Cadence Interface/ Tutorial Guide

Cadence Interface/Tutorial Guide — 0401494 *Printed in U.S.A.*

Preface

About This Manual

This manual explains how to use the Xilinx/Cadence Interface software with Cadence Concept and Verilog-XL.

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

Other publications you can consult for related information are the Cadence manuals, *Concept Schematic User Guide* and *Xilinx FPGA Designer (Concept) User Guide*.

Manual Contents

This manual covers the following topics.

- Chapter 1, "Introduction," describes the Xilinx/Cadence design flow, Xilinx-supplied libraries, Xilinx architecture support, and major features.
- Chapter 2, "Getting Started," explains how to set up Xilinx and Cadence files and environment variables.
- Chapter 3, "Design Entry," describes Cadence design entry in relation to the Xilinx software.
- Chapter 4, "Functional Simulation," explains how to perform functional simulation of your designs using Cadence's Verilog-XL simulator.

- Chapter 5, "Design Implementation," explains how to use CONCEPT2XIL to translate your design into an EDIF file.
- Chapter 6, "Timing Simulation," describes how to prepare for timing simulation using the NGD2VER command. The chapter then explains how to conduct timing simulation using the Cadence Verilog-XL software.
- Chapter 7, "Design and Simulation Techniques," describes various design and simulation techniques.
- Chapter 8, "Manual Translation," summarizes how to do design implementation, functional simulation, and timing simulation from the UNIX command line.
- Chapter 9, "Schematic Design Tutorial," guides you through a typical field-programmable gate array (FPGA) and complex programmable logic device (CPLD) design process from schematic entry to completion of a functioning device.
- Appendix A, "Glossary," describes the basic terminology for the Xilinx / Cadence interface.
- Appendix B, "Program Options," describes Xilinx and Cadence command line programs that pertain to the Xilinx / Cadence interface.
- Appendix C, "Processing Designs with LogiBLOX Components," explains how to translate LogiBLOX modules for use within the Cadence Concept editor.
- Appendix D, "Synopsys / Verilog Design Flow," provides a flow chart that illustrates how to process designs described in Verilog HDL using Synopsys and simulate them with the Cadence Verilog-XL simulator.
- Appendix E, "Files," contains annotated testbench templates.
- Appendix F, "XILINX.PFF Property Filter File Format," describes the structure of the xilinx.pff file.

Conventions

Typographical

This manual uses the following conventions. An example illustrates each convention.

- `Courier font` indicates messages, prompts, and program files that the system displays.

  ```
  speed grade: -100
  ```

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

  `rpt_del_net=`

  `Courier bold` also indicates commands that you select from a menu.

  `File` → **`Open`**

- *Italic font* denotes the following items.

  - Variables in a syntax statement for which you must supply values

    `edif2ngd` *design_name*

  - References to other manuals

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

  - Emphasis in text

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

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

  `edif2ngd` [*option_name*] *design_name*

  Square brackets also enclose footnotes in tables that are printed out as hardcopy in DynaText®.

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

  `lowpwr ={on|off}`

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

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

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

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

- A horizontal ellipsis “. . .” indicates that an item can be repeated one or more times.

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

Online Document

Xilinx has created several conventions for use within the DynaText online documents.

- Red-underlined text indicates an interbook link, which is a cross-reference to another book. Click on the red-underlined text to open the specified cross-reference.
- Blue-underlined text indicates an intrabook link, which is a cross-reference within a book. Click on the blue-underlined text to open the specified cross-reference.
- There are several types of icons.

  Iconized figures are identified by the figure icon.

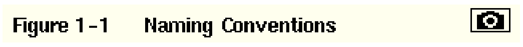


Figure 1-1 Naming Conventions

Iconized tables are identified by the table icon.

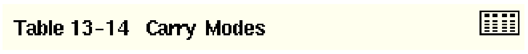
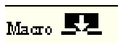


Table 13-14 Carry Modes

The Copyright icon displays in the upper left corner on the first page of every Xilinx online document.

The DynaText footnote icon displays next to the footnoted text.



Macro

Double-click on these icons to display figures, tables, copyright information, or footnotes in a separate window.

- Inline figures display within the text of a document. You can display these figures in a separate window by clicking on the figure.

Contents

Chapter 1

Introduction

This chapter contains the following sections:

- "Architecture Support" section
- "Platform Support" section
- "Features" section
- "Design Flows" section
- "Files" section
- "Tutorials" section
- "Online Help" section
- "Design Approaches" section

Architecture Support

You can use the Cadence interface with the XC3000A / L, XC3100A / L, XC4000E / L, XC4000EX / XL / XV, XC5200, and XC9500 Xilinx architectures.

Platform Support

The Cadence design tools are supported on Sun SPARC and HP Series 9000 workstations.

See the following table for a listing of supported operating systems.

Table 1-1 Platforms Supported by Xilinx/Cadence Interface

Sun 4	Solaris	HP Series 9000
SunOS 4.1.3 SunOS 4.1.4	Solaris 2.5	HP-UX 10.20

Operating system versions listed in this table are based on the setups required to run the Cadence 97A netlisters, CONCEPT2XIL and XIL2CDS. On the HP-UX platform, only version 10.20 is supported. On Solaris, only version 2.5 is supported.

You may use the Xilinx/Cadence interface with either the Cadence release 97A or later.

Features

The following sections describe the major features available in this release.

Xilinx/Cadence Interface

The following table summarizes software supplied by Cadence and Xilinx.

Table 1-2 Xilinx and Cadence Software

Product	Supplied by Xilinx	Supplied by Cadence
Concept Unified Schematic Library	X	
Verilog Unified Simulation Library	X	
Verilog SIMPRIM Library	X	
VAN-Analyzed Verilog Library	X	
Xilinx Core Tools	X	
Concept (schematic editor)		X
Verilog-XL (Verilog simulator)		X
Synergy (synthesis tool)		X
CONCEPT2XIL		X
XIL2CDS		X

Libraries

The Xilinx/Cadence interface supports the Cadence 97A and 97B software releases. However, to process Xilinx designs in conjunction with the 97A and 97B releases, you will also need:

CONCEPT2XIL from Cadence to generate EDIF netlists from Concept designs plus the following libraries from Xilinx:

- Concept Unified Libraries for schematic entry
  - xce3000
  - xce4000e (encompasses XC4000E/L)
  - xce4000x (encompasses XC4000EX/XL/XV)
  - xce5200
  - xce9000
  - xcepads
- VAN-Analyzed Verilog libraries (xc****_syn) for use by CONCEPT2XIL to create EDIF netlists
  - xce3000_syn
  - xce4000e_syn (encompasses XC4000E/L)
  - xce4000x_syn (encompasses XC4000EX/XL/XV)
  - xce5200_syn
  - xce9000_syn
- Verilog Unified Library simulation models for Verilog functional simulation
  - verilogxce3000
  - verilogxce4000e (encompasses XC4000E/L)
  - verilogxce4000x (encompasses XC4000EX/XL/XV)
  - verilogxce5200
  - verilogxce9000
- SIMPRIM-based Verilog simulation models for Verilog timing simulation and post-NGDBuild functional simulation.

Schematic and Verilog Design Entry

The Xilinx / Cadence interface supports these design entry methodologies:

- Concept schematic entry
- Verilog HDL entry (Supported by Cadence)

Direct Generation of Structural Verilog Netlist from Concept Schematics

You can generate a Xilinx Unified Library-based structural Verilog netlist for your design directly from a Concept schematic using the Concept HDL Direct design methodology. See the "Concept Setup Library Files" section of the "Getting Started" chapter for details on how to set up your system for HDL Direct.

Standard EDIF Netlist

The Xilinx core tools read a standard EDIF 2.0.0 netlist as input. EDIF2NGD is the Xilinx tool that translates the EDIF file to a Xilinx NGD (Native Generic Database) file. This netlisting capability makes it easy for you to integrate third party design entry and simulation tools. If you want to implement your design using the Xilinx software, you must first generate a structural Verilog netlist for your design directly from a Concept schematic using HDL Direct. (HDL Direct must be set to On in Concept.) The resulting Verilog netlists are then converted to a standard EDIF netlist using the CONCEPT2XIL netlister.

Concept

Concept is one of the two schematic editors supported by Cadence; Composer is the other schematic entry platform. The Xilinx / Cadence interface for M1 supports only the Concept schematic editor.

Verilog-XL

Verilog-XL is Cadence's Verilog HDL simulator. This simulator is used in the Xilinx / Cadence design flow to verify the functionality of your design. You can use Verilog-XL to perform Unified Library based functional simulation and SIMPRIM-based functional simulation. You may also use Verilog-XL for SIMPRIM-based timing simulation. Timing simulation is performed using a structural Verilog netlist and an SDF file created by NGD2VER. The SDF file contains the timing data for the design.

This release supports the use of Verilog-XL to simulate behavioral Verilog, as well as Verilog gate level netlists composed of SIMPRIM elements. This release also supports gate-level simulation of LogiBLOX components. Gate level netlists are generated by NGD2VER.

CONCEPT2XIL

The command line program, CONCEPT2XIL, is the Cadence Concept EDIF netlister. The CONCEPT2XIL program converts the Verilog (.V) file produced by Concept to an EDIF (.EDF) file, which can then be input to the Xilinx core implementation tools. CONCEPT2XIL is shipped and supported by Cadence Design Systems.

XIL2CDS

XIL2CDS is a command line utility shipped by Cadence that allows you to integrate your chip-level design into a board level schematic.

Contact Cadence for more information about XIL2CDS.

NGD2VER

NGD2VER generates a structural SIMPRIM library-based Verilog netlist that points to the SIMPRIM library when the -ul option is specified. A *design*.tv testbench template file can also be created by specifying the -tf option.You can use the template to create a testbench to verify your design. If there is timing information available in a mapped or routed NGA file, an SDF file is also generated.

Simulation of Synopsys Designs

The Xilinx/Cadence interface supports post-synthesis and timing simulation of Synopsys designs entered in Verilog HDL through the generic Verilog HDL netlister, NGD2VER, which is shipped with the Xilinx core tools. For more information, refer to the "Synopsys/Verilog Design Flow" appendix.

Automatic Library Specification

NGD2VER will generate the Verilog-XL `uselib statement in your Verilog netlist referencing the SIMPRIM library when you specify the -ul option.

Waveform Viewer Support

NGD2VER can add support for the Cadence SimWave Waveform Viewer by writing out $shm_open and $shm_probe directives to your Verilog netlist to create a Simulation History Manager (SHM) database. The SHM directives are incorporated into the test fixture (.tv) created by NGD2VER.

LogiBLOX

LogiBLOX is a Xilinx tool that you can use to create high-level functional modules that can be incorporated into a schematic or an HDL-based design. LogiBLOX is only supported in standalone mode for the Cadence interface. After you create your modules, you must use the Concept genview command to generate bodies for your modules. See the "Processing Designs with LogiBLOX Components" appendix for details.

Timing Constraints

You can specify timing constraints in your Concept schematic to guide the place and route tools; timing constraints can be added as properties. For details about timing constraints, refer to the "Using Timing Constraints" chapter in the *Development System Reference Guide.*

You can also place constraints in an external constraints file (*.ucf extension) that EDIF2NGD can process. For details on user constraint files, refer to the "The User Constraints (UCF) File" chapter in the *Development System Reference Guide.*

Synergy Support

Synergy is Cadence's synthesis tool. Synergy can synthesize designs entered in either Verilog HDL or VHDL. The Xilinx interface to Synergy is available only from Cadence Design Systems.

Design Flows

The design flow you use for performing design entry and simulation depends on whether you use schematic design entry or HDL design entry.

In either case, the easiest and most automatic way to implement your design is to use the Xilinx Design Manager graphical interface. You can also run the various programs in the design flow manually from a UNIX command prompt. The programs in the FPGA design implementation flow are described in the "Program Options" appendix. These commands are also described in detail in the *Development System Reference Guide*.

The programs from the CPLD design flow are described in the *CPLD Schematic Design Guide* and *CPLD Synthesis Design Guide*.

The Xilinx / Cadence interface supports the following design flows:

- Schematic entry with the Unified Libraries components, LogiBLOX components, or both;
- Schematic entry with Concept Unified Library schematic components plus Xilinx-compliant EDIF or NGO blocks

The following two figures show the highlights of the design process for: FPGA design and CPLD design. Note that many of the details in both figures are the same except within the blocks labelled "Design Manager Flow Engine" and "Schematic Entry Design Flow."

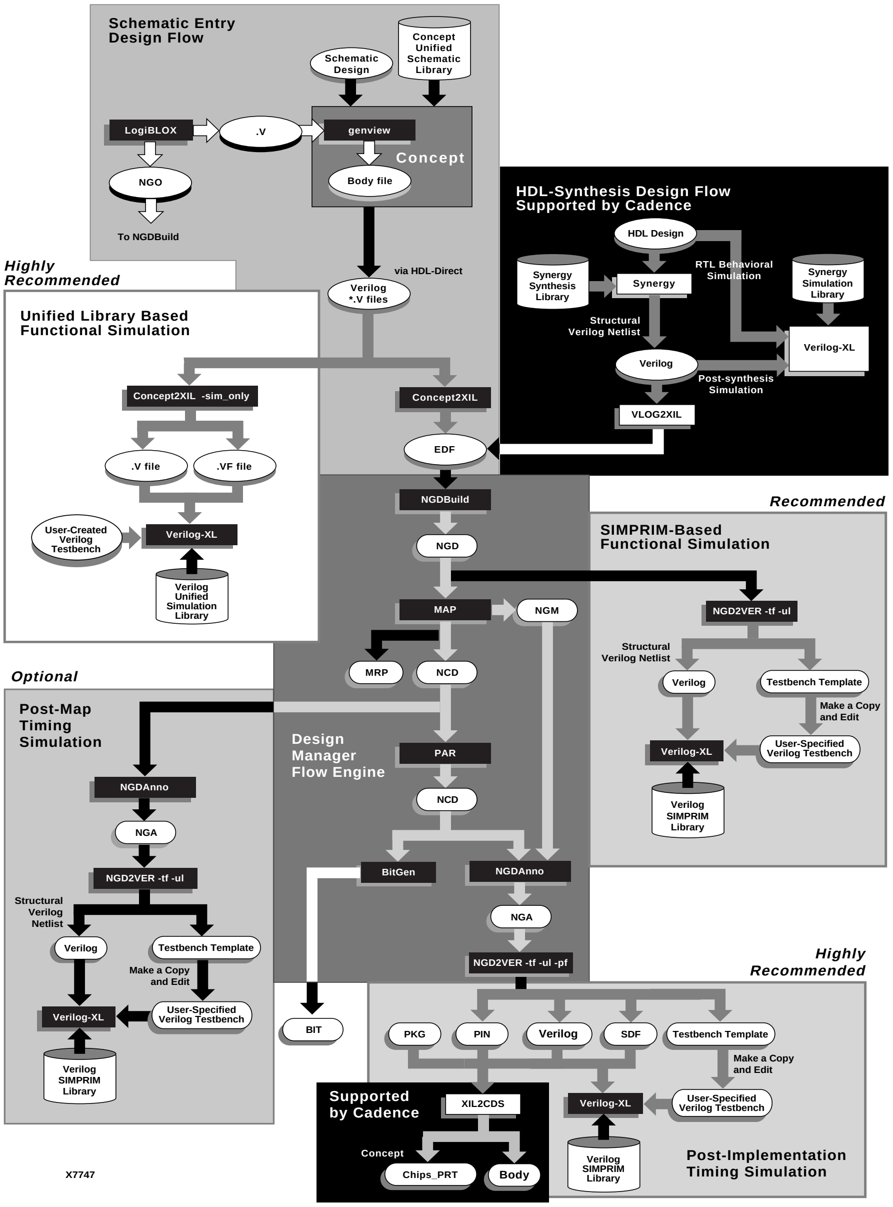


Figure 1-1 Overall Cadence Design Flow — FPGA Design

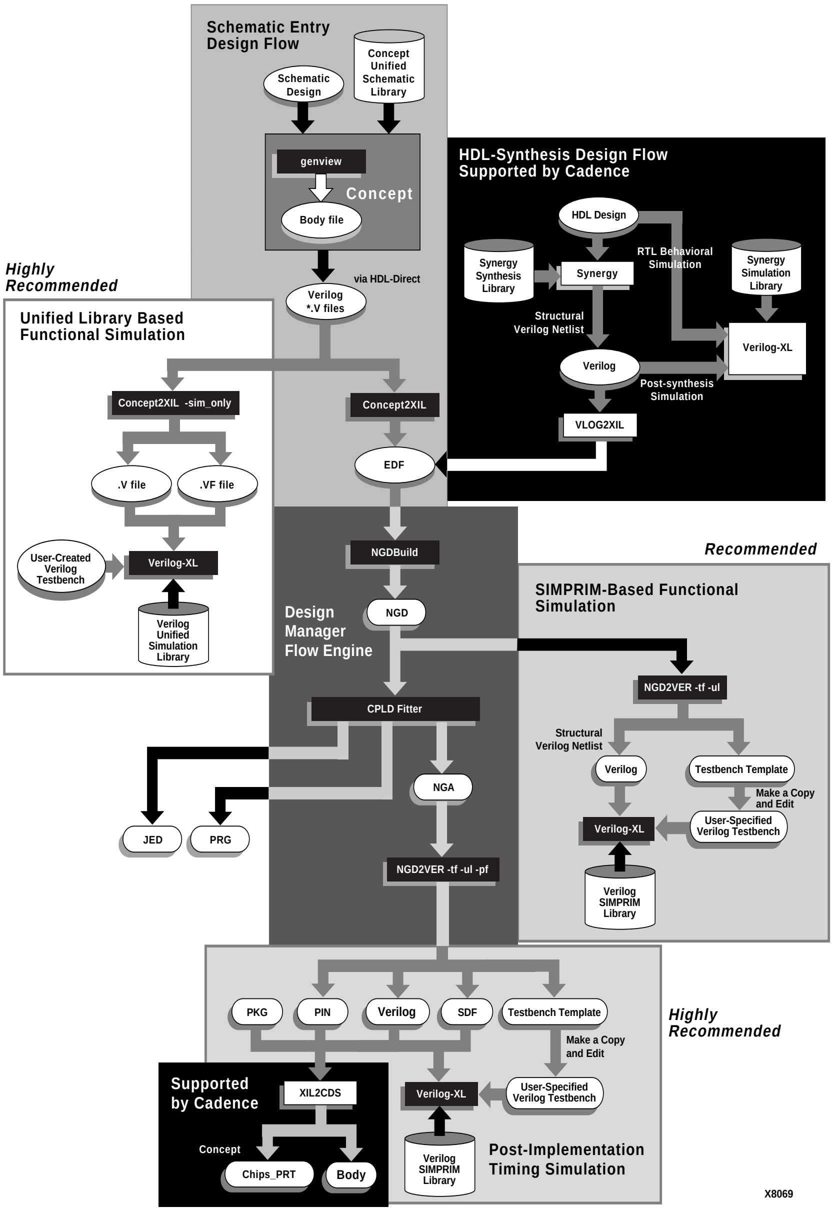

Schematic Entry Design Flow
Schematic Design
Concept Unified Schematic Library
genview
Concept
Body file
via HDL-Direct
Verilog *.V files
HDL-Synthesis Design Flow Supported by Cadence
HDL Design
Synergy Synthesis Library
Synergy
RTL Behavioral Simulation
Synergy Simulation Library
Structural Verilog Netlist
Verilog-XL
Verilog
Post-synthesis Simulation
VLOG2XIL
Highly Recommended
Unified Library Based Functional Simulation
Concept2XIL -sim_only
Concept2XIL
.V file
.VF file
EDF
User-Created Verilog Testbench
Verilog-XL
Verilog Unified Simulation Library
NGDBuild
Recommended
SIMPRIM-Based Functional Simulation
Design Manager Flow Engine
NGD
NGD2VER -tf -ul
CPLD Fitter
Structural Verilog Netlist
Verilog
Testbench Template
Make a Copy and Edit
NGA
User-Specified Verilog Testbench
Verilog-XL
JED
PRG
NGD2VER -tf -ul -pf
Verilog SIMPRIM Library
PKG
PIN
Verilog
SDF
Testbench Template
Highly Recommended
Make a Copy and Edit
Supported by Cadence
XIL2CDS
Verilog-XL
User-Specified Verilog Testbench
Concept
Chips_PRT
Body
Verilog SIMPRIM Library
Post-Implementation Timing Simulation
X8069

Figure 1-2 Overall Cadence Design Flow — CPLD Design

Files

The following files are involved in the processing of a design through the Cadence interface:

- CONCEPT2XIL creates an .EDF file, which is an EDIF netlist file.
- EDIF2NGD creates an .NGO file, which contains netlist information in a proprietary data base format; it is a binary file.
- A .UCF file, an input file to NGDBuild, contains user-specified constraints for the map, place, and route tools. This file contains I/O locations and maximum timing delays.
- NGDBuild creates an .NGD file, which is a Native Generic Database file; it contains a gate-level logical description of the design.
- MAP or PAR create an .NCD file, which is a Native Circuit Design file; it contains a physical description of the design.
- NGDAnno creates an .NGA file, which contains physical timing delay information.
- NGD2VER creates a .v file, .tv, .pin, and .SDF file if the input file is a .NGA file when invoked with the -tf and -pf options.

  The -tf option creates a .tv file, which is a Verilog test bench or stimulus file template.

  The -pf option creates a .pin file which contains pinout information for a design. The -pf option cannot be used if the input to NGD2VER is an NGD file. The .pin file correlates each signal in the design to a pin on a particular Xilinx FPGA package.

  The .v and .sdf files are always created when NGD2VER is run with an NGA file input.

  An .SDF file is a Standard Delay Format file containing delay information.
- A .PKG file defines the pins on a Xilinx FPGA or CPLD package. These files, which are supplied by Xilinx, are located in $XILINX/cadence/data.

Tutorials

Xilinx recommends that you perform the tutorials provided in this manual to become familiar with the basic concepts of design, verification, and implementation.

Online Help

Online help is available in the Concept schematic editor by entering **help** in the Concept command window. You can also access information on Cadence tools (for example, Concept or Verilog-XL) in OpenBook, Cadence's online documentation system, which is shipped by Cadence. To start up OpenBook, simply enter the command **openbook** at the UNIX prompt.

Design Approaches

You can enter a design using schematics or a hardware description language (HDL) such as Verilog.

Schematic Entry

This general procedure describes the schematic entry flow.

1. Enter your design using the Concept schematic editor.

Note: You must have Concept's HDL Direct mode enabled. For details, see the "Getting Started" chapter. With HDL Direct enabled, Concept generates a Verilog netlist automatically when you save a drawing.

2. Process the Verilog files that Concept produces with the CONCEPT2XIL netlister using the -sim_only command line option.
3. Functionally simulate your design using Verilog-XL.
4. Translate your design into EDIF format using CONCEPT2XIL.
5. Use NGDBuild in the Xilinx Design Manager or the command line tool to convert the EDIF file to an NGD file.
6. Implement your design with the Xilinx Design Manager/Flow Engine. (You can also use the command line versions of these programs.)

If you perform a manual translation of your design, you can also do a post-MAP simulation to obtain a rough timing simulation before routing delays are added. Alternatively, you can run TRCE after mapping to evaluate timing before net delays are added. For an explanation of TRCE, see the "TRACE" chapter in the *Development System Reference Guide*.

7. Perform timing simulation on the design using Verilog-XL and a test bench stimulus file.
8. Download your design to the FPGA, or program the CPLD.
9. Optionally, use the XIL2CDS program to integrate your chip-level design into a board level schematic.

Verilog HDL Entry (Synergy)

Verilog HDL entry flows are supported by Cadence Design Systems.

The steps you follow when using HDL to process a design are similar to those you follow for schematic entry.

1. Create the design in Verilog.
2. Conduct an RTL (Register Transfer Level) behavioral simulation of your design.

   RTL level simulation allows you to verify or simulate a description at the system or chip level. At this level designers generally describe the system or chip by using high-level RTL language constructs.
3. Synthesize the design. With Synergy, the output is a .V file.
4. Translate your design into EDIF format using VLOG2XIL.
5. Use NGDBuild in the Xilinx Design Manager or the command line tool to convert the EDIF file to an NGD file, and merge the NGO files with the rest of the design.
6. Optionally, you may generate a post-synthesis Verilog netlist using NGD2VER and perform a SIMPRIM-based functional simulation.
7. Implement your design with the Xilinx Design Manager / Flow Engine. You can also use the command line versions of the individual tools to process the design.

If you use the command line flow, optionally, you can also perform a post-map timing simulation of your design before routing delays are added. Alternatively, you can run TRCE after mapping to evaluate timing before net delays are added.

8. Perform timing simulation on the design using Verilog-XL and a test bench stimulus file.
9. Download your design to the FPGA, or program the CPLD.
10. Optionally, use the XIL2CDS program to integrate your chip-level design into a board level schematic.

Mixed-mode Entry, Top Level Schematic

1. Capture the top level schematic in Concept.
2. Make sure that each non-schematic block is processed to either an NGO, XNF, or EDIF format file. If you have HDL blocks, these must be synthesized first and translated to one of these three formats.
3. Generate a Concept body for each non-schematic block, either manually, or using the genview utility in Concept.
4. Instantiate the body into the appropriate sheet (page) of your schematic design.
5. Save your design. (You must have Concept's HDL Direct mode enabled; for details, see the "Getting Started" chapter.)
6. Add the following line after the part list in your Verilog netlist:

   `parameter cds_action="ignore";`

7. Translate your design into EDIF format using CONCEPT2XIL.
8. Use NGDBuild in the Xilinx Design Manager or the command line tool to convert the EDIF file to an NGD file and merge the NGO files with the rest of the design.
9. Optionally, generate an unrouted post-NGDBuild Verilog netlist using NGD2VER and perform a SIMPRIM-based functional simulation.

   If you conduct a manual translation, you may also want to perform a functional simulation to obtain a rough estimate of delays in the unrouted design.

10. Implement your design with the Xilinx Design Manager/Flow Engine, or perform these steps manually if you prefer.

    If you perform a manual translation, you may also perform a post-Map simulation to get an approximate estimate of delays in the unrouted design.

11. Place and route (PAR) and conduct a back-annotation (NGDAnno) on your design before performing timing simulation.

12. Perform timing simulation on the design using Verilog-XL and a test bench stimulus file.

13. Download your design to the FPGA, or program the CPLD.

14. Optionally, use the XIL2CDS program to integrate your chip-level design into a board level schematic.

Mixed-Mode Entry, Top Level HDL (Verilog)

1. Edit the top level in Verilog
2. Instantiate each non-Verilog block in the design, and specify a “preserve” property on each block that corresponds to a schematic block.
3. Synthesize the design.
4. Write out a Verilog netlist for the design.
5. Translate your design into EDIF format with VLOG2XIL.
6. Implement your design with the Xilinx Design Manager/Flow Engine, or perform these steps manually, if you prefer. If you process your design manually, you can also do a post-MAP timing simulation to obtain a rough idea of whether your timing requirements can be met.
7. Perform timing simulation on the design using Verilog-XL.
8. Download your design to the FPGA or program the CPLD.

Chapter 2

Getting Started

This chapter lists the required software and describes how to configure your system to use the Cadence design tools for creating and processing Xilinx designs. The Xilinx / Cadence Interface supports the following Cadence programs: Concept and Verilog-XL. For Synergy synthesis support, contact Cadence Design Systems.

This chapter contains the following sections:

- "Required Software" section
- "Setting Up Your Environment" section
- "Invoking Concept" section
- "Exiting Concept" section

Required Software

To enter designs using schematics and program Xilinx FPGAs and CPLDs, you need the following software programs:

- Cadence Release 97A or later; see the "Platform Support" section of the "Introduction" chapter for a list showing which version you should use on your particular platform. Your installation of Cadence should include the following components:

  a) Concept

  b) Verilog-XL

  c) CONCEPT2XIL (EDIF netlister)—available from the Cadence ftp site

  d) XIL2CDS (for optional board-level integration)—available from the Cadence ftp site

- Libraries supplied by Xilinx:

a) For schematic entry: Concept Unified Libraries

b) For Verilog functional simulation: Verilog Unified Library simulation models

c) For Verilog timing simulation: SIMPRIM-based Verilog simulation models

- Xilinx Development System software; your installation of Xilinx products must include at least the following executables:

  a) EDIF2NGD

  b) NGDBuild

  c) MAP

  d) PAR

  e) TRCE (Optional for static timing analysis)

  f) NGDAnno

  g) NGD2VER

  h) BitGen

For a description of Xilinx / Cadence platform support, refer to the "Platform Support" section of the "Introduction" chapter.

Setting Up Your Environment

When you have finished the installation of the Xilinx software, verify that your .cshrc or setup file contains lines similar to those outlined in the following subsections.

Note: In the following variable settings, *platform* is **sun** (Sun4), **sol** (Solaris), or **hp** (HP-UX).

Required Environment Variables (All Platforms)

Verify that the following variables are set up;

setenv XILINX *path_to_Xilinx_root_dir*

where *path_to_Xilinx_root_dir* is the location of the Xilinx software.

If you are using the Motif version of DynaText or the EPIC editor, you must set an environment variable to access the set of Key bindings used by a Motif application. You will find an XKeysymDB file has been installed into your $XILINX/bin/*platform* directory by the Xilinx install. You must do the following to access this file:

```
setenv XKEYSYMDB $XILINX/bin/platform/XKeysymDB
```

Failure to set up this environment variable will result in the following types of messages being displayed when you attempt to start up the DynaText viewer. The listed keys are not usable:

```
Warning: translation table syntax error: Unknown
keysym name: osfActivate

Warning: ... found while parsing ':
<Key>osfActivate:ManagerParentActivate()fDown
```

Set your LM_LICENSE_FILE variable to point to the Xilinx license file (license.dat):

```
setenv LM_LICENSE_FILE path_to_Xilinx_license_file ;\
    path_to_Cadence_license_file
```

Make sure the LD_LIBRARY_PATH is set up to point to the Xilinx software. Add the path to your current path for Sun or Solaris.

```
setenv LD_LIBRARY_PATH \
   ${LD_LIBRARY_PATH}:$XILINX/bin/platform
```

Note: The backslash (\) at the end of a line is a continuation character indicating that the line wraps to the next line. If you use the backslash character, it *must* be the last character on the line.

If you are using a Sparc station, set LD_LIBRARY_PATH as follows:

```
setenv LD_LIBRARY_PATH \
   ${LD_LIBRARY_PATH}:$XILINX/bin/platform: \
   /usr/openwin/lib
```

If you are using an HP workstation, set the SHLIB_PATH.

```
setenv SHLIB_PATH ${XILINX}/bin/hp:lib:/usr/lib
```

Concept Environment Setup

In addition, you must set your Concept environment variables in your .cshrc shell or setup file and configure your Concept startup and library files.

Concept Environment Variables

1. Set the CDS_INST_DIR environment variable to the location of your Cadence installation directory in your .cshrc or setup file.

   `setenv CDS_INST_DIR` *location_of_Cadence_tools*

Note: It is common to create a soft link called "tools" under **$CDS_INST_DIR**, and to link it to the directory **$CDS_INST_DIR**/ tools. *platform*, where *platform* is "hppa" (for the HP), "sun4" (for SunOS), or "sun4v" (for Solaris). For example, to create a link called "tools" to your "tools.sun4" Cadence subdirectory (SunOS), use the following commands:

```
cd $CDS_INST_DIR
```

```
ln -s tools.sun4 tools
```

If your Cadence installation directory does not have a link called "tools", you can either add the link yourself, or substitute "tools.*platform*" wherever you see "tools" in the settings.

2. Add CDS_INST_DIR to your path in your .cshrc or setup file.

```
set path = ($CDS_INST_DIR/tools/bin \
   $CDS_INST_DIR/tools/pic/picdesigner/bin \
   $CDS_INST_DIR/tools/editor/lib $path)
```

Concept Setup Library Files

Before you begin design entry, verify that your Concept setup files are set up properly. These four files include *startup.concept*, *cds.lib*, *global.cm*d, and *master.local*. The files should be located in your current working Concept directory. Xilinx has provided examples of each of these files in $XILINX/cadence/examples.

1. Use a text editor to modify your startup.concept file so that you are set up to run HDL Direct. Xilinx recommends that you add the following lines to the file to enable HDL Direct.

```
set hdl_direct on
set hdl_checks on
set check_signames on
set check_net_names_hdl_ok  on
set check_port_names_hdl_ok  on
set check_symbol_names_hdl_ok  on
set capslock_off
```

HDL Direct will run automatically when you write your design. Error and warning messages are written to the Concept HDL Direct window and to the hdldir.log file.

These commands can also be set directly in Concept by entering them in the Concept command window.

