribute translates to a SLOW Xilinx slew rate. The Synopsys NONE attribute translates to a FAST Xilinx slew rate.

## Synopsys Attributes

The following Synopsys attributes are supported for Xilinx EPLD designs when using FPGA Compiler. See the Synopsys Design Compiler manual for more information on using the `set_attribute` command.

### Part Type

This attribute is used to specify the target EPLD device type for FPGA Compiler only. (If you are using Design Compiler, you must specify the part type by using the `-p` option of the `syn2epld` command.)

The format is:

```
set_attribute design_name part -type string  
          dddd-ssppnn
```

where:

- *design\_name* = the name of the top-level design entity.

- *dddd* = basic device type.
- *ss* = speed grade.
- *pp* = package type.
- *nn* = number of package pins.

The currently supported device types are shown in the “EPLD Architecture” appendix. For example:

```
set_attribute SCAN part -type string 7354-10PC44
```

## Pin Assignment

This attribute is used to specify the pins on which to place output signals.

The format is:

```
set_attribute port_name pad_location -type  
string pin_number
```

where:

- *port\_name* = The name of the top-level design port.

The format of *pin\_number* is:

- *Pnn* for PC and PQ packages, where *nn* is the pin number.
- *rc* for PG and BG packages, where *rc* are the row letter and column number.

For example, for PC and PQ packages:

```
set_attribute RDY pad_location -type string P23
```

For example, for PG and BG packages:

```
set_attribute RDY pad_location -type string K13
```

**Note:** The pin assignment attribute overrides previously saved pinouts when running `fitnet` with the `-f` (pin freeze) option.

## Register Initial State

This attribute is used to specify the initial (power up) state of registers in your design.

The format is:

```
set_attribute instance_name  
fpga_xilinx_init_state -type string state
```

where:

- *instance\_name* is the name of a register instance
- *state* is either S (set) or R (reset)

For example:

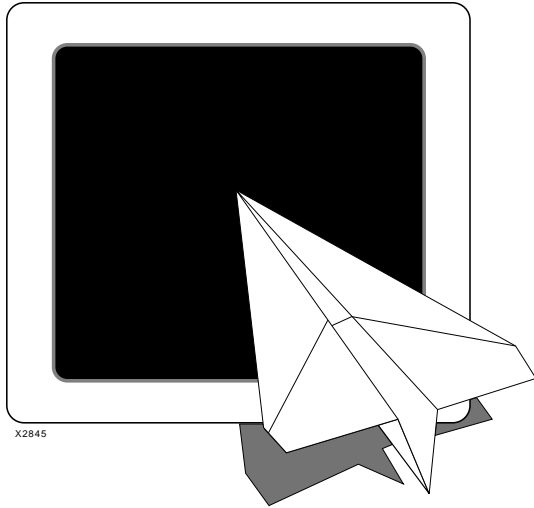
```
set_attribute "QOUT_reg<2>"  
fpga_xilinx_init_state -type string S
```

You can also use the Synopsys `find` function to specify a set of instance names using wildcards, for example:

```
set_attribute find (cell QOUT*)  
fpga_xilinx_init_state -type string S
```







***XEPLD Files***

# ***Xilinx Synopsys Interface EPLD User Guide***

## Appendix D

### XEPLD Files

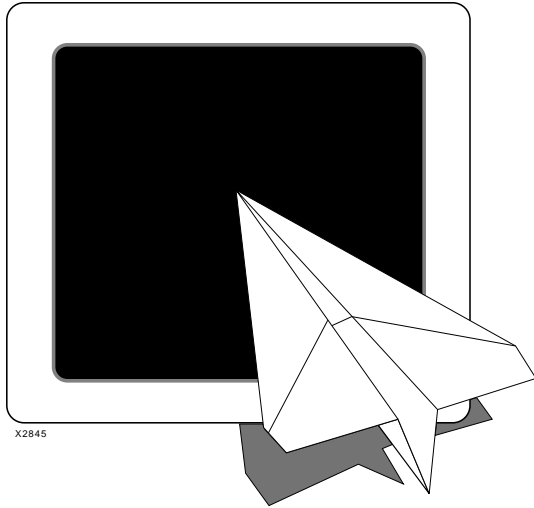
---

This appendix describes the principal files that may be found in the working directory.

- *configuration\_name.sim* — Simulation model file produced by the `vhdlan` command and read by the VSS simulator.
- *design\_name.eqn* — Equation Report showing the detailed final logic implementation (after optimization) in the EPLD, using Xilinx PLUSASM language format.
- *design\_name.err* — Error log showing a list of any errors and warnings that occurred during the fitting process.
- *design\_name.lgc* — Logic Optimizer log showing how the fitter optimized your logic for the EPLD.
- *design\_name.log* — Fitter log containing the details of steps performed during fitting.
- *design\_name.map* — Mapping Report summarizing the contents of all function blocks and macrocells in the target EPLD.
- *design\_name.par* — Partitioner Report showing the details of how EPLD function block resources were allocated.
- *design\_name.pin* — Pin-List Report showing the final pinout of your design.
- *design\_name.prg* — Hex formatted programming bit-map file created by the `makeprg` command. This file is downloaded to a device programmer.
- *design\_name.res* — Resource Report showing the amount of EPLD macrocell and I/O pin resources used and those remaining.
- *design\_name.sedif* — EDIF-formatted logic netlist created by the

Synopsys **write** command when Design Compiler is used. This file is used by the **syn2ep1d** program.

- *design\_name.sxnf* — XNF-formatted logic netlist used by the **syn2ep1d** program. This file is created by the synopsys **write** command when FPGA compiler is used.
- *design\_name.tim* — Static Timing Report showing the calculated worst-case timing for the logic paths in your design.
- *design\_name.v* — Verilog HDL language source design file.
- *design\_name.vhd* — VHDL language source design file.
- *design\_name.vmd* — XEPLD design database file created by the fitter (for XC7272 only). This file contains a complete description of your fitted design.
- *design\_name.vmf* — Pin-save file created by the **pinsave** command. This file, if it exists, is used by the fitter to lock a previous device pinout. See Chapter 4 for more information.
- *design\_name.vmh* — XEPLD design database file created by the fitter (for all devices except the XC7272). This file contains a complete description of your fitted design.
- *design\_name\_vss.vhd* — VHDL timing simulation model created by the **vmh2vss** program. This file is used by the VSS simulator.
- *design\_name\_vss.sdf* — Timing back-annotation file created by the **vmh2vss** program. This file is used by the VSS simulator in conjunction with the *design\_name\_vss.vhd* file.
- *design\_name.xff* — Merged and flattened netlist created by the **syn2ep1d** program. This file is primarily based on the *.sxnf* (or *.sedif*) file and is the primary input to the fitter.
- *design\_name.xnf* — The simulation netlist file created by the **vmh2vss** command. This is an intermediate file used by **vmh2vss** to create the *design\_name\_vss.vhd* timing simulation model.
- **.synopsys\_dc.setup** — Configuration file used by the Synopsys compiler to define the libraries and parameters for synthesis.
- **.synopsys\_vss.setup** — Configuration file used by the Synopsys VSS simulator (and the **vhd1an** command) to define the model library used for simulation.



