XILINX®

ONLINE

CPLD XSI

DESIGN GUIDE

XSTEP™

Synthesis Design Guide

V1.0 for Workstations

Getting Started with Xilinx EPLDs

Designing with EPLDs

Compiling and Fitting Your Designs

Simulating Your Design

Library Component Specifications

Preface

About This Manual

This manual has been created to provide guidance in use of Synthesis Design in the workstation environment.

Manual Contents

This manual covers the following topics.

- Chapter 1. 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 XC7000 or XC9000 EPLD designs using Synopsys.
- Chapter 2. This chapter discusses how to use design techniques, library cells and xepld command parameters to get the best performance from Xilinx XC7000 and XC9000 EPLDs.
- Chapter 3. This chapter describes how to compile your design using the Synopsys Design Compiler shell (DC Shell).
- Chapter 4. The Xilinx EPLD Synopsys Interface supports both functional and timing simulation of VHDL designs using the VSS simulator. This chapter shows you how to prepare designs for simulation and how to use a test bench.
- Appendix A describes each of the Xilinx library components.

Conventions

In this manual the following conventions are used for syntax clarification and command line entries.

- `Courier font` indicates messages, prompts, and program files that the system displays, as shown in the following example.

  ```
  speed grade: -100
  ```

- **`Courier bold`** indicates literal commands that you must enter in a syntax statement.

  `rpt_del_net=`

- *Italic font* indicates variables in a syntax statement. See also, other conventions used on the following page.

  `xdelay` *design*

- Square brackets "[ ]" indicate an optional entry or parameter. However, in bus specifications, such as bus [7:0], they are required.

  `xdelay [`*option*`]` *design*

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

  `xnfprep` *designname* **`ignore_rlocs={true|false}`**

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

  `symbol` *editor* `[bus|pins]`

Other conventions used in this manual include the following.

- *Italic font* indicates references to manuals, as shown in the following example.

  See the *Development System Reference Guide* for more information.

- *Italic font* indicates emphasis in body text.

  If a wire is drawn so that it overlaps the pin of a symbol, the two nets *are not* connected.

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

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

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

  `allow block` *`blockname loc1 loc2 ... locn`* `;`

Table of Contents

Chapter 1 Getting Started with Xilinx EPLDs

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 XC7000 or XC9000 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 Release Document.

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 or XC9000 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 Xilinx EPLD designs are processed.

Note: You will find sample setup files in the `$DS401/tutorial/synopsys/scan` directory. The sample setup files are configured for XC9000 designs. However, before using them, you must edit the `.synopsys_dc.setup` file contained in the tutorial directory by typing the actual `$DS401` path into the `search_path` variable. Your $DS401 variable should point to the XC9500 Pre-release installation directory.

Note: In UNIX and DC Shell commands shown in this book, where text is too long to print on one line, the back-slash (\) character at the end of a line is used to indicate a continuation line. In actual usage, continuation line breaks are optional and may occur at any legal

point in the command line.

The Design Compiler Setup File

For XC9000 designs, your Design Compiler setup file (`.synopsys_dc.setup`) must contain the following lines:

```
search path = { . \
  XACT9500_path/synopsys/libraries/syn \
  $Synopsys_path/libraries/syn}
link_library = {xc9000.db}
target_library = {xc9000.db}
symbol_library = {xc9000.sdb}
compile_fix_multiple_port_nets = true
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"
```

For XC7000 designs, your Design Compiler setup file (`.synopsys_dc.setup`) must contain the following lines:

```
search path = { . \
  XACT9500_path/synopsys/libraries/syn \
  $Synopsys_path/libraries/syn}
link_library = {xc7000.db XC7000.sldb}
target_library = {xc7000.db}
symbol_library = {xc7000.sdb}
synthetic_library = {xc7000.sldb}
define_design_lib xc7000 -path \
  XACT9500_path/synopsys/libraries/dw/lib/xc7000
compile_fix_multiple_port_nets = true
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 *XACT9500_path* is the actual directory path where your XC9500 Pre-release software is installed, and *$Synopsys_path* is the actual path where your Synopsys software is installed.

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

The VSS Simulator Setup File (.synopsys_vss.setup)

For XC9000 designs, your VSS Simulator setup file, `.synopsys_vss.setup`, must contain the following lines:

```
xc9000: $DS401/synopsys/libraries/sim/lib/xc9000
TIMEBASE = NS
TIME_RES_FACTOR = 0.1
```

For XC7000 designs, your VSS Simulator setup file, `.synopsys_vss.setup`, must contain the following lines:

```
xc7000: $DS401/synopsys/libraries/sim/lib/xc7000
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 XC9500 Pre-release 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 Synopsys 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 xepld**

   If **`xepld`** cannot be found, the XC9500 Pre-release software is not installed or is not in your path.

2. **echo $DS401**

   This variable should point to the XC9500 Pre-release directory found in Step 1.

3. **echo $XACT**

   This variable should also point to the XC9500 Pre-release directory found in Step 1.

Verifying Your File Structure

To verify that you have the necessary files for EPLD development,

use the file structure diagram in Figure 1-1.

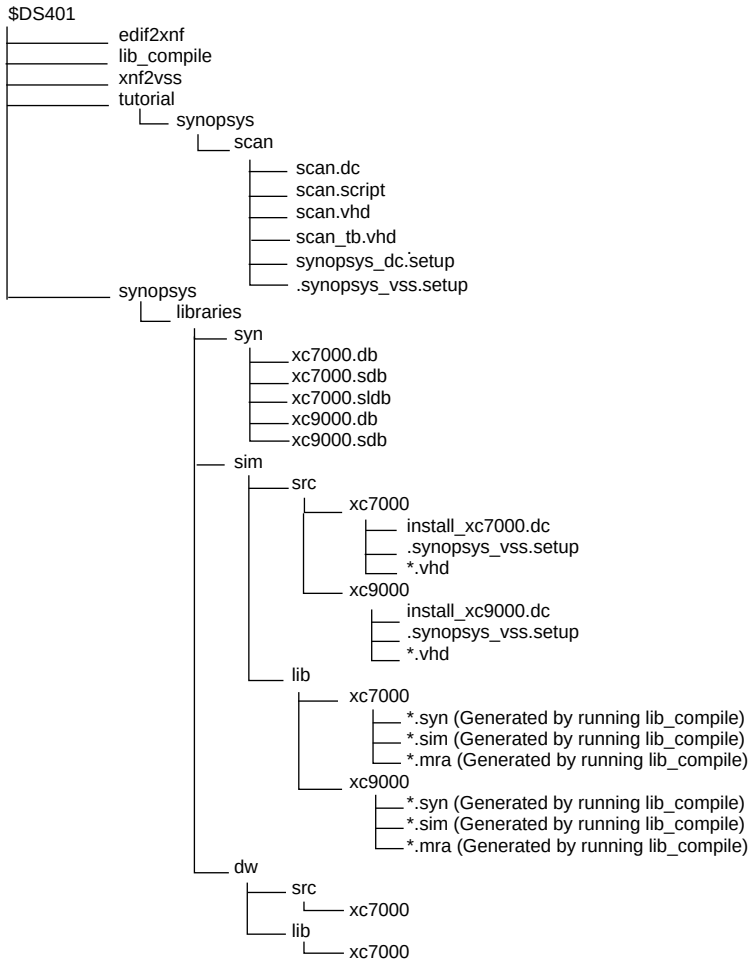


Figure 1-1 File Structure for Synopsys Interface

Xilinx EPLD Design Flow

Figure 1-2 shows the basic design flow for creating XC9500 designs. Each step is described in the following design example.

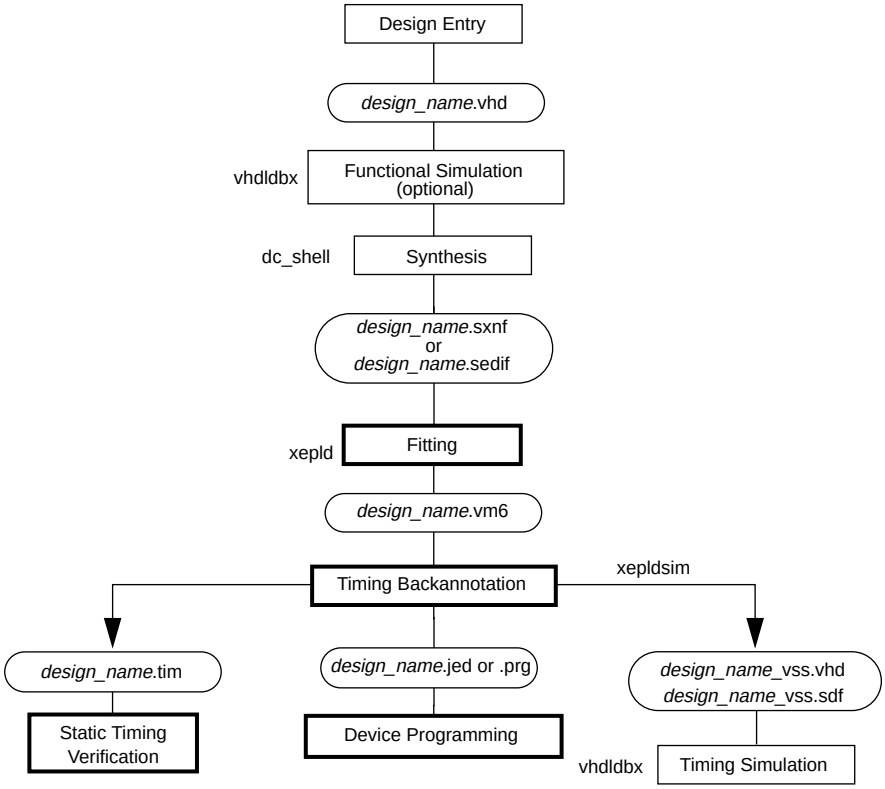


Figure 1-2 Basic EPLD Design Flow

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 can be implemented in a Xilinx XC95108-7PC84 or XC7354-7PC44 device.