Xilinx also recommends that you include the command, `set capslock_off`, in the startup.concept file.

```
set capslock_off
```

With this command, Concept maintains the case of the property strings that you add to the design. This setting is important when you define linked properties that reference other pre-defined properties, as in linked timespecs. The referencing of pre-defined timespec properties is case-sensitive. When HDL Direct writes out these properties to the viewprp file for the specific Concept drawing being saved, it will preserve the case of all properties when the capslock key is set to “off”, maintaining case, to the viewprps.prp file in the logic view (directory) for the appropriate Concept drawing.

It is not required that you enter this command in the startup file. However, if you do not enter the command, make sure that you enter the command at the command line prompt in Concept *before* you assign property values. If you do not enter the capslock_off command first, the values will be converted to upper case by default. This may cause problems when defining new TIMEGRPS from existing groups declared in lower case.

2. Create a cds.lib file in your current working Concept directory that points to the VAN-compiled library for the Xilinx architecture you will be using. Concept and CONCEPT2XIL scripts require cds.lib file configuration.

   CONCEPT2XIL writes out an EDIF netlist for you based on library components it reads out of the appropriate library in cds.lib. The library it reads depends on the value you specify for the -family option. Following is an example cds.lib file for the xce4000x architecture.

```
define xce4000x_syn full_path_to_Xilinx_sofware/cadence/data/xce4000x_syn
```

   In this example, xce4000x_syn is the VAN-compiled library for XC4000EX devices.

3. Update your global.cmd file to point to the appropriate Xilinx Concept libraries and define the default name of the SCALD mapping file with the “use” command.

   For more details about the global.cmd file, refer to the section “Global.cmd file” in Chapter 2, The Editing Environment, in the *Concept Schematic User Guide*. Following is an example file for the an XC4000ex device:

   ```
   master_library "./master.local" ;
   library "xce4000x"
           "hdl_direct_lib" ,
           "xcepads" ,
           "standard" ;
   use "design.wrk" ;
   root_drawing "unnamed" ;
   ```

   Following is a brief description of each of the library elements:

   - xce4000x
     This text points to the architecture-specific Concept XC4000EX / XL / XV library.
   - hdl_direct_lib
     This library contains various components to support HDL Direct methodology, including inports and outports.
   - xcepads
     This text points to the generic Xilinx pad library.
   - standard
     The standard library contains standard Concept components such as drawing and border symbols.

   All four libraries are required.

   When accessing a library, Concept searches through the libraries following a “last read, first out” scheme. For the sample global.cmd file, the libraries are searched in the following order: standard, xcepads, hdl_direct_lib, and finally xce4000x.

   The **use** command specifies the work or project library listing file from which existing design names can be read and viewed in Concept, and to which listings for new design blocks can be written into from Concept. For example, **`use "design.wrk";`** indicates that design.wrk is the work file:

   FILE_TYPE = LOGIC_DIR;

"DESIGN1" '*design1*';
"DESIGN2" '*design*2';

END.

design1 and *design*2 are the names of the design blocks for this project.

4. Enter the references to any user libraries into the master.local file. For more details about the master.local file, see the section "master.local Abbreviations file" in Chapter 2, The Editing Environment in the *Concept Schematic User Guide*. For M1, it is recommended that you list all Xilinx architecture libraries for Concept in the master.local file. Following is an example user library file for M1 Xilinx designs.

```
file_type = master_library;

"xce9000"   'full_path_to_Xilinx_software/cadence/data/xce9000/xce9000.lib';
"xce5200"   'full_path_to_Xilinx_software/cadence/data/xce5200/xce5200.lib';
"xce4000x"   'full_path_to_Xilinx_software/cadence/data/xce4000e/xce4000x.lib';
"xce4000e"   'full_path_to_Xilinx_software/cadence/data/xce4000e/xce4000e.lib';
"xce3000"   'full_path_to_Xilinx_software/cadence/data/xce3000/xce3000.lib';
"xcepads" 'full_path_to_Xilinx_software/cadence/data/xcepads/xcepads.lib';

end.
```

The xce4000x supports the XC4000EX/XL/XV architectures. The xce4000e supports the XC4000E/L architectures.

Verilog Environment Setup

Set up your Verilog environment so that you can perform functional simulation.

- Set the Verilog environment variable, VERILOGEXE, to point to the location of your Verilog executables.

```
setenv VERILOGEXE $CDS_INST_DIR/tools/\
   verilog/bin/verilog
```

- Set VENVHOME to the location where the Verilog 2.0 hierarchy is installed. This variable is used only to assist you with setting the VENV_PATH and VENV_LD_LIB_PATH variables, and is usually the same as CDS_INST_DIR.

```
setenv VENVHOME /tools/cadence97A
```

- Set VENV_PATH to the location of the Verilog executables.

```
setenv VENV_PATH "${VENVHOME}/\
   tools/bin:${VENVHOME}/tools/dfII/bin”
```

- Set VENV_LD_LIB_PATH to the location of libraries used by Verilog.

  ```
  setenv VENV_LD_LIB_PATH ${VENVHOME}/\
     tools/lib:${VENVHOME}/tools/verilog/lib
  ```

- Reset your PATH variable to add the Verilog executables.

  ```
  set PATH="${VENV_PATH}:$PATH"
  ```

- Add the Verilog libraries to the LD_LIBRARY_PATH environment variable.

  ```
  setenv LD_LIBRARY_PATH \
     "${VENV_LD_LIB_PATH}:$LD_LIBRARY_PATH"
  ```

  If you are using an HP platform, use the SHLIB_PATH instead of LD_LIBRARY_PATH.

- (Optional) Set up the environment variable CDSDIR. This variable is required by the Verilog Language Sensitive Editor (LSE).

  ```
  setenv CDSDIR $CDS_INST_DIR/tools
  ```

Dynatext Environment Variables

Set up the EBTRC variable to point to the DynaText browser.

setenv EBTRC $XILINX/bin/*platform***/ebtrc_CD**

Invoking Concept

To enter Concept from the operating system command line, type **concept**.

Exiting Concept

To exit Concept, select **File** → **Quit**. You may also enter “quit” in the Concept command line window.

Chapter 3

Design Entry

This chapter describes Cadence design entry in relation to the Xilinx Development software. This chapter does *not* discuss in any detail how to use the design tools Concept or Synergy. Design entry procedures are described only for Xilinx-specific software, libraries, and features.

For a detailed description of the Concept design entry tools and procedures, refer to the *Concept Schematic User Guide*. For a list of the manuals that describe the Synergy tool, see the "Verilog HDL Design Entry" section in this chapter.

Before using the design entry tools, ensure that you have set up your environment as described in the "Required Software" section of the "Getting Started" chapter.

This chapter contains the following sections.

- "Concept" section
- "Requirements For HDL Direct Compliance" section
- "Using the Xilinx Concept Unified Schematic Libraries" section
- "Renamed Components" section
- "Verilog / Concept HDL Direct Naming Conventions" section
- "VCC and GND Components" section
- "Using the LogiBLOX Libraries" section
- "Specifying Xilinx Properties and Constraints in Concept" section
- "Attaching Signal Names" section
- "Creating Bus Taps" section

- "Using the BSCAN Symbol" section
- "Using the STARTUP Symbol" section
- "Using the CONFIG Symbol to Specify Part Type" section
- "Using HDL Direct Methodology" section
- "Creating Bodies for Non-Schematic Design Blocks" section
- "Verilog HDL Design Entry" section
- "Translating Your Design" section

Concept

Concept is a Cadence schematic entry tool. You can capture designs for implementation in the XC3000A / L, XC3100A / L, XC4000E / L, XC4000EX / XL / XV, XC5200, and XC9500 Xilinx architectures.

The Concept libraries include the following:

- Libraries for designing Xilinx FPGAs and CPLDs named xce*xxxx*
- LogiBLOX modules, which are generated by the lbgui (LogiBLOX GUI) standalone command for use with Concept, include Verilog and NGO files. The Verilog modules generated by lbgui are used as the input to the Concept genview command to create body files. The NGO files created by lbgui are used to implement the LogiBLOX module in your design.
- VAN-analyzed Verilog libraries for Concept HDL Direct Verilog netlist generation support

For a complete description of how to use Concept, see the *Concept Schematic User Guide*. Also see the *Concept Getting Started Tutorial*.

The following illustration shows the schematic entry flow using Concept with the Xilinx-supplied Concept Unified Schematic Libraries. Note that LogiBLOX is not supported for use with CPLDs.

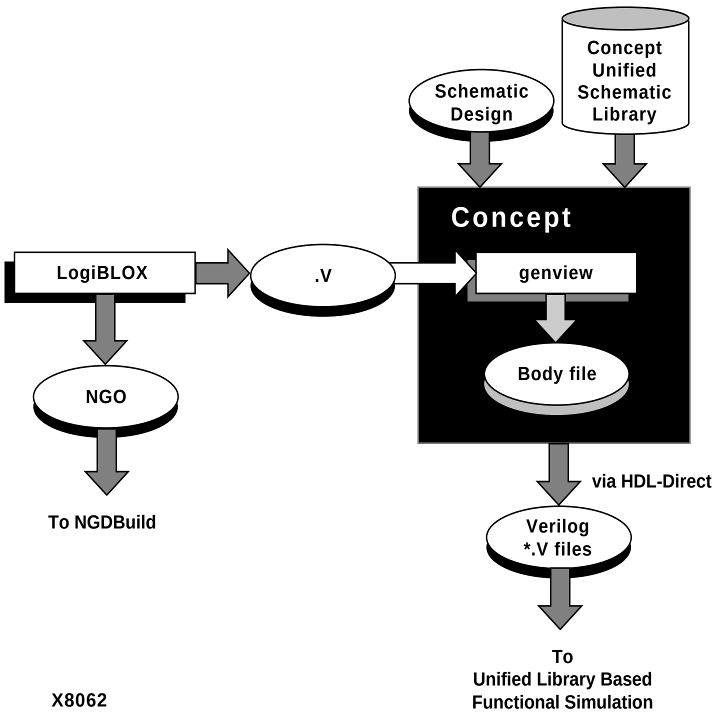


Figure 3-1 Schematic Entry Design Flow

Requirements For HDL Direct Compliance

The Xilinx/Cadence schematic design flow requires that SCALD (Structured Computer Aided Logic Design) schematic designs be converted to HDL Direct compliant.

Your SCALD schematic drawings must adhere to the following rules:

- SCALD BIT TAP symbols must be replaced with the HDL Direct SLICE symbols.
- HDL Direct TAP symbols must be used in place of SCALD LSB TAP and MSB TAP symbols.
- Instead of the SCALD convention of attaching an “\I” suffix to interface signals, HDL Direct port symbols from hdl_direct_lib

must be attached to the signals in a schematic which correspond to pins on a symbol body at an upper-level of hierarchy in the design. INPORTS should be attached to the inputs, OUTPORTS to the outputs, and IOPORTS to the bidirectional signals of the schematic.

- SIGN EXTEND and SLASH symbols must be replaced with the equivalent wiring.
- The “\I” must be removed from all signal names.
- FLAG symbols must be removed from schematics.
- For signals that end with \G (which designates them as global signals), remove the \G and place a forward slash (/) at the beginning of the signal.
- Wires or signals must not be connected to pass-thru pins on pads. Connect them only to the main pin. Use the “bubble_check off” option with the SCALD compiler.
- NOT bodies must be replaced with wires.
- All signals and symbol names starting with numbers must be renamed to valid Verilog identifiers. For example, replace 1MYSIGNAL with MYSIGNAL1.
- SUPPLY_0 and SUPPLY_1 are not supported. Use the GND and VCC symbols from the appropriate Xilinx architecture library.
- The HDL Direct signal concatenation operator ampersand (&) replaces the SCALD signal concatenation operator colon (:).
- The signal replication operator “\R number” and the REPLICATE symbol must be replaced with a concatenation of the required number of signals. The SCALD REPLICATE symbol should be replaced in the same way.
- The HDL Direct ALIAS symbols replace the SCALD SYNONYM symbols; however, ALIAS symbols are not supported by Xilinx.
- Signals and bodies may not share the same name.
- Property values must adhere to Verilog naming rules.

For more detailed and up-to-date information, refer to the “Converting SCALD Schematics into HDL Direct Schematics” of the “Using HDL Direct With SCALD Applications” chapter of the Cadence manual, *HDL Direct User Guide.*

Using the Xilinx Concept Unified Schematic Libraries

Xilinx supplies the Concept Unified Schematic Libraries as part of the Xilinx Development software release. To design Concept schematics using Xilinx devices, you must install the Libraries. In addition, you must also set up your Concept setup files to access these Libraries. Refer to the "Concept Environment Setup" section of the "Getting Started" chapter for details. To verify that Concept is installed and set up so that you can access the Concept Unified Schematic Libraries, perform the following steps.

1. Open Concept by entering the command **`concept`** at the prompt.
2. Select **`Add Part`**.
3. When the Component Browser dialog box displays, click the list box to the right of the Library field. The list of libraries should display the xce series parts (xce4000e, xce4000ex, etc.) including the pads (xcepads).

To create a Xilinx FPGA or CPLD design with Concept, you can use the XC3000A, XC4000E/L, XC4000EX/XL/XV, XC5200, or XC9000 libraries. The corresponding name for these Xilinx families in Concept are xce3000, xce4000e (XC4000E/L), xce4000ex, xce4000x (XC4000EX/XL/XV), xce5200, and xce9000. In addition, the xcepads library contains the basic pads, such as IOPAD, IPAD, OPAD, and UPAD. You can also use the LogiBLOX modules for designs targeting Xilinx architectures that support LogiBLOX. Your design must contain primitives and macros.

You can only use the libraries for one family when creating your design. For example, you cannot use elements from both the XC4000 and the XC5200 libraries in a single design. Also components in these libraries are no longer sizable. See the "SIZE Property" section.

You can specify design libraries by editing the global.cmd file. The first line of the global.cmd file list of libraries begins with the keyword, "library":

`library` "*lib_name*",["*lib_name*"]**;**

Lib_name is a library name like XC4000E. Specify multiple libraries by using a comma (,) to separate each library name. Be sure that a semicolon (;) follows your last entry. Refer to the example in $XILINX/cadence/examples.

FPGA and CPLD Libraries

The XC3000A, XC4000E/L, XC4000EX/XL/XV, and XC5200 libraries are sets of primitives and macros that include all the basic Xilinx FPGA components, such as logic gates and flip-flops. The XC9000 library is the set of primitives and macros for Xilinx CPLDs. You can find descriptions of all FPGA and CPLD elements in the Xilinx *Libraries Guide*.

PAD Library

The PAD library contains the basic Xilinx pad (IPAD, IOPAD, OPAD, and UPAD) primitives, called “xcepads.”

If you want to place properties on a pad, instead you must attach the properties to the corresponding I/O buffer. If you place properties on PAD symbols, these properties will be ignored during the netlisting process—that is, they will be omitted from your design.

Renamed Components

To conform to naming restrictions in Concept and to prevent overlap with built-in Verilog primitives, the following Xilinx library components have been renamed in the Concept and Verilog Unified libraries.

Table 3-1 Renamed Components

Xilinx Component Name	Cadence Component Name	Component Description	Applicable Families
BUF	BUFF	General purpose non-inverting buffer	XC3000 XC4000E/L XC4000EX/XL/XV XC5200 XC9000
PULLDOWN	PULLDOWN1	Resistor to GND for input PADs	XC4000E/L XC4000EX/XL/XV XC5200

Table 3-1 Renamed Components

Xilinx Component Name	Cadence Component Name	Component Description	Applicable Families
PULLUP	PULLUP1	Resistor to VCC for input PADs, open-drain, and tristate outputs	XC3000 XC4000E / L XC4000EX / XL / XV XC5200 XC9000

A translation table located at $XILINX / cadence / data / cadence.ttl maps the Concept component names to the Xilinx component names for these library components.

In addition, several of the component names have changed in the new M1 Xilinx Concept Unified Libraries for consistency with the Xilinx Unified Library conventions. The following table summarizes these name changes.

Old Name	New Name
ILD1	ILD_1
ILDI1	ILDI_1
ILDX1	ILDX_1
ILDXI1	ILDXI_1

Verilog/Concept HDL Direct Naming Conventions

Verilog requires that several rules be followed for user-specified names in Xilinx designs. Additionally, the Xilinx / Cadence interface requires that you use HDL Direct methodology: therefore, you must adhere to these rules when naming objects in your Concept schematics. The rules are defined as follows for Verilog naming:

- A name in Verilog is a sequence of letters, digits, dollar signs ($), and the underscore (_) symbol. More specifically, only the following characters are legal in Verilog names:

  `a-z, A-Z, 0-9, _, and $`

- You may not use a digit or $ as a first character of a name. The name must begin with a letter or an underscore. If you enter your design in Concept, the first character must be a letter.
- Upper and lower case letters are interpreted as different characters (unless the upper case option is used when compiling).
- Names can be a total of up to 1024 characters long.

  Examples of valid names:

  adder
  bus_b
  _signal1
  n$777
  clockin
- Verilog keywords cannot be used as user-specified names. This includes Verilog reserved words like “input”, “output”, and “module”.
- Concept allows block names and signal names to be identical, but in Verilog, block names and signal names cannot be the same.

For more information, refer to section 2.5 in the *Verilog-XL Reference Manual*.

VCC and GND Components

For the Xilinx Development software, the VCC and GND components are located in the device family libraries instead of the PAD libraries.

Using the LogiBLOX Libraries

LogiBLOX is a graphical interactive tool from Xilinx for creating high-level functional modules, such as counters, shift registers, multiplexers, and memories. Memory module creation is a key feature of LogiBLOX. For a description of the LogiBLOX memory modules, refer to the “Module Descriptions” chapter in the *LogiBLOX Reference/User Guide*.

LogiBLOX includes both a library of generic modules and a set of tools for customizing them. The modules you create with LogiBLOX can be used in designs generated with Cadence’s Concept schematic editor. For details, refer to the "Processing Designs with LogiBLOX Components" appendix.

Specifying Xilinx Properties and Constraints in Concept

This section describes various rules and restrictions for using Xilinx properties and constraints within Concept. This section also explains where you can obtain detailed information on how to add properties in Concept.

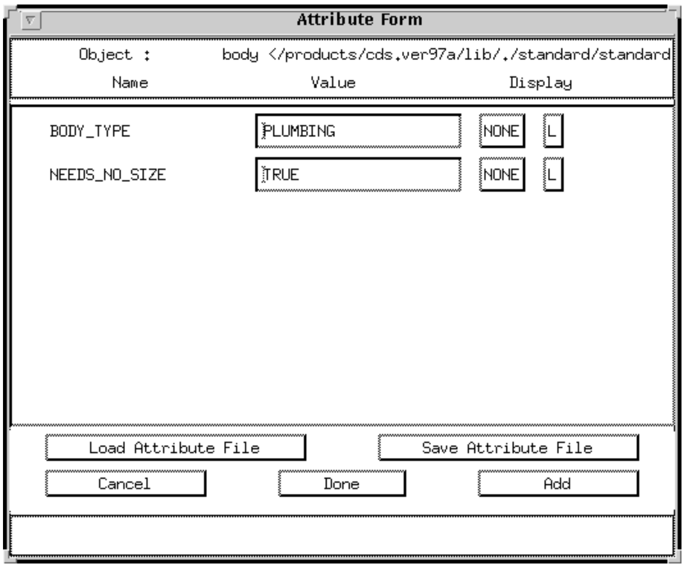
Note: The terms "properties" and "attributes" are used interchangeably.

Adding Xilinx Properties

If you are not familiar with how to add properties within Concept, refer to the section "Using Properties" in Chapter 3, *Creating a Design* in the Cadence document, *Concept Schematic User Guide*.

Following is a brief description of how to add a property to a component in Concept.

1. Within Concept, click on the Attribute button on the left-hand side of the editor window.



Attribute Form

Object : body </products/cds.ver97a/lib/./standard/standard

Name	Value	Display	
BODY_TYPE	PLUMBING	NONE	L
NEEDS_NO_SIZE	TRUE	NONE	L

Figure 3-2 Attributes Form

2. Click on the object to which you want to add a property.
3. When the Attribute Form displays, click the Add button in the lower right hand corner.
4. Scroll down to the bottom of the attribute list and enter the Name and Value in the new fields.

Note: Properties must always be entered as NAME-VALUE pairs.

The buttons in the third column of the Attribute Form indicate what will be displayed in the schematic window—the property NAME only, VALUE only, or BOTH. The default is set to VAL, that is, only the value of the property displays on the schematic. For clarity, Xilinx recommends that you display *both* the NAME and the VALUE of the property.

5. Click Done when you have finished adding properties.

Rules and Restrictions For Using Xilinx Properties in Concept

The following subsections describe rules, limitations, and restrictions for using Xilinx properties.

SIZE Property

The SIZE property is not supported for this release. This property has been removed from all the components in the libraries. When placing an array of bodies, you must use Iterated Instances instead.

To use iterated instances, you must either add the PATH property to a new body, or edit the existing PATH property on a body that has already been saved to the design:

PATH=*body_name*(*n*:*m*)

where *body_name* is the name of the body you want to replicate, and (*n*:*m*) represents the range of indices over which you want to replicate the symbol. For example, if n=3 and m=0, you are replicating the body four times (0 through 3).

For high-level functional modules, including registers, counters, adders, and memory, you can also use the LogiBLOX utility.

See the “Understanding Iterated Instances” section in the “Schematic Guidelines” chapter in the Cadence manual, *HDL Direct User Guid*e for more information.

CONCEPT2XIL Property Filter File

If a Xilinx property is entered on a schematic but not listed in the property filter file ($XILINX/cadence/data/xilinx.pff), the property will not be included in the EDIF file after CONCEPT2XIL translation. The xilinx.pff file is located in $XILINX/cadence/data.

The xilinx.pff property filter file also contains entries for the predefined TIMESPECS TS01 through TS10 and the predefined TIMEGRPS GRP01 through GRP10. If you need additional TSidentifier or TIMEGRP properties, you must add them to the property filter file.

To customize your own xilinx.pff file, perform the following steps:

1. Copy the xilinx.pff file from $XILINX/cadence/data to your design directory.

   cp $XILINX/cadence/data/xilinx.pff
   your_design_directory

Note: The xilinx.pff file that you want to use should be placed in the directory from which you want to run the CONCEPT2XIL command.

The xilinx.pff file located in the directory in which you run CONCEPT2XIL takes precedence over the xilinx.pff file located in $XILINX/cadence/data.

2. Edit the xilinx.pff file to include new properties. If your path is set up properly, the next time you start up Cadence, the xilinx.pff file in your Cadence working directory is used as the primary file.

   For more information on the format of the xilinx.pff property filter file, see the "XILINX.PFF Property Filter File Format" appendix.

   When you enter a new property in the xilinx.pff file, you must ensure that you have the proper format. Since the `set capslock_off` variable is set up in the startup.concept file, you must enter property names in lower case. Following are examples of correct and incorrect entries:

ts01:	"ts01"	String	NORMAL; {Timespec}
TS02:	"ts02"	String	NORMAL;
ts03:	"TS03"	String	NORMAL;
TS04:	"TS04"	String	NORMAL;
TS05:	"TS05"	String	NORMAL;

   The first column corresponds to the name of the property, while the second column corresponds to the output format of the property when written out to the EDIF netlist.

   This is what you will see in the .EDF netlist written by CONCEPT2XIL:

```
(property ts01 (string "dc2s=20ns")
(property TS03 (string "from:pads:to:pads:30ns")
```

   In the schematic file, logic.1.1, you may see the following:

```
FORCEPROP 2 LAST TS01 dc2s=20ns
FORCEPROP 2 LAST TS02 DP2P=30ns
FORCEPROP 2 LAST TS03 from:pads:to:pads:30ns
FORCEPROP 2 LAST TS04 from:ffs:to:pads:20ns
FORCEPROP 2 LAST TS05 from:pads:to:ffs:20ns
```

TS02, TS04, and TS05 are missing from the .edf file because these entries are incorrectly entered in the xilinx.pff file, that is, their names are entered in upper case instead of lower case.

For the property to be translated to EDIF, its name must be in lower case in the first column of xilinx.pff. If it is in upper case, it will not get translated because the netlister will not find a match in the xilinx.pff file with what is written out to the logic drawing (schematic file)

The second column of the xilinx.pff file determines how the property will look when it is written to the .EDF file.

- If it is upper case in the 2nd column, it will get written out in upper case in the .EDF file.
- If it is lower case in the 2nd column, it will get written out in lower case in the .EDF file.

Xilinx Properties Without Values

In Concept, all properties are entered with the following syntax:

property_name=*value*

The Xilinx properties, such as COLLAPSE, DECODE, DOUBLE, FAST, ID, KEEP, MEDDELAY, NODELAY, NOREDUCE, SLOW, USE_RLOC, and WIREAND are inherently associated with a value of TRUE or FALSE, and are defined as Boolean properties in the xilinx.pff file. Set these Boolean properties to a value of TRUE, for example, FAST=TRUE.

Xilinx Properties on Pads

In Concept, pads are comment bodies; they only exist for documentation purposes.

Xilinx properties on PADS will not be translated into the EDIF netlist when running CONCEPT2XIL. As a workaround, you must use LOC properties attached to the corresponding I/O buffers. The type of LOC properties you must specify on I/O buffers depends on the Package type. For example, normally LOC=P17 implies Pin 17, but for Pin and Ball Grid arrays (PG and BGA packages), a pin mnemonic such as B3 or T1 is used. Refer to the "Attributes, Constraints, and Carry Logic" chapter in the Xilinx *Libraries Guide*.

Supported Xilinx Properties

For a complete description of the Xilinx properties that are supported in this release, refer to the "Attributes/Logical Constraints" section in the *Libraries Guide*.

Obsolete Xilinx Properties

The following table lists Xilinx properties that were valid in previous software versions but are no longer supported in the new design flow.

Property	Description
INPUT_LOAD	Internal property used by xcdsprep
OUTPUT_LOAD	Internal property used by xcdsprep
OUTPUT_MODE=CAP	Not supported in this release
OUTPUT_MODE=RES	Not supported in this release
PARAM=C, G, I, K, L, N, SC, W	Not supported in this release
PIN	Used by xnfmerge
PROHLOC	Replaced by the new attribute PROHIBIT
PR_PARAMS=S	Replaced by the property S=TRUE
PR_PARAMS=X	Replaced by the property KEEP=TRUE
SLEW_RATE=FAST	Replaced by the property FAST=TRUE
SLEW_RATE=SLOW	Replaced by the property SLOW=TRUE
VHDL_MODE	Internal property used by xcdsprep
User defined properties other than user-defined TSPEC and TNM properties.	Not supported in this release

Entering Timing Specifications in Schematics

The Cadence netlist writer program (CONCEPT2XIL) converts all property names to lower case letters, and the Xilinx netlist reader EDIF2NGD then converts the property names to uppercase letters. To ensure references from one constraint to another are processed correctly, observe these guidelines:

- A TSidentifier name should contain only upper case letters on a Concept schematic (TSMAIN, for example, but not TSmain or TSMain).
- If a TSidentifier name is referenced in a property value, it must be entered in upper case letters. For example, the TSID1 in the second constraint below must be entered in upper case letters to match the TSID1 name in the first constraint.

```
TSID1 = FROM: gr1: TO: gr2: 50;
TSMAIN = FROM: here: TO: there: TSID1: /2;
```

Creating New Groups from Existing Groups

The Cadence netlist writer program (CONCEPT2XIL) converts all property names to lower case letters, and the Xilinx netlist reader EDIF2NGD then converts the property names to upper case letters. To ensure references from one constraint to another are processed correctly, observe these guidelines:

- Group names should contain only upper case letters on a Cadence schematic (MY_FLOPS, for example, but not my_flops or My_flops).
- If a group name appears in a property value, it must also be expressed in upper case letters. For example, the GROUP3 in the first constraint example must be entered in upper case letters to match the GROUP3 in the second constraint.

```
TIMEGRP GROUP1 = gr2: GROUP3;
TIMEGRP GROUP3 = FFS: except: grp5;
```

Attaching Signal Names

If you do not attach signal names to signals in Concept, HDL Direct will not work properly. Ensure that you attach signal names to all schematic signals.

Creating Bus Taps

When tapping signals off a bus in Concept, unless you maintain the same name for the bits you tap off as the name of the parent bus, the taps will not be interpreted properly by NGDBuild. For example, if a bus is named “foo<2..0>”, the only allowed names of the bits tapped off of the bus are foo<2>, foo<1>, and foo<0>. Any deviation from these names requires that you insert a buffer between the bit tapped off the bus and the new name.

The typical error message issued by NGDBuild / EDIF2NGD when bits are renamed in the process of tapping them off the bus is:

```
Fatal_Error: Duplicate based:basedport.c port xx in
cell alias_bit
```

The error indicates that the bits tapped off the bus are considered to have been renamed.

In the following figure, the bus taps are legitimate configurations within Concept, but will not be processed correctly by NGDBuild.

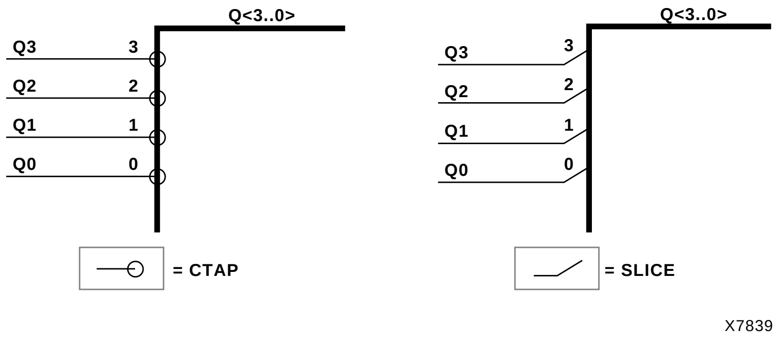


Figure 3-3 Unacceptable Bus Tap Naming For NGDBuild

The next figure illustrates how to use angle brackets to modify the previous bus taps for use with NGDBuild.

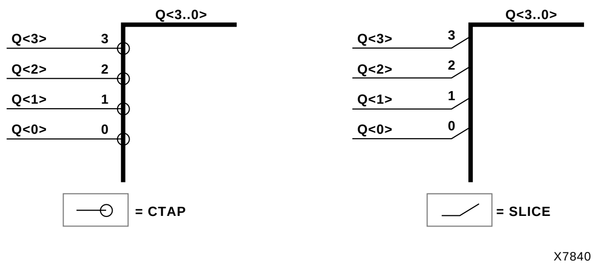


Figure 3-4 One Correct Method of Creating NGDBuild-Legal Bus Taps

The next figure illustrates another unacceptable method for tapping three nets with different names into a bus. Although this method is valid in Concept, NGDBuild will not process this configuration properly.

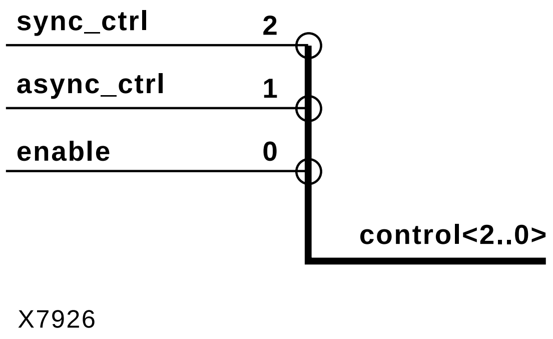


Figure 3-5 Unacceptable Bus Tap Naming

The next figure illustrates how you can modify the tapped bits of the bus in the "Unacceptable Bus Tap Naming" figure by inserting a non-inverting buffer (BUFF) between the bus bit and the new name.

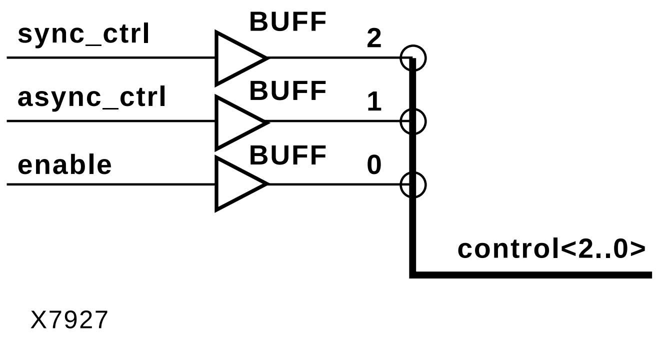


Figure 3-6 Another Correct Method of Creating NGDBuild-Legal Bus Taps

For information on how to set up bus taps, see the following sections in Chapter 3, Creating a Design, in the Cadence manual, *Concept Schematic User Guide*.

- "Using Bus Taps" section
- "Tips on Bus Taps" section
- "Using Other Bus Taps" section

Using the BSCAN Symbol

The BSCAN symbol indicates that boundary scan logic should be enabled after the FPGA configuration is complete. It also provides optional access to some special features of the XC4000E/L, XC4000EX/XL/XV, and XC5200 boundary scan logic.

If you are using the XC4000E/L, XC4000EX/XL/XV, or XC5200 device family, you can also use the BSCAN design element in your design. For details, see the "BSCAN" section of the Xilinx *Libraries Guide*.