***Design Example***

# ***Xilinx Synopsys Interface EPLD User Guide***

## Appendix E

### Design Example

---

This appendix shows the VHDL source file for a PCI bus interface.

#### PCI Bus Interface Design Description

The PCI bus is a 32 or 64 bit interface with multiplexed address and data lines. It is intended for use as an interconnect mechanism between highly integrated peripheral controllers, add-in boards, memory systems, and central processors. This bus is becoming a fundamental building block for high-performance personal computers and workstations.

The following VHDL source code implements a PCI bus interface using the Xilinx XC73108 EPLD.

**Note:** A complete application note describing this design is available.

```
-- PCI Target Interface Design for XC73108 --
-- Synopsys VHDL Solution using Xilinx XC7000 Library --

library IEEE, xc7000;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_misc.all;
use IEEE.STD_LOGIC_arith.all;
use IEEE.STD_LOGIC_components.all;
use xc7000.components.all;

entity pci_108 is
  port (

    -- PCI bus signals --
    CLK : in bit;
    FRAME, IRDY : in boolean;
    TRDY, STOP : inout std_logic;
    DEVSEL : buffer std_logic;
    AD : inout std_logic_vector (31 downto 0);
    CBE : inout std_logic_vector (3 downto 0);

    -- Target interface signals --
    TERM, READY, T_ABORT, TAR_DLY : in boolean;
    ADT : inout std_logic_vector (31 downto 0);
    CBET : inout std_logic_vector (3 downto 0);
    S0, S1 : buffer std_logic;    -- registers --
    RD_WR, BRD_EN : buffer boolean;  -- registers --
    PAR_OE, PERR_OE : out boolean  -- registers --
  );
end pci_108;
```

```

architecture BEHAVIOR of pci_108 is

-- Internal signals --
-- PCI AD bus output register --
    signal PCI_AD : std_logic_vector (31 downto 0);
-- PCI AD bus input register --
    signal ADI : std_logic_vector (31 downto 0);
-- Target AD bus output register --
    signal TARGET_AD : std_logic_vector (31 downto 0);
-- PCI CBE bus output register --
    signal PCI_CBE : std_logic_vector (3 downto 0);
-- PCI CBE bus input register --
    signal CBEI : std_logic_vector (3 downto 0);
-- Target CBE bus output register --
    signal TARGET_CBE : std_logic_vector (3 downto 0);
    signal BA : std_logic_vector (31 downto 24);    -- register --
    signal AVALID, BA_WR, HIT : boolean;
    signal TRDY_OE, AD_OE, ADT_OE : boolean;    -- registers --
    signal TRDY_VAL, STOP_VAL : boolean;
    signal S1_S0 : std_logic_vector (1 downto 0);
    signal COMMAND_TYPE : std_logic_vector (3 downto 1);
    signal IO_SPACE, MEMORY_SPACE, CONFIG_SPACE, MEMORY_MULT,
        MEMORY_CACHE : boolean;
    type TARGET_TYPE is (IDLE_STATE, B_BUSY_STATE, S_DATA_STATE,
        TURN_AR_STATE, BACKOFF_STATE);
    signal TARGET_SEQ : TARGET_TYPE;    -- state register
    signal IDLE, B_BUSY, S_DATA, TURN_AR, BACKOFF : boolean; -- state
        decoders
-- PCI Bus Command Encoding --
    constant IO_SPACE_CMD : std_logic_vector (3 downto 1) := "001";
    constant MEMORY_SPACE_CMD : std_logic_vector (3 downto 1) := "011";
    constant CONFIG_SPACE_CMD : std_logic_vector (3 downto 1) := "101";
    constant MEMORY_MULT_CMD : std_logic_vector (3 downto 1) := "110";
    constant MEMORY_CACHE_CMD : std_logic_vector (3 downto 1) := "111";
    constant BA_AD_SEL : std_logic_vector (7 downto 2) := "000100";
    constant CFG_SPACE_00H : std_logic_vector (31 downto 0) :=
        "00000000000000000000000000000000";
    constant CFG_SPACE_04H : std_logic_vector (31 downto 0) :=
        "00000000000000000000000000000000";

```

```
-- Re-declare attribute cells with boolean ports --

component f
  port (I : in boolean);
end component;

component opt_off
  port (I : in boolean);
end component;

begin

-- XEPLD fitter attributes --
XU1: f port map (IRDY);
XU2: f port map (FRAME);
XU3: opt_off port map (HIT);
XU4: opt_off port map (BA_WR);

-- PCI bus to Target interface data path --
-- Target interface passes address and data transfers synchronously
-- through interface.

S1_S0 <= (S1, S0);

DATA_PATH: process begin
  wait until (CLK'event and CLK = '1');

  if (S1_S0 = "00") then
    PCI_AD <= ADT;
  elsif (S1_S0 = "01") then
    PCI_AD <= CFG_SPACE_00H;
  elsif (S1_S0 = "10") then
    PCI_AD <= CFG_SPACE_04H;
  else
    PCI_AD <= BA (31 downto 24) & "000000000000000000000000";
  end if;
end process;
```





```
-- Target Interface State Machine --

TARGET: process begin
  wait until (CLK'event and CLK = '1');
  case TARGET_SEQ is

-- IDLE -- Idle condition.
    when IDLE_STATE =>
      if (not FRAME) then
        if (not HIT) then
          TARGET_SEQ <= B_BUSY_STATE;
        elsif (not TERM or READY) then
          TARGET_SEQ <= S_DATA_STATE;
        else
          TARGET_SEQ <= BACKOFF_STATE;
        end if;
      end if;

-- B_BUSY -- Agent not involved in current transaction.
    when B_BUSY_STATE =>
      if (FRAME) then
        TARGET_SEQ <= IDLE_STATE;
      elsif (IRDY or HIT) then
        if (not IRDY and HIT and (not TERM or READY)) then
          TARGET_SEQ <= S_DATA_STATE;
        else
          TARGET_SEQ <= BACKOFF_STATE;
        end if;
      end if;

-- S_DATA -- Agent has accepted request and will respond.
    when S_DATA_STATE =>
      if (FRAME and (not TRDY_VAL or not STOP_VAL)) then
        TARGET_SEQ <= TURN_AR_STATE;
      elsif (not STOP_VAL and (TRDY_VAL or not IRDY)) then
        TARGET_SEQ <= BACKOFF_STATE;
      end if;
```