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

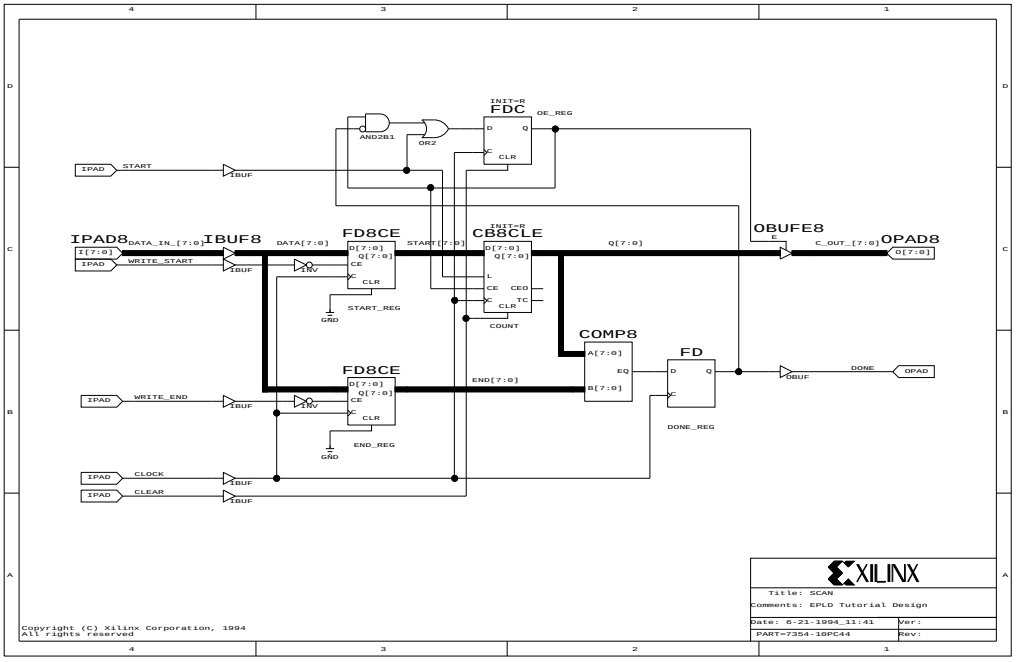


Figure 1-3 Schematic Representation — SCAN Design

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:  XC95108-7PC84                              --
--                                                              --
--   Author:         Xilinx Corporation                         --
--                   Copyright (C) Xilinx Corporation 1995
--                   All rights reserved                        --
--                                                              --
--   Requirements: Xilinx XC9500 Pre-release V1.0
--                   Synopsys Design Compiler v3.1 or later
--                                                              --
------------------------------------------------------------------

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

```
            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/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 installed tutorial area into your current directory as follows:

  ```
  cp -r  $DS401/tutorial/synopsys/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 XC9500 Pre-release 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 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;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_components.all;
use STD.Textio.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:

```
vhdldbx
```

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

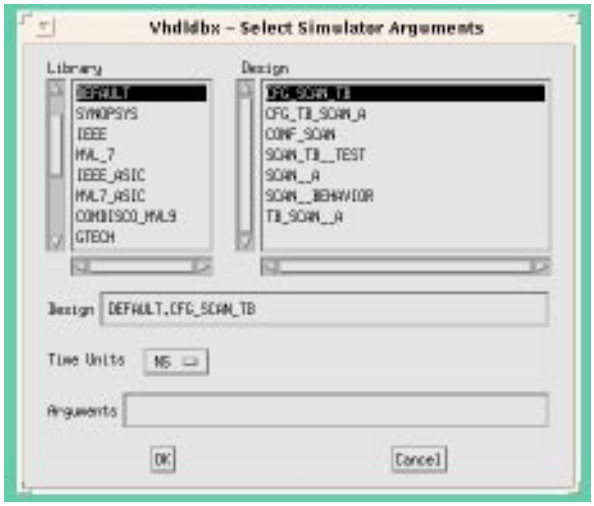


Figure 1-6 VHDLDBX Window

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.

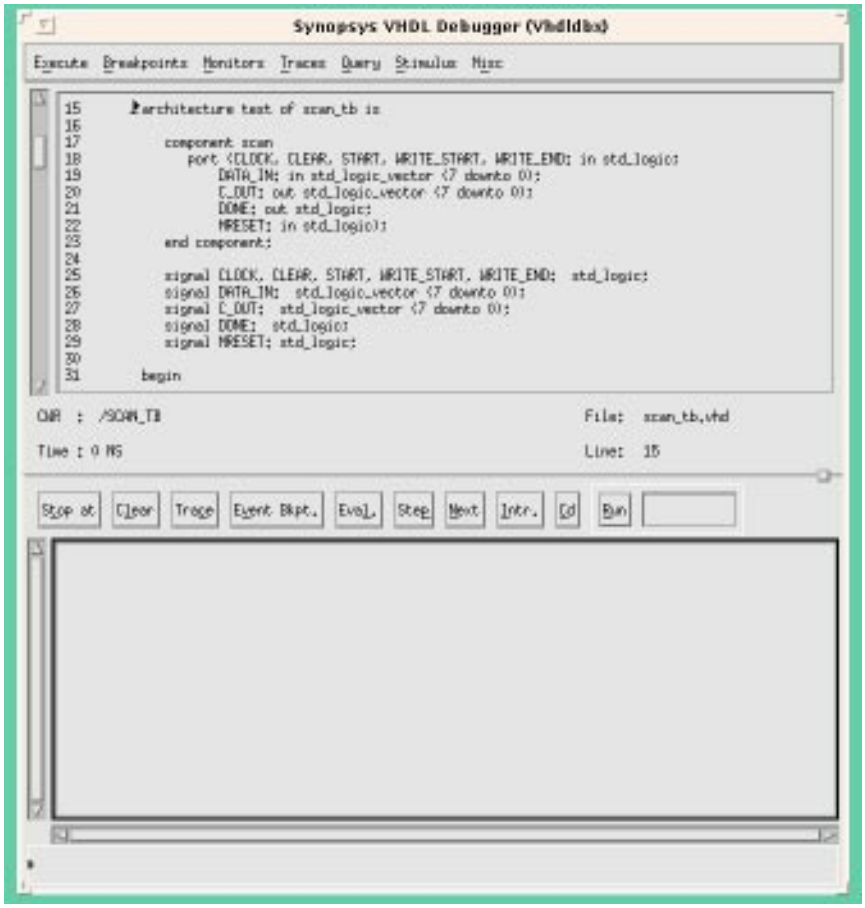


Figure 1-7 Synopsys VHDL Debugger

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.

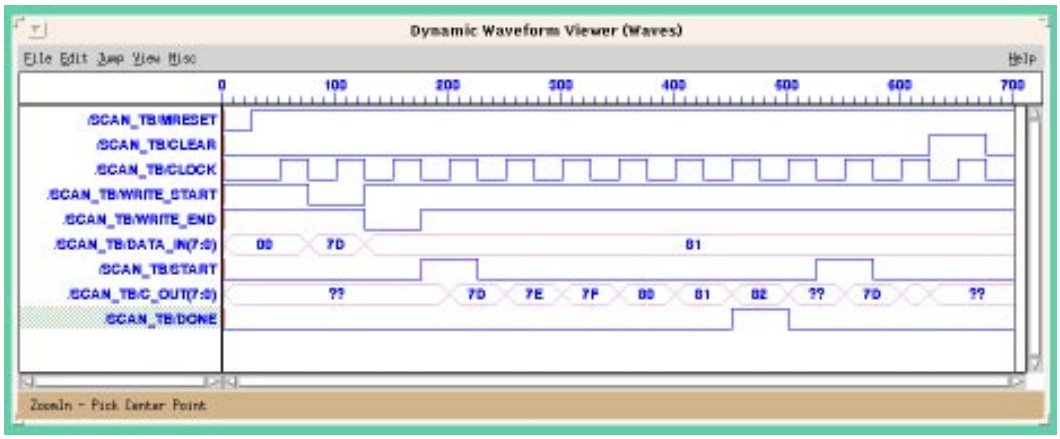


Figure 1-8 Synopsys Dynamic Waveform Viewer (Waves)

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 also 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 or XC9000 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, you can specify the target EPLD device using a Synopsys attribute. Otherwise, you can specify the target device later when you invoke the fitter. Enter the following Synopsys command to specify a target XC9000 device:

```
set_attribute scan part -type string 95108-7pc84
```

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 initial states of the remaining flip-flops are not critical for this design.

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.edif
```

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 fitter step. No speed or area estimates are provided by the XC7000 or XC9000 library. Therefore do not attempt to create a timing report or perform estimated timing simulation at this time.

Fitting Your Design

The XEPLD fitter translates your logical design file (`scan.sxnf or scan.edif`) into a physical device layout.

Step 18 — Fit Your Design

To fit your design into an XC9000 device, enter the following on the UNIX command line:

```
xepld -p 9 scan
```

To fit your design into an XC7000 device, enter the following on the

UNIX command line:

```
xepld -p 7 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, **xepld** 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, and assuming there are no errors, you need only to examine the fitter report file. For XC9000 designs, examine scan.rpt. For XC7000 designs, examine `scan.res`. If you wish, you can also examine the static timing report file, scan.tim.

If you need more information on fitting, see the "Fitting Your Design" chapter.

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 the remainder of this tutorial.

Step 19 — Prepare A Timing Simulation File

The xepld command produces a timing simulation netlist file (scan_tim.xnf) each time the design is successfully implemented. To translate the netlist for VSS simulation, enter the following command at the UNIX prompt:

```
xepldsim -vss scan
```

The xepldsim command produces a structural VHDL file (scan_vss.vhd) and an SDF-formatted timing back-annotation file (scan_vss.sdf).

Step 20 — 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 21 — Analyze Your Back-Annotated Design

Analyze the back-annotated design architecture, produced by the Xilinx **xepldsim** 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 22 — 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 23 — Invoke the VSS Simulator