Note: There is no simulation modelling for Boundary Scan.

Using the STARTUP Symbol

The Xilinx STARTUP macro gives you external control of Global Set / Reset (GSR) for 4000E, 4000EX / XL / XV and Global Reset (GR) for XC5200, global tri-state control (referenced as GTS in the Verilog files), and the user configuration clock. Instantiating a STARTUP symbol in your design allows you to control these functions from an external pin. Its use is optional; however, regardless of whether you use the STARTUP symbol in your design or not, you should trigger Global Set / Reset and global tri-state control in your testbench file during simulation. Refer to "Setting Global Set / Reset and Tri-state Signals (FPGAs)" section of the "Design and Simulation Techniques" chapter.

If you are using the XC4000E / L, XC4000EX / XL / XV, or XC5200 device family, you can use the STARTUP macro with your design. For details, see the "STARTUP" section in the Xilinx *Libraries Guide*.

Using the CONFIG Symbol to Specify Part Type

Properties attached to the CONFIG symbol dictate how the implementation tools should process the design.

In Concept, properties must be attached to a body; they cannot be attached to a "Schematic" or 'Sheet". To specify global properties such as part type, you must attach them to the CONFIG symbol.

To specify the Xilinx part type in your schematic, you designate the part type as a PART property on the CONFIG symbol in the top-level schematic. The PART information is passed from the EDIF file to NGDBuild and MAP. This means that you do not need to run NGDBuild or MAP with the -p option unless you want to override the PART information included in the CONFIG symbol.

To add the PART property to the CONFIG symbol, perform the following steps.

1. Within Concept, select Add Part.

2. When the Library Browser dialog box displays, click the button to the left of the Library field and select the Xilinx architecture-specific library (xce4000x, for example).
3. Select CONFIG from the list of components and place this symbol in the schematic.
4. Display the attributes currently attached to the CONFIG symbol by clicking on the Attribute button and then clicking on the CONFIG symbol.
5. Click ADD. Enter the PART designation in the Name field. Enter the Xilinx part name in the Value field (for example, 4028EXPG299-3). Other forms allowed are as follows:

   ```
   XC4028EX-3-PG299
   XC4028EX-PG299-3
   4028EX-PG299-3
   4028EX-3-PG299
   ```

   The part name can be entered with or without the speed grade. If no speed grade is specified, then a default value is used. Refer to the "Attributes, Constraints, and Carry Logic" chapter in the Xilinx *Libraries Guide* for more information on specifying part type.

Using HDL Direct Methodology

When you use Xilinx/Concept, you must enable the HDL Direct option to ensure that your design is written out properly for the CONCEPT2XIL netlister.

When you add the appropriate lines to your startup.concept file before you start Concept, HDL Direct will run automatically when you save your design. Refer to the "Concept Setup Library Files" section of the "Getting Started" chapter in the *Concept Schematic User Guide* for details. Also see the section on "Requirements for HDL Direct Compliance" in the *HDL Direct User Guide*.

Creating Bodies for Non-Schematic Design Blocks

You must also create a body for each non-schematic block that is contained within your schematics. If you do not create and instantiate a body for a non-schematic block in your schematic, it will not be properly netlisted by CONCEPT2XIL or NGDBuild. The following sections describe how to create a body for your non-schematic blocks using Concept's genview utility.

For details on how to create symbols using genview, refer to the "Generating Design Views", "Using Genview for Top-Down Design", and "Using Genview for Bottom-Up Design" sections in the "Creating Blocks" chapter of the *Concept Schematic User Guide*.

Creating a Body for a Verilog Netlist

To incorporate a Verilog netlist into your design schematic, you need to create a body for that block. Follow these steps to create a body for each block.

Copy the Verilog netlist for your subblock into the project directory, and type the following in the command line window within Concept:

`genview -i` *module_name*.**`v -v logic body verilog`**

This tells Concept to generate a body view for a module named *module_name* from the Verilog netlist, and to copy the associated Verilog netlist to a file named verilog.v in the logic view for this module. Genview also adds a reference to this new block in your .wrk file. This reference to the new block in your .wrk file allows you to access this body file for use in your schematic.

module_name.v may be a Verilog .v file generated by LogiBLOX for functional simulation. See the "Processing Designs with LogiBLOX Components" appendix for details.

Once the module has been generated, you must also edit the resulting verilog.v file in the *logic* subdirectory of the new module directory, and add the definition, **`parameter cds_action="ignore";`** after the module declaration.

Example:

```
module mycount (load, up_dn, clk_en, clock,
async_ctrl, term_cnt, d_in, q_out

  );

parameter cds_action="ignore"; // <---- Add this line

    input load;

    input up_dn;

    input clk_en;
    input clock;
    input async_ctrl;
    output term_cnt;
    input [3:0] d_in;
    output [3:0] q_out;
```

The addition of this parameter definition is a required step. The cds_action=”ignore” parameter tells CONCEPT2XIL that it should not try to look for additional design hierarchy beyond this block, because there are no underlying schematics for the block.

You now have a body for your logic block that you can use.

Generating a Body for a Schematic

To generate a body from a schematic:

1. Within Concept, edit the schematic.

   `edit` *schematic_name*

2. Type **`genview`** in the command window.

   A body drawing is automatically created for your schematic.

Generating a Body for an XNF, NGO, or EDIF File

You can manually create a body for an XNF, NGO, or EDIF file using the Concept **badd** command. For details, refer to the section, “Creating a Block Diagram” in *Chapter 5, Creating Blocks* of the Cadence manual, *Concept Schematic User Guide*.

Verilog HDL Design Entry

Verilog HDL Synthesis using Cadence Synergy is supported by Cadence Design Systems. For a complete description of Synergy, refer to the following Cadence manuals:

- *Synergy HDL SmartBlocks Interface User Guide*
- *Synergy HDL Synthesizer and Optimizer Modeling Style Guide*
- *Synergy HDL Synthesizer and Optimizer Tutorial*

- *Synergy Libraries Development Guide*
- *Synergy SmartBlocks Data Book*

The following figure shows the design flow for Synergy. For details, contact Cadence.

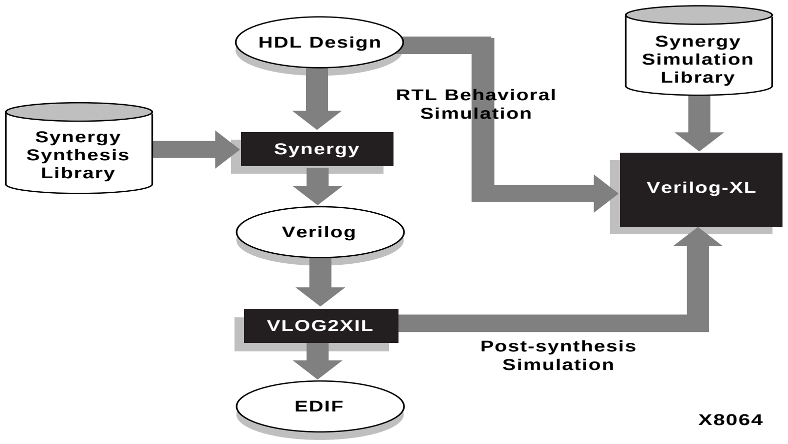

HDL Design
Synergy Simulation Library
RTL Behavioral Simulation
Synergy Synthesis Library
Synergy
Verilog-XL
Verilog
VLOG2XIL
Post-synthesis Simulation
EDIF
X8064

Figure 3-7 HDL Synthesis Design Flow

Verilog HDL Synthesis flows using Synopsys are shown in the "Synopsys / Verilog Design Flow" appendix and discussed in detail in the Xilinx *Synopsys (XSI) Interface/Tutorial Guide.*

Translating Your Design

Once you have completed and saved your design schematics, you will automatically have an HDL Direct-generated Verilog netlist for every block in your design. The next step is to use the CONCEPT2XIL program with the -sim_only option to configure these Verilog netlists so that you can perform a functional simulation. Refer to the "Functional Simulation" chapter for details.

After completing functional simulation, use CONCEPT2XIL to prepare your design for use with the Xilinx design implementation tools. Refer to the "Converting the Concept Design to an EDIF File" section of the "Design Implementation" chapter for details.

Chapter 4

Functional Simulation

This chapter contains the following sections:

- "Introduction" section
- "Unified Library Based Functional Simulation" section
- "SIMPRIM Library Based Functional Simulation" section

Introduction

This chapter explains how to perform functional simulation on your designs using Verilog-XL. Functional simulation provides an effective means of identifying logic errors in your design before the design is implemented in a Xilinx device. Since timing information for the design is not available in a functional simulation, you should conduct your functional simulation using unit delays. Doing functional simulation before routing your design saves debugging time later in the design process by verifying that your design is functionally correct.

This chapter describes two types of functional simulation.

- Unified Library Based Functional Simulation

  This simulation is performed on the output Verilog (.V) files from Concept HDL Direct and the Verilog configuration file (.VF) for a purely schematic design, following a post-processing step by CONCEPT2XIL. See the "Unified Library Based Functional Simulation" figure.

- SIMPRIM-Based Functional Simulation

  This simulation is performed on .v and .tv (option -tf for NGD2VER) files created by NGD2VER on the NGD file output from NGDBuild. SIMPRIM-based functional simulation can be performed on any type of design. (On mixed mode designs, it is the only method available.) See the "SIMPRIM-Based Functional Simulation" figure.

You can usually perform both types of simulation on either FPGA or CPLD designs.

Note: There is no simulation modelling for Boundary Scan.

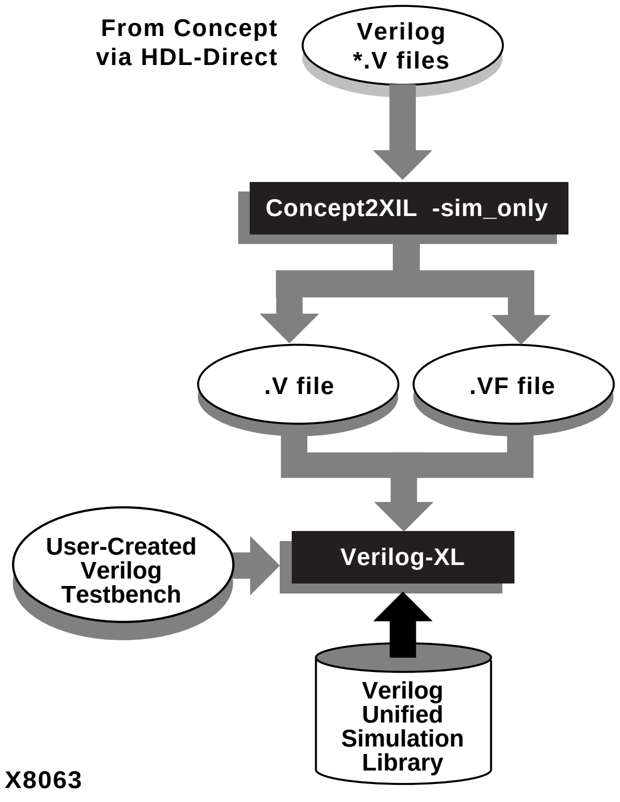

From Concept via HDL-Direct
Verilog *.V files
Concept2XIL -sim_only
.V file
.VF file
User-Created Verilog Testbench
Verilog-XL
Verilog Unified Simulation Library
X8063

Figure 4-1 Unified Library Based Functional Simulation

Unified Library Based Functional Simulation

This section describes Unified Library based functional simulation for pure Concept schematic designs and Concept designs with Logi-BLOX modules.

Pure Concept Schematic Without LogiBLOX Elements

The functional simulation of a pure schematic design requires that HDL Direct be enabled when you save the various schematic blocks in your design. With HDL Direct enabled, a Verilog HDL view is automatically generated for a schematic or symbol body when you save it. You then run CONCEPT2XIL with the -sim_only option to resolve the design hierarchy and generate the required *design*.vf Verilog configuration file and *design*.v file containing the required global modules.

Steps:

1. Enable HDL Direct
2. Save schematic
3. Run the command:
   concept2xil -sim_only -family *architecture design_name*
4. Create testbench file manually
5. Simulate design

Creating HDL Views for the Design/ Netlisting the Design

When you save a schematic in Concept with HDL Direct enabled, Concept creates a directory for the schematic with an underlying *logic/* directory. The *logic/* directory contains a Verilog HDL view for that block of your design. You must then run CONCEPT2XIL -sim_only to generate the required files for functional simulation.

Following is the CONCEPT2XIL syntax:

`concept2xil -family` *technology* **`-sim_only`** *design_name*

The -family option specifies the library that should be used (for example, xce4000x). To target the Xilinx XC4000EX, XC4000XL and XC4000XV architectures, specify "xce4000x" as the target library.

The -sim_only option tells CONCEPT2XIL to generate only the files needed for functional simulation.

The CONCEPT2XIL program generates the following two files in the Xilinx run directory (default is *xilinx.run*)

- *design*.vf, a Verilog configuration file
- *design*.v

The configuration file, *design*.vf, contains information on the location of the Verilog netlists for each block of the design.

The *design*.v file defines the following modules which, although not supported by CONCEPT2XIL, are nevertheless written out by the embedded Cadence EDIF netlist writer:

- alias_bit — a module that is used by HDL Direct if you rename a net. This happens most often when you tap bits off a bus in conjunction with a TAP, SLICE, or MERGE body. Note that alias bits are *not* supported by the EDIF netlister. Because alias bits are not supported, if you rename a net (for instance when you tap a bit off a bus and do not keep the same bus name), you must separate the original net name and the new net name in your schematic by inserting a buffer (BUFF) component between the new and existing nets.
- alias_vector — defines a module that is used if you rename buses. Bus renaming is usually done in conjunction with a SLICE or MERGE body. Alias vectors are *not* supported by the M1 EDIF netlister.

Creating a Testbench File

You will also need to create a testbench file in a text editor for your functional simulation, usually named *design*.stim.

Note: The term "testbench" and "test fixture" are used interchangeably.

In the M1.4 Verilog interface, a new methodology is used to support the global (set/) reset and global tri-state signals in the FPGAs, and global reset in the CPLDs. The procedure for driving these global signals is described in detail in the "Setting Global Set/Reset and Tri-state Signals (FPGAs)" section of the "Design and Simulation Techniques" chapter. Most of what is required is implemented automatically when you use the testbench template generated by NGD2VER; however, if you are performing Unified Library functional simulation directly from CONCEPT2XIL, a testbench template does not get generated automatically. You may either create the testbench file manually, or, as a workaround, process your design through

NGDBuild, then run NGD2VER -tf -ul on the *design*.ngd file to generate a template that you can use for your testbench.

Example:

ngdbuild -p 4028ex-3-pg299 *design*

ngd2ver -tf -ul *design***.ngd**

For an example of a testbench file, see the "Sample Test Fixture - XC4000EX Unified Library Functional Simulation (GSR and GTS simulation)" section of the "Files" appendix.

Running the Functional Simulation

To functionally simulate the design with Verilog-XL, you must:

- Specify the full pathname of

  a) a copy of the testbench file named *design*.stim

  b) the design Verilog netlist (*design*.v)

  c) the Verilog configuration file (*design*.vf)

  It is usually simpler to navigate to the directory in which these files reside and run the simulation locally.

- It is also recommended that you specify the +delay_mode_unit option to do a unit delay simulation. This prevents race conditions if there are any feedback loops in your design.

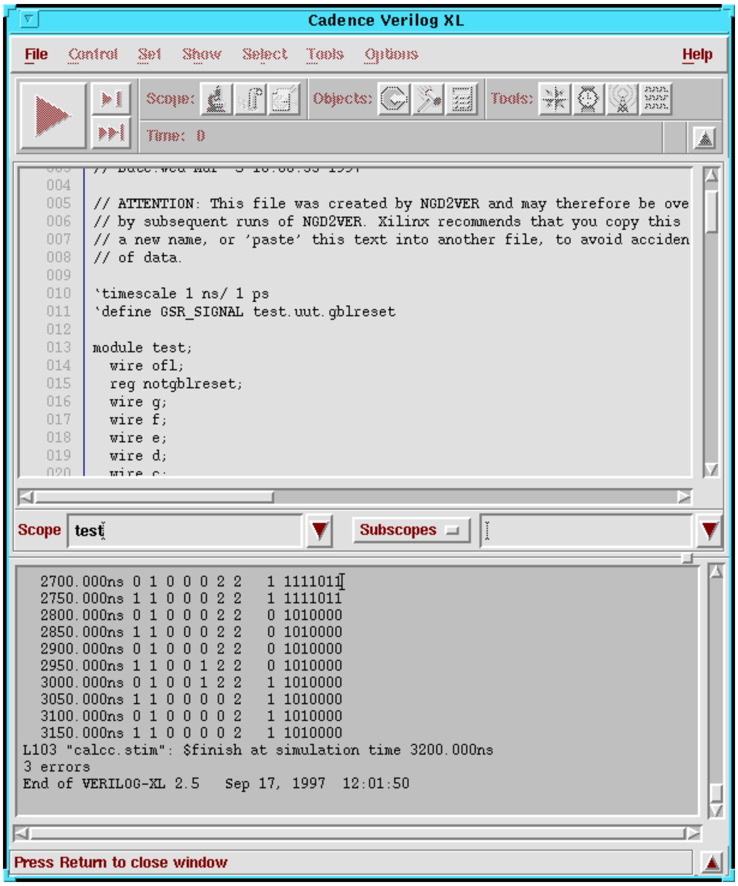
Following is the syntax:

```
verilog +delay_mode_unit design.stim \
    full_path_to_design_name.v \
    -f full_path_to_design_name.vf
```

(The "\"at the end of a line indicates that the line following the current one can be typed on the same command line.)

If available, it is recommended that you specify the +gui option to invoke the Verilog Environment. (+gui is supported beginning in the 97A release.)

In the following figure, the Verilog-XL Control Window (VCW) displays when you use the +gui option.



```
// ATTENTION: This file was created by NGD2VER and may therefore be ove
// by subsequent runs of NGD2VER. Xilinx recommends that you copy this
// a new name, or 'paste' this text into another file, to avoid acciden
// of data.

`timescale 1 ns/ 1 ps
`define GSR_SIGNAL test.uut.gblreset

module test;
  wire ofl;
  reg notgblreset;
  wire g;
  wire f;
  wire e;
  wire d;
  wire c;
```

```
  2700.000ns 0 1 0 0 0 2 2   1 1111011
  2750.000ns 1 1 0 0 0 2 2   1 1111011
  2800.000ns 0 1 0 0 0 2 2   0 1010000
  2850.000ns 1 1 0 0 0 2 2   0 1010000
  2900.000ns 0 1 0 0 0 2 2   0 1010000
  2950.000ns 1 1 0 0 1 2 2   0 1010000
  3000.000ns 0 1 0 0 1 2 2   1 1010000
  3050.000ns 1 1 0 0 0 0 2   1 1010000
  3100.000ns 0 1 0 0 0 0 2   1 1010000
  3150.000ns 1 1 0 0 0 0 2   1 1010000
L103 "calcc.stim": $finish at simulation time 3200.000ns
3 errors
End of VERILOG-XL 2.5   Sep 17, 1997  12:01:50
```

Figure 4-2 Part of the Verilog Environment

A convenient way of running a Verilog simulation is to create a script containing a command line with all the required options. To run the simulation, simply invoke the script.

Example:

For a design named "calc", targeting an XC4000EX component, say you generate a Verilog netlist called "calc.v" with CONCEPT2XIL and create a testbench file called "calc.stim". Assume your simulation library is located in the directory, $XILINX/cadence/data/verilogxce4000x.

To run a functional simulation, you would navigate to the xilinx.run directory and run the simulation using this command line:

```
verilog +delay_mode_unit calc.stim calc.v \
-f calc.vf
```

To specify the simulation library, $XILINX/cadence/data/verilogxce4000x, you can either add the following `uselib directive to your .v file.

`uselib dir = *explicit_path_to_Xilinx***/cadence/data/verilogxce4000x libext=.v**

or you can specify the library path as a-y command line option to Verilog -XL.

Adding SimWave Support to the Testbench File

Your simulation waveforms are displayed by a waveform display application that is separate from Verilog-XL. Given the appropriate directives, Verilog-XL writes the waveform data to a simulation history directory (*design*.shm). The waveform viewer application (usually SimWave) reads this data from the simulation history database and displays the waveforms.

If you wish to view your simulation waveforms graphically while performing your functional simulation, you must add an “initial” block to the testbench file containing directives to create a simulation history database for the waveform viewer.

Note: NGD2VER will add this “initial” block to a stimulus template file (*designf*.tv) when invoked with the -tf option. For pre-NGDBuild functional simulation flows, this block must be added to your testbench manually.

Sample "initial" block adding Simulation History Manager support:

```
initial
begin
   $shm_open(“calc.shm”);
   $shm_probe(“AS”);
end
```

The $shm_open command creates the database directory, “calc.shm”. $shm_probe(“AS”) directs the simulation history manager to probe *all* signals, thus making them available for viewing in the waveform viewer.

To invoke SimWave, enter the following command:

```
simwave &
```

For a complete description of SimWave, refer to the "Using SimWave" section of the "Schematic Design Tutorial" chapter.

Global Reset

Global reset should always be toggled at the beginning of a simulation to ensure that all flip-flops and latches initialize to a known state. See the "Setting Global Set / Reset and Tri-state Signals (FPGAs)" section of the "Design and Simulation Techniques" chapter for information on toggling global reset for XC3000A / L, XC3100A / L, XC4000 / E / EX / XL / XV, XC5200, and XC9500 devices.

Pure Concept Schematic Designs With LogiBLOX

Refer to the "Processing Designs with LogiBLOX Components" appendix for detailed information.

Mixed Mode Designs

Mixed mode designs can be functionally simulated using a slightly different flow. If the top level of your design is a schematic, each non-schematic block (from Synopsys or other third party tool) must be available in, or processed down to XNF, .NGO, or .EDF format. Each non-schematic block must also be represented with its own body (or symbol) in the design schematics. The steps are as follows:

1. Process all non-schematic blocks to XNF, .NGO or .EDF format using the appropriate translation tool. For example, a Synopsys block must be processed down to XNF format.

2. Run NGDBuild to merge the schematics with the non-schematic blocks.

   `ngdbuild -p` *part_type design_name*

   The –p option specifies the Xilinx device or architecture into which your design will be implemented. The –p option may specify an architecture only (for example, XC4000EX), a complete part specification (device, package, and speed-- for example, xc4028ex-pg299-3), or a partial specification (device and package only--for example, XC4028EX)

The input *design_name* may be an XNF or EDIF 2.0.0 netlist. If the input netlist is in a format that the Netlister Launcher recognizes, the Netlister Launcher reads in the netlist, determines the format of the input netlist, and then invokes the appropriate netlist reader, EDIF2NGD or XNF2NGD.

3. Run NGD2VER to generate the functional simulation netlist.

   `ngd2ver -tf -ul` *design_name*

   The -tf option generates a testbench template file (.tv) for you containing an instantiation of your design as well as support for the Cadence Simulation History Manager.

   Make a copy of this testbench file, name it (*designf*.stim), and use it as a starting point for your simulation stimulus. For an example of a testbench file, see the “Sample Test Fixture - XC4000EX Post-NGDBuild Simulation (GSR and GTS simulation)” section of the “Files” appendix.

   The -ul option directs NGD2VER to write out a `uselib directive which tells Verilog-XL where to load the SIMPRIM based simulation libraries when compiling the simulation.

   See the “Design and Simulation Techniques” chapter for information on driving the global signals GSR, GR, GTS and PRLD.

Running the Simulation

To functionally simulate the design with Verilog-XL, you must:

- Specify the full pathname of

  a) a copy of the testbench file (*designf*.stim)

  b) the design Verilog netlist (*designf*.v)

  It is usually simpler to first navigate to the directory where these files reside and run the simulation locally.

- It is also recommended that you specify the +delay_mode_unit option to do a unit delay simulation. This option prevents race conditions if there are any feedback loops in your design.

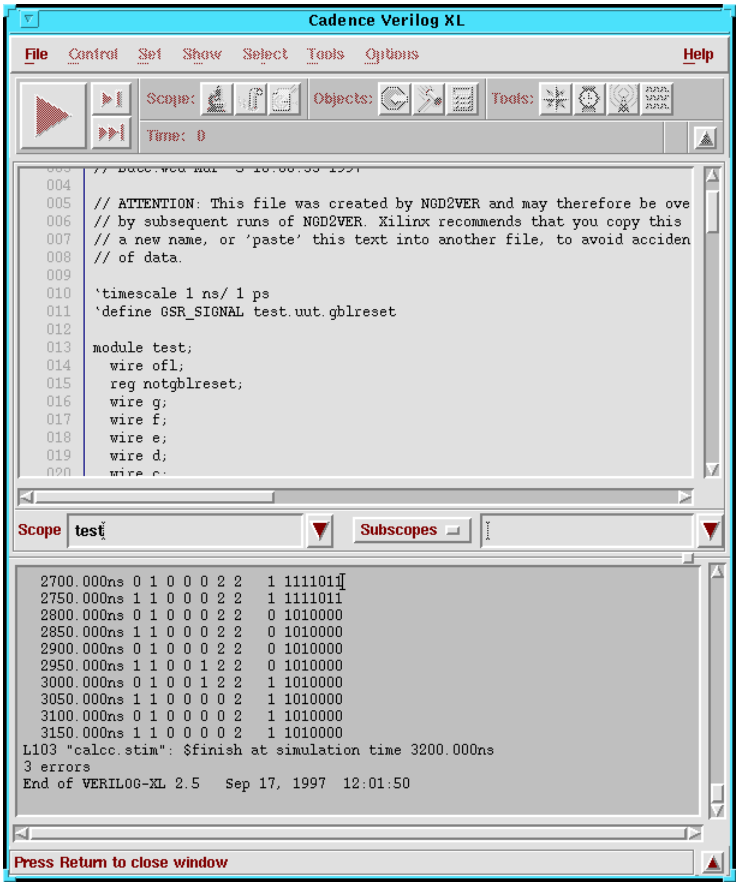
Following is the syntax:

```
verilog +delay_mode_unit design_namef.stim \
  full_path_to_design_namef.v
```

(The “\”at the end of a line indicates that the line following the current one can be typed on the same command line.)

If available, it is recommended that you specify the +gui option to invoke the Verilog Environment.

In the following figure, the Verilog-XL Control Window (VCW) displays when you use the +gui option.



```

// ATTENTION: This file was created by NGD2VER and may therefore be ove
// by subsequent runs of NGD2VER. Xilinx recommends that you copy this
// a new name, or 'paste' this text into another file, to avoid acciden
// of data.

`timescale 1 ns/ 1 ps
`define GSR_SIGNAL test.uut.gblreset

module test;
  wire ofl;
  reg notgblreset;
  wire g;
  wire f;
  wire e;
  wire d;
```

```
  2700.000ns 0 1 0 0 0 2 2   1 1111011
  2750.000ns 1 1 0 0 0 2 2   1 1111011
  2800.000ns 0 1 0 0 0 2 2   0 1010000
  2850.000ns 1 1 0 0 0 2 2   0 1010000
  2900.000ns 0 1 0 0 0 2 2   0 1010000
  2950.000ns 1 1 0 0 1 2 2   0 1010000
  3000.000ns 0 1 0 0 1 2 2   1 1010000
  3050.000ns 1 1 0 0 0 0 2   1 1010000
  3100.000ns 0 1 0 0 0 0 2   1 1010000
  3150.000ns 1 1 0 0 0 0 2   1 1010000
L103 "calcc.stim": $finish at simulation time 3200.000ns
3 errors
End of VERILOG-XL 2.5   Sep 17, 1997  12:01:50
```

Press Return to close window

Figure 4-3 Verilog-XL Control Window

A convenient way of running the simulation is to create a script containing a command line with all the required options. To run a simulation, simply invoke the script.

Example:

For a design named “calc”, targeting an XC4000EX component, say you generate a Verilog netlist called “calcf.v” with NGD2VER, and create a testbench file called “calcf.tv”.

```
ngd2ver -tf -ul calc.ngd calcf
```

Copy the testbench template file "calcf.tv" to a file named "calcf.stim".

To run a functional simulation, you would navigate to the xilinx.run directory and enter the following command:

```
verilog +delay_mode_unit calcf.stim calcf.v
```

To invoke SimWave, enter the following command:

```
    simwave &
```

For a complete description of SimWave, refer to the “Using SimWave” section of the “Schematic Design Tutorial” chapter.

Global Reset

Global reset should always be toggled at the beginning of a simulation to ensure that all flip-flops and latches initialize to a known state. See the “Setting Global Set/Reset and Tri-state Signals (FPGAs)” section of the “Design and Simulation Techniques” chapter for information on toggling global reset for XC3000A/L, XC3100A/L, XC4000/E/EX/XL/XV, XC5200, and XC9500 devices.

The XC3000A/L architecture always contains a GR (Global Reset) port for SIMPRIMS-based functional simulation. However, for Unified library-based functional and timing simulation, GR is hidden for this architecture. Therefore, you cannot use the same testbench file for Unified library simulation and SIMPRIM-based simulation.

SIMPRIM Library Based Functional Simulation

SIMPRIM-based functional simulation requires an NGD file as a starting point. You can use NGDBuild to generate an NGD file from the CONCEPT2XIL EDIF file generated by the CONCEPT2XIL command:

ngdbuild -p **XC***xxxx design***.edf**

where *xxxx* corresponds to an architecture, for example, 4000EX, 3000, or part type (for example, 4028ex-pg299-3).

For details on how to use the command line version of NGDBuild, see the "NGDBuild" chapter of the *Design System Reference Guide*.

For details on using the Design Manager to run NGDBuild, see the "Using the Design Manager" chapter in the *Design Manager/Flow Engine Reference/User Guide*.

Once you have created an NGD file, you can conduct a SIMPRIM-Based functional simulation. The following diagram illustrates the design flow.

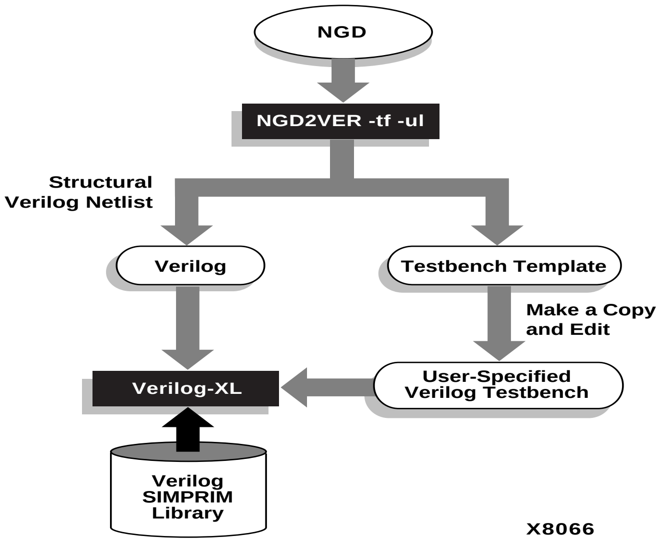


Figure 4-4 SIMPRIM-Based Functional Simulation

The basic simulation steps are as follows:

1. Use NGD2VER to create a structural Verilog netlist file and a test-bench stimulus template.
2. Copy the testbench stimulus template (*designf*.tv) to a file named "*designf*.stim" and use this copy as your user-specified testbench file. For an example of a testbench template file, see the "Sample Test Fixture - XC4000EX Post-NGDBuild Simulation (GSR and GTS simulation)" section of the "Files" appendix.
3. Run the Verilog-XL simulator in unit delay mode using the Xilinx-supplied Verilog SIMPRIM Library.

verilog +delay_mode_unit *design_namef***.stim**
*full_path_to_design_namef***.v**

Using NGD2VER

The NGD2VER program translates your NGD file to a Verilog HDL netlist which describes the design in terms of Xilinx simulation primitives. This Verilog netlist corresponds to an unmapped design which contains no timing information. You must use NGD2VER with the -tf and -ul options to create the appropriate files for use with the Cadence Verilog-XL simulator.

The following syntax translates your design to a Verilog file:

ngd2ver -tf -ul *infile***.ngd** *outfile*

- The -tf option generates a testbench template file. The output testbench template / file has a .tv extension. For an example of a testbench template file, see the annotated template in the "Sample Test Fixture - XC4000EX Post-NGDBuild Simulation (GSR and GTS simulation)" section of the "Files" appendix.
- The –ul option causes NGD2VER to write a `uselib directive pointing to the appropriate simulation library. This directive is written to the output Verilog file (.v). The path is written as shown below:

  `` `uselib dir=``*path_to_Xilinx*`/verilog/data libext=.vmd`

  where *path_to_Xilinx* is the location of the Xilinx software.