---

```

-- TURN_AR -- Completed transaction on bus.
  when TURN_AR_STATE =>
    if (FRAME) then
      TARGET_SEQ <= IDLE_STATE;
    elsif (not HIT) then
      TARGET_SEQ <= B_BUSY_STATE;
    elsif (not TERM or READY) then
      TARGET_SEQ <= S_DATA_STATE;
    else
      TARGET_SEQ <= BACKOFF_STATE;
    end if;
-- BACKOFF -- Agent busy, unable to respond at this time.
  when BACKOFF_STATE =>
    if (FRAME) then
      TARGET_SEQ <= TURN_AR_STATE;
    end if;
  end case;
end process;
IDLE <= (TARGET_SEQ = IDLE_STATE);
B_BUSY <= (TARGET_SEQ = B_BUSY_STATE);
S_DATA <= (TARGET_SEQ = S_DATA_STATE);
TURN_AR <= (TARGET_SEQ = TURN_AR_STATE);
BACKOFF <= (TARGET_SEQ = BACKOFF_STATE);

-- DEVSEL is asserted from address hit until either TURN_AR or T_ABORT is
-- asserted by the Target agent.

DEVSEL <= 'Z' when (not TRDY_OE) else
          '0' when ((BACKOFF or S_DATA) and not T_ABORT) else
          '1';

-- Target agent ready to complete current data transaction.
-- TRDY is asserted while the Target transfers data (S_DATA)
-- and remains asserted while READY asserted and T_ABORT de-asserted.

TRDY <= 'Z' when (not TRDY_OE) else
       '1' when (not (READY and not T_ABORT and S_DATA and TAR_DLY)) else
       '0';

TRDY_VAL <= (TRDY = '1');

```

```
-- Target requests Master to stop the current transaction.
-- STOP is asserted in response to T_ABORT
-- and remains asserted until entering TURN_AR.

STOP <= 'Z' when (not TRDY_OE) else
        '1' when (not (BACKOFF or (S_DATA and (T_ABORT or TERM)
and
        TAR_DLY))) else
        '0';
STOP_VAL <= (STOP = '1');

-- PCI Bus Command Decoder

COMMAND_TYPE <= CBEI (3 downto 1);
IO_SPACE <= (COMMAND_TYPE = IO_SPACE_CMD);
MEMORY_SPACE <= (COMMAND_TYPE = MEMORY_SPACE_CMD);
CONFIG_SPACE <= (COMMAND_TYPE = CONFIG_SPACE_CMD);
MEMORY_MULT <= (COMMAND_TYPE = MEMORY_MULT_CMD);
MEMORY_CACHE <= (COMMAND_TYPE = MEMORY_CACHE_CMD);

AVALID <= (IO_SPACE or MEMORY_SPACE or CONFIG_SPACE or
MEMORY_MULT or
        MEMORY_CACHE) and BRD_EN;

OE: process begin
    wait until (CLK'event and CLK = '1');

    TRDY_OE <= BACKOFF or S_DATA or TURN_AR;
    AD_OE <= S_DATA and TAR_DLY and RD_WR;

    ADT_OE <= (IDLE and FRAME and RD_WR and (AD(1 downto 0) =
"00")) or
        (HIT and not RD_WR) or (S_DATA and not RD_WR and not
        T_ABORT);
```

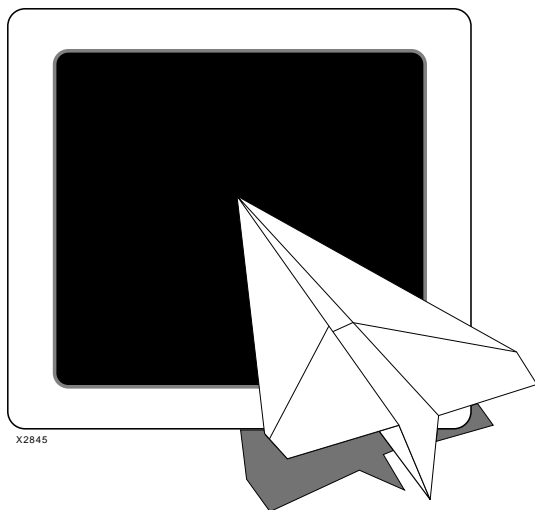
```
    if (IDLE) then
        RD_WR <= (CBE(0) = '0');
    end if;

    if (not S_DATA) then
        S0 <= '0'; S1 <= '0';
        if (IDLE and not FRAME) then
            if (MEMORY_SPACE or MEMORY_MULT or MEMORY_CACHE)
then
                S0 <= '1';
            end if;
            if (IO_SPACE) then
                S1 <= '1';
            end if;
        end if;
    end if;

    PAR_OE <= S_DATA and not TRDY_VAL and RD_WR;
    PERR_OE <= not IRDY and not RD_WR;
end process;
end BEHAVIOR;
```

**Figure E-1 PCI Controller Source File**





***Fitter Reports***

# ***Xilinx Synopsys Interface EPLD User Guide***

## Appendix F

### Fitter Reports

---

The primary fitter reports that you will use are:

- *design\_name.res* — Resource Report showing the amount of EPLD macrocell and pin resources remaining.
- *design\_name.tim* — Static Timing Report showing the calculated worst-case timing for the logic paths in your design.
- *design\_name.pin* — Pin-List Report showing the final pinout of your design.

Examples of these three reports are provided in the following sections.



## Resource Report

Use this report to determine the amount of EPLD resources used by your design, and the amount of remaining resources.

```

XEPLD, Version 5.0                                Xilinx Inc.
Resource Report
Circuit name: SCAN
Target Device: XC7354-10PC44                      Integrated: 6-20-94, 12:37P
  
```

### LOGIC RESOURCES

	Required	Used	Remaining
Function Blocks	6	6	0
Macrocells	26	26	28

### PIN RESOURCES:

Type	Req	-----Used-----							-----Remaining-----						
		I	O	I/O	Fclk	Foe	Cen	Tot	I	O	I/O	Fclk	Foe	Cen	Tot
Inputs	12	8		4				12	0		15				15
Outputs	9		0	6	1	1	1	9	0	15		0	0	0	15
I/Os	0			0				0		15					15
Fclks	1				1			1				0			0
Foes	0					0		0					0		0
Cens	0						0	0						0	0
		22	8	0	10	2	1	1	22						

Note: The design requires 0 pins with Fast Input capability.  
 This device has 11 pins with Fast Input capability.  
 The design requires 0 pins with Fast Output capability.  
 This device has 0 FO and 1 I/FO remaining from original 0 FO and 16 I

End of Resource Report

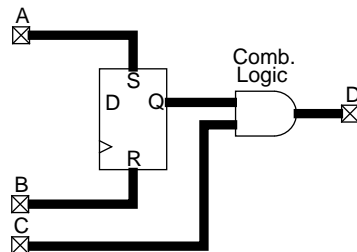
**Figure F-1 The Resource Report — SCAN Design**

## The Static Timing Report

Use this report to verify your design timing. The types of reported timing parameters are described in the following sections.