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

```
vhdldbx -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 **xepldsim**. The **-sdf_top** parameter specifies the level in the test bench hierarchy at which the back annotation information will be applied, which is the “UUT” instance of the scan design.

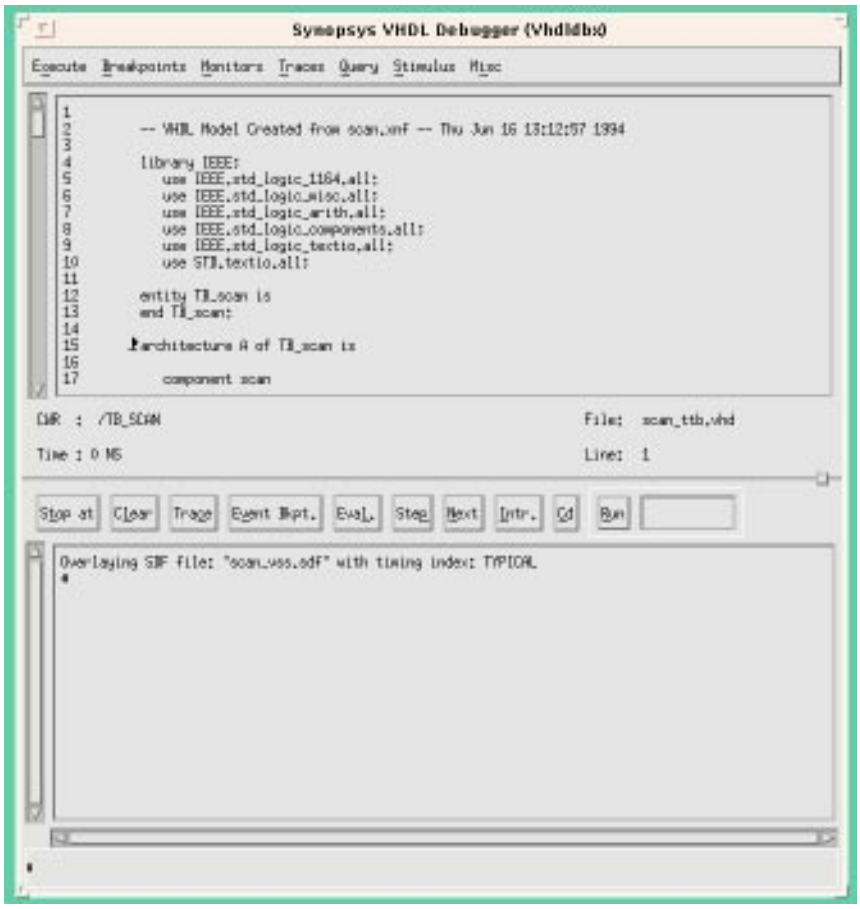


Figure 1-9 Synopsys VHDL Debugger

Step 24 — 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 25 — 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.

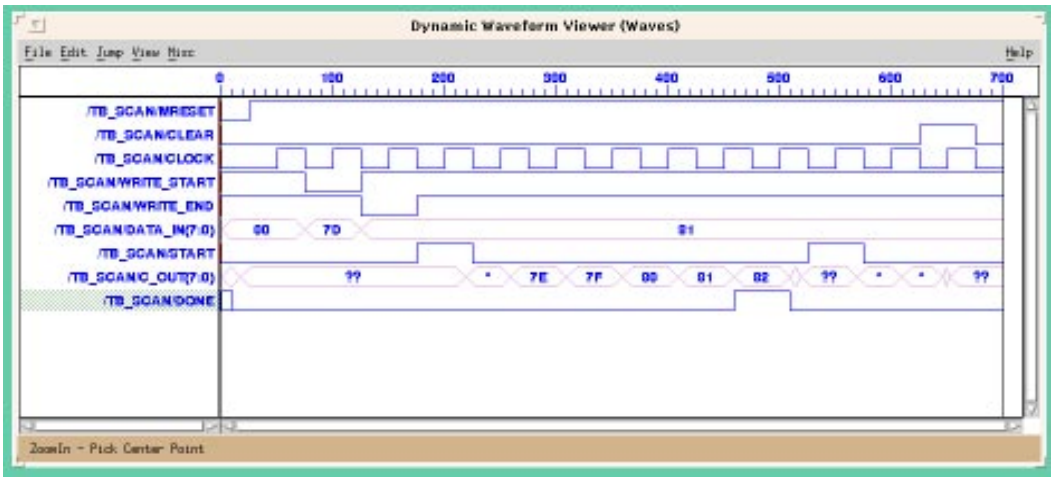


Figure 1-10 Synopsys Dynamic Waveform Viewer (Waves)

Step 26 — 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.

Chapter 2

Designing with EPLDs

This chapter discusses how to use design techniques, library cells and xepld command parameters to get the best performance from Xilinx XC7000 and XC9000 EPLDs. For more information on library components, see the “Library Component Specifications” appendix. For more information on attributes, see the “Attributes” appendix.

This chapter describes how you can control the following aspects of design implementation:

- Device selection
- Register clocking and asynchronous set / reset
- Device output enable control
- Register power-up state
- Macrocell power / speed trade-off
- Output slew rate
- Pin assignment and pin freezing
- Logic collapsing
- Timing specifications and optimization

VHDL Design File Requirements

The XC7000 library contains an assortment of scalable arithmetic and counter components that support efficient logic implementations in that architecture. If you plan to instantiate any components from the XC7000 library you will need to declare the Xilinx `XC7000.components` package in your design source file. 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;
```

The XC9000 library does not provide specialized components for arithmetic or counter functions; synthesized generic logic solutions are used instead.

Target Device Selection

By default, the fitter will automatically select an XC9500 family device for you, choosing, in general, the smallest part that will satisfy the needs and constraints of your design.

You can optionally specify a target device on the xepld command line when you run the fitter. The format of the part-type parameter on the xepld command line is:

xepld -p *ddddd-sspppp design_name*

where

ddddd is the device code, for example 95108

ss is the speed grade, for example 10

pppp is the package type and pin count, for example PC84

You may specify either a unique device code, a range of eligible devices or an entire EPLD family from which the fitter will automatically choose. To specify just the EPLD family, use just “7” (for XC7000) or “9” (for XC9000) as the part-type parameter value.To specify a range of devices, you can use an asterisk (*) as a wildcard character. You may also specify an enumerated list of devices, separated by commas.

If you use the asterisk character or an enumerated list in the xepld command, you must enclose the parameter string in quotes. If you use an asterisk in the part code field, your string must begin with a “7” (for XC7000) or “9” (for XC9000). For example, the following are valid part-type parameter specifications:

```
xepld -p 95108-10PC84 design1
xepld -p “95108-*PC84” design1
xepld -p “95108-10PC84,95108-7PQ*” design1
xepld -p 7 design1
```

Note: You cannot mix device codes from the XC7000 and XC9000

families in the same xepld command.

As an alternative, if you are using FPGA Compiler, you can set the "part" attribute in DC Shell, as follows:

set_attribute *design_name* **part -type string** *part_type*

You cannot, however, specify wildcards or lists using the DC Shell PART attribute as you can in the xepld command line. If you want the fitter to use the part specified in the DC Shell part attribute, you must include the parameter "-p indesign" on your xepld command line. For example:

```
xepld -p indesign design1
```

If you specify any part code in the xepld -p parameter (instead of "indesign"), it will override any PART attribute set in DC Shell.

Refer to the Release Document for a list of EPLD device codes supported by the current version of the fitter software.

EPLD-Specific I/O Ports

Ordinarily, you need only to declare ports in your top-level entity to represent I/O pins on the EPLD device. The Synopsys set_port_is_pad and insert_pads commands automatically infers IBUF, OBUF, OBUFE and IOBUFE cells from the XC7000 or XC9000 library to represent the I/O ports in the XNF netlist.

The following sections describe special global control pins on EPLD devices that can be used for register clocking, 3-state control and register set/reset, instead of ordinary IBUF inputs. Unless otherwise specified, the fitter automatically allocates these special global pins, if possible, when input ports in your design are used to perform these control functions.

Clock Inputs

To use a device input as a clock source, you can simply refer to a top-level input port as the clock condition in a process. For example:

```
entity xyz is
  port (CLOCK:in std_logic; ...
...
process (CLOCK)
  begin
```

```
    if (CLOCK'event and CLOCK='1') then
      ...
```

The fitter automatically uses one of the global clock pins (GCK for XC9000 or FCLK for XC7000) whenever possible.

For XC9000 devices, a global clock input signal may perform negative-edge clocking. For example:

```
process (CLOCK)
  begin
  if (CLOCK'event and CLOCK='0') then
    ...
```

The same clock input may even be used both as both positive-edged and negative-edged to clock different processes on opposite edges of the clock signal. Global clock inputs may also be used as ordinary input signals to other logic elsewhere in the design.

If an input port signal passes through any logic function (other than an inverter) before it is used as a clock, the input cannot be routed to the flip-flops using the global clock path. Instead, the flip-flop's clock signal will be routed through the logic array.

For XC7000 devices, input port signals must be used only as positive-edge clocks and for no other functions in the design in order for the global clock path (FastClock) to be used.

There are a limited number of global clock pins on each EPLD device (consult the device data sheet). If you need to explicitly control the use of global clock pins, you can instantiate the BUFG cell from the XC7000 or XC9000 library and pass your input port signal through it. For example:

```
U1: BUFG port map (O=>GLOBAL_CLOCK, I=>CLOCK);
process (GLOBAL_CLOCK)
  begin
  if (GLOBAL_CLOCK'event and GLOBAL_CLOCK='0') then
    ...
```

The global clock pins provide much shorter clock-to-output delays than clocks routed through the logic array. Routing a clock through the logic array also uses up one extra p-term for each flip-flop.

If you want to prevent the fitter from automatically using the global clock pins, specify the "-nogck" parameter on the xepld command line as follows:

xepld -nogck *design_name*

If -nogck is specified, input ports used as clocks will always pass through the array. You can still instantiate BUFG cells to explicitly specify global clock inputs.

Output Enable Signals

To use a device input to control three-state device outputs, you can simply refer to a top-level input port signal as a 3-state condition in your design. For example:

```
entity xyz is
  port (ENABLE:in std_logic; ...
...
Q <= Q_VALUE when (ENABLE='1') else 'Z';
```

The fitter automatically uses one of the global three-state control pins (GTS for XC9000 or FOE for XC7000) whenever possible.

For XC9000 devices, a global 3-state control input signal may perform an active-low output-enable. For example:

```
Q <= Q_VALUE when (ENABLE='0') else 'Z';
```

The same 3-state control input may even be used both active-high and active-low to enable alternate groups of device outputs. Global 3-state control inputs may also be used as ordinary input signals to other logic elsewhere in the design.

If an input port signal passes through any logic function (other than an inverter) before it is used as an output enable, the input cannot be routed to the device output drivers using the global 3-state control path. Instead, the output enable signal will be routed through the logic array.

For XC7000 devices, input port signals must be used only as active-high output enables, and for no other functions in the design, in order for the global 3-state control path (FOE) to be used.

There are a limited number of global 3-state control pins on each EPLD device (consult the device data sheet). If you need to explicitly control the use of global 3-state control pins, you can instantiate the BUFGTS cell (for XC9000) or BUFFOE cell (for XC7000) and pass your input port signal through it. For example:

```
U1: BUFGTS port map (O=>GLOBAL_ENABLE, I=>ENABLE);
```

```
Q <= Q_VALUE when (GLOBAL_ENABLE='1') else 'Z';
```

The global 3-state control pins provide much shorter input-to-output-enable delays than 3-state controls routed through the logic array. Routing a 3-state control signal through the logic array also uses up one extra p-term for each output.

If you want to prevent the fitter from automatically using the global 3-state control pins, specify the "-nogts" parameter on the xepld command line as follows:

`xepld -nogts` *design_name*

If -nogts is specified, input ports used for 3-state control will always pass through the array. You can still instantiate BUFGTS or BUFFOE cells to explicitly specify global 3-state control inputs.

Asynchronous Clear and Preset

To use a device input as an asynchronous clear or preset source, you can simply refer to a top-level input port as the set or reset condition in a clocked process. For example:

```
entity xyz is
  port (CLOCK, RESET : in std_logic; ...
...
process (CLOCK, RESET)
  begin
  if (RESET='1') then Q <= '0';
  elsif (CLOCK'event and CLOCK='1') then
    ...
```

For XC9000 designs, the fitter automatically uses the global set/reset pin (GSR) whenever possible. A global set/reset input signal may perform active-low clear or preset. For example:

```
process (CLOCK, PRESET)
  begin
  if (PRESET='0') then Q <= '1';
  elsif (CLOCK'event and CLOCK='1') then
    ...
```

A global set/reset inputs may also be used as an ordinary input signal to other logic elsewhere in the design.

If an input port signal passes through any logic function (other than an inverter) before it is used as an asynchronous clear or preset, the input cannot be routed to the flip-flops using the global set/reset

path. Instead, the flip-flop's clear / preset signal will be routed through the logic array. Routing a clear or preset through the logic array uses up one extra p-term for each flip-flop.

There is only one global set / reset pin on each XC9000 device. If you need to explicitly control the use of the global set / reset pin, you can instantiate the BUFGSR cell from the XC9000 library and pass your input port signal through it. For example:

```
U1: BUFGSR port map (O=>GLOBAL_RESET, I=>RESET);
process (CLOCK, GLOBAL_RESET)
  begin
 if (GLOBAL_RESET='1') then Q <= '0';
  elsif (CLOCK'event and CLOCK='1') then
    ...
```

Note: If a flip-flop has both a clear and preset condition and you assert both the clear and preset concurrently, its Q-output is unpredictable. This is because the fitter may arbitrarily invert the logic stored in a flip-flop to achieve better logic optimization. Individual clear and preset operations still produce the correct final logic state as dictated by the user design. Functional simulation does not accurately predict the ultimate behavior of the chip when clear and preset are asserted concurrently. Timing simulation, however, is performed after logic optimization and behaves exactly as the chip will when programmed.