  If you do not specify the –ul option, the `uselib line will not be written into the Verilog file. The alternative to specifying the `uselib directive in your .v file is to specify the -y option when you invoke Verilog-XL and to specify the path to the simulation library (or libraries) as its argument.
- *infile*.**ngd** is the input NGD. If you enter a file name with no extension, NGD2VER will look for a file with an .nga extension having the name you specified. If you wish to translate an NGD file, you must enter the .ngd extension.
- *outfile* indicates the file name to which the output of NGD2VER will be written. Default is *infile*.v (*infile* is the root name of the input file).

To prevent overwriting an existing Verilog netlist generated by HDL Direct, it is recommended you specify the name of the post-NGDBuild Verilog functional simulation netlist as "infilef.v":

`ngd2ver -tf -ul` *infile*.**`ngd`** *infile***`f`**

The output Verilog netlist will be named infilef.v, and the testbench template file, infilef.tv.

Running a Verilog Functional Simulation

Edit a copy of the testbench stimulus template and add your stimulus to it to create a user-specified testbench file.

To run the Verilog simulation, use the procedure described in the "Running the Simulation" section of the "Mixed Mode Designs" section described earlier in this chapter. Use the filenames *designf*.v and *designf*.stim as the inputs to your simulation.

Global Reset

The XC3000A / L architecture always contains a GR (Global Reset) port for SIMPRIMS-based functional simulation. However, for Unified library-based functional and timing simulation, GR is hidden for this architecture. Therefore, you cannot use the same testbench file for Unified library simulation and SIMPRIM-based simulation.

Chapter 5

Design Implementation

This chapter contains the following sections:

- “Converting the Concept Design to an EDIF File” section
- “Implementing the Design” section

Once you complete functional simulation, you are ready to implement your design using the Xilinx core tools. The primary netlist format supported by the Xilinx core tools is EDIF 2.0.0, so you must first convert your design into an EDIF (.EDF) file using the CONCEPT2XIL command line script.

Note: XNF format netlists are also currently supported by the Xilinx Development implementation tools. However, support for XNF netlists will be phased out in future releases.

After you have created an EDIF file, you can then use either the NGDBuild command or the Xilinx Design Manager to generate an NGD file. You are then ready to implement your design. Implementation can be performed by the Xilinx Design Manager. See the *Design Manager/Flow Engine Reference/User Guide* for details. You may also use the Xilinx command line versions of the individual tools to implement your design from a UNIX shell. Refer to the *Development System Reference Guide* for details on FPGA designs. For CPLD designs, refer to the *CPLD Schematic Design Guide*.

Xilinx implementation tools first translate the design into a flattened or hierarchical netlist, then optimize, place, and route the design. The tools annotate delay data for timing simulation and generate physical (bitstream) design data for downloading. See the “FPGA Design Implementation” figure for an overview of the process for FPGAs. See the “CPLD Design Implementation” figure for an overview of the process for CPLDs.

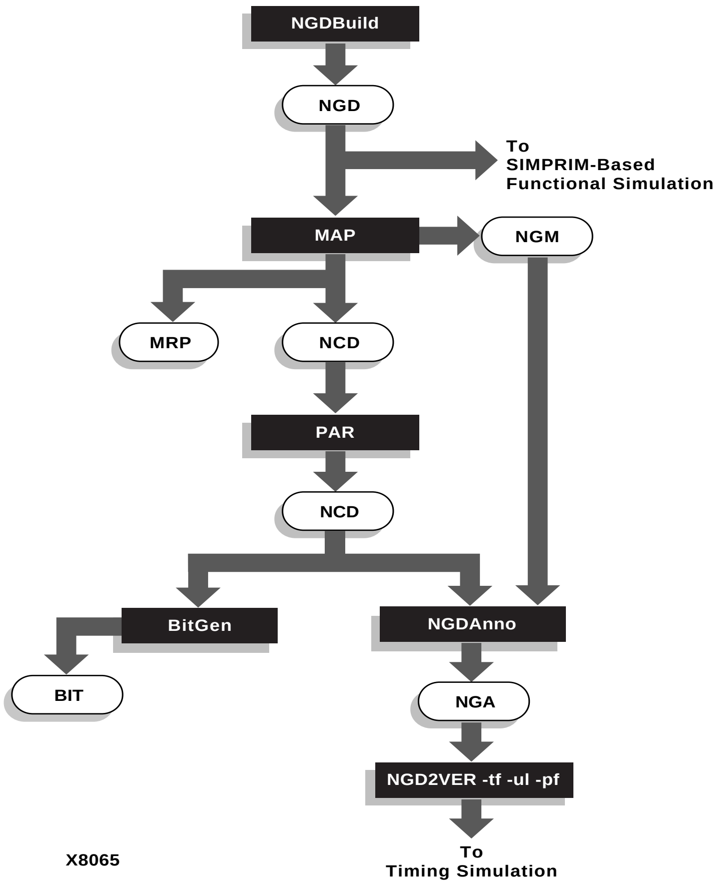


Figure 5-1 FPGA Design Implementation

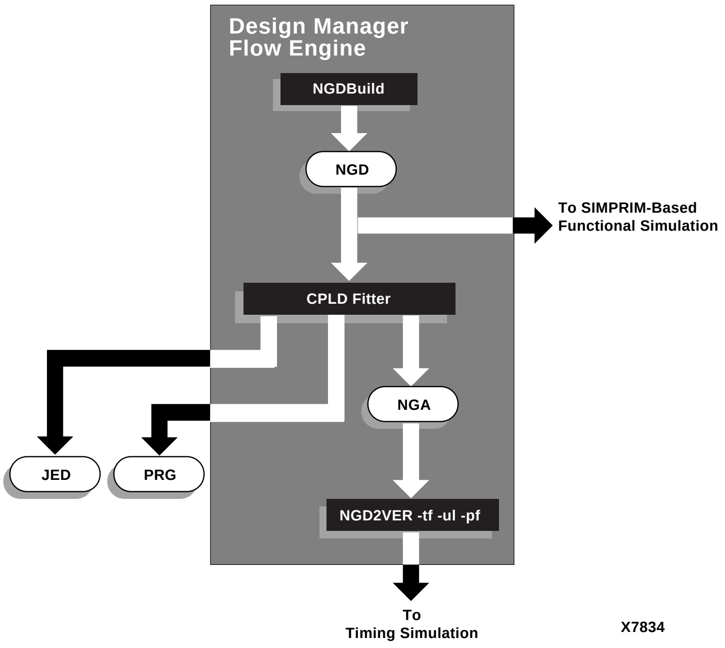


Figure 5-2 CPLD Design Implementation

This chapter describes how to convert your Concept Verilog file into an EDIF file using CONCEPT2XIL. The chapter also provides specific references to the command line design implementation tools for FPGAs and CPLDs.

Converting the Concept Design to an EDIF File

Once your Concept schematic is complete, you must convert the Verilog file (.V) that it outputs into an EDIF file before you can use the Xilinx design implementation tools. To translate the Verilog file, you must run the CONCEPT2XIL command line utility. The design flow for this conversion is shown in the following figure:

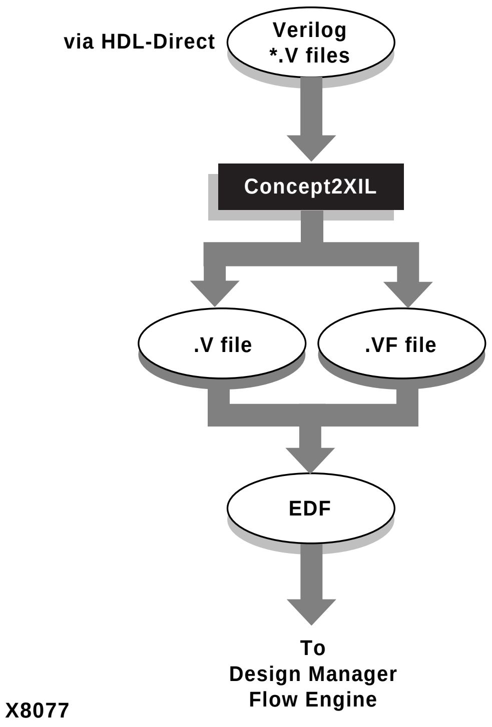


Figure 5-3 CONCEPT2XIL Design Flow

Following is the syntax for the CONCEPT2XIL command:

> **concept2xil** [**-cdslib** *lib_map_filename*]
> [**-gcmd** *command_filename*] [**-help**] [**-log** *log_filename*]
> [**-rundir** *run_directory*] **-family** *technology design_name*

- -cdslib *lib_map_filename* indicates the name of the library map file, which points to a technology-specific VAN-compiled Verilog library or libraries whose Verilog names are of the form, xce*xxx*_syn. The default is cds.lib. This parameter is optional.
- -gcmd *command_filename* specifies the name of the global.cmd command file, which indicates the Concept libraries available to the netlister. The default is global.cmd. This parameter is optional.

- -help allows you to obtain more information on CONCEPT2XIL and its options.
- -log *log_filename* specifies the name of the log file. The default is concept2xil.log. This parameter is optional.
- -rundir *run_directory* specifies the name of the directory in which CONCEPT2XIL is run. The default is xilinx.run. This parameter is optional.
- -family *technology* specifies the target architecture. Valid technologies are xce3000, xce4000e, xce4000x, xce5200, and xce9000. This parameter is optional.
- *design_name* is the input design name.

Here is an example:

```
concept2xil -family xce4000x block
```

Note: The CONCEPT2XIL command also generates a .v file and a .vf file as well as an EDIF (.edf) file. It will overwrite any existing .v and .vf files with the same name in the destination directory. Be sure that you do not overwrite any existing files that you want to preserve.

Once you have generated an EDIF file, you can use the Xilinx design tools to implement your design.

Implementing the Design

This section does *not* discuss the following design implementation tools, NGDBuild, Map, PAR, BitGen, NGDAnno, and CPLD Fitter (CPLD). The command line options for each of these tools can be found in the Xilinx manual, *Development System Reference Guide,* with the exception of the CPLD command, which is discussed in the "Fitter Command and Option Summary" appendix of the *CPLD Schematic Design Guide.*

You can find the following commands in the *Development System Reference Guide*:

- NGDBuild -- "NGDBuild" chapter
- Map -- "MAP—The Technology Mapper" chapter
- PAR (Place and Route) -- "PAR—Place and Route" chapter
- BitGen -- "BitGen" chapter

- NGDAnno -- “NGDAnno” chapter

You can also use the Xilinx Design Manager to implement your design. Refer to the “Using the Design Manager” chapter in the Xilinx manual, *Design Manager/Flow Engine Reference/User Guide*.

Chapter 6

Timing Simulation

Timing simulation verifies a placed and routed design by using worst-case routing and block delay information. The delay information is extracted from the routed design and passed to the back-annotated simulation netlist so it can be used during timing simulation. Timing simulation reduces the need for hardware debugging by determining whether or not the design works under worst-case conditions.

Timing simulation is also a useful tool for determining the device speed grade required for a particular application. Timing simulation verifies design functionality by using delay information from the routed NCD file created during design implementation.

This chapter describes how to prepare for timing simulation using the NGD2VER command and then conduct timing simulation within the Cadence simulation environment using Verilog-XL. It also describes how to load SimWave to view the simulation signals in a waveform format.

This chapter contains these sections.

- "Post-Map Timing Simulation (FPGAs)" section
- "Post-Implementation Timing Simulation" section

Note: There is no simulation modelling for Boundary Scan.

Post-Map Timing Simulation (FPGAs)

An optional simulation that you may perform for FPGAs is post-map timing simulation. The post-MAP simulation gives you a rough idea of whether your timing requirements can be met before delays due to routing are added.

Note: After mapping, Xilinx strongly recommends that you either perform a Static Timing Analysis using TRCE, or conduct a post-map simulation. A Static Timing Analysis can often be conducted faster than a simulation and is especially useful for first-time designs. Refer to the Xilinx *Development System Reference Guide* for information on running static timing analysis. On the other hand, if you already have a testbench file for post-NGDBuild functional simulation, you can use that testbench file for post-MAP and post-implementation timing simulation with some minor modifications.

The following figure provides a flow chart of post-map simulation.

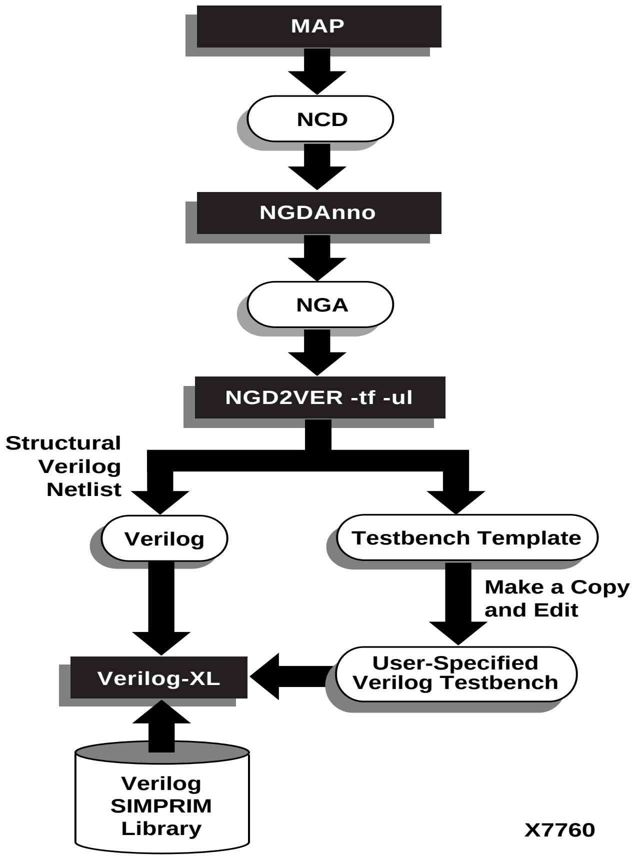


Figure 6-1 Post-Map Timing Simulation

Mapping and Back-Annotation

To perform a post-map simulation, you must have previously mapped and back-annotated your design. To perform mapping and back-annotation, use the MAP and NGDAnno commands as follows:

`map` *design*.**`ngd`**

`ngdanno` *design*.**`ncd`** *design*.**`ngm`**

For details on mapping and back-annotation, see the following chapters in the *Development System Reference Guide*.

- "MAP—The Technology Mapper" chapter
- "NGDAnno" chapter

Back-annotation is the method by which physical design information, including timing delays, is distributed back to the logical design for simulation.

In the Xilinx Development System, back-annotation for FPGA designs is performed as described below.

- The NGDAnno program distributes delays, setup and hold times, and pulse widths found in the physical NCD design file onto the logical design view represented in the NGD file. Physical component locations for PADs are also combined with the information in the NGD file.

  The output of NGDAnno is an NGA (Generic Annotated) file containing the logical design with annotations.
- The annotated design NGA file is used as an input to the netlist writer NGD2VER, which translates the back-annotated information into a Verilog netlist for simulation.

Running NGD2VER

The NGD2VER program translates your design into a Verilog HDL file containing a netlist description of the design in terms of Xilinx simulation primitives (SIMPRIMS). The Verilog file is then used to perform a simulation with the Cadence Verilog-XL simulator. You must use NGD2VER with the -tf and -ul options to create the appropriate files for use with the Cadence Verilog-XL simulator.

NGD2VER also generates an SDF (Standard Delay Format) version 2.1 file containing delays obtained from the fully-implemented input file. NGD2VER will only generate an SDF file if the input is an NGA file, which contains timing information.

The SDF file produced is intended solely for use with the Verilog file generated by NGD2VER. Do not attempt to use the SDF file in conjunction with the original design or the product of another netlist writer.

Recommended procedure:

ngd2ver -tf -ul *design***.nga**

cp *design***.tv** *design***t.stim**

Edit *design*t.stim and add your test vectors.

The following syntax translates your design to a Verilog file:

ngd2ver [**-tf**] **-ul** *infile*[**.nga**] [*outfile*.**v**]

- The -tf option generates a testbench template file. The file has a .tv extension. Xilinx recommends that you copy this testbench template to a file named *design*t.stim, then edit this .stim file and add your simulation vectors.

  For an example of a testbench template file, see the annotated example in the "Sample Test Fixture - XC4000EX Post-NGDBuild Simulation (GSR and GTS simulation)" section of the "Files" appendix.

- The –ul option causes NGD2VER to write a `uselib library directive into the output Verilog file. The path will be written as shown below to point to the Verilog SIMPRIM library:

  ```
  `uselib dir=$XILINX/verilog/data libext=.vmd
  ```

  where *$XILINX* is the location of the Xilinx software, and $XILINX/verilog/data is the location of the Verilog SIMPRIM library.

- *infile* is the input NGA file. If you enter a file name with no extension, NGD2VER will automatically look for a file with the name you specified plus an .NGA extension.

- *outfile* indicates the filename to which the output of NGD2VER will be written. Default is *infile*.v (*infile* being the same root name as the input file).

  See the "Setting Global Set/Reset and Tri-state Signals (FPGAs)" section of the "Design and Simulation Techniques" chapter for information on driving the global signals GSR, GR, GTS and PRLD.

Running the Verilog Timing Simulation

Make sure you edit a copy (*design*t.stim) of the testbench stimulus template to create a user-specified testbench file.

cp *design***.tv** *design***t.stim**

edit *design***t.stim**

See the “Files” appendix for an example of a testbench template. The example is only a template; it does not contain test vectors.

To simulate the design with Verilog-XL, you must:

Specify the full pathname of

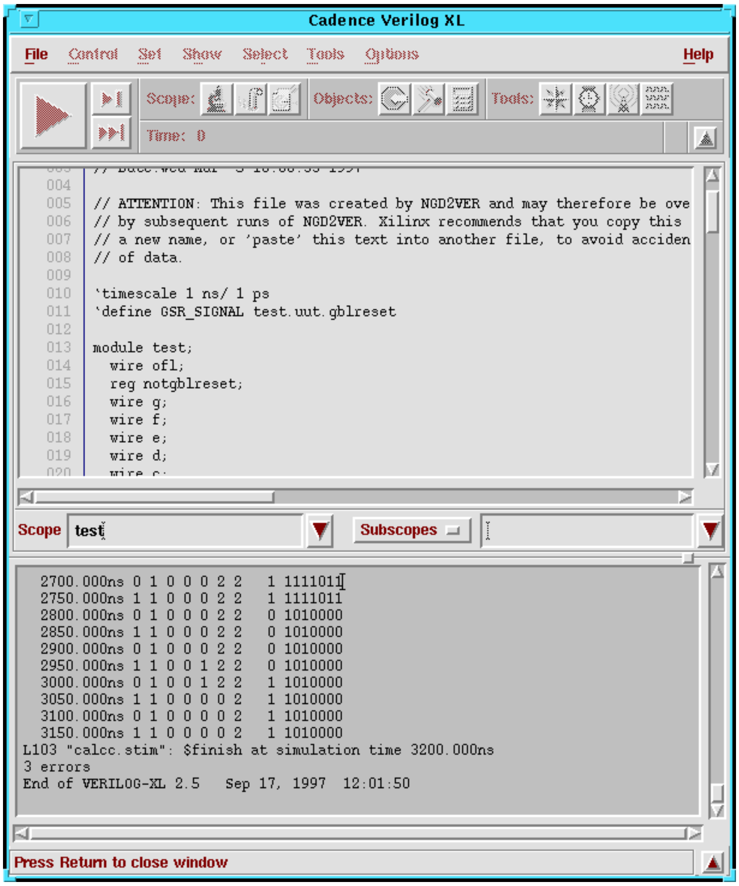
- the design Verilog netlist (*design*t.v)
- a copy of the testbench file (*design*t.tv) named *design*t.stim

Following is the syntax:

`verilog` *full_path_to_design_namet*`.stim` *full_path_to_design_namet*`.v`

If available, it is recommended that you specify the +gui option to invoke the Verilog Environment.

In the following figure, the Verilog-XL Control Window (VCW) displays when you use the +gui option.



```
// ATTENTION: This file was created by NGD2VER and may therefore be ove
// by subsequent runs of NGD2VER. Xilinx recommends that you copy this
// a new name, or 'paste' this text into another file, to avoid acciden
// of data.

`timescale 1 ns/ 1 ps
`define GSR_SIGNAL test.uut.gblreset

module test;
  wire ofl;
  reg notgblreset;
  wire g;
  wire f;
  wire e;
  wire d;
  wire c;
```

```
   2700.000ns 0 1 0 0 0 2 2   1 1111011
   2750.000ns 1 1 0 0 0 2 2   1 1111011
   2800.000ns 0 1 0 0 0 2 2   0 1010000
   2850.000ns 1 1 0 0 0 2 2   0 1010000
   2900.000ns 0 1 0 0 0 2 2   0 1010000
   2950.000ns 1 1 0 0 1 2 2   0 1010000
   3000.000ns 0 1 0 0 1 2 2   1 1010000
   3050.000ns 1 1 0 0 0 0 2   1 1010000
   3100.000ns 0 1 0 0 0 0 2   1 1010000
   3150.000ns 1 1 0 0 0 0 2   1 1010000
L103 "calcc.stim": $finish at simulation time 3200.000ns
3 errors
End of VERILOG-XL 2.5   Sep 17, 1997  12:01:50
```

Figure 6-2 The Verilog Environment (+gui)

A convenient way of running a Verilog simulation is to create a script containing a command line with all the required options. To run a simulation, simply invoke the script.

Example:

For a design named “calc” targeting an XC4000EX component, say you generate a Verilog netlist called “calct.v” and create a testbench file called “calct.tv”. Make a copy of “calct.tv” named “calct.stim” and add your stimulus to this file.

To run a timing simulation, you would use this command line:

```
verilog xilinx.run/calct.stim xilinx.run/calct.v
```

To invoke SimWave, enter the following command:

```
simwave &
```

For a complete description of SimWave, refer to the "Using SimWave" section of the "Schematic Design Tutorial" chapter.

Global Reset

Global reset should always be toggled at the beginning of a simulation to ensure that all flip-flops and latches initialize to a known state. See the "Setting Global Set/Reset and Tri-state Signals (FPGAs)" section of the "Design and Simulation Techniques" chapter for information on toggling global reset for XC3000A/L, XC3100A/L, XC4000/E/EX/XL/XV, XC5200, and XC9500 devices.

Post-Implementation Timing Simulation

You can perform a post-implementation timing simulation on both FPGA and CPLD designs. For FPGAs, you must have first mapped, routed, and back-annotated your design. For CPLDs, you must have run the design through the CPLD fitter.

Note: Post-implementation timing simulation for FPGAs is also referred to as post-route timing simulation.

The following figure illustrates the design flow for post-implementation timing simulation.

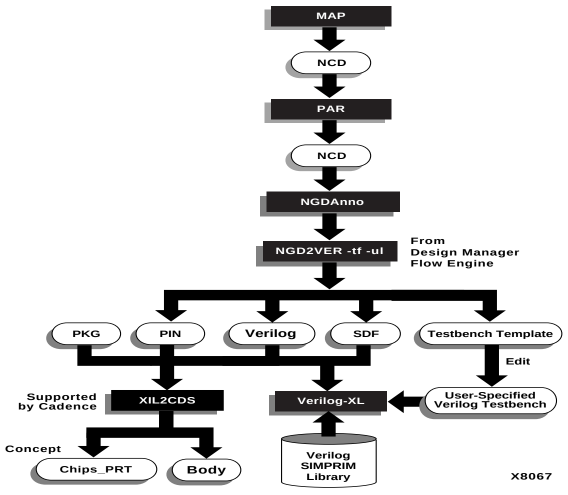


Figure 6-3 Post-Implementation Timing Simulation

Running NGD2VER

The NGD2VER program translates your design into a Verilog HDL file containing a netlist description of the design in terms of Xilinx simulation primitives. The Verilog file is then used to perform a simulation with the Cadence Verilog-XL simulator. You must use NGD2VER with the -tf and -ul options to create the appropriate files for use with the Cadence Verilog-XL simulator.

The following syntax translates your design to a Verilog file:

ngd2ver [**-tf**] **-ul** **-pf** *infile*[**.nga**] [*outfile*.**v**]

- The -tf option generates a testbench file. The file has a .tv extension. You must make a copy of this testbench template and edit it by adding your simulation vectors, For an example of a testbench template file, see the annotated example in the “Sample Test Fixture - XC4000EX Post-NGDBuild Simulation (GSR and GTS simulation)” section of the “Files” appendix.
- The –ul option causes NGD2VER to write a `uselib library directive into the output Verilog file. The path will be written as shown below to point to the Verilog SIMPRIM library:

```
`uselib dir=$XILINX/verilog/data libext=.vmd
```

where *$XILINX* is the location of the Xilinx software and $XILINX/verilog/data is the location of the Verilog SIMPRIM library.

- The –pf option will write out a pin file—a Cadence signal-to-pin mapping file. NGD2VER will generate a PIN file if the input file contains routed external pins and you have specified a –pf command line option. The file will have a .pin extension. The file is used as an input to XIL2CDS. Specifying this option is required only if you need to run XIL2CDS.
- *infile* is the input NGA file. If you enter a file name with no extension, NGD2VER will automatically look for a file with the name you specified plus an .NGA extension.
- *outfile* indicates the filename to which the output of NGD2VER will be written. Default is *infile*.v (*infile* being the same root name as the input file).

NGD2VER also generates an SDF (Standard Delay Format) file containing delays obtained from the fully-implemented input file. NGD2VER will only generate an SDF file if the input is an NGA file which contains timing information. The SDF file generated by NGD2VER is based on SDF version 2.1.

Note: The SDF file produced is intended solely for use with the Verilog file generated by NGD2VER. Do not attempt to use the SDF file in conjunction with the original design or the product of another netlist writer.

The .PIN, .PKG, . SDF and .V files output files are used by the Cadence command, XIL2CDS to generate a 1) a chips_prt file, which contains physical information about the component, and 2) a body file, which is a Concept symbol file. XIL2CDS enables you to integrate your chip-level design into a board level schematic.

Chapter 7

Design and Simulation Techniques

This chapter describes various design and simulation techniques. The sections in the chapter are as follows.

Replicating Components in a Design (SIZE)

In the Xilinx Development System release libraries, the SIZE property is no longer supported due to potential performance reductions in simulation time associated with SIZE in the context of HDL Direct methodology requirements.

To replicate bodies in your design, you must use the *Iterated Instance* methodology. To use iterated instances, edit the PATH property associated with the body (this property is assigned by Concept when you save the design) and attach an index range (*n*:*m*) to the path value:

PATH=*body_name*(*n*:*m*)

where *body_name* is the instance name of the body you want to replicate and (*n*:*m*) represents the range of indices over which you want to replicate the symbol. For example, to replicate a body four times with an index range 0 to 3, set n=3 and m=0:

```
PATH=P14(3:0)
```

For high-level functional modules, including registers, counters, adders, and memory, you can also use the LogiBLOX utility to generate modules of the desired size.

See the “Understanding Iterated Instances” section in the “Schematic Guidelines” chapter in the Cadence manual, *HDL Direct User Guid*e for more information.

Retargeting a Design to a Different Family

You can retarget your designs from one device family to another, provided both your source (original) and target designs include only Unified Library components. Since most of the symbols in the Unified Libraries have the same footprint and naming across all device families, you can easily convert your designs without doing extensive design re-entry.

In the following example, you retarget your design to an XC5200 device. The original design is an XC4000E device.

Follow these steps to convert your designs:

1. Start Concept and open your design.
2. Enter the following Concept commands in the command line window:

   ```
   ignore xce4000e
   library xce5200
   get
   write
   ```

   The **ignore** command removes a specified library from the active search list.

   The **library** command adds the specified library to the search list.

   When specified with no arguments, the **get** command reads in the object references in the design and displays the design using components from the currently active libraries.

   When specified with no arguments, the **write** command writes or saves the active design view to disk. In this case, the active view is the logic, or schematic view, logic.1.1.

Alternatively, you can convert the design to the target family by first changing your global.cmd file to point to the target family library, then reading in the design and saving it with the new library references.

Sample global.cmd file:

```
library “xce5200”,
        “xcepads”,
        “hdl_direct_lib”,
        “standard” ;
use “design.wrk” ;
```

Note that both device architecture libraries must be defined in either your master.lib or master.local file.

When accessing a library, Concept searches through the libraries using a “last read, first out” protocol. For the sample global.cmd file, the libraries are searched in the following order: standard, hdl_direct_lib, xcepads, and finally xce5200.

The **use** command specifies the SCALD library map file to be used for the design.

After the conversion process, the symbols that are common to both the source (XC4000E, in this case) and target families should maintain their relative location and pin position in the target design schematic. This is due to the uniformity in size, shape and naming of symbols in the Unified Libraries across all device architectures or families. Pins on these symbols should also retain their connectivity to the nets they were originally attached to in the source design.

3. Next, you must manually replace symbols that are not common to your source and target families with equivalent logic. For example, if a GCLK was used earlier in an XC3000A design and is then retargeted for use in an XC4000E device, the GCLK symbol must be manually replaced with a BUFGP, BUFG, or BUFGS.
4. You must also manually replace components that are macros in the source (or target) library, but primitives in the other. For example, an AND5 is a primitive in the XC4000E family, but a macro in the XC5200 library. If you are converting an XC4000E design to the XC5200 family, you must manually replace all instances of AND5s in the target design after you have completed

the initial library retargeting step described in Step 2. Use the "replace" command in Concept to make your replacements.

Example:

select the AND5 symbol
replace and5

5. For an XC4000E / L / EX / XL / XV to XC5200 conversion, you must also replace all XC4000 wide decoder macros (DECODExxx) with XC5200 DECODExxx macros.

   In the case of an XC4000 to XC5200 conversion, components which must be manually replaced are listed in the following table:

Table 7-1 XC4000E/L/EX/XL/XV and XC5200 Components Requiring Manual Replacement

and5	and5b1	and5b2	and5b3	and5b4	and5b5
nand5	nand5b1	nand5b2	nand5b3	nand5b4	nand5b5
or5	or5b1	or5b2	or5b3	or5b4	or5b5
nor5	nor5b1	nor5b2	nor5b3	nor5b4	nor5b5
ifd	ild1	ofd	ofdt	cy4	fdpe
bufgs	bufgp				

Merging Design Files from Other Sources

You can enter specific blocks of your design in some form other than Concept schematics, such as HDL, or other third party format. However, whatever form of entry you use for a design block, you must convert into one of the following formats before you can incorporate it into your Concept schematic: NGO, V, XNF or EDIF. Incorporating a .V (Verilog) behavioral description into your Concept schematic is supported by Cadence only.

To incorporate these netlist files into your schematic, you must create a body for the netlist file and place it on your schematic as you would any other component. For a description of how to generate a body for a design block, refer to "Creating Bodies for Non-Schematic Design Blocks" section of the "Design Entry" chapter.

XC4000 Flip-flop Initialization

The following subsections describe IOB and CLB flip-flop initialization.

IOB Flip-flop Initialization

To set the initial value of XC4000E/L/EX/XV/XL IOB flip-flops at power-up, follow these instructions:

Set the IOB flip-flops to power-up either High or Low on power-up by attaching the appropriate INIT property (INIT=1 or INIT=0) to each IOB flip-flop in your schematic. The default value of INIT is 0. When you activate the global reset signal (GSR) during simulation, the IOB flip-flops will initialize to this assigned value.

CLB Flip-flop Initialization

To set the initial value of XC4000E/L/EX/XV/XL CLB flip-flops at power-up, follow these instructions:

To get a flip-flop which will initialize to 0 at power-up, select a flip-flop macro or primitive with an asynchronous RESET input (FDCE, for example, has an asynchronous reset input called “CLR”).

To get a flip-flop which will initialize to 1 at power-up, select a flip-flop macro or primitive with an asynchronous SET input (FDPE, for example, has an asynchronous preset pin called “PRE”).

XC9500 Flip-flop Initialization

The following subsections describe XC9500 flip-flop initialization.

IOB Flip-flop Initialization

XC9500 devices do not contain IOB flip-flops.

Macrocell Flip-flop Initialization

To set the initial value of XC9500 macrocell flip-flops at power-up in your simulation, follow these instructions:

For Unified Library and post-NGDBuild functional simulation, you must explicitly drive the macrocell flip-flops High or Low in your test fixture file.

For timing simulation, set the macrocell flip-flops to power-up either High or Low on power-up by attaching the appropriate INIT property (“S” for “SET” and “R” for “RESET”) to each macrocell flip-flop in your schematic. The default value of INIT is R (Reset). When you activate the global preload signal (PRLD) during a timing simulation, the macrocell flip-flops will initialize to the assigned value.

Setting Global Set/Reset and Tri-state Signals (FPGAs)

The way you set the global set / reset, global reset and global tri-state signals depends on what flow you are using (Cadence Concept HDL Direct, or other), the part type you are using, and whether your design contains a STARTUP component (XC4000E / L / EX / XL / XV and XC5200 devices only). The various approaches are described in the following subsections.