### Combinational Pad-to-Pad Delays

A combinational pad-to-pad delay is calculated from an input pad through any combinational logic to an output pad. Combinational paths include any asynchronous Set and Reset inputs to registered outputs as shown below.



...

Summary of Combinational Pad-to-Pad Delays (In Best to Worst Order)

From	To	Delay(nsec)
C	D	7.5
A	D	15.0
B	D	15.0

...

**Figure F-2 Combinational Pad-to-Pad Delay Report Example**

## Setup-to-Clock Time

The setup time is calculated using the fastest clock path and the slowest data path arriving at any given register. The timing analyzer checks all registers and reports the worst-case setup time for each pair of clock and data signals.

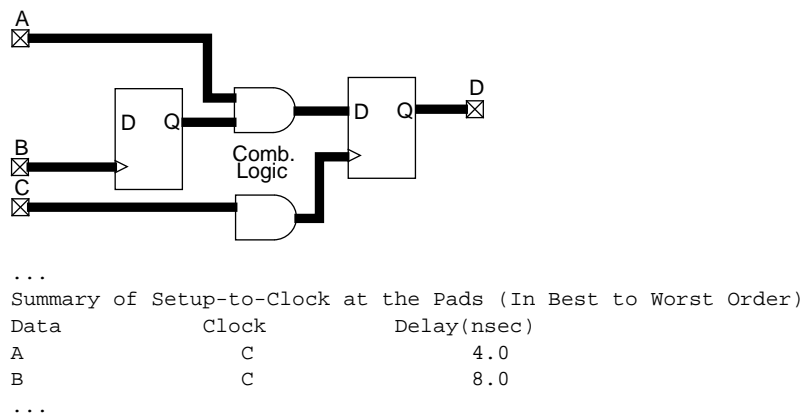


Figure F-3 Setup-to-Clock Time Report Example

## Clock-to-Output Delays

Clock-to-output delays are calculated from the input pad of a clock signal to the output pad.

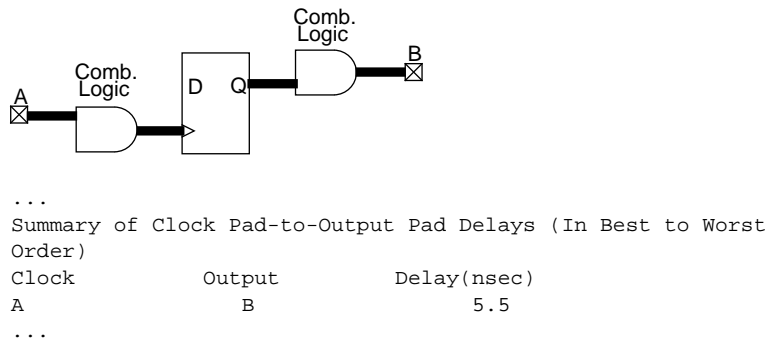
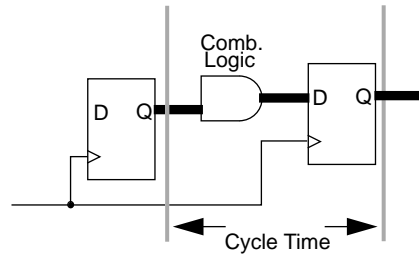


Figure F-4 Clock-to-Output Time Report Example

## Cycle Time

The cycle time is calculated between two registers that share the same clock. The timing report does not show cycle times for circuits that do not have a register to register path.



```
...
Summary of Cycle Time Delays (In Best to Worst Order)
(See .map file for signal names)
From      To      Delay(nsec)
FB3_9     FB3_4     8.0
FB3_6     FB3_7     8.0
...
```

**Figure F-5 Cycle Time Report Example**

**Note:** You should also consider the setup and clock-to-output delays when determining the maximum device speed in your system.

## Example Timing Report

XEPLD, Version 5.0E

Xilinx Inc.

### Timing Report

Circuit name: scan

Target Device: XC7354-10PC44

Report Date: 6-20-94, 13:41:24

Slowest Combinational Pad-to-Pad	28.5 nsec	(Worst Case)
Slowest Setup-to-Clock at the pads	13.0 nsec	(Worst Case)
Slowest Clock-to-Output(Pad-to-Pad)	20.0 nsec	(Worst Case)
Maximum Clock Frequency CLOCK	76 Mhz	(Worst Case)

#### Summary of Combinational Pad-to-Pad Delays (In Best to Worst Order)

From	To	Delay(nsec)
CLEAR	C_OUT<7>	28.5
CLEAR	C_OUT<6>	28.5
CLEAR	C_OUT<5>	28.5
CLEAR	C_OUT<4>	28.5
CLEAR	C_OUT<3>	28.5
CLEAR	C_OUT<2>	28.5
CLEAR	C_OUT<1>	28.5
CLEAR	C_OUT<0>	28.5

#### Summary of Setup-to-Clock at the Pads (In Best to Worst Order)

Data	Clock	Delay(nsec)
WRITE_START	CLOCK	11.0
WRITE_END	CLOCK	11.0
DATA_IN<7>	CLOCK	11.0
DATA_IN<6>	CLOCK	11.0
DATA_IN<5>	CLOCK	11.0
...	...	...
DATA_IN<3>	CLOCK	11.0
DATA_IN<2>	CLOCK	11.0
DATA_IN<1>	CLOCK	11.0
DATA_IN<0>	CLOCK	11.0
START	CLOCK	13.0

## Summary of Clock Pad-to-Output Pad Delays (In Best to Worst Order)

Clock	Output	Delay(nsec)
CLOCK	DONE	10.0
CLOCK	C_OUT<7>	20.0
CLOCK	C_OUT<6>	20.0
CLOCK	C_OUT<5>	20.0
CLOCK	C_OUT<4>	20.0
CLOCK	C_OUT<3>	20.0
CLOCK	C_OUT<2>	20.0
CLOCK	C_OUT<1>	20.0
CLOCK	C_OUT<0>	20.0

## Summary of Cycle Time Delays (In Best to Worst Order)

(See .map file for signal names)