Controlling Register Initial State

All registers in an EPLD device are initialized when the device is powered up. XC7000 devices also have a Master Reset pin that re-initializes the whole device. The initial state (preload value) of each register is programmable (except for IOB registers in XC7000 devices).

Unless otherwise specified in your design, each register in an XC9000 device will initialize to the zero (reset) state at power-up.

The initial states of registers in XC7000 devices are, by default, selected by the fitter based on the most efficient use of resources. If you want all registers in an XC7000 design to preload to zero by default, you can specify the "-preload" parameter on the xepld command line, as follows:

`xepld -p 7 -preload` *design_name*

The -preload parameter has no effect on XC7000 IOB registers (like the IFD cell) that you instantiate in your design. You can use the DC Shell initial state attribute (described below) to override the xepld preload parameter for selected registers.

Initial State Attribute

You can specify the preload states of selected registers in your design by setting the DC Shell initial state attribute. If you are using FPGA Compiler, enter the following in DC Shell:

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

where:

- *instance_name* is the name(s) of a cell instance(s) 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
```

The initial state attribute is ignored if it is applied to any cell in your design that is not a flip-flop.

Controlling Power Consumption

The power consumption of each macrocell in an EPLD device is programmable. The standard (default) setting consumes more power and produces shorter propagation delay. The low-power setting reduces power consumption for less speed-critical paths.

By default, all macrocells in the design will operate in standard power mode. You can change the global power setting to use the low power mode throughout the design by specifying the "lowpwr" parameter on the xepld command line as follows:

xepld -lowpwr *design_name*

Controlling Output Slew Rate

For all XC9000 devices and some XC7000 devices, each output is programmable to operate either at full speed or with limited slew rate. (Consult device data sheets for applicability.) Limiting the slew rate reduces output switching surges in the device. Slew rate control becomes important when your design uses a large number of outputs or you have transmission lines on your board which are sensitive to fast edge rates.

By default, all registers have limited (slow) slew rate. If you want to disable the slew rate limitation of a device output to increase its switching speed, use the "set_pad_type" command in DC Shell. For either FPGA Compiler or Design Compiler, enter the following in DC Shell:

set_pad_type -slewrate NONE {*port_list***}**

where port_list is a list of output ports that are to operate with fast output slew rate. The keyword "NONE" signifies that no slew rate limitation should be applied to the named output ports.

If you need to set the slew rate back to slow for some output ports, you can enter the following in DC Shell:

set_pad_type -slewrate HIGH {*port_list***}**

The keyword "HIGH" signifies that full slew rate limitation should be applied to the named output ports, thereby slowing output transitions.

Note: The set_pad_type command must be invoked before the insert_pads command in DC Shell.

Controlling the Pinout

When you first run the fitter before your pinout is committed, the software automatically selects pin locations for your I/O signals. Pin locations are selected which will give you the greatest flexibility to iterate your design without having to move any of the pins. Each time the fitter successfully implements your design, it creates a guide file (*design_name*.gyd), which contains all the resulting pinout information. After you commit your pinout, subsequent design iterations

cause the guide file to be read by the fitter and your committed pinout will be preserved.

We strongly recommend that you allow the software to automatically generate your initial pinout. Attempting to select your own initial pin preferences reduces the ability of the fitter to implement your design successfully the first time. It further reduces the amount of logic changes you could make after freezing your pinout.

Pin Freezing

If you have successfully fit a design into an EPLD device and you build a prototype containing the device, you will probably want to "freeze" the pinout. The next time you iterate that design, you should specify the "pinfreeze" option on the xepld command line, as follows:

`xepld -pinfreeze` *design_name*

The -pinfreeze parameter tells the fitter to read and obey the pinout from the guide file that was saved the last time the fitter completed. The fitter will not move any of the pins contained in the guide file, even if it prevents the design from successfully mapping.

The pin locations stored in the guide file are specified based on the top-level port names in your design. If you change the name of any of your ports, the corresponding pin will no longer be constrained to the location stored in the guide file. Renaming a port is a way you can relax the pinfreezing constraints on a design if your logic changes prevent the design from successfully fitting with your original pinout.

When you iterate your design while your pins are frozen, you are free to delete existing ports and/or add new ports. The fitter will automatically select the best locations for any new ports you add, after placing all the existing ports constrained by the guide file.

Note: If you iterate your design and your pinout is not yet committed (you haven't built a prototype containing the device), you should not specify the pinfreeze option. Instead, allow the software to redefine the pinout of your modified design. This will continue to give you the greatest flexibility to iterate your design again after you commit your pinout.

Pin Assignment

You can assign explicit locations for pins in your design using the DC Shell pad_location attribute. If you are using FPGA Compiler, enter the following in DC Shell:

set_attribute *port* **pad_location -type string** *pin_name*

where *port* is the name of the port being assigned.

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

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

For PC and PQ type packages, the *pin_name* takes the form "P*nn*" where *nn* is a number. For example, for the PC84 package, the valid range for *pin_name* is P1 through P84. For grid array type packages (PG and BG), the *pin_name* takes the form "*rc*", where *r* is the row letter and *c* is the column number.

When your design contains pad_location attributes, you should specify the target device type either using the xepld command's -p parameter or the DC Shell PART attribute (see Target Device Selection in this Chapter). The pad_location attributes are typically not compatible when retargeting a design between different package types, device types or device families.

The pad_location attributes are unconditional in that the software will not attempt to relocate a pin if it cannot achieve the specified assignment. If you specify a set of pad_location attributes that the fitter cannot satisfy, the fitter will terminate with an error.

The pad_location attributes override the pin assignments in the guide file if you specify the pinfreeze option. This allows you to make explicit changes to your committed pinout. If you override the guide file using pad_location attributes, the software will issue a warning.

If your objective is to preserve a previously created pinout, we recommend you use the pinfreeze feature instead of creating a set of pad_location attributes with the existing pin locations. The guide file saved from the previous design implementation contains additional information to help the fitter to successfully fit your modified design.

If you used pad_location attributes when compiling your netlist but you want to let the fitter automatically assign all I/O pins, you can specify the -ignoreloc parameter on the xepld command line:

`xepld -ignoreloc` *design_name*

The ignoreloc option allows you to temporarily ignore all the pad_location attributes in your netlist. This is useful if you want to test how your design fits a different target device without re-compiling your design.

Pin Assignment Precautions

You can apply the pad_location attribute to as many ports in your design as you like. However, each pin assignment further constrains the software making it more difficult for the fitter to automatically allocate logic and I/O resources for the remaining I/O signals in your design.

When you manually assign output and I/O pins, you force the software to place associated logic functions into specific macrocells and specific function blocks. If the associated logic does not exceed the available function block resources (macrocells, product terms, and FastCONNECT inputs), the logic is mapped into the macrocell and the design will route in the FastCONNECT.

It is usually best to allow the fitter to automatically assign most or all of the pins based on the most efficient placement of logic in the device. The fitter automatically establishes a pinout which best allows for future design iterations without pin relocation. Any manual pin assignments you make in your design may not allow as much tolerance for changes in the logic associated with those pins, and in the logic physically mapped to nearby locations in the device.

If you are assigning pin locations to ports used as clocks, asynchronous set/reset, or output enable in your design, you should assign them to the GCK, GSR and GTS pins on the device if you want to take advantage of these global resources. The fitter will still automatically assign other clock, set/reset and output enable inputs to remaining GCK, GSR and GTS pins if available.

Controlling Logic Optimization

When you create combinational logic functions, the software

attempts to collapse as much of the logic as possible into the smallest number of EPLD macrocells. Any combinational logic function bounded between device I/O pins and flip-flops is subject to complete or partial collapsing. Collapsing the logic improves the speed of the logic path and can also reduce the amount of logic resources (macrocells, p-terms and FastCONNECT inputs) required to implement the function.

The process of collapsing logic into other logic functions is called "logic optimization".

Collapsing Product Term Limit

When a larger combinational logic function consisting of several levels of AND-OR logic is completely collapsed (flattened), the number of product terms required to implement the function may grow considerably. By default, the fitter limits the number of p-terms used as a result of collapsing to 15 for XC9000 devices, and 17 for XC7000 devices. If the collapsing of a logic level results in a logic function consisting of more than 15 or 17 p-terms (after Boolean reduction), then the collapsing of that logic level is not performed and the function will be implemented using two or more levels of AND-OR logic.

The overall extent to which logic is collapsed throughout the design can be controlled using the "-pterms" parameter on the xepld command line:

`xepld -pterms` *nn design_name*

where *nn* is the maximum allowable number of p-terms that can be used to implement a logic function after collapsing. (The default *nn*=15 for XC9000 devices, and 17 for XC7000 devices)

If you find that the path delay of a larger, multi-level logic function is not satisfactory, try increasing the pterms parameter to allow the larger functions to be flattened further. For example, you may try increasing the p-term limit to 25 when rerunning the fitter, as shown:

```
xepld -pterms 25 design1
```

For XC9000 designs, the fitter report (*design_name*.rpt) indicates the number of p-terms used for each logic function. You should see these numbers increase as you raise the pterms limit, until the design is

fully flattened. At the same time, you'll see the internal combinational nodes eliminated until none remain.

Preventing Collapsing of a Logic Node

Flattening typically increases the overall amount of p-term resources required to implement the design. Some designs which fit the target device initially may fail to fit if flattened too much. Other designs can be flattened completely and still fit. If you cannot increase the xepld pterms parameter enough to sufficiently flatten a critical path and still fit the target device, you may try applying logic optimization control at specific nodes in your design.

A special cell is provided in the XC7000 and XC9000 libraries, named OPT_OFF, which is used to apply a logic optimization constraint to any signal connected to it. The OPT_OFF cell has one port, which is an input named "I". By instantiating an OPT_OFF cell and connecting its "I" port to a signal in the middle of a combinational logic function, you can prevent that signal from being collapsed.That is, you prevent the cell that drives the signal from being collapsed forward into any of its fanouts. The OPT_OFF cell is instantiated as follows:

label`: OPT_OFF port map (`*signal_name*`);`

Note: The connected signal does not actually pass through the OPT_OFF cell. The OPT_OFF cell only applies an attribute to the signal when the fitter reads the netlist.

In the following example, an OPT_OFF cell is used to prevent the logic for an address decoder from being collapsed into the select input of a 16-bit multiplexer:

```
component OPT_OFF port (I : in std_logic);
end component;
...
DECODE1 <= '1' when (ADDR_BUS = ADDR_1) else '0';
DATA_BUS <= A_BUS(0 to 15) when (DECODE1='1') else B_BUS;
U1: OPT_OFF port map (DECODE1);
```

By preventing logic optimization, the fitter will not duplicate the logic of the address decoder in each bit of the multiplexer.

You can use OPT_OFF to break logic chains in non-speed-critical paths and prevent those functions from using too many p-terms. If you set the pterms parameter too high and your design no longer fits, try using OPT_OFF to reduce the size of selected non-critical paths.

Controlling Timing Paths

There are two mechanisms that can improve the timing of your design:

- Global Timing Optimization
- XACT Performance (timing constraints)

Timing Optimization

By default, the fitter performs global timing optimization on logic paths in your design. Timing optimization will shorten your critical paths as much as it can. In general, timing optimization optimizes logic and allocates the fastest available resources for the longest paths in your design, assuming all paths are equally critical. In some cases, the fitter trades off density for a speed advantage.

If you do not want the fitter to perform global timing optimization, you can specify the "-notiming" parameter on the xepld command line as follows:

`xepld -notiming` *design_name*

XACT Performance

The Synopsys FPGA Compiler generates timing specifications that it writes into the XNF netlist for the XEPLD fitter. You can enter timing constraints in the Design Analyzer, the DC Shell command window, or a DC Shell script file. FPGA Compiler does not use your timing constraints to optimize your logic or infer library cells during compilation. All timing optimization is performed by the XEPLD fitter after reading your timing specifications from the XNF netlist.

The following path types can be specified:

Pad-to-pad delay	Input port to an output port
Register setup time	Input port to the data pin of a flip-flop
Register-to-register	Output pin of a flip-flop to the data pin of a flip-flop
Clock-to-output delay	Output pin of a flip-flop to an output port

This section lists the Synopsys commands that enable you to create timing specifications for your Xilinx EPLD designs. Examples are provided to demonstrate how implemented Synopsys commands are passed to the fitter. For a complete listing of all options and arguments for each command, refer to the Synopsys documentation.

Create Specifications for Input Ports and Clock Net

You can use the following commands to place a timing specification on all input ports and a specified clock net.

- Create Clock — This command creates a setup time specification on each input port, and a cycle time specification on the specified clock signal as follows:

  create_clock {*clock***} -period** *delay*

  where *clock* is the name of the clock signal and *delay* is the clock cycle time in nanoseconds.

- Max Period — This command creates a setup time specification on each input port, and a cycle time specification on the specified clock signal as follows:

  max_period *delay* **{***clock***}**

  where *clock* is the name of the clock signal and *delay* is the clock cycle time in nanoseconds.

Create Specifications for Input and Output Ports

The Set Max Delay command creates a combinational pad-to-pad delay specification on each input and output port. You can also use this command to affect register setup times, cycle times, and clock-to-output delay specifications if you list the flip-flop cell names with either the –from or –to options.

set_max_delay *delay* **-from {***input_port***} -to {***output_port***}**

Create Tighter Constraints on Output Ports

The Set Output Delay command creates clock-to-output delay specifications using the values from the Create Clock or Max Period constraints and creates tighter constraints for the output ports as follows:

set_output_delay *delay* **–clock {***clock***} {***output_port***}**

This command also changes the values of pad-to-pad delay specifications created by the Set Max Delay command.

Create Tighter Constraints on Input Ports

The Set Input Delay command changes the values of register setup time specifications created by the Create Clock or Set Max Delay commands and creates tighter constraints on all input ports as follows:

set_input_delay *delay* **–clock {***clock***} {***input_port***}**

This command also changes the values of pad-to-pad delay specifications created by the Set Max Delay command.

Prevent Specifications on Indicated Paths

The Set False Path command prevents the FPGA Compiler from generating timing specifications for specified paths as follows:

set_false_path –from {*input_port***}**

Create Clocks on All Input Ports

The Derive Clocks command automatically creates clocks on all input ports that source clock pins on flip-flops. If performing timing optimization, set constraints on all clocks in your design. You can run the Derive Clocks command to make sure that you have not missed any clocks in your design as follows:

derive_clocks

Disabling Timing Specifications

If you used timing constraints when compiling your netlist but want to run the fitter without using your timing specifications, you can temporarily disable XACT Performance by specifying the “-ignorets” parameter on the xepld command line as follows:

xepld -ignorets *design_name*

Reducing Levels of Logic

The XC9500 architecture, like most CPLD devices, is organized as a large, variable-sized combinational logic resource (the AND-array and XOR gate) followed by a register. If you place combinational logic before a register in your design, the fitter maps the logic and register into the same macrocell. The output of the register is then directly available at an output pin of the device. If, however, you place logic between the output of a register and the device output pin, a separate macrocell must used to perform the logic, decreasing both the speed and density of your design. The following example shows two functionally similar styles for designing a selectable divide-by-2 or divide-by-4 counter. The first design style is inefficient for CPLD architectures; the second example is more efficient.

```
-- Inefficient style for CPLDs:
process (CLOCK)
  begin
  if (CLOCK'event and CLOCK='1') then
    DIV2 <= not DIV2;
    DIV4 <= DIV4 xor DIV2;
  end if;
end process;
DIV_OUT <= DIV2 when (SELECT='0') else DIV4;

-- More efficient style for CPLDs:
process (CLOCK)
  begin
  if (CLOCK'event and CLOCK='1') then
    DIV2 <= not DIV2;
    DIV_OUT <= (DIV_OUT xor DIV2) when (SELECT='1')
      else (not DIV_OUT);
  end if;
end process;
```

XC7000 Input Pad Registers

The XC7000 architecture allows you to implement registers within function block macrocells and within input pads. In XC9000 devices, all registers are implemented in macrocells. This section shows you how to assign specific register types in an XC7000 design.

The fitter uses input pad registers to implement flip-flops 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 data and clock inputs must come directly from input ports or I/O ports. The clock input signal must not be used for anything other than register clocking, and the data input signal must not be used as input to any other function.

Preventing Register Optimization

To prevent the fitter from automatically assigning any registers to the input pads, specify the "-noifd" parameter on the xepld command line, as follows:

`xepld -p 7 -noifd` *design_name*

Note: The -noifd parameter does not prevent you from instantiating explicit input pad register cells (like IFD) in your design.

Using Input Pad Registers

If you want to assign a specific register in your design to an input pad, instantiate the IFD cell from the XC7000 library. The Clock input must be driven by a BUFG cell (global FastClk). Except for signals declared as FastInputs, the D input signal must not be used as input to any other function in the design.

Chapter 3

Compiling and Fitting Your Design

The Synopsys interface 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 with regard to implementation efficiency. FPGA Compiler does, however, support certain Xilinx-specific attributes used to control design implementation, such as timing constraints and register initial states, that are not supported by Design Compiler. 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 using the VSS simulator before trying to synthesize it. See the "Simulating Your Design" chapter for information on functional simulation.

Compiling a Synopsys EPLD Design

This section describes the procedure for compiling a complete EPLD design based on VHDL or HDL. If you are preparing a VHDL/HDL-based module for inclusion in a schematic-based design, refer to the section "Compiling Behavioral Modules for Schematics" later in this chapter.

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. Unless otherwise specified, this procedure applies to both XC7000 and XC9000 designs.

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 vhdl` *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 vhdl 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 described in Chapter 2.

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

Step 3 — Elaborating the Design

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

`elaborate` *entity_name*

where *entity_name* is the name of your top-level entity in your design.

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 or XC9000 target library.

Step 4 — Compiling Your Design

When you compile your design, the Synopsys synthesizer uses the components in the Xilinx XC7000 or XC9000 technology library to create an actual implementation of your design. The library used during compilation is defined by the DC Shell target_library variable, typically specified in your .synopsys_dc.setup file.

To synthesize your design based on target EPLD 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 — 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 xepld command line).
- Register initial states.
- Pin assignments.
- Output slew rate

- Timing constraints (for XACT Performance)

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

```
set_attribute scan part -type string 95108-7pc84
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.

Step 6 — 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 for which you instantiated 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 infers the appropriate I/O buffer cells into your design:

```
insert_pads
```

Note: If you want to control output slew rate, the DC Shell **set_pad_type** command must be invoked before the insert_pads command.