Setting Global Set/Reset

At the beginning of an FPGA design simulation, you must toggle the global set / reset signal (GSR in XC4000E / L / EX / XL / XV designs), or the GR global reset signal in XC5200 or XC3000A / L and XC3100A / L designs; toggling the global set / reset emulates the power-on reset of the FPGA. If this action is not performed, the flip-flops and latches in your simulation may not function correctly.

The global set / reset net is present in the implemented design whether or not you instantiate the STARTUP block in your design. The function of the STARTUP is to give you the option to control the global reset net from an external pin.

If you wish to select the global set / reset pulse width so that it reflects the actual amount of time it takes for the chip to go through the reset process when power is supplied to it (T_{mrw}), refer to *The Programmable Logic Data Book* for the device you are simulating for this information.

The general procedure for specifying global set / reset or global reset during a pre-NGDBuild Verilog Unified Library simulation involves defining the global reset signals with one of the following Verilog macros: GSR_SIGNAL or GR_SIGNAL. This is necessary because these global nets do not exist in the Concept Unified Libraries, and as a result, also do not exist in a netlist generated directly from Concept schematics which have only been processed by running CONCEPT2XIL with the -sim_only option. In addition, you must also declare the global set/reset signal either as a Verilog *wire* or *reg*. Your choice of *wire* or *reg* depends on whether the design contains a STARTUP component.

Note: In the Xilinx M1.4 release, the Verilog Unified Library is only used in pre-NGDBuild simulation of Cadence Concept or Concept/Synergy mixed mode designs immediately after running CONCEPT2XIL with the -sim_only option. Simulation at all other points of the flow utilizes the Verilog SIMPRIM Libraries.

For pre-NGDBuild Unified Library functional simulation, you must set the value of the appropriate Verilog macro (GSR_SIGNAL or GR_SIGNAL) to the name of the GSR or GR net, qualified by the appropriate scope identifiers. (GSR_SIGNAL and GR_SIGNAL are used in the Verilog Unified Libraries to emulate the global reset signals.) The scope identifiers are made up of some combination of the test module scope, and the design instance scope. The scope qualifiers are required because the scope information is needed when the GSR_SIGNAL and GR_SIGNAL macros are interpreted by the Verilog Unified Library simulation models to emulate a global reset signal.

The net name you specify, and whether you specify the net as a Verilog *reg* or a *wire,* depends on whether or not you have instantiated a STARTUP block in the design.

1. If no STARTUP block is present in the design, it is recommended that you name the global (set/)reset net *test*.GSR or *test*.GR (remember that Verilog is case-sensitive!), and the signal should be declared as a Verilog *reg* data type.

2. If there is a STARTUP block in the design and the GSR pin is connected to a net, the value of GSR_SIGNAL should be set to the net connected to the GSR pin on the STARTUP symbol.

   The signal you actually toggle at the beginning of the simulation will be the port or signal in the design that is used to control global set/reset. This is usually an external input port in the

Verilog netlist, but may also be a wire if global reset is controlled by logic internal to your design.

3. When invoking Verilog-XL to run the simulation, it is recommended that the testbench file be specified before the Verilog netlist for the design in order that the simulation work properly.

   Example, Unified Library simulation:

   ```
   verilog design.stim design.v -f design.vf
   ```

   Example, post-NGDBuild functional simulation:

   ```
   verilog design.stim designf.v
   ```

4. It is recommended that the name of the main module in the testbench file be named "test" to be consistent with name of the testbench module that will be written out downstream in the design flow by NGD2VER when you do post-NGDBuild, post-MAP, or post-route simulation. If this naming consistency is maintained, you should be able to use the same testbench file for simulation at all stages of the design flow with minimal modification.

Note: For Unified Library functional simulation, you must always define the appropriate Verilog macro (GSR_SIGNAL or GR_SIGNAL) for the global set / reset signal. (This macro is not used in timing simulation when there is a STARTUP block in the design.)

Note: The GSR signal in XC4000E / L / EX / XL / XV devices and the GR signal in XC5200 devices is active High, whereas the GR signal (XC3000A / L and XC3100A / L designs) signal is active Low.

Note: For post-NGDBuild and post-route timing simulation, the testbench template (.tv file) produced by running NGD2VER with the -tf option already contains most of the code described previously (and illustrated in the following examples) for defining and toggling GSR or GR. However, in the case where you have a user signal controlling the STARTUP block, you must manually edit the testbench template file generated by NGD2VER to specify the user control signal connected directly to the GSR or GR pin on the STARTUP block symbol as GSR_SIGNAL (4K) or GR_SIGNAL (5K).

Designs with No STARTUP Block

When there is no STARTUP block in the design, you can use the same testbench file with little or no modification at all stages of the design flow if you follow these guidelines:

Example 1: XC4000E/L/EX/XL/XV Unified Library Functional Simulation (No STARTUP Block)

The following illustrates how you can drive the GSR signal in a Verilog-XL testbench file at the beginning of a pre-NGDBuild Unified Library functional simulation.

Note: The terms “testbench” and “test fixture” are used synonymously throughout this manual.

You should reference the global set/reset net as “GSR” in an XC4000E/L/EX/XL/XV design when there is no STARTUP block in the design, and the Verilog macro defining the global net must be called “GSR_SIGNAL”, since this is how it is modeled in the Verilog Unified Simulation Library:

- Within the testbench file, GSR should be declared as a Verilog register within the “test” module:

```
module test;
reg GSR;
```

- Set a macro called GSR_SIGNAL to *test*.GSR (the name of the global set/reset signal, qualified by the name of the testbench module) using the **`define** compiler directive.

```
`define GSR_SIGNAL test.GSR;
```

GSR should be toggled High, then Low in an “initial” block:

```
module test;

reg GSR;
`define GSR_SIGNAL test.GSR;

initial
  begin
     GSR = 1;       // reset the device
     #100 GSR = 0;
```

In this example, the active High GSR signal in the XC4000E/L/EX/XL/XV device is activated by driving it High. 100 ns later, it is deactivated by driving it Low. (100 ns is an arbitrarily chosen value.)

Alternatively, you may reference the macros instead of the “GSR” signal name in the initial block:

```
initial
  begin
     ‘GSR_SIGNAL = 1;       // reset the device
     #100 ‘GSR_SIGNAL = 0;
```

You may use the same test fixture for simulating at other stages of the design flow when there is no STARTUP block in the design.

Example 2: XC5200 Unified Library Functional Simulation (No STARTUP Block)

For pre-NGDBuild Unified Library functional simulation, the active High GR net in XC5200 devices should be simulated in the same manner as GSR for XC4000E/L/EX/XL/XV. Substituting GR for GSR, and GR_SIGNAL for GSR_SIGNAL gives:

```
module test;

reg GR;
`define GR_SIGNAL test.GR;

initial
  begin
     `GR_SIGNAL = 1;      // reset the device
     #100 `GR_SIGNAL = 0;
```

You may also use the same test fixture for simulating your design at other stages of the design flow when there is no STARTUP block in the design.

Example 3: XC3000A/L and XC3100A/L Unified Library Functional Simulation (No STARTUP Block)

Asserting global reset in XC3000A/L and XC3100A/L designs is almost identical to the procedure for asserting global reset in XC5200 designs, except that GR in XC3000A/L and XC3100A/L devices is active Low. (Also note that the STARTUP block is not supported on XC3000A/L and XC3100A/L devices):

```
module test;

reg GR;
`define GR_SIGNAL test.GR;
initial
  begin
```

```
      `GR_SIGNAL = 0;     // reset the device
     #100 `GR_SIGNAL = 1;
```

Note: The Global Reset (GR) signal in the XC3000A / L architecture is modelled differently in Unified Library functional simulation netlists and SIMPRIM library based netlists generated by NGD2VER. In the Verilog Unified Library, GR is modelled as a wire within a global module, while in a SIMPRIM-based netlist, it is always modelled as an external port. As a result, you cannot use the same testbench file to do both Unified library simulation and SIMPRIM-based simulation.

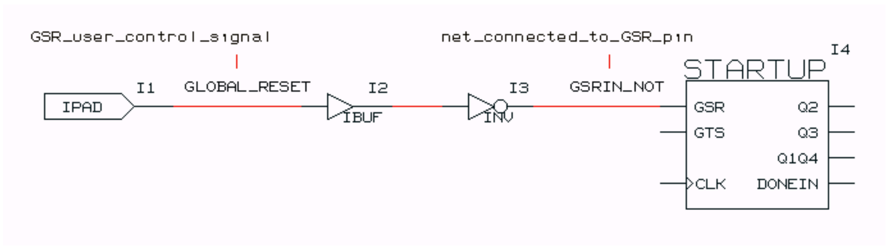
Designs With STARTUP block (XC4000E/L/EX/XL/XV and XC5200 Devices Only)

Asserting global set / reset when the STARTUP block is specified in the design is similar to asserting global set / reset *without* a STARTUP block in the design. There are two differences, however. The first difference is that the `*define* statement must now specify the name of the net attached to the GSR pin (XC4000E / L / EX / XL / XV devices) or GR pin (XC5200 devices) on the STARTUP block.

`` `define  GSR_SIGNAL `` *net_connected_to_GSR_pin*

The other difference is that the signal you toggle is now either the external input signal or internal signal that controls the "*net_connected_to_GSR_pin*" (or "*net_connected_to_GR_pin*") on the STARTUP block. If the signal is a user-specified external input as shown in the following figure, it will appear in your Verilog netlist as an input port, and can be driven as follows:

```
initial
begin
   GSR_user_control_signal = 1;
   #100 GSR_user_control_signal = 0;
```



If the signal is an internal net, use the Verilog *force* command to toggle it, being sure to specify the appropriate scope qualifier for the design instance (usually “uut”):

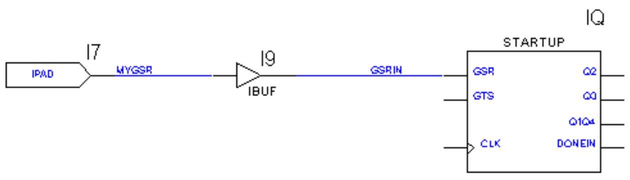
```
initial
begin
   force uut.GSR_user_control_signal = 1;
   #100 force uut.GSR_user_control_signal = 0;
```

Example 1: XC4000E/L/EX/XL/XV

Unified Library Simulation (With STARTUP)

The following is an example of how to drive the global set/reset signal in a Verilog-XL test fixture file at the beginning of a pre-NGDBuild Unified Library functional simulation when there is a STARTUP block in an XC4000E/L/EX/XL/XV design.

In the following illustration, a signal called “mygsr” is the *GSR_user_control_signal*. In this case “mygsr” is an external user signal that controls GSR. “mygsr” sources an IBUF, which in turn sources a signal called “gsrin”. “gsrin” represents the *net_connected_to_GSR_pin* — the pin that directly sources the GSR pin of the STARTUP block:



- The design allows you to control global set/reset in the device by driving the external input port, “mygsr”. In the test fixture file, “mygsr” will appear as a Verilog *reg* within the “test” module:

  ```
  module test;
    reg mygsr;
  ```

- In addition, for XC4000E/L/EX/XL/XV designs, a Verilog macro called GSR_SIGNAL must be declared to make the connection between the user logic and the global GSR net embedded in the Unified Library models. This is done by using a `define directive to set GSR_SIGNAL to the following:

test_module_name.design_instance_name.gsr_pin_signal

gsr_pin_signal corresponds to the name of the signal connected to the GSR pin on the STARTUP block (in this case, "gsrin". The scope qualifier in this case also includes the name of the design instance ("uut") in anticipation that the net will appear as an internal net of the design in the post-NGDBuild, post-Map and post-route simulations further down in the flow:

```
`define GSR_SIGNAL test.uut.gsrin;
```

- The global set / reset control signal should be toggled High, then Low in an "initial" block:

```
module test;
  reg mygsr;
   `define GSR_SIGNAL test.uut.gsrin;

initial
  begin
     mygsr = 1;      // reset the device
     #100 mygsr = 0;
```

Post-NGDBuild Functional, Post-Map Timing, and Post-Route Timing Simulation (With STARTUP Block)

For post-NGDBuild functional simulation, post-Map timing simulation, and post-route timing simulation, the procedure is identical to Unified Library functional simulation, except that you must omit the `define statement for GSR_SIGNAL as shown. This must be done because the net connections will already exist in the post-NGDBuild design. Leaving in the Verilog macro definition will cause a conflict with these connections. In the following example the macro definition has been commented out to avoid this conflict.

```
module test;
  reg mygsr;

    // `define GSR_SIGNAL test.uut.gsrin;
```

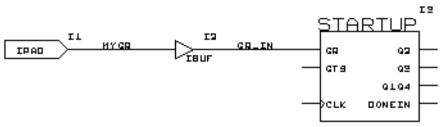
```
initial
  begin
     mygsr = 1;      // reset the device
     #100 mygsr =0;
```

Example 2: XC5200

Unified Library Functional Simulation Designs with STARTUP Block

For an XC5200 design containing a STARTUP block (similar to the previous Example 1 for XC4000 designs), the net controlling GR should be stimulated in the same manner as for the XC4000E / L / EX / XL / XV.

Substitute GR_SIGNAL for GSR_SIGNAL, *mygr* for *mygsr,* and *gr_in* for *gsr_in* in Example 1 to obtain the test fixture fragment for stimulating GR in a Verilog Unified Library simulation.



```
module test;
  reg mygr;

  `define GR_SIGNAL test.uut.gr_in;

initial
  begin
     mygr = 1;      // reset the device
     #100 mygr = 0;
```

Post-NGDBuild Functional, Post-Map Timing, and Post-Route Timing Simulation (With STARTUP Block)

For post-NGDBuild functional simulation, post-Map timing simulation, and post-route timing simulation, the procedure is identical to Unified Library functional simulation, except that you must omit the `define statement for GR_SIGNAL. This must be done because the net connections will already exist in the post-

NGDBuild design, and leaving in the macro definition would cause a conflict with these connections. In the following example the Verilog macro definition has been commented out to avoid this conflict.

```
module test;
  reg mygr;

    // `define GR_SIGNAL test.uut.gr_in;

initial
  begin
     mygr = 1;      // reset the device
     #100 mygr = 0;
```

Example 3: XC3000A/L and XC3100A/L designs

STARTUP is not supported in XC3000A/L and XC3100A/L designs. Follow the procedure for XC3000A/L and XC3100A/L designs without STARTUP blocks.

Setting Global Tri-state (XC4000 and XC5200 Outputs Only)

XC4000E/L/EX/XL/XV devices also have a global control signal (GTS) that tri-states all output pins. This capability allows you to isolate the actual XC4000E/L/EX/XL/XV part during board level testing. You can also tri-state the FPGA device outputs during board level simulation to assist in simulation debug. In most cases, however, GTS is usually deactivated so that the outputs are active.

Although the STARTUP component also gives you the option of controlling the global tri-state net from an external pin, more often than not it is only used for controlling global reset. When this is the case, the GTS pin may be left unconnected at the design entry phase. In this case it will float to its inactive state level. The global tri-state net GTS is present in an implemented design whether or not a STARTUP block has been instantiated. If desired, GTS can be explicitly deactivated by driving it Low in your test fixture file, or connecting the GTS pin to GND in your input design.

The general procedure for specifying GTS is similar to that used for specifying the global set/reset signals GSR and GR: define the global tri-state signal with the Verilog macro, GTS_SIGNAL. This is necessary because this global net is not modelled in a netlist generated directly from Concept schematics which have only been processed by running CONCEPT2XIL with the -sim_only option. In addition, you must also declare the global tri-state signal either as a Verilog *wire* or *reg*. Your choice of *wire* or *reg* depends on whether the design contains a STARTUP component.

The specific net name you specify, and whether you specify the net as a Verilog *reg* or a *wire,* depends on whether or not you have instantiated a STARTUP block in the design, and whether you have connected a signal to the STARTUP GTS pin.

1. If no STARTUP block is present in the design, it is recommended that you name the global tri-state net *test.GTS* (remember that Verilog is case-sensitive!), and declare the signal as a Verilog *reg* data type.
2. If there is a STARTUP block in the design and the GTS pin is connected to a net, the value of GTS_SIGNAL should be set to the name of the net connected to the GTS pin on the STARTUP symbol. The signal you actually toggle at the beginning of the simulation will be the port or signal in the design that is used to control global tri-state. This is usually an external input port in the Verilog netlist, but may alternatively be a wire if global tri-state is controlled by internal logic in your design.
3. It is recommended that the name of the main module in the test-bench file be named “test” to be consistent with the name of the test fixture module that will be written out downstream in the design flow by NGD2VER when you do post-NGDBuild, post-Map (optional) and post-route simulation. If this naming consistency is maintained, you should be able to use the same test fixture file for simulation at all stages of the design flow with minimal modification.

Note: For Unified Library functional simulation, you must always define the appropriate Verilog macro (GTS_SIGNAL).

Note: The GTS signal in XC4000E/L/EX/XL/XV devices and XC5200 devices is active High. This macro is not used in timing simulation when there is a STARTUP block in the design and the GTS pin is connected up.

Note: For post-NGDBuild and post-route timing simulation, the test-bench template (.tv file) produced by running NGD2VER with the -tf option already contains most of the code described above (and illustrated below) for defining and driving GTS. However, in the case where you have a user signal controlling the STARTUP block, you must manually edit the test fixture template file generated by NGD2VER to specify the user control signal for GTS.

Designs with No STARTUP Block

When there is no STARTUP block in the design, you can use the same test fixture file with little or no modification if you follow these guidelines in the following examples:

Example: XC4000E/L/EX/XL/XV and XC5200 Unified Library Functional Simulation (No STARTUP Block)

The following is an example of how you can drive the GTS signal in a Verilog-XL test fixture file at the beginning of a pre-NGDBuild Unified Library functional simulation. The global tri-state net is called “GTS” in an XC4000E/L/EX/XL/XV or XC5200 design when there is no STARTUP block in the design. The Verilog macro defining the global net must be called “GTS_SIGNAL”, since this is the name of the predefined macro used to model the global tri-state signal in the M1 Verilog Unified Library simulation models:

- Within the test fixture file, GTS should be declared as a Verilog register within the “test” module:

  ```
     module test;
     reg GTS;
  ```

- Set a macro called GTS_SIGNAL to *test.GTS* (the name of the global tri-state signal, qualified by the name of the test fixture module), using the **`define** compiler directive.

  ```
  `define GTS_SIGNAL test.GTS;
  ```

  GTS should be driven Low in an “initial” block:

  ```
  module test;

  reg GTS;
  `define GTS_SIGNAL test.GTS;

  initial
    begin
       GTS = 0;
  ```

In this example, the active High GTS signal is deactivated by driving it Low to activate the outputs of the design.

Alternatively, you can reference the GTS_SIGNAL macro instead:

```
initial
  begin
     `GTS_SIGNAL = 0;
```

Designs With STARTUP block (XC4000E/L/EX/XL/XV and XC5200 Devices Only)

Asserting global tri-state when the STARTUP block is specified in the design is similar to asserting global tri-state *without* a STARTUP block in the design. There are two differences, however. The first difference is that if the GTS pin on the STARTUP block is connected, the `define statement must now set GTS_SIGNAL to the name of the net attached to the GTS pin on the STARTUP block.

`` `define GTS_SIGNAL `` *net_connected_to_GTS_pin*

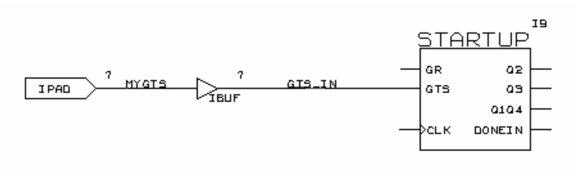
The other difference when the GTS pin on the STARTUP block is connected is that the signal you drive is now either the external input port or internal signal that controls the "*net_connected_to_GTS_pin*" on the STARTUP block. If it is an external input, it will appear in your Verilog netlist as an input port. To tri-state your outputs, drive this signal High; to activate your outputs, drive it Low:

```
initial
begin
   GTS_user_control_signal = 1;
   #100 GTS_user_control_signal = 0;
```

Example 1: XC4000E/L/EX/XL/XV and XC5200

Unified Library Functional Simulation (With STARTUP, GTS Pin Connected)

In the following figure, the design contains a STARTUP block, and the GTS pin on STARTUP is connected to an external input called *mygts*:



The external input, "mygts", is declared as a Verilog register and a `*define* directive setting GTS_SIGNAL to the name of the net connected to the GTS pin is required to connect the user logic to the global GTS model in the Unified Library simulation models for output buffers (OBUF, OBUFT, etc.). Your test fixture should look something like this:

```
module test;
  reg mygts;

    `define GTS_SIGNAL test.uut.gts_in;
.
.
.
initial
  begin
     mygts = 1;  // if you wish to tri-state the
                 // device;
     #100 mygts = 0;     // deactivate GTS
```

Post-NGDBuild Simulation of GTS (With STARTUP, GTS pin connected)

For post-route timing simulation, the procedure is identical, except that you must omit the `define statement for GTS_SIGNAL because it will cause contention with the GTS net driver.

```
module test;
  reg mygtsr;

    // `define GTS_SIGNAL test.uut.gtsin

initial
  begin
     mygts = 1;  // if you wish to tri-state the
```

```
                       // device;
    #100 mygts = 0; // deactivate GTS
```

Example 2: XC4000E/L/EX/XL/XV and XC5200

Unified Library Simulation (With STARTUP, GTS Pin NOT Connected)

For Unified Library functional simulation, define a wire called GTS, and set the GTS_SIGNAL macro to *test*.GTS. Toggle GTS_SIGNAL as shown:

```
module test;
  wire GTS;
  `define GTS_SIGNAL test.GTS

initial
  begin
   force `GTS_SIGNAL = 1;    // if you wish to
                               // tri-state the
                               // device;
   #100 force `GTS_SIGNAL = 0; // deactivate GTS
```

Post-NGDBuild Simulation of GTS (With STARTUP, GTS Pin Unconnected)

For post-NGDBuild functional simulation, the actual net exists and must be further qualified by the design instance scope, “uut”:

```
module test;
  // wire GTS;
  // `define GTS_SIGNAL test.GTS

  `define GTS_SIGNAL test.uut.GTS

initial
  begin
    force ‘GTS_SIGNAL = 1;    // if you wish to
                               // tri-state the
                               // device;
   #100 force `GTS_SIGNAL = 0;  // deactivate
GTS
```

For post-route timing simulation, you can use the same test bench, unaltered.

Setting Global PRLD (CPLD Designs)

Before you simulate a CPLD design, you must force the global reset, or “PreLoad” signal (PRLD) signal. Forcing the global preload signal emulates the power-on initialization of the CPLD. If you do not force global preload, the flip-flops and latches will have undefined initial states. The PRLD signal is active High, and should be forced High, then Low to properly initialize the flip-flops in your design at the beginning of a simulation.

If you wish to select the PRLD pulse width so that it reflects the actual amount of time it takes for the chip to go through the reset process when power is supplied to it (T_{wmr} for XC9500 devices), refer to *The Programmable Logic Data Book* for the device you are simulating for this information.

Unified Library Functional Simulation

To simulate PRLD in Verilog-XL using the M1 Unified Library simulation models, define a reg called “*PRLD*” and set a Verilog macro called “*PRLD_SIGNAL* to “*test.PRLD*” as follows:

```
module test;
reg PRLD;
`define PRLD_SIGNAL test.PRLD;

Toggle this newly defined signal in an initial block:

initial
  begin
     PRLD = 1;     // reset the device
     #100 PRLD = 0;
```

Alternatively, you may reference the PRLD_SIGNAL macro in your initial block:

```
initial
  begin
     `PRLD_SIGNAL = 1;     // reset the device
     #100 `PRLD_SIGNAL = 0;
```

Post-NGDBuild and Post-Implementation Timing Simulation

To simulate the global PRLD signal in a Verilog post-NGDBuild or post-implementation timing simulation, you can re-use the same test fixture as the one you create for Unified Library functional simulation.

Oscillator Functions (OSC, OSC4, OSC5)

The OSC (X3000A/L), OSC4 (XC4000E/L/EX/XL/XV) and OSC5 (XC5200) oscillator components do not have Verilog simulation models associated with them. In the case of the OSC, the clock signal frequency is derived from an external crystal-controlled oscillator. On the other hand, both the OSC4 and OSC5 are internal oscillators which are useful in applications where timing is not critical.

To simulate these oscillators, you must reference the net attached to the output of the oscillator component. Toggle this net at the desired frequency in your Verilog test fixture using a **`force`** command within an **`always`** block.

Example: Given an oscillator output net called "osclk" attached to an oscillator symbol (OSC, OSC4, or OSC5), and assuming a timescale unit of 1ns, use the following "always" block to emulate an oscillator with a 10 Mhz clock frequency:

```
always #100 force osclk = ~osclk;
```

Manual Translation

You can access the programs required to simulate and implement your design in command line mode. This chapter summarizes the design implementation, functional simulation, and timing simulation procedures from the UNIX command line for different types of designs. Program syntax and options are discussed in the "Program Options" appendix.

This chapter contains the following sections:

- "Functional Simulation" section
- "Design Implementation" section
- "Timing Simulation" section

Functional Simulation

This section briefly describes the steps for running Unified Library Based and SIMPRIM-based functional simulation.

Unified Library Based Functional Simulation

Unified Library-based functional simulation can only be performed on pure schematic designs without LogiBLOX elements.

Schematic Designs Without LogiBLOX Elements

1. Enable HDL Direct in Concept.
2. Save schematic in Concept.
3. Run concept2xil -sim_only. The syntax of the command is:

   `concept2xil -family` *technology* **`-sim_only`** *design_name*

This command creates a .v and .vf file in the xilinx.run directory unless specified otherwise using the -rundir option

4. Run the functional simulation using Verilog-XL.

   `verilog +delay_mode_unit` *full_path_to_design_name* **`.v \`**
   `-f` *full_path_to_design_name* **`.vf`** *design_name* **`.stim`**

SIMPRIM Library Based Functional Simulation

SIMPRIM-based functional simulation requires an NGD file as an input. You can use NGDBuild to generate an NGD file from an EDIF, XNF, or NGO file.

1. Submit the design to NGDBuild.

   `ngdbuild -p` *part_type design_name*

   The part_type with the -p option does not need to be specified if there is a PART specified in the schematic in the CONFIG block.

2. Use NGD2VER to create a structural Verilog netlist and a testbench stimulus template. Specify the output netlist name as *design_name***f** to avoid overwriting any Unified Library simulation netlists.

   `ngd2ver -tf -ul` *design_name* **`.ngd`** *design_name***`f.v`**

3. Make a copy of the testbench stimulus template and name it *design*f.stim. Edit the copy of the testbench file to create a user-specified testbench file.

4. Run the Verilog-XL command.

   `verilog +delay_mode_unit` *full_path_to_design***`f.stim \`**
   *full_path_to_design***`f.v`**

   (The “\”at the end of a line indicates that the line following the current one must be typed on the same command line.)

Mixed Mode Designs

Functional simulation of mixed-mode schematic designs containing SXNF, XNF, or EDIF subblocks can be performed using the Xilinx architecture-independent Verilog SIMPRIM simulation libraries. More generally, SIMPRIM library functional simulation can be performed on any type of design regardless of the design entry method. The procedure is as follows:

1. Generate an XNF or EDF netlist for each non-schematic block using the appropriate translation tool. For example, a Synopsys block must be written out in SXNF or SEDIF format.
2. Run NGDBuild to merge the schematics with the non-schematic blocks.

   ngdbuild -p *part_type design_namef*

3. Proceed to Step 2 of the "SIMPRIM Library Based Functional Simulation" section.

Design Implementation

This section explains how to implement schematic designs for FPGAs and CPLDs.

Schematic Designs (FPGA)

1. Convert the Concept design from Cadence to EDIF with CONCEPT2XIL.

   Here is an example:

   concept2xil [**-cdslib** *lib_map_filename*] **-family** *technology design_name*

   The EDIF (.EDF) file is written to a xilinx.run directory by default.

   Refer to the "Converting the Concept Design to an EDIF File" section of the "Design Implementation" chapter for details.

2. Navigate to the xilinx.run directory and submit the design to NGDBuild. NGDBuild reads a file in EDIF or XNF format, reduces all the components in the design to Xilinx primitives, runs a logical design rule check on the design, and writes an NGD file as output.

   ngdbuild -p *technology design_name*

   For example:

   ngdbuild -p xc4000ex test

or

ngdbuild -p 4028exhq240-3

The -p part type does not need to be specified if it has already been specified in the schematic.

3. Map the logic to the components in the FPGA by typing the following syntax:

   map -p *partname design_name***.ngd**

   For example:

   map -p 4028exhq240-3 test.ngd

   The -p part type does not need to be specified if it has already been specified in the schematic or as an option to NGDBuild.

4. Place and route the design:

   par test.ncd testt.ncd

   The first file is created by the MAP utility, and PAR creates the other one.

5. Proceed to the "Post-Implementation Timing Simulation" section.

6. Run BitGen. The BitGen program produces a bitstream for Xilinx FPGA device configuration. After the FPGA design has been completely routed, it is necessary to configure the device so that it can execute the desired function. This is done using the bitstream generated by BitGen, Xilinx's bitstream generation program. BitGen takes a fully routed NCD (Circuit Description) file as its input and produces a configuration bitstream—a binary file with a .bit extension.

Schematic Designs (CPLD)

When using CPLDs, the procedure for implementing pure schematic designs, designs with XNF, EDIF, or NGO elements, and mixed-mode schematic-at-top designs is the same. Follow these steps.

1. Convert the Concept design from Cadence to EDIF with CONCEPT2XIL:

   Here is an example:

concept2xil [**-rundir** *run_directory*] **-family** *technology design_name*

Refer to the "Converting the Concept Design to an EDIF File" section of the "Design Implementation" chapter for details.

2. Submit the design to the CPLD fitter.

   cpld -p *partname design_name*

3. Proceed to the "Post-Implementation Timing Simulation" section.

HDL Top Level Designs

1. Synthesize the HDL modules in your design treating any non-HDL subblocks as black boxes and write out an EDIF or XNF file from the synthesis tool.

2. Any instantiated subblocks should be processed down to EDIF, XNF, or NGD format. Merge all the EDIF, XNF, and NGO files for the submodules with the top level XNF or EDIF file by running NGDBuild.

   ngdbuild -p *part_number design_name*

   For example:

   ngdbuild -p XC4000E test (where test is the root name for the EDIF or XNF file)

   or

   ngdbuild -p 4028exhq240-3 test

3. Proceed to the "Post-Implementation Timing Simulation" section.

Pure HDL Designs

1. Synthesize the HDL file and create an EDIF or XNF file for the synthesized design.

2. Convert the EDIF or XNF file to an NGD file by using ngdbuild:

   ngdbuild -p *technology design_name*

   or

   ngdbuild -p *part_number design_name*

For example:

ngdbuild -p XC4000E test (where test is the root name for the EDIF or XNF file)

or

ngdbuild -p 4028exhq240-3

3. Proceed to the "Post-Implementation Timing Simulation" section.

Timing Simulation

This section explains how to conduct post-map and post-implementation timing simulations.

Post-Map Timing Simulation (FPGAs Only)

Before performing a post-map timing simulation, you must have a mapped and back-annotated netlist for your design. Then perform the following steps.

1. Run NGDAnnno to obtain a back-annotated netlist.

   ngdanno *design design_m* [design.ngm]

2. Use NGD2VER to create a structural Verilog netlist and a testbench stimulus template.

   ngd2ver [**-tf**] **-ul** *design_m***.**[**nga**]

   The -tf option is only needed if you did not generate a testbench template during functional simulation.

3. Make a copy of the testbench stimulus template and edit it to create a user-specified testbench file.

4. Invoke Verilog-XL with the following command.

   verilog *full_path_to_design***t.stim** *full_path_to_design***t.v**

Post-Implementation Timing Simulation

For FPGAs, you must have mapped, routed and back-annotated your design. For CPLDs, you must have run the design through the CPLD fitter.

1. For FPGAs, use NGDAnno to back-annotate your design. Assuming the routed NCD file is called “designt”, enter the following command:

   ngdanno designt design.ngm

   If your design is for a CPLD, use the NGA file output from the CPLD Fitter as an input to the NGD2VER command in the next step.

2. Use NGD2VER to create a structural Verilog netlist and a test-bench stimulus template.

   ngd2ver -tf -ul [**-pf**] *design_name***.nga** *design_name***t.v**

   The -pf option is only required if you plan to integrate the design into a board-level schematic.

   The output is renamed to *design_name***t** to avoid overwriting any simulation netlists generated for Unified Library functional simulation. Refer to the “Running NGD2VER” section of the “Timing Simulation” chapter.

3. Make a copy of the testbench template, naming it *design*t.stim and add your test vectors to the copy. The copy will be your testbench file.