From	To	Delay(nsec)
FB1_9	FB1_9	10.0
FB1_1	FB1_1	10.0
FB1_2	FB1_2	10.0
FB1_3	FB1_3	10.0
FB1_4	FB1_4	10.0
...	...	...
FB1_4	FB6_4	13.0
FB6_4	FB6_4	13.0
FB4_8	FB6_4	13.0
FB4_9	FB6_4	13.0
FB1_1	FB6_4	13.0

\*\*\*\*\*

## Combinational Pad-to-Pad Delays(nsec)

\From	C
	L
	E
	A
	R
To	\-----

C_OUT<0>	28.5
C_OUT<1>	28.5
C_OUT<2>	28.5
C_OUT<3>	28.5
C_OUT<4>	28.5
C_OUT<5>	28.5
C_OUT<6>	28.5
C_OUT<7>	28.5

\*\*\*\*\*

```

Clock Pad-to-Output Pad Delays(nsec)

      \Clock          C
      /              L
     /              O
    /              C
   /              K
  /
Output \-----

C_OUT<0>      20.0
C_OUT<1>      20.0
C_OUT<2>      20.0
C_OUT<3>      20.0
C_OUT<4>      20.0
C_OUT<5>      20.0
C_OUT<6>      20.0
C_OUT<7>      20.0
DONE          10.0
*****
```

Register-to-Register Delays(nsec)

\From \ \ \ To		F	F	F	F	F	F	F	F	F	F
		B	B	B	B	B	B	B	B	B	B
		1	1	1	1	1	1	1	1	1	2
		1	2	3	4	5	6	7	8	9	1
FB1_1		10.0									
FB1_2			10.0								
FB1_3				10.0							
FB1_4					10.0						
FB1_5						10.0					
FB1_6							10.0				
FB1_7								10.0			
FB1_8									10.0		
FB1_9										10.0	
FB2_1											10.0
FB2_2											
FB2_3											
FB2_4											
FB2_5											
FB2_6											
FB2_7											
FB2_8											
FB3_3		13.0									
FB3_5		13.0									
FB4_7						13.0	13.0	13.0	13.0		
FB4_8		13.0									
FB4_9		13.0				13.0					



From \ To	F	F	F	F	F	F	F	F	F	F
	B	B	B	B	B	B	B	B	B	B
	2	2	2	2	2	2	2	3	3	4
	—	—	—	—	—	—	—	—	—	—
	2	3	4	5	6	7	8	3	5	7

\*\*\*\*\*

Register-to-Register Delays(nsec)

\From	F	F	F	F	F	F
B	B	B	B	B	B	B
4	4	5	5	5	6	
8	9	7	8	9	4	
To	-----					

FB1_1						
FB1_2						
FB1_3						
FB1_4						
FB1_5						
FB1_6						
FB1_7						
FB1_8						
FB1_9						
FB2_1						
FB2_2						
FB2_3						
FB2_4						
FB2_5						
FB2_6						
FB2_7						
FB2_8						
FB3_3	13.0	13.0			13.0	13.0
FB3_5	13.0	13.0				13.0
FB4_7	13.0	13.0	13.0	13.0	13.0	13.0
FB4_8	13.0	13.0				
FB4_9		13.0				
FB5_7	13.0	13.0	13.0	13.0	13.0	13.0
FB5_8	13.0	13.0		13.0	13.0	13.0
FB5_9	13.0	13.0			13.0	13.0
FB6_4	13.0	13.0				13.0
*****						

```
Setup Delays(nsec)

  \Clock          C
  \              L
  \              O
  \              C
  \              K
Data \-----

DATA_IN<0>      11.0
DATA_IN<1>      11.0
DATA_IN<2>      11.0
DATA_IN<3>      11.0
DATA_IN<4>      11.0
DATA_IN<5>      11.0
DATA_IN<6>      11.0
DATA_IN<7>      11.0
START           13.0
WRITE_END       11.0
WRITE_START     11.0

End of Timing Report
```

**Figure F-6 Static Timing Report Example**

## Pin-List Report

Use this report to see the final EPLD pin assignments.

XEPLD, Version 5.0

Xilinx Inc.

Pin-List Report

Circuit name: SCAN

Target Device: XC7354-10PC44

Integrated: 6-20-94, 12:38PM

Pkg Pin	Pin Type	Pin Use	Pin Name
---	----	---	----
1	MR		
2	I	I	DATA_IN<6>
3	I	I	DATA_IN<5>
4	I	I	DATA_IN<4>
5	CLK	I	CLOCK
6	CLK	O	C_OUT<2>
7	I	I	DATA_IN<3>
8	I/O	I	DATA_IN<2>
9	I/O	I	DATA_IN<1>
10	VSS		
11	I/O	I	DATA_IN<0>
12	I/O	I	CLEAR
13	I/O	tie	(unused)
14	I/O	tie	(unused)
15	I/O	tie	(unused)
16	I/O	tie	(unused)
17	I/O	O	C_OUT<7>
18	I/O	tie	(unused)
19	I/O	tie	(unused)
20	I/O	tie	(unused)
21	VCC		
22	I/O	O	C_OUT<6>
23	VSS		
24	I/O	O	C_OUT<4>
25	I/O	O	DONE
26	I/O	O	C_OUT<1>
27	I/O	O	C_OUT<0>
28	I	I	WRITE_START
29	I/O	tie	(unused)
30	I/O	tie	(unused)
31	VSS		
32	VCC		

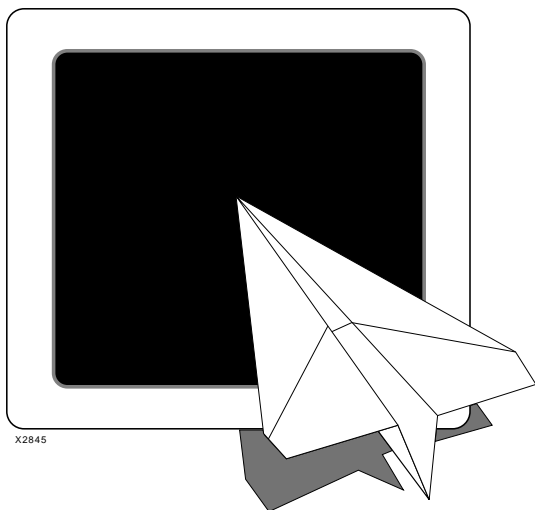
```
33  I/O  tie  (unused)
34  I/O  tie  (unused)
35  I/O  tie  (unused)
36  I/O  tie  (unused)
37  I/O  tie  (unused)
38  I/O  tie  (unused)
39  CEN   O    C_OUT<5>
40  FOE   O    C_OUT<3>
41  VCC
42  I     I     WRITE_END
43  I     I     START
44  I     I     DATA_IN<7>
```

Pin Use Legend:

```
I      - input
O      - output
I/O    - input/output
I-L    - input uses latch
I-R    - input uses register
I/O-L  - input/output uses latch
I/O-R  - input/output uses register
NC     - not connected/not available
tie    - unused pin must be tied to VCC or GND
(O)    - unused pin attached to used macrocell
```

End of Pin-List Report

**Figure F-7 The Pin-List Report**



## ***Glossary***

# ***Xilinx Synopsys Interface EPLD User Guide***

## Appendix G

### Glossary

---

This appendix provides definitions of the words and terms used in this manual and in Xilinx EPLD data sheets.