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 compiled 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 required processing in DC Shell. 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
```

Note: None of the Synopsys timing or area analysis reports are useful at this time because the EPLD technology libraries do 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 section.

Fitting Your Design

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

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

xepld [*options***]** *design_name*

Invoking the xepld command with no parameters produces a listing of all available command-line options.

The *design_name* is the name of the netlist file produced by the Synopsys compiler, without path qualifiers, and either with or without extension.

If *design_name* is specified without extension, the xepld command automatically searches for and reads netlists with file extensions .sxnf and .sedif as produced by Synopsys FPGA Compiler and Design Compiler.

If you do not specify any optional parameters, the fitter assumes you are running an XC9000 design and automatically selects a device from the XC9000 family which fits your design (if possible). You can designate an XC7000 design by simply including the parameter "-p 7" on the xepld command line:

xepld -p 7 *design_name*

When you specify "-p 7", the fitter automatically selects a device from the XC7000 family which fits your design (if possible).

The xepld command performs the following functions:

- Reads the netlist file (*design_name*`.sxnf` or *design_name*`.sedif`) produced by the Synopsys compiler.
- Minimizes and collapses the combinational logic of your design so that it requires the least number of macrocell and product term resources.

- Partitions and maps your design to fit within the architecture of the EPLD, optionally selecting the target device.
- Creates a device programming file (*design_name*`.prg` for XC7000 or *design_name*`.jed` for XC9000).
- Creates a fitter report (*design_name*.rpt for XC9000 or *design_name*.res for XC7000) that shows you information such as the type and quantity of device resources used.
- Creates a Static Timing Report (*design_name*.tim) that shows the calculated worst-case timing for all signal paths in your design.
- Creates a guide file (*design_name*.gyd) that is used to lock signal names to device pins, allowing you to keep the device pinouts during design iterations.
- Creates a timing simulation netlist file that can be translated into structural VHDL for the Synopsys VSS simulator.

Whenever the xepld command is invoked, it copies any existing fitter report file (.rpt or .res), timing report file (.tim), guide file (.gyd), pinlist report (.pin) and programming file (.jed or .prg) to the backup directory.

XEPLD Command Parameters

The [*options*] field of the xepld command represents an optional list of one or more command-line parameters. The following are the xepld command-line parameters that apply to synthesis design entry:

- **-detail** — produces a detailed path timing report (*design_name*.tim) instead of the default summary report.
- **-ignoreloc** — temporarily ignores all pad_location attributes in the design, allowing the fitter to assign the locations of all I/O pins.
- **-ignorets** — temporarily ignores all timing specification attributes in the design.
- **-lowpwr** — uses the low-power mode for all macrocells in the design (default is standard power).
- **-nogck** — prevents the fitter from optimizing inputs used as clocks onto the device's global clock input pins (GCK or FCLK).

- **-nogts** — prevents the fitter from optimizing inputs used for 3-state output enable control onto the device's global 3-state control input pins (GTS or FOE).
- **-noifd** — prevents the fitter from optimizing registers in your design into input-pad flip-flops in XC7000 devices.
- **-nota** — bypasses the timing analyzer so that no static timing report is generated.
- **-notiming** — inhibits the default global timing optimization performed by the fitter; only paths with timing specifications are optimized to improve timing.
- **-p** *part_type* — specifies the target EPLD device type or set of devices from which to choose (default is automatic device selection from the XC9000 family); where *part_type* can be:
  - "*ddddd-sspppp*" — where *ddddd* is the device code (such as 95108), *ss* is the speed grade, *pppp* is the package code (such as PQ160), and an asterisk (*) can be used as a wildcard string (quotes required around *part_type* when asterisk is used).
  - "*ddddd-sspppp,ddddd-sspppp ...*" — a list of valid part-type specifications as defined above (quotes required).
  - indesign — signifies that the target device is specified by a PART attribute in the netlist.
- **-pinfreeze** — uses the guide file (*design_name*.gyd) from the last successful invocation of the fitter to reproduce the same pin locations (default is automatic pin assignment).
- **-preload** — inhibits preload optimization in XC7000 designs so that all registers are initialized to the preload states defined in the Library appendix (default allows the fitter to change unspecified register preload states to improve XC7000 mapping efficiency).
- **-pterms** *nn* — sets the limit to *nn* for the number of product terms allowed as a result of collapsing (default=15 for XC9000 and 17 for XC7000).
- **-s** *signature* — specifies the user signature string to be programmed into the device for identification purposes (default is the design name).

Compiling Behavioral Modules for Schematics

If you are developing a schematic-based design using some other schematic entry tool (such as Viewlogic), you can include module symbols in your schematic that are functionally defined using Synopsys VHDL or HDL. These are called “behavioral modules”.

This section describes how to prepare a synthesis-based behavioral module using Synopsys FPGA Compiler. Behavioral modules are represented by custom symbols in the schematic design. In general, the names of the pins on your behavioral module symbol should match the names of the top-level entity ports in your Synopsys source file. Refer to the *XC9500 Schematic Design Guide* for information on how to include the behavioral module symbol in your schematic design.

The procedure for compiling a behavioral module is similar to the procedure for compiling a complete EPLD design, as described earlier in this chapter. For behavioral modules, however, you do not specify device I/O pads; you would therefore omit the set_port_is_pad and insert_pads commands. Also, many of the DC Shell attributes, such as PART and pad_location, are not applicable to behavioral modules.

Behavioral modules are compiled and written as XNF netlists by performing the following steps:

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 Module

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

`analyze -format vhdl` *module_name*`.vhd`

or, for Verilog HDL modules:

`analyze -format verilog` *module_name*`.v`

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 described in Chapter 2.

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

Step 3 — Elaborating the Module

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

`elaborate` *entity_name*

where *entity_name* is the name of your top-level entity in your module.

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

You are now ready to compile your module using the XC7000 or XC9000 target library.

Step 4 — Compiling Your Module

When you compile your module, the Synopsys synthesizer uses the components in the Xilinx XC7000 or XC9000 technology library to create an actual implementation of your module. The library used during compilation is defined by the DC Shell target_library variable, typically specified in your .synopsys_dc.setup file.

To synthesize your module based on target EPLD 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 — Specifying Attributes

The only attribute that you may set for behavioral modules is:

- Register initial states.

See the "Attributes" appendix for details.

Step 6 — Writing the Netlist

Write your synthesized module file in XNF netlist format by entering the following Synopsys command:

`write -format xnf -hierarchy -output` *module_name*`.xnf`

where:

- **`-format xnf`** specifies the XNF file format.
- **`-hierarchy`** specifies that all levels of the module hierarchy are to be written.
- **`-output`** *module_name*`.xnf` specifies your output file name, which should be the same as your source file name, with the extension `.xnf`.

The XNF file produced by FPGA Compiler will be read when the fitter finds the behavioral module symbol in your schematic design.

Chapter 4

Simulating Your Design

This software supports both functional and timing simulation of VHDL designs using the VSS simulator. This package also provides a Verilog library and interface supporting timing simulation using the Cadence Verilog 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 should 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 source information necessary for timing simulation.
5. Prepare the timing model. The **xepldsim** command prepares the timing model of your design for simulation.

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 library.

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

All registers in Xilinx EPLDs are initialized when power is applied. XC7000 devices also have a Master Reset pin that re-initializes the whole device when pulsed. You must perform the necessary steps to initialize the registers in your design at beginning of timing simulation for proper simulation results.

The following sections show you how to set up your design to perform register initialization for both functional and timing simulation.

Preparing for Timing Simulation

When you generate your timing simulation model, **`xepldsim`** 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 include the MRESET signal in your test bench file for functional and timing simulation, you must also include MRESET in your port declarations in your source design file as follows:

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

MRESET is not used anywhere else in your design. It will be ignored 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 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 because the initial values are not written into the netlist.

Note: For XC7000 designs, 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 4-1. 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 **xepldsim** (after **xepld**) 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.

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 4-1 shows a typical configuration declaration in a test bench file. If the test bench is always used to simulate the design source file, the design does not need its own configuration declaration.

```
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 4-1 Simulation Test Bench — SCAN Design

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

Functional Simulation Using VSS

Functional simulation is used to debug your logic before fitting your design into an EPLD. The Xilinx EPLD Synopsys Interface fully supports functional simulation using the Synopsys VSS simulator, including all instantiated cells from the XC7000 and XC9000 libraries.

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 (**vhdldbx**).

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:

   vhdldbx

   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 vhdldbx selector window appears as shown in Figure 4-2.

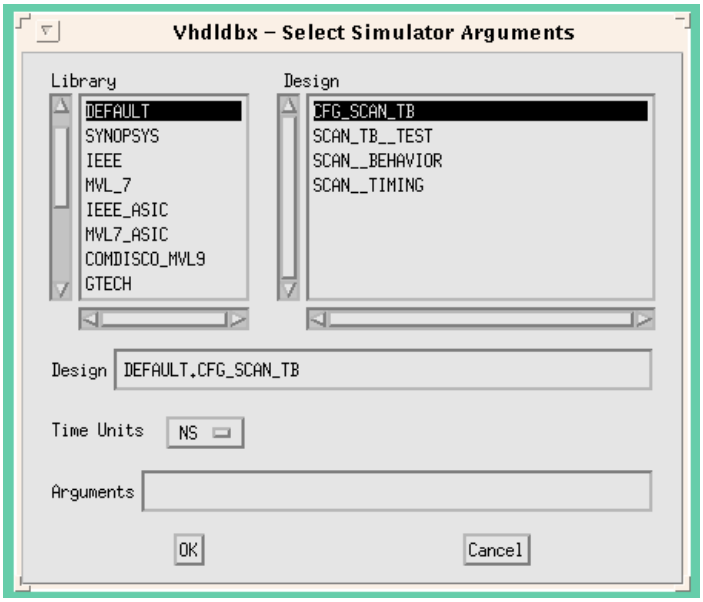


Figure 4-2 Selector Window (vhdldbx)

After you click OK, the vhdldbx user interface window appears as shown in Figure 4-3.

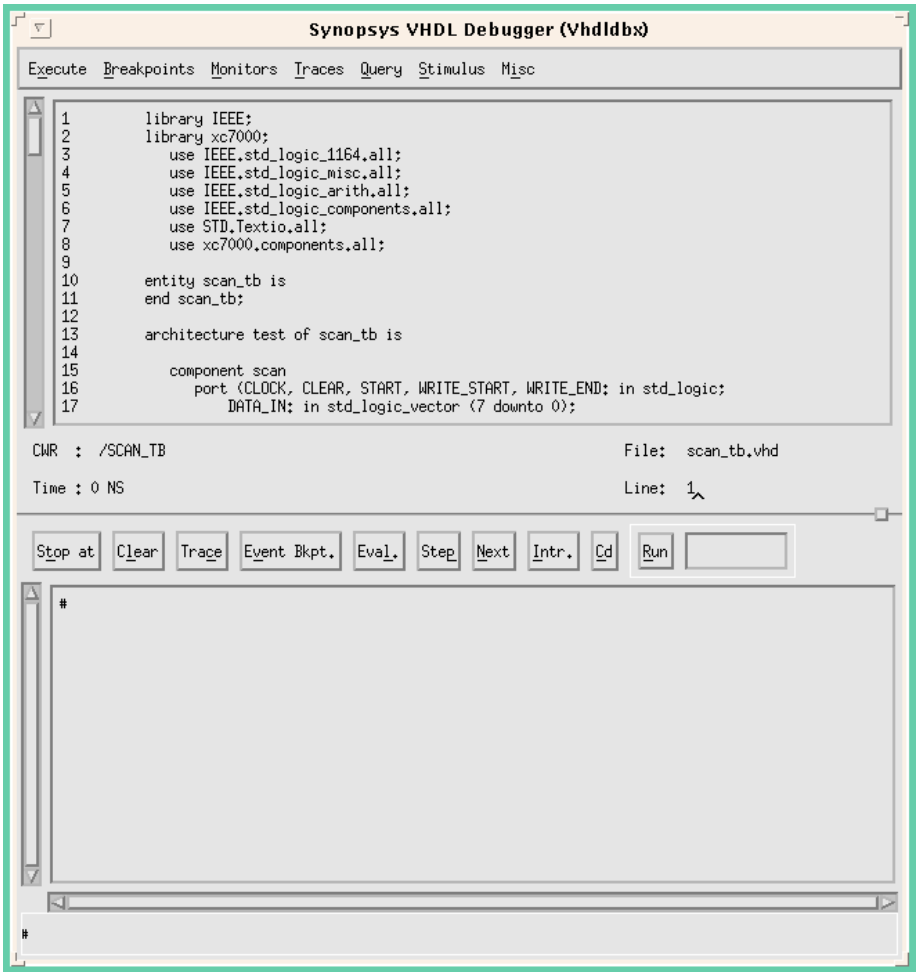


Figure 4-3 User Interface Window (vhdldbx)

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 vhdldbx 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 4-4

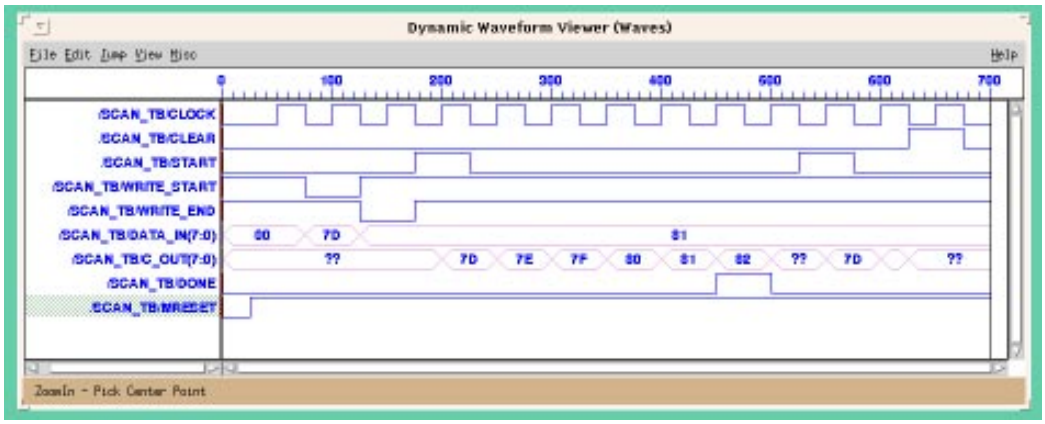


Figure 4-4 Functional Simulation Waveforms — SCAN Design

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 or XC9000 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 or XC9000 library, and create a netlist.
3. Run the Xilinx fitter, using the **`xepld`** command to process the netlist.

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

Examine the appropriate fitter report files to verify that the fitter completed successfully. 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 Simulation Model

The xepldsim command prepares timing simulation models file for the Synopsys VSS simulator and the Cadence Verilog simulator. The xepldsim command translates the timing simulation netlist file (*design_name*_tim.xnf) produced by the xepld command into the required simulation output file(s).