4. Submit the design to the Cadence Verilog-XL simulator to conduct a timing simulation.

   verilog *full_path_to_design***t.stim** *full_path_to_design***t.v**

Chapter 9

Schematic Design Tutorial

This chapter contains the following sections:

- “Making Incremental Design Changes” section
- “Command Summaries” section
- “Further Reading” section

Introduction

This chapter guides you through a typical Field Programmable Gate Array (FPGA) and Complex Programmable Logic Device (CPLD) design procedure from schematic entry to completion of a functioning device. It uses a design called Calc, a 4-bit processor with a stack. In the first part of the tutorial, you use Concept, the Cadence design entry tool, to create the schematics and symbols for the Calc design. Next you use Verilog-XL, the Cadence Verilog HDL simulator, to perform a functional simulation on the design. In the third step, you use the Xilinx Design Manager to implement the design. Finally, you verify the design’s timing by again using Verilog-XL. The simple design example used in this tutorial demonstrates many system features that you can apply to more complex FPGA and CPLD designs as well.

Note: Although this tutorial describes creating and processing FPGA designs, you can apply most of the steps to CPLD designs.

This tutorial includes instructions on the following:

- Installing the tutorial files
- Targeting the tutorial design (Calc) to a XC4000E or an XC9500 device
- Using Concept
- Completing the ALU1 block in the Calc design
- Adding the STARTUP block to tie signals to the global reset
- Adding device information in the Calc design
- Exploring Xilinx library elements
- Exploring the XC4000E oscillator
- Controlling device layout from the schematic
- Modifying the Calc design for a non-XC4000E/EX device
- Converting the design to an EDIF file using CONCEPT2XIL

- Performing functional simulation on the Calc design in Verilog-XL
- Implementing the design using Design Manager
- Configuring the Xilinx Design Manager / Flow Engine
- Performing timing simulation on the routed Calc design in Verilog-XL
- Examining routed designs with the Editor for Programmable ICs (EPIC)
- Verifying the Calc design on a demonstration board
- Making incremental design changes
- Command summaries

Required Background Knowledge

This tutorial assumes that you have a basic understanding of the following:

- UNIX operating system
- Motif Windows. Cadence applications conform to the Motif window style.

Design Flow

See the "Introduction" chapter for the design flow when using the Cadence interface for describing the general steps for creating a design using the Cadence interface.

An incremental design methodology is described in this tutorial. In incremental design, the design is processed completely through the flow at least once; a small change is then made to the design; and the design is processed again. Place and route information from the previous design processing cycle is used to constrain subsequent cycles of the same design. When this method is used, timing information in a design remains relatively stable through many processing cycles. Also, place and route time is usually reduced since much of the processing has already been done in previous cycles.

The tutorial design can be targeted for an XC4000E or XC9500 device. You can use a Xilinx demonstration board to test the functionality of your design. Make sure your demonstration board and software support your selected device. To determine compatibility, refer to the release notes that came with your software package.

In this tutorial the following conventions are used to refer to the various device families:

- XC3000 family: includes XC3000A, XC3000L, and XC3100A, XC3100L devices
- XC4000 family: includes XC4000E, XC4000EX, XC4000L, XC4000XL, and XC4000XV devices
- XC5200 family: includes XC5200
- XC9500 family: includes XC9500

Software Installation

Required Software

The following versions of software are required to perform this tutorial:

- Cadence 9604 or later, including Concept, Verilog-XL, as well as the program needed to write EDIF netlists (concept2xil) from Cadence.
- Xilinx/Cadence Interface Version M1
- Xilinx Development System Version M1

Before Beginning the Tutorial

Before beginning the tutorial, set up your workstation to use Cadence and Xilinx Development System software as follows:

1. Verify that your system is properly configured. Consult the release notes that came with your software package for more information.
2. Install the following sets of software:
   - Xilinx Development System Version M1
   - Xilinx/Cadence Interface Version M1

- Cadence 9604 or later, including Concept, Verilog-XL, as well as the program needed to write EDIF netlists (concept2xil). CONCEPT2XIL requires that you have a valid Concept license only.

3. Verify the installation, using the "Setting Up Your Environment" section of the "Getting Started" chapter as a guide.

Standard Directory Structure

When a schematic is saved in Concept, a directory is created in the project directory with the same name as the schematic. This directory contains body files, logic files, and HDL Direct generated Verilog text files. HDL Direct will create a structural Verilog netlist of the schematic sheet, which the Xilinx toolset requires for translation.

Note: In this tutorial, file names and directory names are in lower case and the design example is referred to as Calc.

Tutorial Directory and Files

You will complete the Calc design in this tutorial. During the tutorial installation, the $XILINX/cadence/tutorial directory is created; design directories are created; and the tutorial files needed to complete the design are copied to the calc_sch directory. Some of the files you need to complete the tutorial design are not copied, because you will create these files in the tutorial. However, solutions directories with all input and output files are provided. They are located in the $XILINX/cadence/tutorial directory and are listed in the following table:

Table 9-1 Solutions Directories

Directory	Description
calc_sch	Schematic (Concept) tutorial directory
calc_4ke	Schematic solution directory for XC4003E-PC84
calc_9k	Schematic solution directory for XC95xxx-PC84
calc_blx	Schematic solution directory using LogiBLOX

The solution directories contain the design files for the completed tutorial, including schematics and the bitstream file. To conserve disk space, some intermediate files are not provided, except in the calc_4ke directory, which is complete. Different intermediate files are created for different device families. Do not overwrite any files in the solutions directories.

The calc_sch directory contains the incomplete copy of the tutorial design. The installation program copies a few intermediate files to the calc_sch tutorial directory, and you will create the remaining files when you go through the tutorial. As described in a later step, you will copy the calc_sch directory to another area and will work through the tutorial in this new area. The following table lists and describes the directories and files in the calc_4ke solution directory.

Table 9-2 Directories and Files in calc_4ke

Directory or File Name	Description
calc	Top-level design directory
control	Design directory for control module
statmach	Design directory for state controller module
alu1	Design directory for ALU1 module
alu_blox	LogiBLOX version of ALU1 design component
muxblk2	Design component for arithmetic function in ALU1
andblk2	Design component for arithmetic function in ALU1
orblk2	Design component for arithmetic function in ALU1
xorblk2	Design component for arithmetic function in ALU1
muxblk5	Design component for multiplexer arithmetic outputs in ALU1
muxlbk2a	Design component for multiplexer operator function in control
stack	Design component for stack
seg7dec	Design component for 7-segment decoder
debounce	Design component for debounce circuit
osc_3k	Design component interface to RC circuit on demonstration boards; generates clock
xilinx.run	Default run directory for CONCEPT2XIL
calcf.stim	Test fixture for use in Verilog-XL simulation
calc.edf	EDIF netlist files created by CONCEPT2XIL

Table 9-2 Directories and Files in calc_4ke

Directory or File Name	Description
concept2xil .log	CONCEPT2XIL log file
calc.ngo	Native Generic Object created by EDIF2NGD
calc_4ke.ucf	User Constraints File
calc.ngd	Native Generic Design created by NGDBuild
calc.mrp	Mapping report created by MAP
calc.pcf	Physical Constraints File created by MAP
calc.ncd	Native Circuit Description created by MAP
calc_r.ncd	Routed NCD file created by PAR
calc_r.twr	Timing report created by Trace (TRCE)
time_sim.v	Structural Verilog netlist of Calc for simulation
time_sim.sdf	Standard Delay Format file for timing simulation

Note: The terms "testbench" and "test fixture" are used synonymously throughout this manual.

Copying the Tutorial Files

Perform the following steps to make a working copy of the tutorial files:

1. At the UNIX prompt, create a local directory that will serve as the location of the working copy of the tutorial files.
2. Copy the files from $XILINX/cadence/tutorial/calc_sch to your local directory, as well as the subdirectories. For example, you might enter:

   cp -r $XILINX/cadence/tutorial/calc_sch *local_tutorial_directory*

Setting Up for Concept

Concept requires several files to be present in the design directory. The $XILINX/cadence/tutorial/calc_sch directory will contain these files, but you must modify some of them for your environment.

- global.cmd

You will need a global.cmd that references the proper libraries. Here is an example global.cmd:

```
master_library "./master.local" ;
library "xce4000e" ,
        "xcepads",
        "hdl_direct_lib",
        “standard” ;
use "my_design.wrk" ;
root_drawing "my_design" ;
```

- The actual path to each library is defined in the *master.local* file. The *master_library* line, (the first line in the global.cmd file), points to this local library file, which is located in the design directory.
- Also note that one of the entries in the “library” listing should point to the library supporting the Xilinx architecture you are using. In the example above, this corresponds to the *xce4000e* library, which supports the Xilinx XC4000E/L architectures.
- Note the presence of the “hdl_direct_lib” library; SLICE, INPORT, OUTPORT, and IOPORT bodies required for HDL Direct compliance are provided here. SLICEs should be used in place of taps and ctaps in the “standard” library for tapping bits off a bus, and INPORT, OUTPORT, and IOPORT bodies are used on interface signals in a logic drawing that connects to a higher level symbol body.
- The “xcepads” library contains the Xilinx pad symbols.
- The “use” line points to a file (typically with a .wrk extension) that Concept can use to store information about your design.

- master.local

  This file contains the actual UNIX paths to the libraries referenced in global.cmd. It does not need to contain the path to libraries that are local. The following is an example master.local file for a 4000E design:

```
file_type = master_library;
"xce4000e" '/xlx/cadence/data/xce4000e/xce4000e.lib';
"xcepads" '/xlx/cadence/data/xcepads/xcepads.lib';
end.
```

Other Cadence supplied Concept libraries (such as the HDL Direct library) do not need to be referenced here, assuming there is an entry in the $CDS_INST_DIR / lib / master.lib file pointing to those libraries. Do not use variables (such as $XILINX) in this file; absolute path names are required.

For the Calc design, you must modify master.local to point to *location_of_Xilinx_software* / cadence / data / xce4000e / xce4000e.lib.

- *design*.wrk

  The actual filename for this file may vary, but the extension is always '.wrk'. This file contains the correlation between the names of the various schematics and the UNIX directory in which they are stored. This file is automatically created by Concept if it is referenced in a "use" line in the global.cmd. No alterations are necessary.

- cds.lib

  This file is required by CONCEPT2XIL, and it must point to the location that contains VAN-compiled (Verilog Analyzer-Compiled) library files for the compiler. As an example, here is a sample cds.lib file for a 4000E design:

  ```
  define xce4000e_syn /xlx/cadence/data/xce4000e_syn
  ```

  The format for entries in this file is:

  define *target_tech*_syn *path_to_XILINX* / cadence / data / *target_tech_syn*

  where *target_tech* is xce3000, xce4000e, xce4000x, xce5200, or xce9000.

Using HDL Direct

The M1 release of Xilinx tools does not support SCALD methodology when entering designs. One of the implications of this is that the HDL Direct library component counterparts must be used in place of any "standard" library components whenever one exists. Furthermore, HDL Direct must be enabled whenever a schematic sheet is saved. It is best to put the following commands in your startup.concept file to activate HDL Direct every time Concept is invoked:

```
set hdl_direct on
set hdl_checks on
set check_signames on
set check_net_names_hdl_ok on
set check_port_names_hdl_ok on
set check_symbol_names_hdl_ok on
set capslock_off
runopl installation_path_to_cadence/tools/fet/concept/
hdl_direct/bin/autosym
```

When processing designs entered using SCALD methodology, refer to Appendix C of the *Concept User Guide* (from Cadence) for guidelines on converting these designs for HDL Direct compliance.

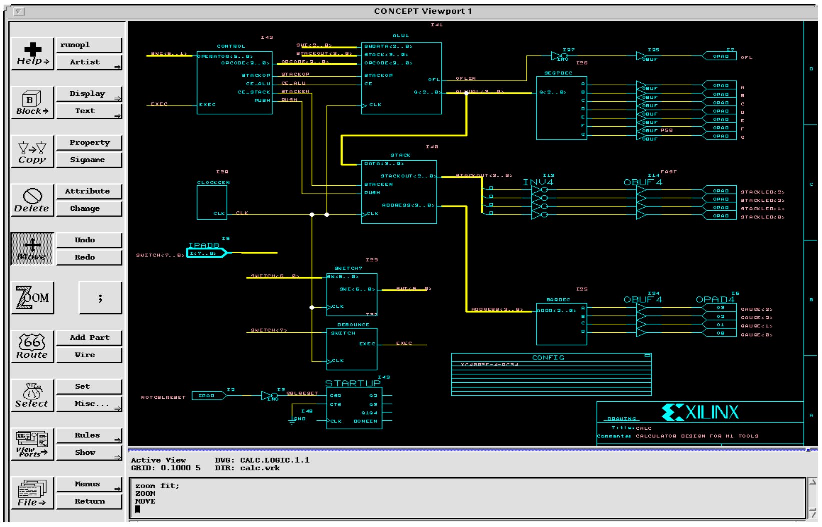
Also note that in this release, the SIZE property is not supported. *Iterated instances* should be used instead (which essentially consists of adding a bus index to the PATH attribute of the symbol body instance). Refer to the Cadence *HDL Direct User Guide* for more information on iterated instances.

Starting Concept

To open the Calc design in Concept, simply type

```
concept &
```

on the UNIX command line, in the design directory. Resize the Concept window to cover the entire screen.



```
Active View      DWG: CALC.LOGIC.1.1
GRID: 0.1000 5   DIR: calc.wrk
```

```
zoom fit;
ZOOM
MOVE
```

Figure 9-1 Top-Level Schematic for Calc

Using the Mouse in Concept

Left Mouse Button

Use the left mouse button to operate on individual items, and to start and terminate wires at the nearest grid intersection. For instance, if you wanted to terminate a wire at the nearest grid point in the schematic, then you would use the left mouse button.

Middle Mouse Button

Use the middle mouse button to operate on groups, and to modify the action of the Wire and Add commands. For example, pressing the middle mouse button while the Wire command is active will cause the wire to be routed directly, instead of orthogonally. In the case of the Add command, the middle mouse button causes the object to be added to rotate.

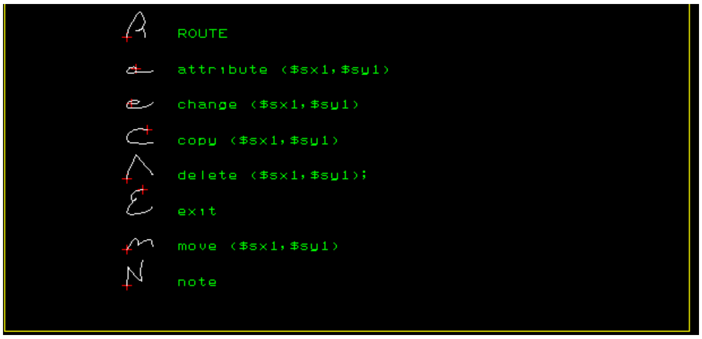
Using the middle mouse button when a group of objects has been selected causes the operation to be performed on the entire group, not just the nearest individual object. For example, if Select is used to highlight several objects, and Delete is then selected, the middle mouse button will delete the entire highlighted group.

Right Mouse Button

Use this button to attach wires to the nearest pin or wire (as opposed to the left mouse button, which will attach to the nearest grid point).The right mouse button will also pan across the schematic if held down while moving the mouse.

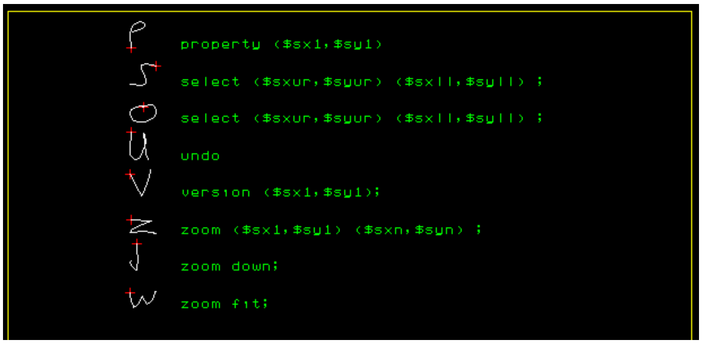
Strokes

Use the left or middle mouse button to perform actions known as strokes. You can use strokes as shortcuts to perform common tasks. Perform a stroke by pressing and holding the middle mouse button while moving the mouse to draw a line with a specific shape. For instance, the stroke with the shape “Z” will zoom in. When applicable, strokes are used in this tutorial. You can view all the default strokes in the Stroke Editor by entering **sted $CDS/tools/fet/concept/ concept.strokes** at the UNIX prompt.



```
ROUTE
attribute ($sx1,$sy1)
change ($sx1,$sy1)
copy ($sx1,$sy1)
delete ($sx1,$sy1);
exit
move ($sx1,$sy1)
note
```

Figure 9-2   STED Default Strokes, page1



```
property ($sx1,$sy1)
select ($sxur,$syur) ($sxll,$syll) ;
select ($sxur,$syur) ($sxll,$syll) ;
undo
version ($sx1,$sy1);
zoom ($sx1,$sy1) ($sxn,$syn) ;
zoom down;
zoom fit;
```

Figure 9-3   STED Default Strokes, page2

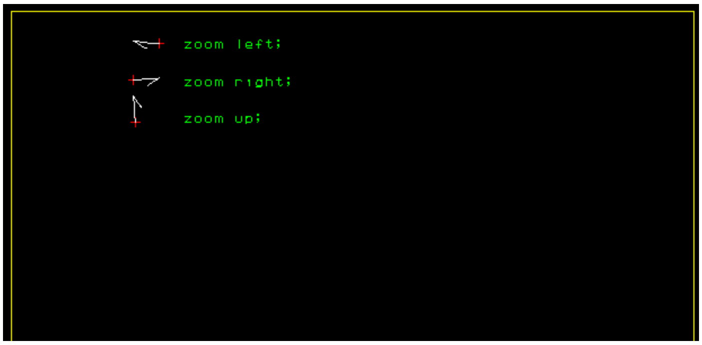


Figure 9-4 STED Default Strokes, page3

Selecting Commands from the Menu Bar

Use the left mouse button to select commands from the menu bar at the left of the screen, which is called the Command Menu.

Entering Commands from the Keyboard

You can type commands in the Concept Message Window. The Message Window is located at the bottom of the screen. For example, a schematic sheet can be opened by typing the command “edit *sheet_name*” in the window.

Cancelling Commands

When you select a command, it is highlighted in the Command Window and echoed to the Message Window. You can cancel commands by selecting the “;” button in the Command Window, typing “;” in the Message Window, or selecting a new command.

Manipulating the Screen

To zoom in on a specific area of the screen, select the Zoom command in the Command Window, and use the mouse to create a box around the area you want to zoom on.To view the entire schematic, press F2 or enter the stroke shape “W”. (You can also perform the command by entering “zoom fit” in the Message Window.)

Saving a Design Directory

To save the current logic or body drawing, select **File** → **Write** in the Command Menu. Alternatively, you may enter “wr” or “write” in the Concept Message Window at the bottom of the Concept screen.

Quitting Concept

To exit out of Concept, select **File** → **Quit** in the Command Menu.

Completing the Calc Design

To complete the tutorial design, you need to add a few design objects to the Calc schematic using Concept.

If you need to stop the tutorial at any time, be sure to save the work you have done by first selecting **File** → **Write** from the Command Menu. A window appears that reports the results of HDL Direct. After reviewing the contents of this window, press Return inside that window to close it. If it reports errors, you may ignore them and write it out anyway, if you wish.

Design Description

The top-level schematic of the Calc tutorial design has been created for you. Each of the blocks in the schematic, such as CONTROL or ALU1, is linked to a second-level module that describes its logic. Additionally, any second-level module can contain another block that references a third-level drawing, and so on. This organization is known as a hierarchical structure.

In this tutorial, you add three symbols to the ALU1 block schematic to complete it. First, you create the ANDBLK2 and ORBLK2 symbols and their underlying schematics and then add them to the schematic. Additionally, you add the FD4RE symbol from the Unified Libraries to the ALU1 block. After the ALU1 block is finished, you add the STARTUP block to the top-level Calc schematic to tie the device’s global reset network to a device pin. To complete design entry, you add a CONFIG block, which lists a set of instructions that dictate how the implementation tools should process the design.

The Calc design is a four-bit processor with a stack. The processor performs functions between an internal register and either the top of the stack or data input from external switches. The results of the various operations are stored in the register and displayed in hexadecimal on a seven-segment display. The top value in the stack is displayed in binary on a bar LED. A count of the items in the stack is displayed as a “gauge” on another bar LED.

The design consists of the following functional blocks:

- ALU1

  The arithmetic functions of the processor are performed in this block.

- CONTROL

  The opcodes are decoded into control lines for the stack and ALU1 in this module.

- STACK

  The stack is a four-nibble storage device. It is implemented using synchronous RAM in the XC4000E design. You can substitute the RAM4_9K module, which uses flip-flops, in place of the RAM16X4S macro in the STACK schematic to implement the stack in an XC9500 or other non-XC4000E device.

- DEBOUNCE

  This circuit debounces the “execute” switch, providing a one-shot output.

- SEG7DEC

  This block decodes the output of the ALU1 for display on the 7-segment decoder.

- CLOCKGEN

  CLOCKGEN uses an internal oscillator circuit in XC4000E devices to generate the clock signal. In XC9500 designs, it is replaced by an input pad and a clock buffer.

Note: The XC3000 and XC5200 FPGA families also have on-board oscillators. See the CLKGEN3K and CLKGEN5K components included in the calc_sch directory to see how the oscillators in these families are used.

- BARDEC

  BARDEC shows how many items are on the stack on a “gauge” of four LEDs.

- SWITCH7

  SWITCH7 is a user-defined module consisting of seven input flip-flops used to latch the switch data.

Before proceeding, close (quit) the Calc schematic window. If a dialog box appears asking if you want to save any changes, choose NO.

Targeting XC9500 Devices

If you wish to target an XC9500 part, you may proceed with this tutorial (even though it is set up for a 4000E part). There will be a discussion later in this tutorial on what is needed to convert the design for a XC9500 instead of the XC4000E.

Creating Schematics for ANDBLK2 Symbol

You will need to create schematics and symbols for ANDBLK2 and ORBLK2. The schematics can then be referenced in a higher-level schematic by instantiating the corresponding symbol bodies.

Opening a Schematic

To open a new schematic sheet for ANDBLK2, type “edit andblk2” in the Message Window.

Adding the First Component to a Schematic

1. From the Command Menu, select Add Part. The Component Browser appears. See figure below.

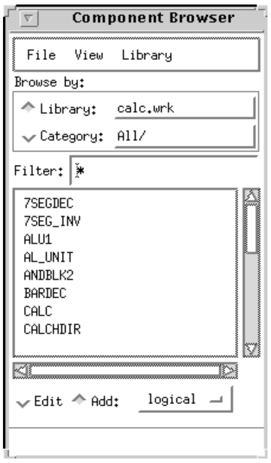


Figure 9-5 Component Browser Menu

2. Select components from the correct library for the device you are targeting (xce4000e for this tutorial). There are several libraries defined in the Component Browser; to select the library you wish to add components from, use the left mouse button on the Component Browser button next to “Library:”, as seen in the following figure. Select the xce4000e library.
3. Scroll down and select AND2.
4. The outline of a 2-input AND gate appears in the schematic window.
5. Move the symbol outline roughly to the center of the schematic entry window and then click the left mouse button to place the object.

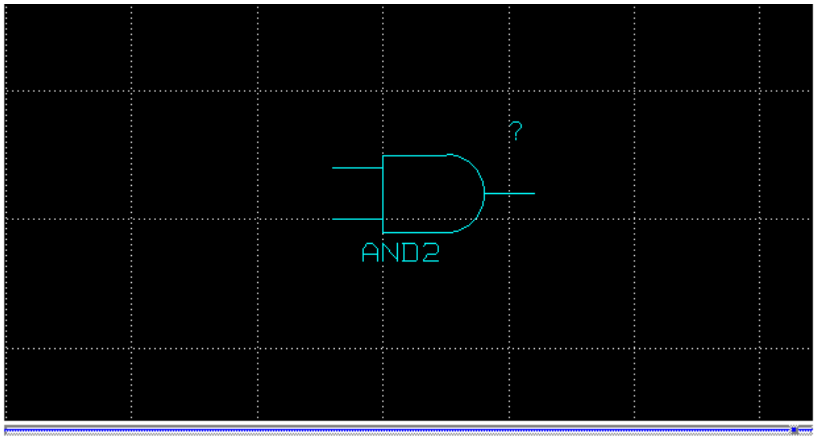


Figure 9-6 Placing a Component

Placing Additional Components

To select another component of the same type, click the middle mouse button (while you are still in "Add Part" mode) anywhere in the schematic window. Then position the component, and use the left button to place it on the sheet. Using this method, select and place a second AND2 symbol.

Copying a Component

Use the Copy command to add more components by copying a component that already appears on the schematic.

1. Select the Select command in the Command Menu with the left mouse button.
2. Move the mouse above and to the left of the two symbols on the sheet, and click the left mouse button.
3. Move the mouse below and to the right of the two symbols. A white box appears surrounding the two symbols.
4. Click the mouse button again to select the objects inside the box. All selected items should now be highlighted. Note that if you zoom or pan across your schematic at this point, all selected components will be deselected.
5. Select the Copy command in the Command Menu.

6. Click the middle mouse button to copy the entire highlighted group (left mouse button would only copy the nearest component, and not the entire group). Place the two copied gates above the original two and click the left mouse button. If necessary, use the right mouse button to pan across the schematic (hold it down while moving mouse).
7. Press F2 to view the entire schematic (or use stroke "W"). The schematic now looks like the following figure.

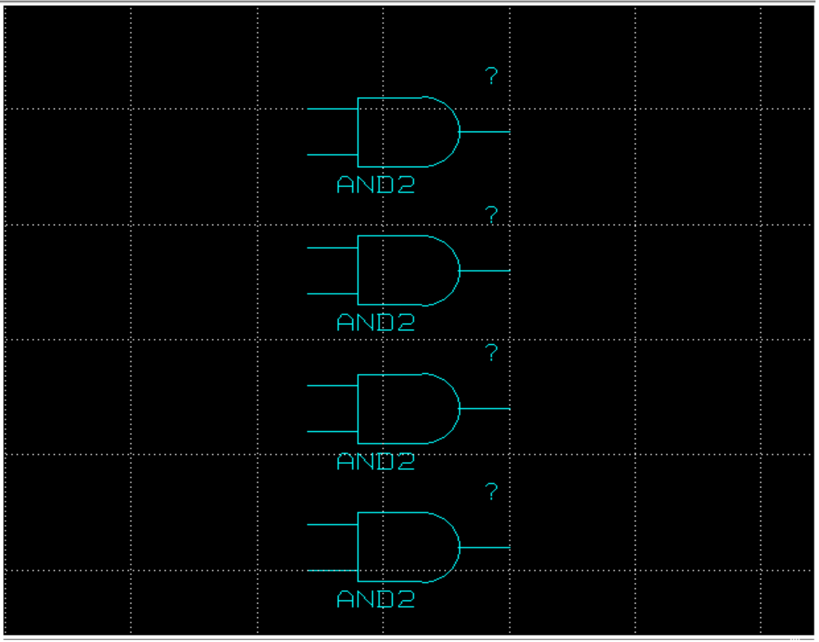


Figure 9-7 Component Placements for ANDBLK2

Moving a Component

If you make a mistake when placing a component, you can use the menu commands to move the component.

1. Select the Move command in the Command Window, and click on the component to be moved with the left mouse button.
2. Move the component to the desired location, and again click the left mouse button.

Adding and Labeling Buses in a Schematic

Sometimes it is convenient to draw a set of signals as a bus rather than as several separate wires. It is not necessary to physically connect a bus to the nets that make up the bus. There are several schematics in the Calc design that have short bus segments that are not connected to anything. This is done to establish pin connectivity to a bus pin on the symbol corresponding to the schematic. A bus must exist on the schematic if you wish to use a bus pin to represent a set of signals.

In Concept, a bus is put on the schematic as a wire. When a label with a bus index is attached to the wire, Concept will automatically change the wire to a bus (it is redrawn as a heavier line).

Add buses to the schematic as follows:

1. Select Wire from the Command Menu, or enter “wire” in the Message Window.
2. Draw a wire by clicking the left mouse button to specify the starting point, moving the mouse to a new position, and then clicking the button again to make a bend in the wire or to connect it to a pin. Terminate the wire by clicking the mouse button in the same place twice. Add the three buses shown in the following figure (add wires where you see a bus in the figure; they will be converted to buses in the next step). You may want to zoom the schematic view out before adding the buses.

   If you make a mistake, select Delete in the Command Menu and click on the wire to be removed.
3. After adding the three buses, select Signame in the Command Menu.
4. Click on the leftmost wire (labeled A<3..0> in the figure). A small red box should appear next to the wire. In the Message Window, type “A<3..0>” and hit Return to label the bus.
5. Use a procedure similar to the previous step to label the other wires “B<3..0>” and “Q<3..0>”.

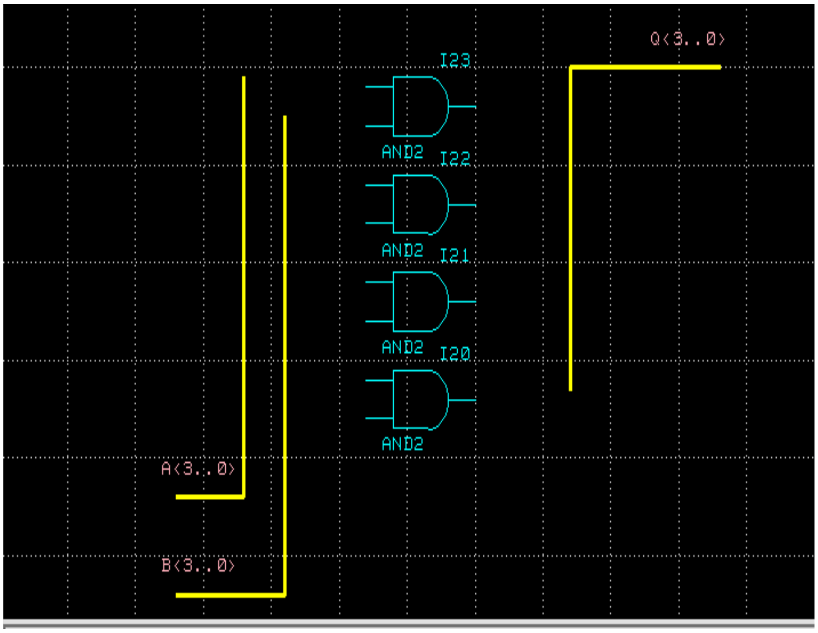


Figure 9-8 ANDBLK2 Schematic with Buses

Adding Wires and SLICEs to a Schematic

Next, wires must be added to attach the appropriate pins on the gates to the buses. You may want to enlarge the view of the components to make it easier to draw the nets.

In order to connect a wire to a bus, a SLICE should be added from the hdl_direct_lib. The SLICE can connect a bus to a wire.

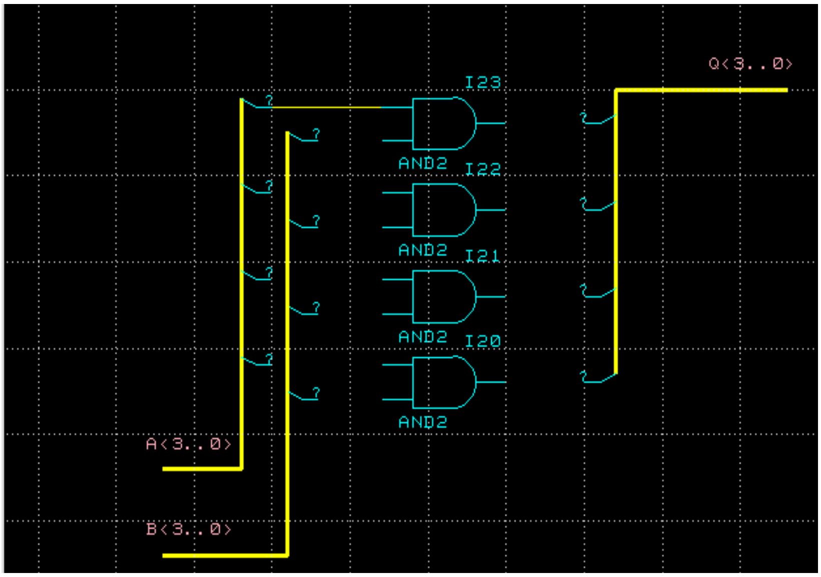
1. Select Add Part in the Command Menu. The Component Browser should come up.
2. Click the left mouse button on the button to the right of "Library:" in the Component Browser. Select hdl_direct_lib. Scroll down the menu to select SLICE with the left mouse button.

Note: Do not use the TAP from the older SCALD standard library; HDL Direct compliance requires that you use the appropriate corresponding hdl_direct components in place of SCALD components wherever applicable.

3. Move the cursor into the schematic window, and click the middle mouse button to rotate the SLICE symbol. Click the mouse until the SLICE achieves the orientation shown in the following figure (for the A<3..0> bus). Then, place the SLICE so the left diagonal side touches the bus, and the right horizontal side is in line with one of the AND2 pins. Click the left mouse button to place.