**Architecture** — A description of the functionality of a block of logic.

**Attributes** — Auxiliary information applied to individual signals or the entire design that influences how the fitter maps the design into an EPLD.

**Backannotation (timing)** — Applying timing data from the fitter to a logic model to perform timing simulation.

**Cells** — Primitive library components.

**Configuration** — A selected architecture applied to define the functionality of a given entity.

**Entity** — A description of the interface signals of a block of logic.

**EPLD** — Erasable Programmable Logic Device.

**Fast Carry** — Arithmetic carry functions using the dedicated fast carry chain that interconnects macrocells. These signals do not pass through the UIM.

**Fast Function Block (FFB)** — EPLD function blocks providing fast pin-to-pin logic throughput for critical decoding and ultra-fast state machine applications (XC7300 family only). The output pins associated with Fast Function Blocks have high current drive capability.

**Fast Output Enable (FOE)** — 3-state control signals that use the dedicated high speed FOE wiring of the device, not the UIM wiring.

**FastCLK** — A clock signal that uses the dedicated high speed FastCLK wiring of the device, not the UIM wiring.

**FastInput** — Inputs to the device that connect directly to the function block inputs, bypassing the UIM.

**Fitter** — The software that maps (fits) a logic design into a target EPLD.

**Flattening (netlist)** — Reducing a netlist to its most elemental specifications by completely expanding all contained macros.

**Function Block** — The High Density Function Blocks of the device, containing nine macrocells, designed to provide the maximum logic density and flexibility, including arithmetic carry logic.

**I/O Blocks** — The input/output logic of the device containing pin drivers, registers and latches, and 3-state control functions.

**Inference** — The automatic selection and placement of a library cell into a gate-level design by the synthesizer, based on a behavioral description in the source design. For example,  $A + B$  infers an adder.

**Input Pad** — The input interface logic of the device containing registers and latches.

**Input Pad Registers and Latches** — D Type registers located in the I/O pad sections of the device. Input pad registers can be used instead of macrocell registers to increase logic density and provide shorter setup time.

**Instance** — A single occurrence of any specific library component.

**Instantiation** — Manual selection and placement of a library cell into a gate-level design by a structural reference in the source file.

**Macrocell** — The basic unit of logic in the device. A macrocell can implement both combinational and registered equations. High Density Function Block macrocells also contain an ALU for implementing arithmetic functions.

**Minimization** — The process of reducing a logic function to a sum-of-products expression consisting of the least number of product terms.

**Netlist** — The specification of all the instances in a design or block of logic and the interconnections (wiring) between them.

**Node** — Any signal used only internally within a block of logic.

**Optimization** — The process of reducing your design to the minimal required device resources. Optimization includes collapsing of



combinational logic nodes into device outputs and registers, assigning signals to global FASTCLOCK and FOE nets, utilization of I/O buffer registers, and the creation of UIM-AND functions.

**Partitioning** — The process of placing symbolic logic into the physical structures of the device. The basic partition is the Function Block or Fast Function Block.

**Pin** — The physical XC7000 device pins (external connections).

**Pin Feedback** — Specifies that the associated signal comes from the actual device pin and not from the UIM.

**PLD** — Programmable Logic Device.

**PLUSASM** — The Xilinx native behavioral design language for EPLD development.

**Port** — Any signal used as an external interface (input or output) to a block of logic.

**Product Term Cascading** — The process of passing product terms (in groups of four) from one macrocell to another for the purpose of increasing the number of usable product terms.

**Target Device** — The physical device (EPLD) in which your logic design is implemented.

**Test Bench** — A VHDL file used to test a VHDL logic design, containing signal transitions, time delays, and an instance of the device under test.

**Universal Interconnection Matrix (UIM)** — The primary device resource used to interconnect macrocells. Propagation delays through the UIM are constant and independent of the interconnections. AND functions can also be implemented in the UIM.

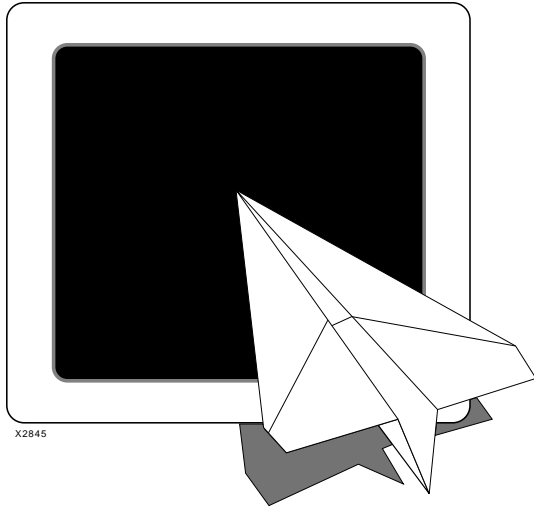
**UIM-AND Function** — An AND gate created from the inherent wired-AND structure of the UIM; requires no macrocell resources.

**UIM Feedback** — Specifies that the associated signal comes from the macrocell and not from the device pin.

**Wired-AND Functions** — AND gates (and their DeMorgan equivalents) produced by the inherent structure of the UIM.

**XEPLD** — Xilinx EPLD development software which includes the fitter.





# ***Xilinx Synopsys Interface EPLD User Guide***

***Index***

# Index

---

## Symbols

- + operator, 2-8, B-4, B-5
- +1 operator, B-27
- operator, B-5, B-36

## Numerics

- 1 operator, B-19
- 3-state control inputs, 2-6, C-2
- 3-state I/O buffer, component, B-29

## A

- ACC, component, 2-8, B-3
- ADD, component, B-4
- adder, component, B-4
- adder/accumulator, component, B-3
- adder/subtractor, component, B-5, B-6
- ADSU, component, B-5
- ADSUR, component, 2-8, B-6
- ALU, A-4, A-5
- analyze
  - design, 1-10, 1-17, 1-21
  - test bench, 1-10, 1-22
- AND gate
  - component, B-7
  - optimization, C-4
- AND2-AND8, components, B-7
- area estimation, iii
- arithmetic functions, creating, 2-8
- attributes
  - F, 2-14, C-3
  - global, C-1
  - H, C-4
  - LOWPWR, 2-14, C-1
  - MRINPUT, 5-5, C-1