Invoke the xepldsim command on the UNIX command line as follows:

`xepldsim -`*target design_name*

Invoking the xepldsim command with no parameters produces a listing of all available command-line options.

The *design_name* is the name of the design as specified when running the xepld command, without path qualifiers and without extension.

If you specify **`-vss`** as the target parameter, **`xepldsim`** produces a structural VHDL file (*design_name*_vss.vhd) and an SDF-formatted timing back-annotation file (*design_name*_vss.sdf), based on the *design_name*_tim.xnf netlist, for use with the Synopsys VSS simulator. A procedure for using the VSS simulator is described below.

If you specify **`-verilog`** as the target parameter, **`xepldsim`**

produces a structural Verilog HDL file (*design_name*_**tim.v**) and and SDF-formatted timing back-annotation file (*design_name*_**tim.sdf**), based on the *design_name*_**tim.xnf** netlist, for use with the Cadence Verilog simulator. Consult the Cadence documentation for information on using the Verilog simulator.

Note: When the fitter processes your design, some of your original nodes may be removed or replaced due to logic optimization. Such nodes cannot be viewed or stimulated during timing simulation. All of the device I/O port signals are always maintained.

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 **xepldsim** command cannot regenerate a timing model complete with your original bus port declarations. Therefore, it 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, are 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, you can perform timing simulation using the same test bench and chip interface as used for functional simulation.

Timing Simulation Using VSS

If you prepared your test bench properly, you can use the same test bench for timing simulation as used for functional simulation. By using the same test bench you can easily verify that the functionality of the device after mapping matches the functionality of your source design. You also eliminate any risk of errors from accidental differences between the test bench files.

1. Analyze your source design file to 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 **xepldsim**:

   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:

   vhdldbx

   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 back-annotated 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 **vhdldbx** 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 back-annotated timing is applied:

  `-sdf_top` *chip_instance_name*

  For example:

  ```
  -sdf_top /scan_tb/UUT
  ```

  All back-annotated 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 **`vhdldbx`** (as shown in Figure 4-5), or on the UNIX command line as you invoke **`vhdldbx`**.

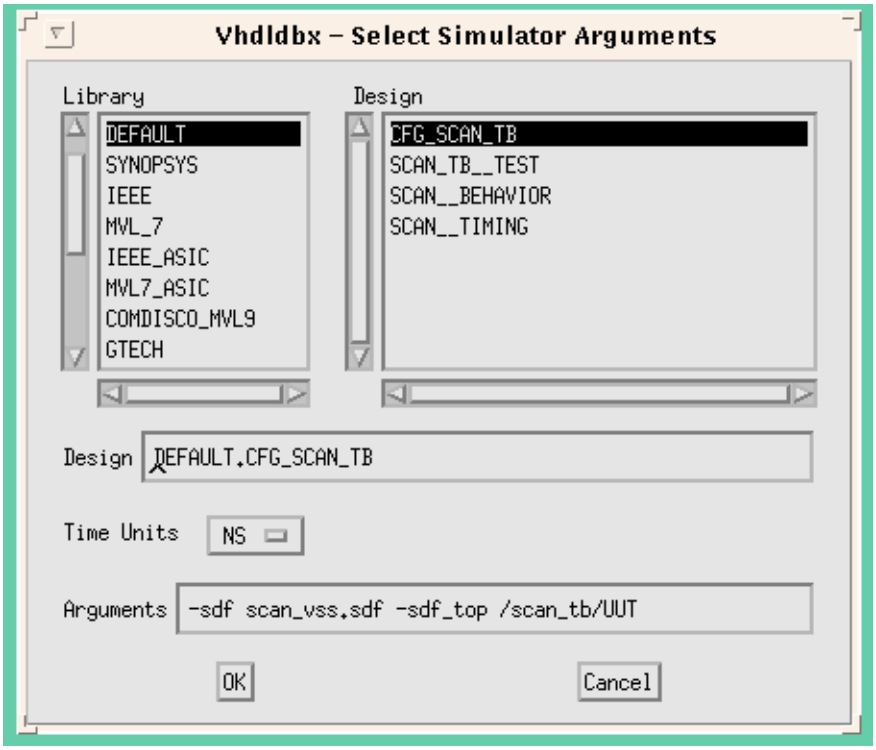


Figure 4-5 Selector Window with Timing Backannotation Parameters Entered (vhdldbx)

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:

`vhdldbx -sdf` *design_name_vss*`.sdf -sdf_top \`
chip_instance_name configuration_name

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

```
vhdldbx -sdf scan_vss.sdf -sdf_top \
  /scan_tb/UUT CFG_SCAN_TB
```

Now you can run the same simulation vectors for timing simulation

as you ran for functional simulation. However, in timing simulation, the registers are set to their initial states in response to the active-low pulse on the MRESET input.

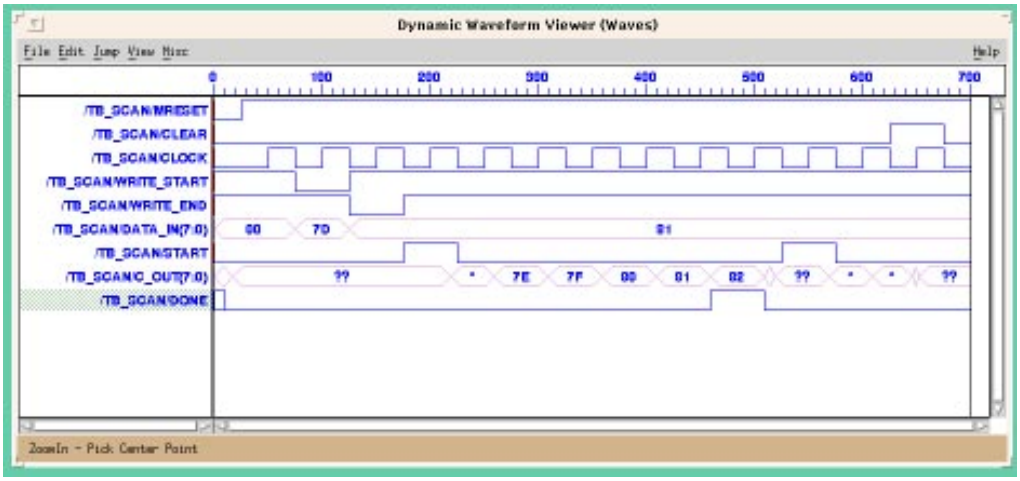


Figure 4-6 Timing Simulation Waveforms — SCAN Design

Appendix A

Library Component Specifications

This appendix describes each of the components (cells) in the Xilinx XC7000 and XC9000 synthesis libraries which are summarized in the following table.

				Devices used with		
Component Name	**Component Description**	**Scalable**	**Inferable**	**7318 7336**	**7354 7372 73108 73144**	**9500 (all)**
ACC	Adder / Subtracter / Accumulator	X			X	
ADD	Adder	X	X		X	
ADSU	Adder / Subtracter	X	X		X	
ADSUR	Adder / Subtracter with Registered Outputs	X			X	
AND2-AND8	AND Gates		X	X	X	X
BUF	Buffer			X	X	X
BUFCE	Clock Enable Inp. Buff. for Input Pad Reg.				X	
BUFE	3-State Buffer		X	X	X	X
BUFFOE	Fast Output Enable Input Buffer			X	X	
BUFGSR	Global set / reset input buffer					X
BUFGTS	Global 3-state control input buffer					X
BUFG	Global clock (FastCLK) Input Buffer			X	X	X
CBX1	Up / Down Counter with Asynchronous Clear	X		X	X	
CBX2	Up / Down Counter with Asynchronous Reset	X		X	X	
COMPEQ	Equal-To Comparator	X		X	X	
COMPLE_TC	Less-Than-Or-Equal Comparator, 2's Comp.	X	X		X	
COMPLE_US	Less-Than-Or-Equal Comparator, Unsigned	X	X		X	
COMPLT_TC	Less-Than Comparator, 2's Complement	X	X		X	
COMPLT_US	Less-Than Comparator, Unsigned	X	X		X	
COMPNE	Not-Equal Comparator	X		X	X	
DEC	Decrementor	X	X	X	X	
FDCP	Edge-Triggered D-Type Flip-Flop with Asynchronous Clear and Preset		X	X	X	X
FDCPE	Edge-Triggered D-Type Flip-Flop with Clock Enable, Async. Clear and Preset				X	X

				Devices used with		
Component Name	**Component Description**	**Scalable**	**Inferable**	**7318 7336**	**7354 7372 73108 73144**	**9500 (all)**
FDPC	Edge-Triggered D-Type Flip-Flop with Asynchronous Clear and Preset		X	X	X	X
IBUF	Input Buffer		X	X	X	X
IFD	Input Pad Register				X	
IFDX1	Input Pad Register with Clock Enable				X	
ILD	Input Pad Latch				X	
INC	Incrementer	X	X	X	X	
INV	Inverter		X	X	X	X
IOBUFE	Bi-Directional I/O Buffer		X	X	X	X
IOBUFE_F	Bidirectional I/O Buffer--fast slew rate		X		note 1	X
IOBUFE_S	Bidirectional I/O Buffer--slow slew rate		X		note 1	X
LD	D-Type Latch				X	X
OBUF	Output Buffer		X	X	X	X
OBUF_F	Output Buffer--fast slew rate		X		note 1	X
OBUF_S	Output Buffer--slow slew rate		X		note 1	X
OBUFE	3-State Output Buffer		X	X	X	X
OBUFE_F	3-state Output Buffer--fast slew rate		X		note 1	X
OBUFE_S	3-state Output Buffer--slow slew rate		X		note 1	X
OR2-OR8	OR Gates		X	X	X	X
SUBT	Subtracter	X	X		X	
XOR2-XOR8	XOR Gates		X	X	X	X

Note 1. Consult XC7000 device data sheets for applicability of output slew rate control.

ACC

ACC is an adder / subtracter / accumulator. (XC7000 only)

Inferencing

The synthesizer does not use this component by inference.