Note: It is not necessary that the SLICE should be in any particular orientation, except that the diagonal portion should be connected to the bus.

Click the left or middle mouse button after placing the SLICE to select a new SLICE. Note this is a copy of the previous SLICE, so there is no need to rotate it. Place this SLICE below the previous SLICE, so it aligns with the next AND2's top pin.



4. Place the remaining 2 SLICEs for the A<3..0> bus, and add 4 SLICEs for the B<3..0> bus and Q<3..0> bus. You may perform this step either by continuing to click the left or middle mouse button (assuming you have not left Add Part mode), or by selecting SLICE each time in the Component Browser. Another method is to type "add slice" in the Message window.

Note the following figure shows a different orientation for the Q<3..0> SLICE. Simply click the middle mouse button until this orientation is achieved.

If you make a mistake, you may select either Delete or Move.

5. Select “wire” from the Command Menu or enter “wire” in the message window.
6. Move the cursor to the top input pin of the top AND2 gate, then click the left mouse button.
7. Move the cursor to the left of the SLICE at the top of the leftmost bus, so that the wire connects to the output of the SLICE. Click the left mouse button to terminate the wire.
8. Add the remaining nets as shown in the following figure.

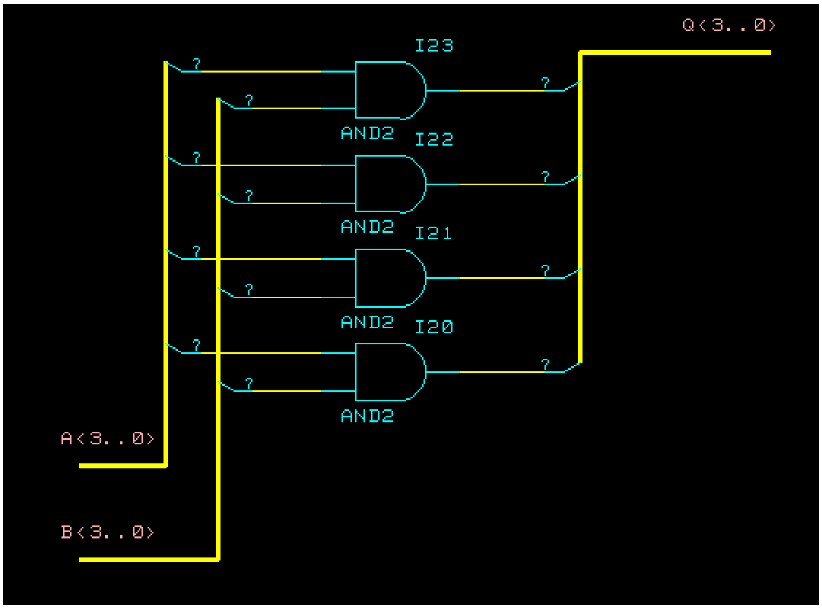


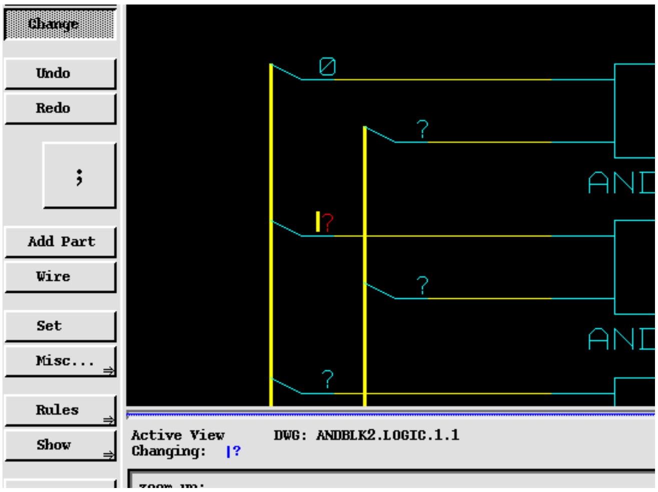
Figure 9-9 ANDBLK2 with All Wires and Buses Connected

Adding Values to SLICE Symbols

At this point, all buses in the ANDBLK2 schematic have been labeled. However, for the nets that have been ripped off the bus via the SLICE symbol, there needs to be an attribute attached to the SLICE to indicate which bit of the bus the net represents. The current default value is “?”; this will need to be changed to a number within the bus bounds.

You may change these values by individually changing the BN attribute on each SLICE, or by using the “bustap” command to change this attribute on a set of SLICEs, all at once.To change each value individually:

1. Select Change in the Command Menu. Use the left mouse button to select the top SLICE of A<3..0>. The “?” (default undefined BN value) should turn red, and a cursor should appear beside it. Hit the right cursor followed by the BackSpace key to delete the current value, and type in “0” followed by Return. The SLICE should now appear as in the figure below. If you accidently select any elements besides the SLICE you wish to edit, select the “;” key in the command menu and reselect Change.



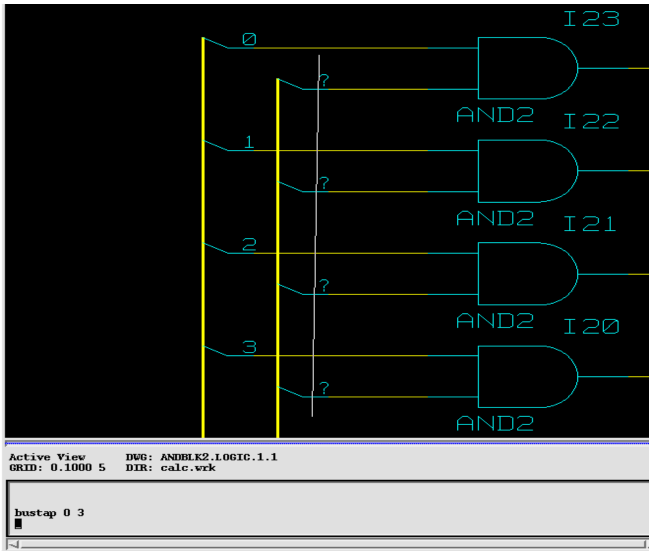
```
Active View      DWG: ANDBLK2.LOGIC.1.1
Changing:   |?
```

Figure 9-10 Adding an Attribute to a SLICE

2. Repeat this procedure for all SLICEs for the A<3..0> bus only. Label from top to bottom, with “0” on top and going sequentially to “3” on the bottom. Reference the “Using the Bustap command” figure.

   The label sizes are increased as shown in the “Using the Bustap command” figure.

3. Although we can use the same procedure to add values to the SLICEs of the other two buses, we can alternatively use another command called “bustap”. Type “bustap 0 3” in the Message Window.



```
Active View      DWG: ANDBLK2.LOGIC.1.1
GRID: 0.1000 5   DIR: calc.wrk
```

```
bustap 0 3
```

Figure 9-11 Using the Bustap command

4. Position the mouse cursor above the first “?” on the top SLICE of the B<3..0> bus, and click the middle mouse button. Position the mouse below the last “?” on the bottom SLICE of the B<3..0> bus, as shown in the figure.
5. Click the middle mouse button again. You should see all the “?” change to numbers, beginning from 0 and going to 3.
6. Use either Change or bustap to modify the SLICEs on the Q<3..0> bus.

Adding Ports

Port symbols must be added to nets and buses to define the connectivity between a schematic and its associated symbol. For the ANDBLK2 schematic, all three buses need ports. Input signals are given INPORTs and output signals are given OUTPORTs.

Add ports to the schematic as follows:

1. Select Add Part mode from the Command Menu, and select the hdl_direct_lib library in the Component Browser.
2. Scroll down to select the INPORT symbol.
3. Position the INPORT so that the symbol pin's right side touches the A<3..0> bus's left end. Click the left mouse button to place.
4. Place another INPORT at the end of the B<3..0> input bus, on the left side of the window.
5. Next select an OUTPORT symbol from the library and place it at the end of the output bus. Instead of selecting it from the Component Browser, try typing "add outport" in the Message Window.
6. Press F2 to view the entire schematic. The schematic appears as in the following figure.

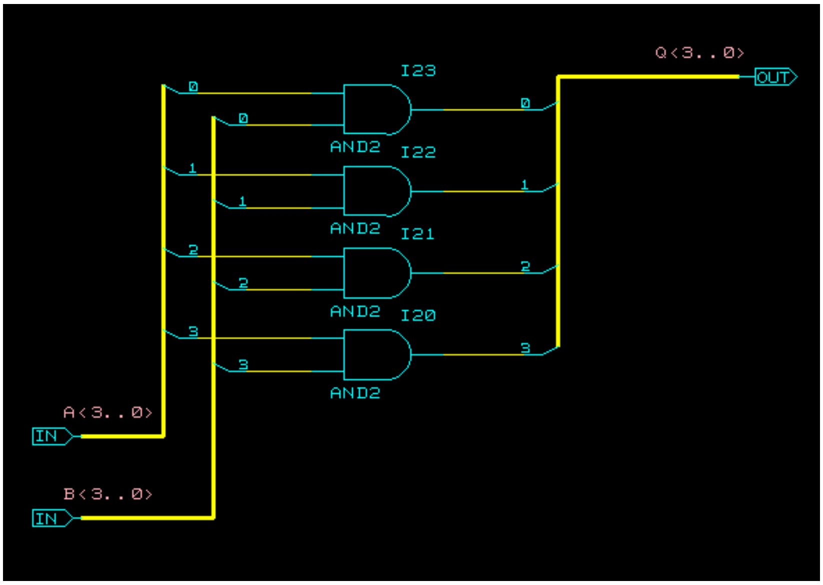


Figure 9-12 Adding Ports

Saving the Schematic

The schematic is now complete. Check and save the schematic as follows:

1. Select **File** → **Write** in the Command Menu. If errors occur, recheck the schematic against the figure below. You may use the command “error” to have Concept show on the schematic itself what errors exist in the drawing. Once Concept has finished checking the schematic, it will automatically call HDL Direct to write out the Verilog netlist. You should see a separate Xterm pop up, and this check must complete successfully also. If not, then you must correct the schematic (the Concept command “error” does not point out errors found in the HDL Direct window).

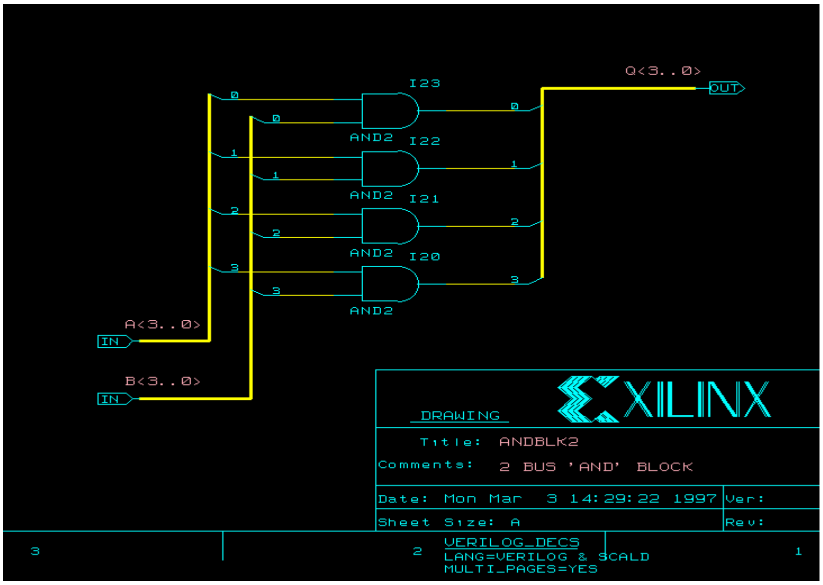
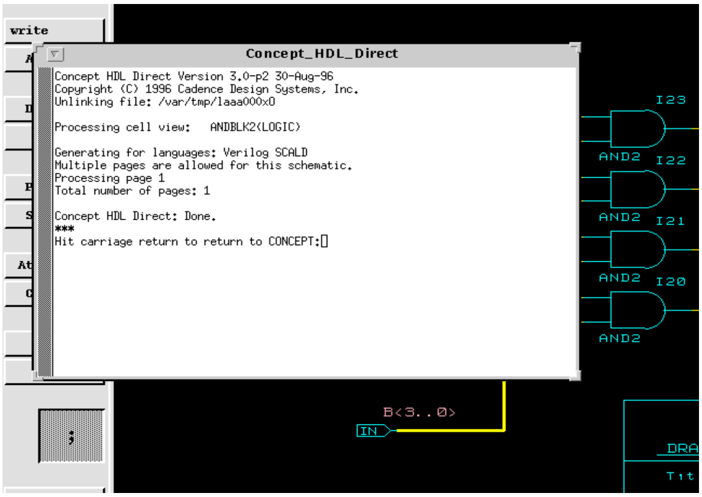


Figure 9-13 Completed ANDBLK2 Schematic

2. Once all schematic errors have been corrected, write the design again if necessary.



```
Concept HDL Direct Version 3.0-p2 30-Aug-96
Copyright (C) 1996 Cadence Design Systems, Inc.
Unlinking file: /var/tmp/laaa000x0

Processing cell view:   ANDBLK2(LOGIC)

Generating for languages: Verilog SCALD
Multiple pages are allowed for this schematic.
Processing page 1
Total number of pages: 1

Concept HDL Direct: Done.
***
Hit carriage return to return to CONCEPT:
```

Figure 9-14 Successful Write with HDL Direct

Creating Schematics for ORBLK2 Symbol

The ORBLK2 schematic is similar to the ANDBLK2 schematic. To create schematics for the ORBLK2 symbol, you can use the ANDBLK2 schematic and simply replace the four AND2 gates with OR2 gates.

1. Press the F2 key to unselect everything on the ANDBLK2 schematic and to view the entire schematic.
2. Enter in the Message Window the command “replace or2”. This will cause any component selected to be replaced by the OR2 symbol.
3. Click on the four AND2 gates with the left mouse button. They should be converted to OR2 gates.
4. Enter the command “write orblk2” in the Message Window. This should cause the schematic to be saved under the new name “orblk2”.

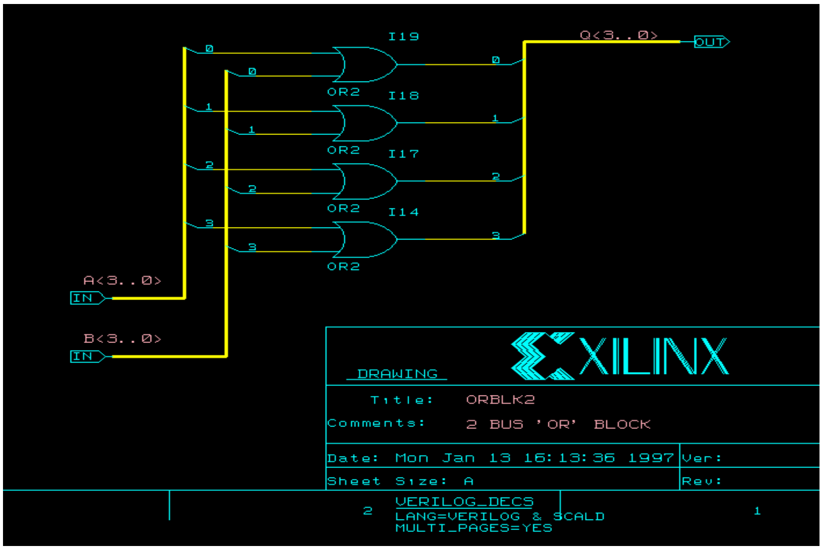


Figure 9-15 Completed ORBLK2 Schematic

Creating the ANDBLK2 Symbol

The following subsections explain how to create the ANDBLK2 symbol.

Creating the Symbol Outline

1. Enter the command "edit andblk2.body" in the Message Window. Previously, when you typed in the command "edit", you did this to create or modify a schematic. When you type "edit *component_name*", it is understood that you mean "edit *component_name*.logic." The "logic" extension references the schematic, whereas the "body" extension references the symbol.

2. A blank grid should appear, with the name of the symbol and an origin "X" in the center. We can now begin to draw the symbol's body with the "wire" command. Select "wire" in the Command Menu.

3. Draw an enclosed shape (usually a rectangle) around the origin. Make it large enough to have 2 inputs on the left side and 1 output on the right side.

4. Add wire stubs to represent the symbol pins. Each pin should be a wire segment that begins on the outline of the main body, and points away from the body. Each pin should also begin and end on a grid point. For the andblk2 symbol, it is best to place the two input pins on the left and one output pin on the right. You should have a total of three pins.
5. Enter the command "dot" in the Message Window. Position the mouse over the dangling end of a pin, and click to place the dot. Place a dot at the end of all other pins.
6. Select the command "signame". Click on the upper left dot, and type in the string "A<3..0>". This should now appear on the drawing. The name of the pin must exactly match the name of the signal connected to the INPORT or OUTPORT symbol. Do the same for the lower left pin (B<3..0>) and the right pin (Q<3..0>).

Adding Text

1. Adding signal names (signames) to the pins correlates the pins to the underlying schematic, but these signames are not visible on the symbol when it is used. To add a visible label to the pin, use the command "note". After this command has been entered, type in the string "A<3..0>", and position it near the top left pin, and inside the symbol.
2. If the text is too large for your taste, type in the command "set size 0.5" to halve the size of the text. Note this will not reduce the size of existing text; if you wish to replace the existing text with smaller text, you must delete it and re-enter it with the "note" command after you have entered "set size 0.5".
3. Place the text "B<3..0>" and "Q<3..0>" by their respective pins. Also add the name of the symbol ANDBLK2 to the body drawing by attaching it to the body as another NOTE at the top of the symbol.

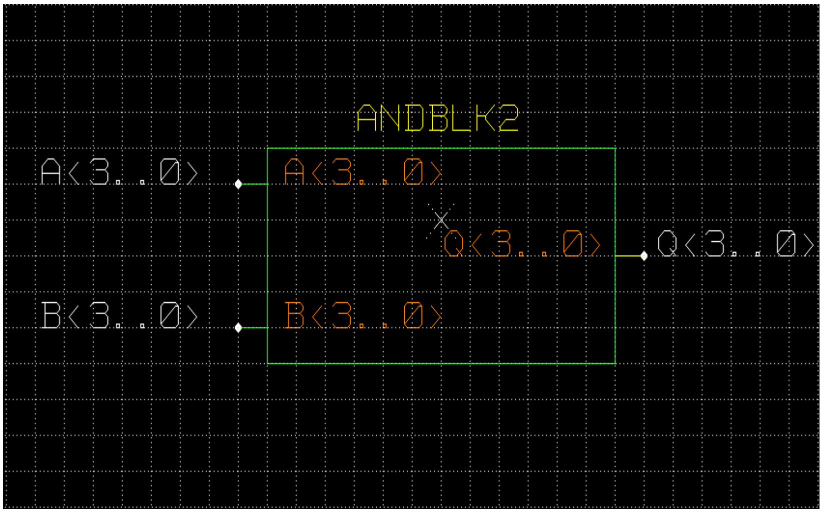


Figure 9-16 Completed ANDBLK2 Symbol

4. Once you are satisfied with the drawing, do a “write”.
5. To return to the normal text size, enter “set size 1”.

Creating the ORBLK2 Symbol

The next step is to create the symbol for ORBLK2, as shown in the following figure. Since ORBLK2 is similar to ANDBLK2, use the ANDBLK2 symbol and modify the text.

1. Select the Change mode. Select the text ANDBLK2, and convert it to ORBLK2. No other modifications should be necessary.
2. Enter in the command “write orblk2.body”. This command will cause Concept to save the modified schematic under the new name.

Editing the ALU1 Schematic

So far you have created symbols for ANDBLK2 and ORBLK2. You have also created underlying schematics for these symbols. The next step is to place the symbols in the ALU1 block schematic.

1. Enter “edit calc” in the Message Window (remember that this is the same as “edit calc.logic”).

2. If the Component Browser is already up, select the calc.wrk library and click on the ALU1 component. Select the logic view (ALU.LOGIC.1.1). The ALU1 schematic should now appear. If the Component Browser had been previously closed, then you must type “edit” once to bring it up.

Note: We could have entered “edit alu1” instead (replacing both step 1 and step 2), but this demonstrates that the mouse can be used to descend hierarchy when in the “edit” mode. This will only work if the Component Browser is up; if it is not, then you must type edit once to bring up the Component Browser.

Placing User-Created Components

The ANDBLK2 and ORBLK2 symbols can now be placed on the schematic as shown in the figure below. The symbols can be placed using the same procedure used to place the AND2 gate from the Xilinx libraries when you created the ANDBLK2 schematic.

1. Use the Zoom button to zoom into the empty area near the center of the schematic, between the XORBLK2 and ADSU4 symbols.
2. Select Add Part. From the Component Browser, select the calc.wrk library, and scroll down to choose the ANDBLK2 component. All user-created components come from the *design*.wrk library (or whatever file was referenced in the “use” line in the global.cmd file). Alternatively, we could have entered “add andblk2” in the Message Window.
3. Move the cursor to the correct location as shown in the figure below.

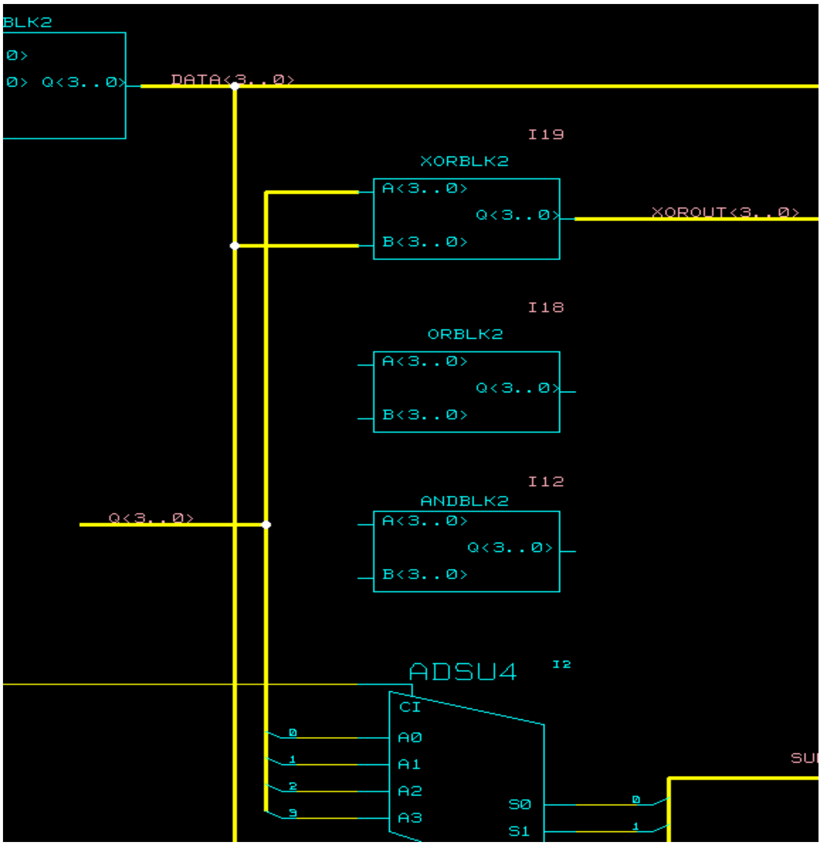


Figure 9-17 Adding ANDBLK2 and ORBLK2 to ALU Schematic

4. Press the left mouse button to place the component.

5. Follow the same procedure to add the ORBLK2 symbol. Refer to the ALU1 schematic in the preceding figure for proper placement. If you make a mistake when placing a component, enter Move mode and select the component to move it.

The next step in the tutorial is to add the FD4RE and AND5B2 components to the ALU1 schematic. The FD4RE component is available in the xce4000e Xilinx Unified Libraries and consists of four flip-flops with a clock enable. The AND5B2 component is a five-input AND gate with two inputs inverted (or “bubbled,” hence the “b” in the component name).

Note: These components are available in all libraries, including those for the XC4000E and XC9500 devices.

1. Use F2 to display the entire ALU1 schematic. Use the Zoom button to zoom into the open area in the lower right-hand corner.
2. Select Add Part from the Command Window.
3. Select the appropriate family library (xce4000e or xce9000) in the Component Browser.
4. Choose FD4RE from the menu. Move the cursor into the schematic window; an outline of the FD4RE component appears.
5. Move the component to lower right corner of the schematic, approximately to the location shown in the figure below.
6. Press the left mouse button to place the component.
7. Repeat steps 4 through 6 to place the AND5B2 component next to the FD4RE as shown in the following figure. Note that you may instead use the command "add and5b2", if you wish.

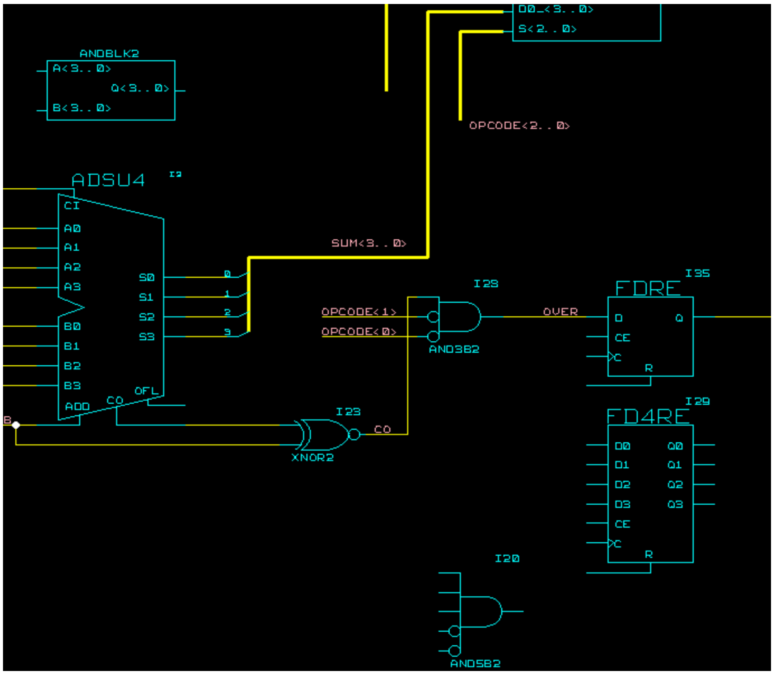


Figure 9-18 Adding FD4RE and AND5B2 to ALU Schematic

Adding Nets, Buses, Ports and Labels

FD4CE and AND5B2

Next complete the addition of the FD4RE and AND5B2 symbols by adding nets, buses, and labels as follows:

1. Add the necessary nets and buses to complete connections for FD4RE and AND5B2 as you did for the previous schematic. The following figure displays the labeled nets and buses for FD4RE and AND5B2.
2. Add ports to the nets and buses attached to the FD4RE and AND5B2, as shown in the following figure. INPORTs should be attached to nets and buses corresponding to ALU1 inputs, and OUTPORTs to ALU1 outputs.

Note: You should always use the IPAD / OPAD symbols for your external FPGA / CPLD ports; INPORT and OUTPORT are used for connections to different levels of hierarchy in your design.

3. To add net names to nets, remember to use Signame in the Command Window, and type in the name of the signal. The red box over the net should be replaced with your net name after you press Return.

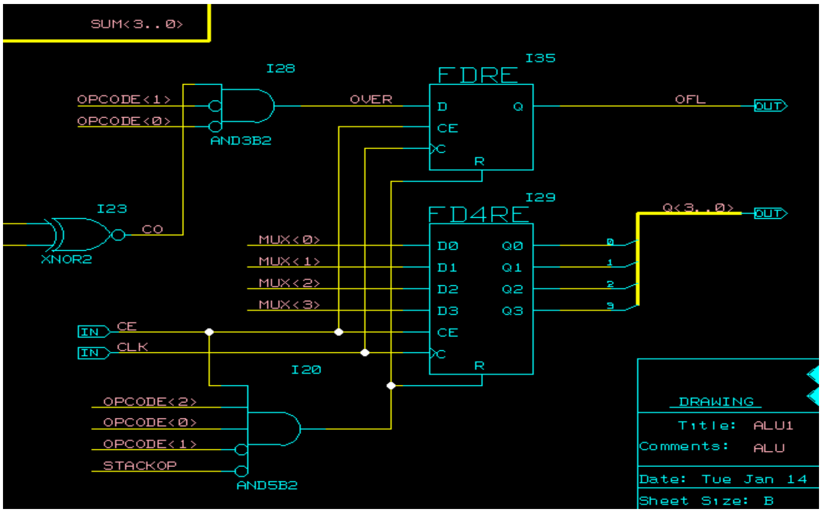


Figure 9-19 Nets, Buses, and Ports for FD4RE and AND5B2

Warning: Whenever you take an existing schematic and add or remove INPORTs or OUTPORTs, or IPADs or OPADs, HDL Direct will report:

```
Error #171: Port exists in entity declaration
but not in the schematic. The entity declaration
may need to be updated.

Would you like to overwrite the entity declara-
tion calc_9k/calc/entity/vhdl.vhd with a new
entity declaration derived from this schematic?
[y/n]
```

Answer “y” to this query. This is not an actual error. If you want to be sure everything is OK, then do another “write”. You should get no error messages this time.

ANDBLK2 and ORBLK2

Next, complete the addition of ANDBLK2 and ORBLK2 to the ALU1 schematic.

1. Add the necessary buses to complete connections to ANDBLK2 and ORBLK2. The following figure displays the labeled nets and buses for ANDBLK2 and ORBLK2.
2. Use the following figure to name the added buses by using the same Signame command from the previous section. You only need to label the output buses of the two components, since the inputs to these components are connected to pre-labeled buses.

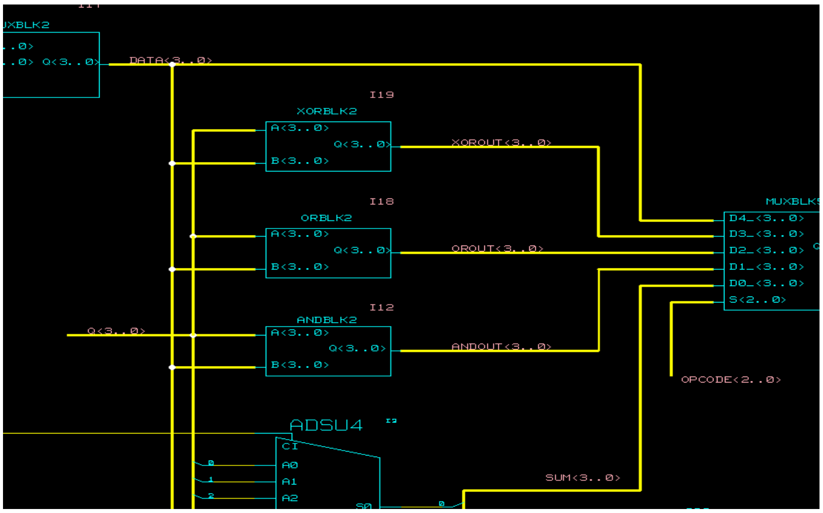


Figure 9-20 Nets, Buses and Labels for ANDBLK2 and ORBLK2

Adding Labels to Components

It is important to add labels to components. Error and warning messages often reference component labels, and labels also appear in simulation netlists. Also, net names at lower levels of hierarchy are referenced using the following format:

```
...component_label/component_label/net_label
```

In the ALU1 schematic, labels have already been added to the MUXBLK2, XORBLK2, and MUXBLK5 blocks.

To add a label to the ORBLK2 instance, follow these steps.

1. Press the Attributes button in the Command Menu.
2. Use the left mouse button to select the ORBLK2 symbol. The following dialog box should appear.

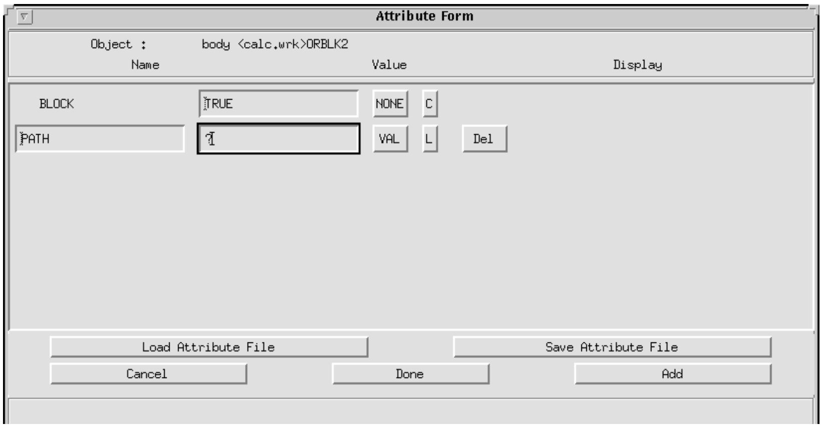


Figure 9-21 Attribute Form

3. In the field under Value and to the right of Path, replace the “?” value (or whatever string is in this field) with the value “ORBLK2”.
4. Select Done to close the Attribute Form.
5. You may use the Move command to position the text as shown in the following figure. Click the left mouse button to select and to place the text.
6. Label the ANDBLK2 symbol the same way using the label ANDBLK2, as shown in the following figure.
7. Give the FD4RE component the label ALUVAL.