- NO\_FCLK, 2-13, C-2
- NO\_FOE, 2-6, C-2
- NO\_IFD, 2-2, C-2
- OPT\_OFF, C-4
- OPT\_UIM, C-4
- part type, C-5
- pin assignment, C-6
- PRELOAD, 2-4, C-3
- register initial state, C-6
- signal, C-3
- slewwrate, C-4
- Synopsys, C-5

## B

- backannotation, timing, 1-21
- binary counters, 2-6
- BUF, component, B-8
- BUFCE, component, B-9, B-25
- BUFE, component, B-10
- buffer
  - component, B-8, B-10
  - global CE, B-9
  - global clock, B-12
  - global FOE, B-11
  - I/O, B-30
  - input, B-12, B-23
  - output, B-32, B-33
- BUFFOE, component, 2-5, B-11, B-30, B-34
- BUFG, component, 2-5, B-12, B-26

## C

- carry chain, A-5
- carry lookahead, A-6

- CBX1, component, 2-6, B-13
- CBX2, component, 2-6, B-14
- clock enable, A-12
- clocks, high speed, 2-12
- comparator component, B-15, B-16, B-17, B-18
- comparators, using, 2-8
- COMPEQ component, 2-8, B-15
- compiler, 4-2
- compiling designs, 3-1
- COMPLE component, 2-8
- COMPLE\_TC component, B-16
- COMPLE\_US component, B-16
- COMPLT component, 2-8
- COMPLT\_UC component, B-17
- COMPLT\_US component, B-17
- COMPNE component, B-18
- COMPNE components, 2-8
- components
  - ACC, 2-8, B-3
  - ADD, B-4
  - ADSU, B-5
  - ADSUR, 2-8, B-6
  - AND2-AND8, B-7
  - BUF, B-8
  - BUFCE, B-9, B-25
  - BUFE, B-10
  - BUFFOE, 2-5, B-11, B-30, B-34
  - BUFG, 2-5, B-12, B-26
  - CBX1, 2-6, B-13
  - CBX2, 2-6, B-14
  - COMPEQ, 2-8, B-15
  - COMPLE, 2-8
  - COMPLE\_TC, B-16
  - COMPLE\_US, B-16
  - COMPLT, 2-8
  - COMPLT\_UC, B-17
  - COMPLT\_US, B-17
  - COMPNE, 2-8, B-18
  - DEC, B-19
  - device-specific, 2-14
  - FDCP, B-20
  - FDCPE, B-21
  - FDPC, B-22
  - IBUF, 2-5, B-23
  - IFD, 2-5, B-24
  - IFDX1, 2-5, B-9, B-25
  - ILD, B-26
  - INC, B-27
  - INV, B-28
  - IOBUFE, 2-5, B-29
  - IOBUFEX1, B-11, B-30
  - LD, B-31
  - OBUF, 2-5, B-32
  - OBUFE, B-33
  - OBUFEX1, B-11, B-34
  - OR2-OR8, B-35
  - SUBT, B-36
  - XOR2-XOR8, B-37
- conventions, syntax, v
- counters, creating, 2-6
- D**
  - D1, D2 ALU inputs, A-4
  - DC Shell, 1-16
    - using, 3-1
  - Debugger, VHDL, 1-14
  - DEC, component, B-19
  - decrementor, component, B-19
  - delay calculation
    - clock-to-output, F-4
    - cycle time, F-5
    - pad-to-pad, F-3
    - setup-to-clock, F-4
  - design
    - compilation, 1-16
    - entry, 1-9
    - example, 1-5, E-1
    - flow, EPLD, 1-5
    - implementation, 5-9
    - iteration, 4-1, 4-5
    - verification, 4-4
- Design Compiler, Synopsys, i

- 
- Design Rule Checker, 2-14
  - device
    - programming, 4-1, 4-5
    - resource estimation, 2-10
    - selection, 1-19, A-2
    - selection, chart, A-2
  - down counters, creating, 2-7
  - D-type flip-flop, B-21, B-22, B-24, B-25, B-26, B-31
  - E**
  - elaborate, Synopsys command, 1-17
  - EPLD
    - architecture, A-1
    - design flow, 1-5
    - programming, 1-24
  - example design, 1-5, E-1
  - F**
  - F attribute, 2-14, C-3
  - fast carry, definition, G-1
  - Fast Function Block, A-7
    - attributes, C-3
    - definition, G-1
  - fast inputs, specifying, 2-14
  - Fast Output Enable, definition, G-1
  - FastCLK
    - definition, G-1
    - signals, A-6
  - FastClock
    - inputs, C-2
    - pins, 2-5
  - FastInput
    - definition, G-2
    - pins, A-3
  - FDCP, component, B-20
  - FDCPE, component, B-21
  - FDPC, component, B-22
  - features
    - unsupported, iii
    - Xilinx software, ii
  - file translation, 4-2
  - syn2epld, 1-19
  - files
    - .eqn, D-1
    - .lgc, D-1
    - .log, D-1
    - .map, D-1
    - .par, D-1
    - .pin, D-1
    - .res, D-1
    - .tim, D-1
    - pinsave, 4-1
    - used by the fitter, D-1
    - XFF, 4-2
  - fitnet, XSI command, 1-20, 4-3, 5-10
  - fitter
    - definition, G-2
    - operation, 4-2
  - fitting, 1-20
    - overview, 4-1
  - flip-flop, component, B-21, B-22, B-24, B-25, B-26, B-31
  - flip-flops, EPLD architecture, A-6
  - FOE
    - input, usage, C-2
    - pins, 2-5
  - Function Blocks
    - EPLD architecture, A-3
    - specifying, 2-13
  - functional simulation, 1-9
  - G**
  - global Chip Enable buffer, B-9
  - global clock, A-11
    - buffer, B-12
    - enable, A-12
  - global FOE buffer, B-11
  - global optimization, inhibiting, 2-5
  - Glosasry, G-1
  - H**
  - H attribute, C-4
  - HDL, i

High Density Function Block

- attributes, C-4
- description, A-3

**I**

- I/O block, A-10
  - definition, G-2
- I/O buffer cells, placing, 1-18
- I/O buffer, component, B-29, B-30
- I/O ports
  - using, 2-5
- I/O signals, defining, 3-3
- IBUF, component, 2-5, B-23
- IFD, component, B-24
- IFDX1, component, 2-5, B-9, B-25
- ILD, component, 2-5, B-26
- INC, component, B-27
- include, Synopsys command, 1-17
- incrementor, component, B-27
- inference, definition, G-2
- initial state specification, register, 1-18
- input buffer, component, B-23
- input pad registers, 2-5
  - usage, C-2
- Inputs
  - fast, using, 2-14
  - hanging, 2-14
- Inputs, 3-state control, C-2
- installation, verification, 1-3
- INV, component, B-28
- inverter, component, B-28
- IOBUFE, component, 2-5, B-29
- IOBUFEX1, component, B-11, B-30