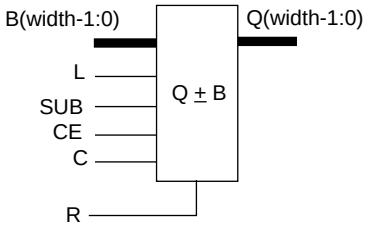
Component Instantiation

```
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. (XC7000 only)

Inferencing

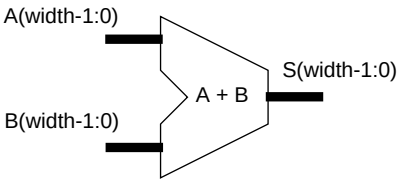
```
sum_signed <= in1_signed + in2_signed;
```

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/subtracter. ADSU is bound to the "+" and "-" operators. (XC7000 only)

Inferencing

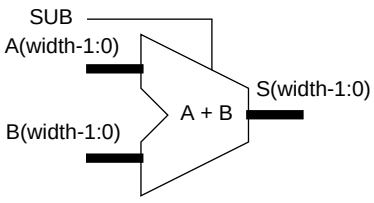
```
if (sub_ctl = '0') then
     sum_signed <= in1_signed + in2_signed;
else
     sum_signed <= in1_signed - in2_signed;
end if;
```

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/subtracter. (XC7000 only)

Inferencing

The synthesizer does not use this component by inference.

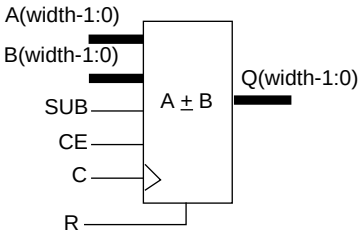
Component Instantiation

```
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

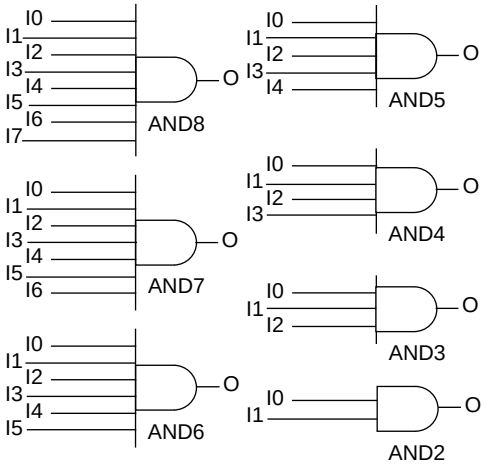
The synthesizer uses these components when creating functions that require AND gates.

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 non-inverting buffer.

Inferencing

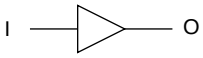
The synthesizer does not use this component by inference.

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 XC7000 input pad registers. BUFCE may only be used to drive the CE input of IFDX1 components.

Inferencing

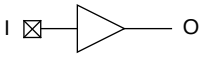
The synthesizer does not use this component by inference.

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, with active-high enable.

Inferencing

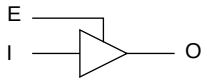
The synthesizer uses these components when creating functions that require 3-state buffers that drive internal signals.

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 OBUFE and IOBUFE components

Inferencing

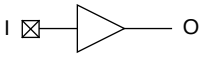
The synthesizer does not use this component by inference.

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 clock signal (GCK or FastCLK).

In XC7000 designs, BUFG can only drive register clock inputs (including IFD and the G input of ILD components). It cannot drive the LD component, or any other logic functions in the design. In XC9000 designs, BUFG signals may be used active-high or active-low (inverted), and for any other logic functions in the design.

Inferencing

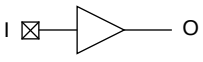
The synthesizer does not use this component by inference.

Component Instantiation

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

Truth Table and Logic Symbol

I	O
0	0
1	1



BUFGSR

BUFGSR is an input buffer used to drive the Global set/reset signal in XC9000 designs. BUFGSR signals can drive the CLR and PRE inputs of FDCP components (global set/reset), and any other logic functions in the design.

Inferencing

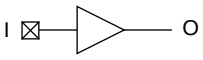
The synthesizer does not use this component by inference.

Component Instantiation

```
U1: BUFGSR port map (O=>global_sr, I=>in_port);
```

Truth Table and Logic Symbol

I	O
0	0
1	1



BUFGTS

BUFGTS is a an input buffer used to drive the global 3-state control signal (GTS) for XC9000. BUFGTS may be used to drive the E input of OBUFE and IOBUFE components (global 3-state control), and any other logic functions in the design.

Inferencing

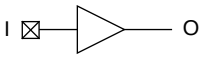
The synthesizer does not use this component by inference.

Component Instantiation

```
U1: BUFGTS port map (O=>global_oe, 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. (XC7000 only)

Inferencing

The synthesizer does not use this component by inference.

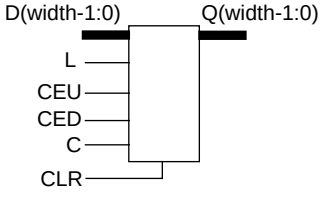
Component Instantiation

```
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. (XC7000 only)

Inferencing

The synthesizer does not use this component by inference.

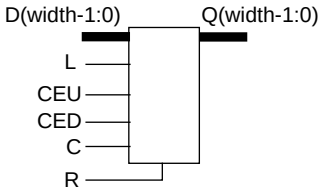
Component Instantiation

```
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. (XC7000 only)

Inferencing

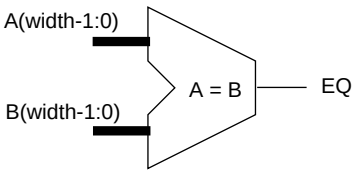
The synthesizer does not use this component by inference.

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. (XC7000 only)

Inferencing

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

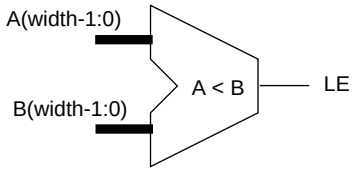
Component Instantiation

```
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. (XC7000 only)

Inferencing

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

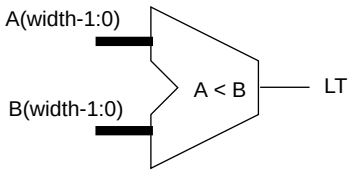
Component Instantiation

```
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. (XC7000 only)

Inferencing

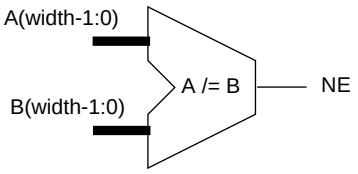
The synthesizer does not use this component by inference.

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. (XC7000 only)

Inferencing

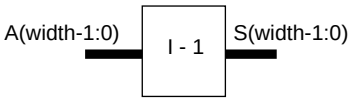
```
sum_signed <= in_signed - 1;
```

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

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

Inferencing

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

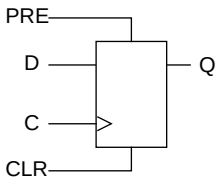
Component Instantiation

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

Truth Table and Logic Symbol

CLR	PRE	C	Q*
1	X	X	0
0	1	X	1
0	0	↑	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.

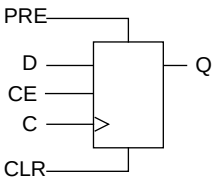
Component Instantiation

```
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 or FDCP for all functions that require D-type registers.

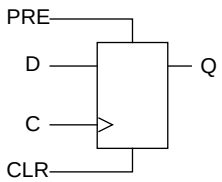
Component Instantiation

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

Truth Table and Logic Symbol

CLR	PRE	C	Q*
X	1	X	1
1	0	X	0
0	0	↑	D

* The initial state is “0”.



IBUF

IBUF is an input buffer.

Inferencing

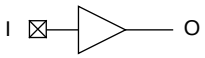
The synthesizer uses these components to receive inputs from device pins.

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 XC7000 Input Blocks.

Inferencing

The synthesizer does not use this component by inference.

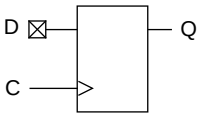
Component Instantiation

```
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 “1”.



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 XC7000 Input Blocks.

Inferencing

The synthesizer does not use this component by inference.

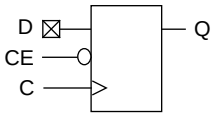
Component Instantiation

```
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 “1”.



ILD

`ILD` is a D-type transparent latch available in the XC7000 Input Block. The `G` input must be driven by a `BUFG` buffer.

Inferencing

The synthesizer does not use this component by inference.

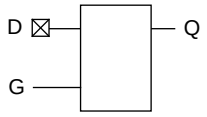
Component Instantiation

```
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 “1”.



INC

INC is an Incrementer. It is bound to the “+1” operator. (XC7000 only)

Inferencing

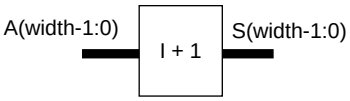
```
sum_signed <= in_signed + 1;
```

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

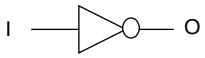
The synthesizer uses this component for signal inversion.

Component Instantiation

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

Truth Table and Logic Symbol

I	O
0	1
1	0



IOBUFE, IOBUFE_F, IOBUFE_S

`IOBUFE` is a non-inverting 3-state I/O buffer with active-high enable. In devices that support programmable output slew rate control, slow output slew rate is performed.

`IOBUFE_F` is an I/O buffer with fast output slew rate.

`IOBUFE_S` is an I/O buffer with slow output slew rate.

Inferencing

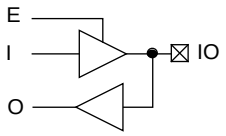
The synthesizer uses these components to create bidirectional I/O.

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



LD

LD is a D-type latch.

Note: In XC7000 designs 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

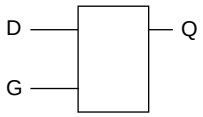
Component Instantiation

```
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_F, OBUF_S

OBUF is an output buffer. In devices that support programmable output slew rate control, slow output slew rate is performed.

OBUF_F is an output buffer with fast output slew rate.

OBUF_S is an output buffer with slow output slew rate.

Inferencing

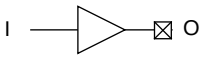
The synthesizer uses this component when creating external outputs to device pins.

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_F, OBUFE_S

OBUFE is a 3-state output buffer with active-high enable. In devices that support programmable output slew rate control, slow output slew rate is performed.

OBUFE_F is a 3-state output buffer with fast output slew rate.

OBUFE_S is a 3-state output buffer with slow output slew rate.

Inferencing

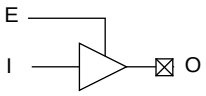
The synthesizer uses this component when creating 3-state external outputs which connect to device pins.

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



OR2 — OR8

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

Inferencing

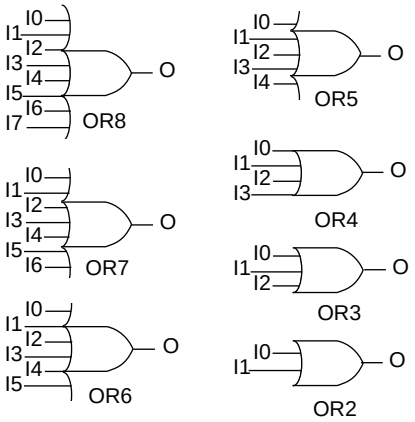
The synthesizer uses these components when creating functions that require OR gates.

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. (XC7000 only)

Inferencing

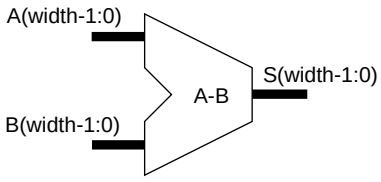
```
diff_signed <= in1_signed - in2_signed;
```

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

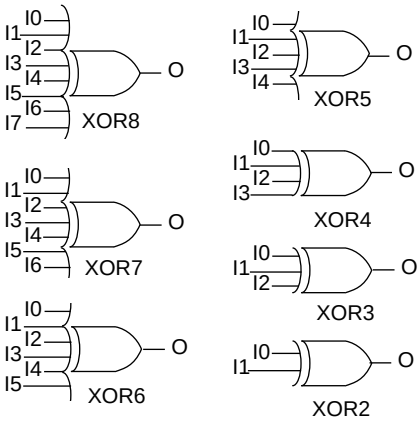
The synthesizer uses these components when creating functions that require XOR gates.

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



U

V

W

X