**L**

- latches
  - input pad, 2-3
  - macrocell, 2-3
  - using, 2-1
- LD, component, B-31
- library
  - availability chart, B-1

- declaration, 2-1

- LOWPWR attribute, 2-14, C-1

**M**

- macrocell, definition, G-2
- makeprg, XSI command, 1-25, 4-5
- mapping effort, 1-18
- Mapping Report, 4-4
- mapping, equations, 4-1
- Master Reset
  - pin, 2-3
  - simulating, 5-2
- Master Reset pin, C-1
- minimization, equations, 4-1
- MRESET input, 5-3, 5-15
- MRINPUT attribute, 5-5, C-1

**N**

- netlist
  - flattened, 1-19, 4-2
  - outputting, 1-19
- NO\_FCLK, attribute, 2-13, C-2
- NO\_FOE, attribute, 2-6, C-2
- NO\_IFD, attribute, 2-2, C-2
- node, definition, G-2

**O**

- OBUF, component, 2-5, B-32
- OBUFFE, component, B-33
- OBUFFEX1, component, B-11, B-34
- one-hot-encoding, state machine, 2-7
- operators
  - , B-5, B-36
  - +, 2-8, B-4, B-5
  - +1, B-27
  - 1, B-19
- OPT\_OFF, attribute, C-4
- OPT\_UIM, attribute, C-4
- optimization
  - AND gates, C-4
  - clock, 2-13
  - definition, G-2
  - equations, 4-1

- inhibiting, 2-5, C-4
- output enable, 2-6
- register/latch, 2-2
- technology specific, iii
- timing-constraint-driven, iii
- OR gates, B-35
- OR2-OR8, components, B-35
- output buffer, component, B-32, B-33, B-34
- output enable signals, 2-6
- P**
- part type, attribute, C-5
- Partitioner Report, 4-4
- partitioning
  - definition, G-3
  - equations, 4-1
- PCI bus example design, E-1
- pin assignment, 2-11
  - attribute, C-6
- pin feedback, definition, G-3
- pin, definition, G-3
- pinouts, saving, 4-1, 4-5
- pins
  - FastClock, 2-5
  - FOE, 2-5
- pinsave file, 4-1
- pinsave, XSI command, 2-12, 4-5
- pipelines, register, 2-8
- PLD, definition, G-3
- PLUSASM, definition, G-3
- Port components
  - design rules, 2-15
- power usage, macrocells, C-1
- PRELOAD, attribute, 2-4, C-3
- product term cascading, definition, G-3
- product terms
  - expansion, A-9
  - shared, A-4
- programming, EPLD, 1-24, 4-1, 4-5

**R**

- register
  - initial state, attribute, C-6
  - initial state, controlling, 2-3, 5-2
  - initial values, C-3
  - input pad, 2-2, 2-5, C-2
  - macrocell, 2-2
  - pipelines, 2-8
  - specifying initial states, 1-18
  - using, 2-1

**Reports**

- Mapping, 4-4
- Partitioner, 4-4
- Pin-list, F-1
  - example, F-13
- Resource, 2-10, F-1
  - example, F-2
- Static Timing, 2-10, 4-1, 4-4, F-1
  - example, F-3, F-6
- Resource Report, 2-10
- ripple carry delay, A-6
- Rules for XEPLD designs
  - device-specific components, 2-14
  - hanging inputs, 2-14
  - port components, 2-15

**S**

- set up files
  - creating, 1-1
  - Design Compiler, 1-2
  - VSS Simulator, 1-2
- set\_attribute, Synopsys command, C-5
- shared product terms, A-4
- simulation, 5-1
  - functional, 1-9, 5-6
  - Master Reset, 5-2
  - strategy, 5-1
  - timing, 5-11
  - verilog, iii
- slewwrate attribute, C-4
- state machines, creating, 2-7



Static Timing Report, 2-10, 4-1, 4-4  
    asynchronous set and reset, F-3  
    clock-to-output, F-4  
    cycle time, F-5  
    example, F-6  
    pad-to-pad delays, F-3  
    setup-to-clock, F-4

SUBT, component, B-36

subtractor, component, B-36

syn2epld, XSI command, 1-19, 1-20,  
2-10, 4-2

Synopsys commands

- analyze, 1-17, 3-2
- compile, 1-18, 3-3
- dc\_shell, 1-16, 3-2
- elaborate, 1-17, 3-2
- exit, 1-19, 3-5
- include, 1-17
- insert\_pads, 1-18, 3-4
- set\_attribute, 1-18, 2-4, 2-11, 3-4
- set\_port\_is\_pad, 1-18, 3-3
- trace, 1-15, 1-23, 5-8
- vhdlan, 1-10, 1-21, 5-6
- vhldbx, 1-14, 1-22, 5-6, 5-13
- write, 1-19, 3-4

Synopsys Design Compiler, 4-2

synthesizing, design, 1-18

## **T**

target device, specifying, 1-18, 2-9, 4-2

test bench

- configuration declaration, 5-5
- creating, 5-4
- initializing registers, 5-4

timing

- backannotation, 1-21
- calculated, 4-1
- information, iii
- simulated, 4-1
- simulation, 1-21

timing model preparation, 5-10

## **U**

UIM, A-3

- AND function, definition, G-3

- attributes, C-4

- feedback, definition, G-3

Universal Interconnection Matrix, definition, G-3

up counter, 2-6

up/down counter

- component, B-13, B-14

- creating, 2-7

## **V**

verification

- design timing, 4-4

- file structure, 1-4

- software installation, 1-3

VHDL, i

vmh2vss, XSI command, 1-21, 4-4, 5-1

VSS timing simulator, i, 4-1

## **W**

Waves, Dynamic Waveform Viewer,  
1-15

wired-AND functions, definition, G-3

## **X**

XEPLD, 4-1

- definition, G-3

XFF file, 4-2

XNF netlist, 1-19

XOR gates, B-37

XOR2-XOR8, components, B-37

XSI, 4-1

XSI commands

- fitnet, 1-20, 4-3, 5-10

- makeprg, 1-25, 4-5

- pinsave, 2-12, 4-5

- syn2epld, 1-19, 1-20, 2-10, 4-2

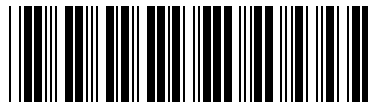
- vmh2vss, 1-21, 4-4, 5-1



The Programmable Logic Company<sup>SM</sup>

2100 Logic Drive, San Jose CA 95124-3400

Tel: (408) 559-7778 FAX: (408) 559-7114



0401331