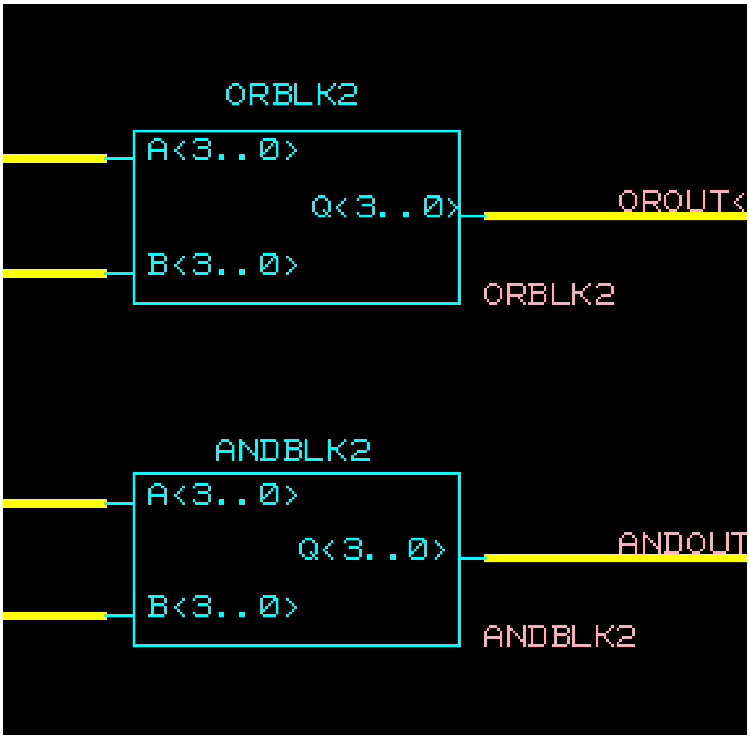


Figure 9-22 Adding Component Labels to ALU1 Schematic

The completed ALU1 schematic is shown in the following figure.

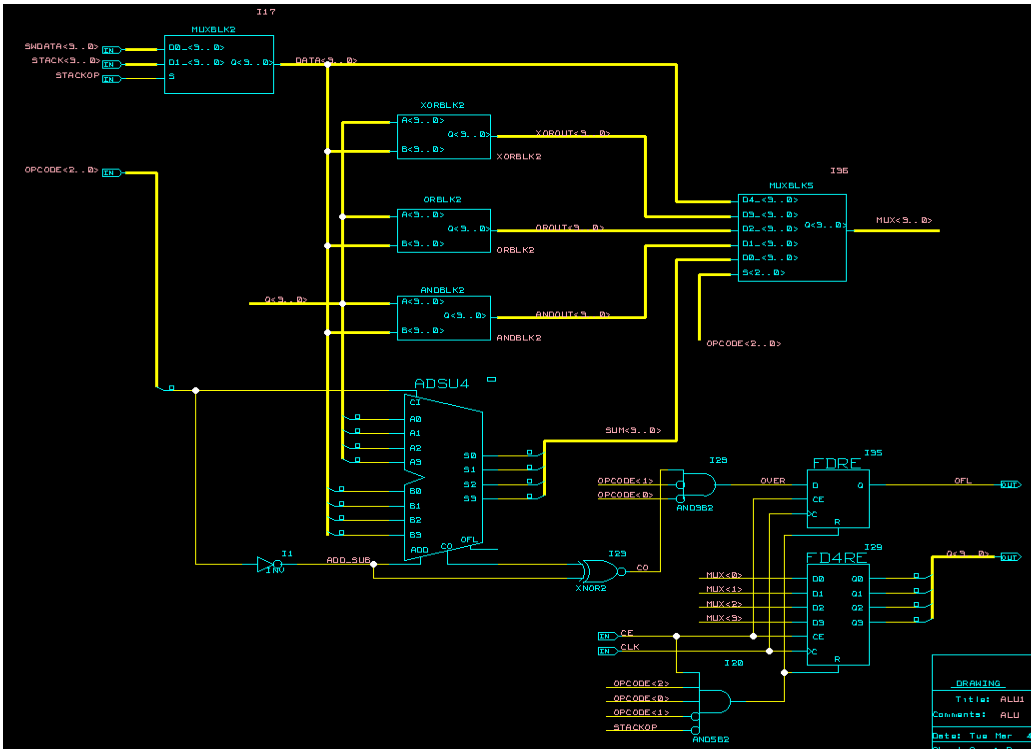


Figure 9-23 Completed ALU1 Schematic

Saving the ALU1 Schematic

Write the schematic. If errors occur, resolve them and then write the schematic again.

Exploring Xilinx Library Elements

The Xilinx libraries contain three types of elements. Primitives are basic logic elements such as the AND2 and OR2 gates that you previously placed in ANDBLK2 and ORBLK2. Soft macros are schematics created by combining primitives and other soft macros. Relationally Placed Macros (RPMs) are soft macros that contain placement information. RPMs are currently only available in the XC4000 family library.

All three types of library elements are placed on a schematic in exactly the same way.

Viewing a Xilinx Soft Macro Schematic

Soft macro schematics include schematics such as you might make for your own designs. In fact, you can load one of these schematics and use the "write *new_filename*" command to save it under another name, and then edit this new schematic to customize it to your needs.

Open the schematic underneath the FD4RE symbol as follows:

1. Enter the command "edit". Remember you will need to type it again if the Component Browser was previously closed (it is OK to iconize it).
2. Select FD4RE with the left mouse button. As shown in the following figure, FD4RE consists of four fdre symbols.
3. Enter "ret" to return to the previous schematic.

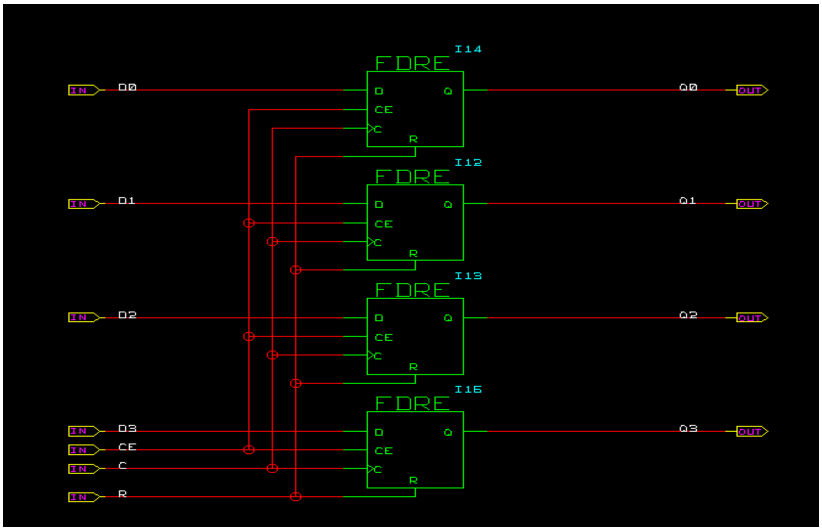


Figure 9-24 FD4RE Schematic from XC4000E Library

Viewing a Xilinx RPM (XC4000E Family Only)

Note: The following description of RPMs contains detailed information on the XC4000E architecture. Refer to *The Programmable Logic*

Data Book for more information on the XC4000E CLB structure and fast carry logic.

If your design is not targeted for the XC4000E family, read this section, but do not perform any of the commands. Continue the tutorial with the "Opening the Calc Schematic" section.

The ALU1 contains a component from the Xilinx library, ADSU4, which is a four-bit wide adder/subtracter. If your design is targeted for the XC4000E library, this schematic is implemented as a Relationally Placed Macro (RPM). If your design is not targeted for the XC4000E library, ADSU4 is implemented without this placement information.

Like schematic of soft macros, RPM schematics are schematics such as you might make for your own designs. To modify an existing Xilinx RPM schematic, save the schematic and associated symbol to a different name, then edit this new schematic to customize it to your needs.

Elements placed in the ADSU4 RPM schematic include CY4 components and FMAPs. The CY4 symbol gives you the ability to specify fast carry logic functionality from the schematic. Fast carry logic is a hardware feature in XC4000E parts that allows very fast arithmetic-type functions.

The FMAPs map logic functions to function generators in Configurable Logic Blocks (CLBs), which are arranged in a rectangular grid in the die. Both the CY4 symbols and FMAP symbols in ADSU4 have RLOC attributes. RLOCs are attached to the symbols that assign relative locations to the CLBs. You can use carry symbols as well as FMAPs and other mapping components in your own schematics. However, knowledge of them is not necessary to use Xilinx Library RPMs. Only expert users should create macros containing carry logic and FMAPs. For a description of these components, see the Xilinx *Libraries Guide.*

Push into the ADSU4 schematic as follows:

1. Enter "edit adsu4" (or "edit" and use the mouse).
2. Use the Zoom button (or stroke "Z") to zoom into the upper portion of the schematic as shown in the following figure.
3. Select the Attribute button.
4. Select the FMAP component in the upper right corner.

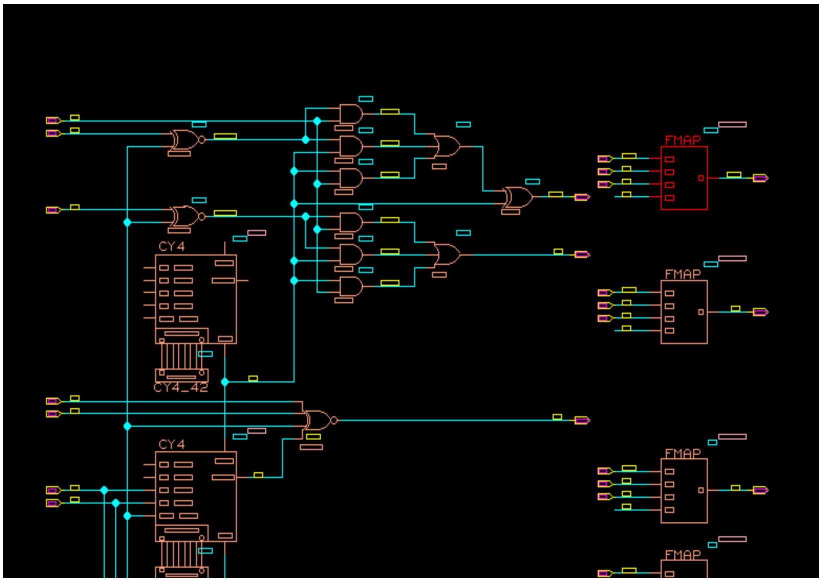


Figure 9-25 Upper Portion of the ADSU4 RPM Schematic

5. The Attribute Form window appears displaying the attributes on the symbol, as shown in the "Upper Portion of the ADSU4 RPM Schematic" figure. The RLOC attribute is set to R0C0.G, indicating that this function is mapped to the G function generator of the upper-left corner (row zero, column zero) CLB in the RPM. RPM origins reference the upper left-hand corner of the macro. (You can also call up the Attribute Form with the stroke that looks like a lowercase cursive "a".)

6. Close the window to return to the ADSU4 schematic window.

7. Hold down the right mouse button to pan around the schematic and look at the RLOCs (or use the arrow keys). Note that logic is mapped to three CLBs, designated as R0C0, R1C0, and R2C0. Therefore, this RPM uses three CLBs that are arranged in a column. Information on the number of CLBs used and the shape of the logic block is available for each RPM in the Xilinx *Libraries Guide*. Note that these locations are relative, not absolute. The macro is not being constrained to be placed in the uppermost CLB in the left most column. Regardless of the RPM's absolute location, the logic associated with the FMAP with the location R0C0 is always at the top, R1C1 is in the CLB directly below, and so on.

8. Close the ADSU4 schematic and return to the ALU1 schematic, using the “ret” command.

Opening the Calc Schematic

Close all open schematic or symbol windows except for the top-level Calc schematic window. If the Calc window is closed, open it. The Calc schematic appears on the screen.

Using the XC4000E Oscillator

If your design is not targeted for the XC4000E family, read this section, but do not perform any of the commands.

The XC4000E family devices contain an on-chip clock generator, which makes it unnecessary to use an external circuit for this purpose. The on-board clock circuitry is not precise, but is suitable for designs that do not need a highly accurate clock, such as the Calc design.

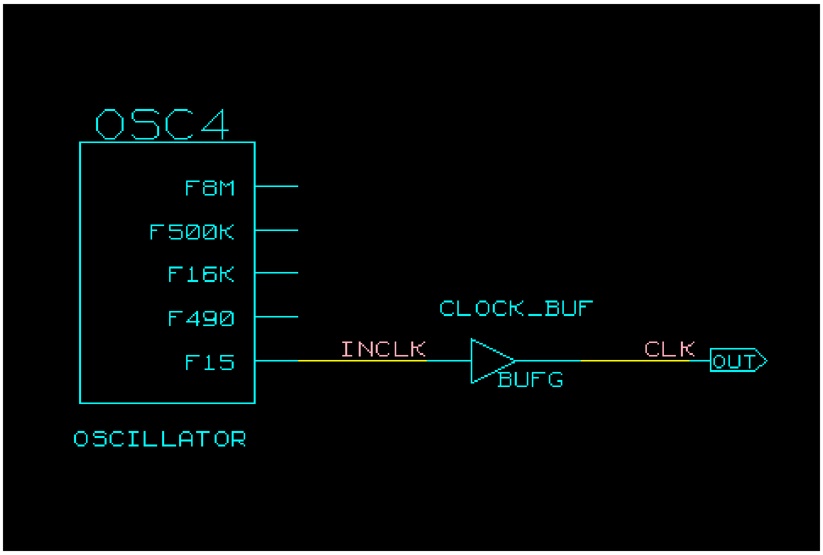


Figure 9-26 CLOCKGEN Schematic

The CLOCKGEN schematic contains an XC4000E library part, OSC4. This symbol represents the on-chip oscillator that generates nominal clock frequencies of 8 MHz, 500 KHz, 16 KHz, 490 Hz, and 15 Hz. The Calc design uses the 15-Hz output from this component when targeted for XC4000E family designs. The clock output from OSC4 is buffered through a BUFG global clock buffer to minimize clock skew.

XC4000E family devices have eight on-chip clock buffers, one BUFGP (primary global buffer) and one BUFGS (secondary global buffer) in each corner of the device. Although it is possible to use them for other purposes, BUFGPs are best used to route externally-generated clock signals. BUFGSs have more flexibility, and can be used to route any large fan-out net, even if it is internally sourced. A BUFG symbol can represent either type of buffer, and allows the implementation software to choose the type of global buffer that is best in each situation. BUFG also facilitates design retargeting to other Xilinx device families, since it can represent any type of global buffer in any family. The BUFG in the Calc design is substituted for a BUFGS during design implementation, because the clock is generated internally by the on-chip oscillator. See the Xilinx *Libraries Guide* and the *Programmable Logic Data Book* for more information on global clock buffers for Xilinx devices.

Controlling FPGA/CPLD Layout from the Schematic

This section discussed FPGA/CPLD schematic layout.

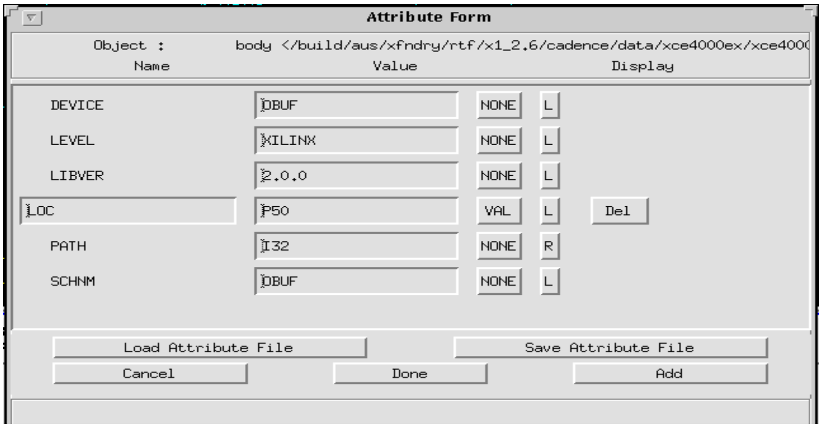
Assigning Pin Locations

It is highly recommended that you let the automatic placement and routing program, PAR, define the pinout. Pre-assigning locations to the I/Os can sometimes degrade the performance of the place and route tools. However, it is usually necessary, at some point, to lock the pinout of a design so that it can be integrated into a board design. The initial pinout should be defined by running the place and route tools without pin assignments, then locking down the I/O placement so that it reflects the locations chosen by the tools. As a general rule, inputs should be placed on the left side of the die, and outputs on the right. I/O in the tutorial schematics must be assigned pin locations so that the Calc design can function in the Xilinx demonstration boards. Because the design is fairly simple, these pin assignments do not adversely affect the ability of PAR to place and route the design completely.

Pin locations are specified by attaching a LOC property to the net attached to the pad. LOC properties should not be attached directly to I/O pads.

Add the LOC property to the OBUF associated with the 7-Segment Display F signal on the Calc schematic as follows:

1. Display the Calc schematic (“edit calc”).
2. Enter into the Attribute mode.
3. Select the OBUF to the right of the pad attached to net F. Refer to the following figure. The Attribute Form should appear.
4. Select Add in the Attribute form. In the resulting blank Name field, type “LOC”. In the Value field, type P50.


Attribute Form

Object : body </build/aus/xfndry/rtf/x1_2.6/cadence/data/xce4000ex/xce400

Name	Value	Display		
DEVICE	OBUF	NONE	L	
LEVEL	XILINX	NONE	L	
LIBVER	2.0.0	NONE	L	
LOC	P50	VAL	L	Del
PATH	I32	NONE	R	
SCHNM	OBUF	NONE	L	

Figure 9-27 Attribute Form for Adding a LOC

5. Click on Done to execute the command.
6. Use the Move command to move the “P50” text to a position closer to the OBUF.

For simplicity, the other pin locations for the Calc design have been placed in a data file known as a constraint file, which is described in a later section. You can leave the other location values undefined. Valid pin locations vary depending on the package. PLCC, HQFP, and other “numeric-only” package pins are designated with a P followed by the pin number, such as P17. PGA and other grid-array package pins use alphanumerics such as A12. The *Programmable Logic Data Book* lists the pinouts of each FPGA and CPLD for each package that Xilinx supplies.

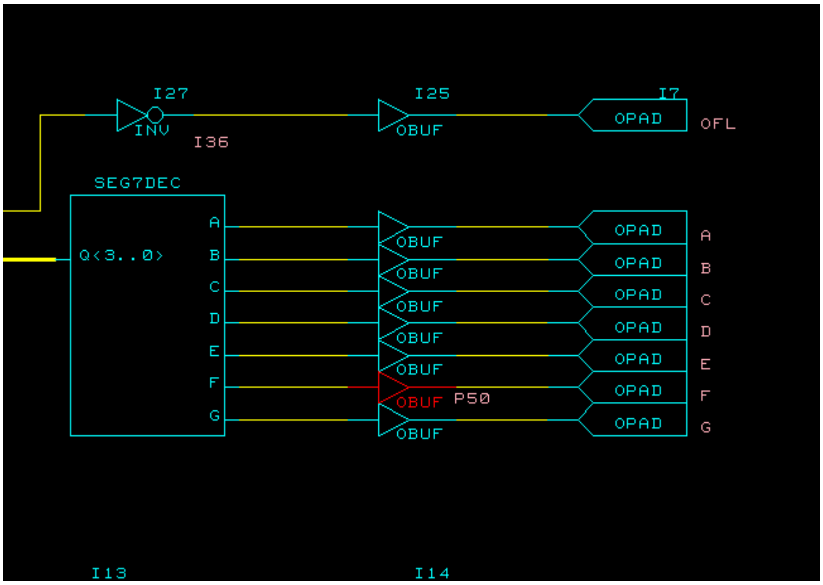


Figure 9-28 Assigning a Location to an Output Net

Designating FAST Pads

Output slew rate can be modified by assigning a FAST attribute to the output buffer, as shown in the following figure. The default slew rate is SLOW. “Fast” pads have different timing specifications and draw more current than “slow” (slew-rate-limited) pads. Slow pads are used by default. See *The Programmable Logic Data Book* for timing specifications for the various slew rate modes.

Add a FAST attribute to the LED output display drivers attached to the STACKLED (3:0) bus as follows:

1. Press F2 to display the entire Calc schematic.
2. Enter the Attribute mode.
3. Click the left mouse button on the OBUF4 symbol attached to the STACKLED (3:0) bus.
4. In the Attribute Form, click on the Add button, and enter “FAST” in the resulting Name field and “TRUE” in the Value field. Click on the button under Display (to the left of the “TRUE” test you just entered) until it displays “Name”.

5. Press Done to execute the command.
6. Move the text to be near the OBUF4 symbol, as shown in the following figure. Since the property is attached to the OBUF4 symbol, it affects all four of the LED outputs.

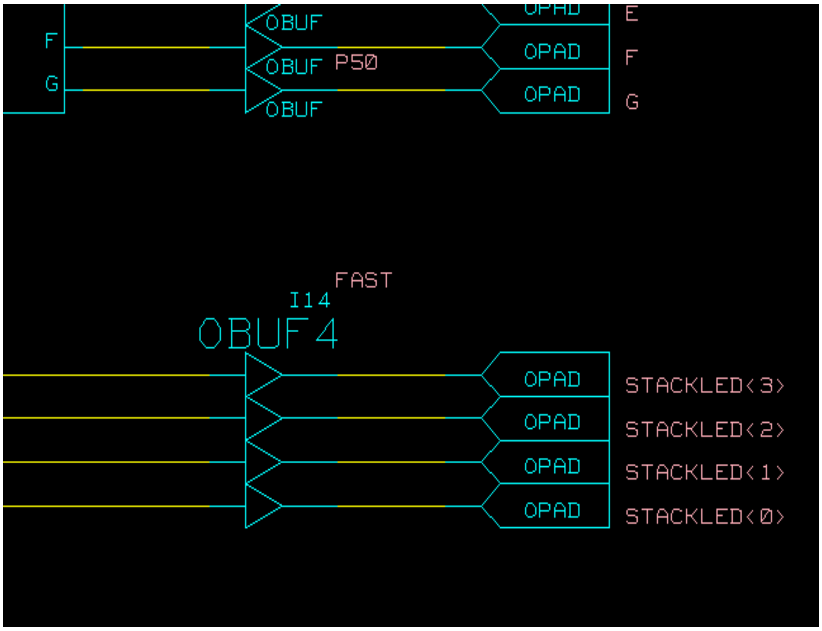


Figure 9-29 Designating a FAST Pad

Using the I/O Flip-Flops

Xilinx XC3000A and XC4000E devices have two flip-flops in each Input Output Block (IOB), an input flip-flop and an output flip-flop. You can also configure input flip-flops as latches and output flip-flops as 3-state. You access these elements using the library components IFD, ILD, OFD, and OFDT, as well as other higher-level macros that contain these components. For more information on these library elements, consult the Xilinx *Libraries Guide.*

IOB flip-flops are used whenever possible to free up internal CLB resources. In the Calc design, IOB flip-flops are used to register the switch inputs. As shown in the figure below, the SWITCH7 macro attached to the input bus SW<6:0> in the lower-left area of the schematic has an underlying schematic that consists of seven IFD (input flip-flip D-type) Xilinx primitives. If similar flip-flops, such as FDs, had been used instead, the flip-flops in the IOBs would be wasted and would occupy valuable CLB resources.

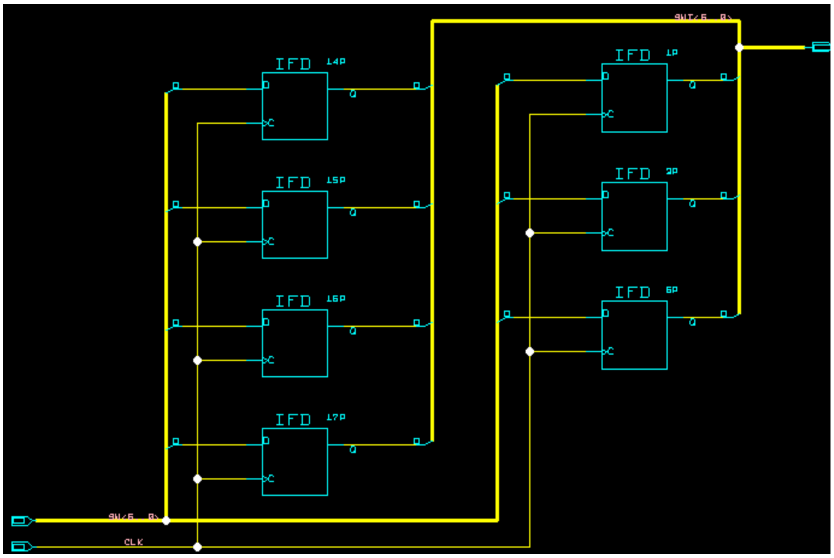


Figure 9-30   SWITCH7 Schematic Using Input Flip-Flops

Saving the Calc Schematic

Before continuing, save the changes made to Calc by doing a “write”, as shown earlier in this tutorial.

Modifying the Design for non-XC4000E/EX Devices

At this point in the tutorial, you have created or edited the following four schematic files: calc, alu1, andblk2, and orblk2. The design, at this point, is suitable for use only in an XC4000E or XC4000EX device. This is because these devices have several advanced features not found in other Xilinx device families. Two of these advanced features are the on-chip memory built into the XC4000E CLB and wide-edge decoders.

Targeting the Design for the XC9500 Family

The incomplete calc_sch design is configured for an XC4003E-PC84 part. If you want to target a demonstration board with this device, go to the "RAM Stack Implementation" section. If you are targeting the tutorial design for an XC95108-PC84 (no demonstration board available) or other device family, you must convert the design to reference the XC9500 library instead of the XC4000E library.

The procedure provided below allows you to change every Xilinx component in the Calc design from the XC4000E library to the XC9500 library. Since the designs were created using the Unified Libraries, the parts in the XC4000E and XC9500 libraries have identical footprints and pinouts. This allows you to easily retarget designs to a different device family, provided only library parts common to the two families are used. You must manually replace any library parts that are not common to both families. This example shows a situation where this may happen.

Note: Although an XC4000E-to-XC9500 conversion is shown here, this procedure may be used to retarget from any family to any other family.

To retarget the Calc design to the XC9500 family:

1. Exit from Concept by entering "quit" in the Message Window.
2. Open the global.cmd file, and change the "xce4000e" reference to "xce9000".
3. Open the master.local file, and replace all "xce4000e" references with "xce9000". If your Xilinx tools were installed in /xilinx, for example, then the following would be a valid master.local:

   ```
   file_type = master_library;
   "xce9000" '/xilinx/cadence/data/xce9000/xce9000.lib';
   "xcepads" '/xilinx/cadence/data/xcepads/xcepads.lib';
   end.
   ```

4. Modify the cds.lib file to have "xce9000" references. If your Xilinx tools are installed in /xilinx, for example, then the following would be a valid cds.lib:

   ```
   define xce9000_syn /xilinx/cadence/data/xce9000_syn
   ```

5. Open Concept once more. You may see in the Message Window such warnings as:

   ```
   The default property PINTYPE (with value IN) is
   no longer on the body OBUF.
   ```

You may ignore these warnings; when you write the schematic again, these warnings should disappear.

Targeting the Design for the 3000A and 5200 Family

If you wish to target the XC3000A, XC3100A, or XC5200 family, you may follow the above procedure for the XC9500 family to retarget the design. Simply substitute "xce3000" or "xce5200" where you see "xce9000" in the procedure above.

RAM Stack Implementation

The RAM stack is implemented using a 16x4 RAM macro from the XC4000E library. Although the stack is 4x4, RAM and ROM are only available in 16x1 or 32x1 increments, so only one fourth of the memory addresses are used. A stack four times as deep could be implemented using only two CLBs. An equivalent flip-flop implementation would require 64 flip-flops or 32 CLBs. In this case, with a stack only four words deep, using the static memory feature of the XC4000E CLB still reduces the stack from eight CLBs to two CLBs.

To view the XC4000E stack implementation, follow these steps:

1. Make sure the top-level "Calc" schematic is displayed, and type in "edit". Select the STACK symbol with the mouse. Alternatively, you can enter "edit stack".
2. On the stack schematic is a RAM16X4S component, which represents four 16x1 synchronous RAMs. Select this component while in "edit" mode to view its schematic.

   The schematic for RAM16X4S is shown below.

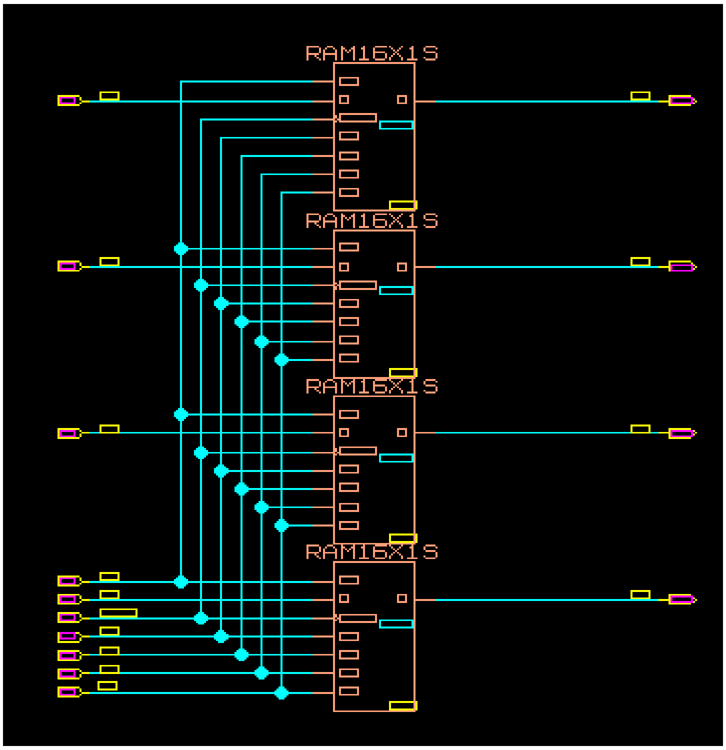


Figure 9-31 RAM16X4S, XC4000E Implementation

Using the Device-Independent Register File

The device-independent stack is implemented by replacing the RAM16X4S with a register file that emulates a synchronous RAM with a set of flip-flops and multiplexers. This implementation can be used for any Xilinx device, even one from the XC4000E family.

If you are targeting an XC4000E device, you may skip this section to take advantage of the RAM feature of the XC4000E.

Make the stack a device-independent schematic as follows:

1. Return to the stack schematic (using “ret”). Enter the command “replace ram4_9k”.
2. Use the left mouse button to select the RAM16X4S.

3. The RAM16X4S is replaced with the device-independent RAM4_9K as shown below. Note that the label RAM_BLOCK has been removed by Concept. Change the PATH property on this component in the same manner as was done to the components in the ALU1 schematic.

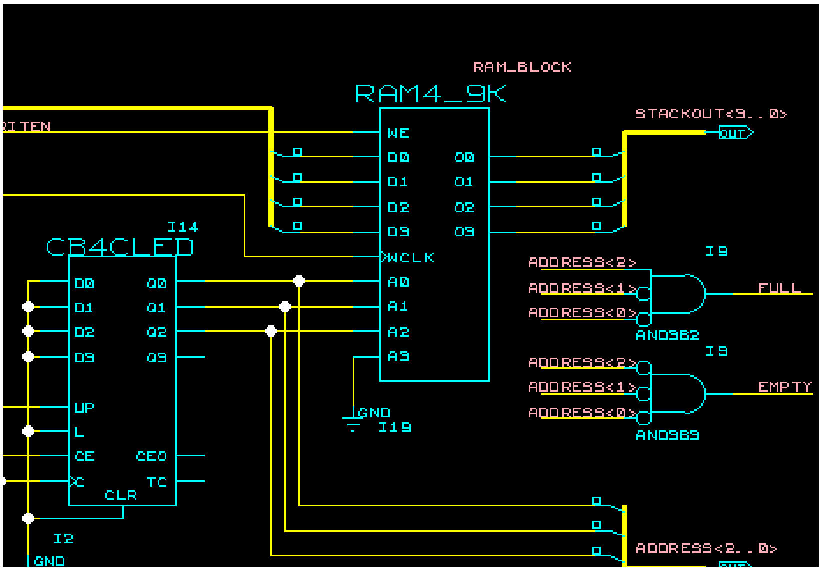


Figure 9-32 Replacing RAM with REG_9K

4. The unused A3 pin that exists on RAM16X4S does not exist on RAM4_9K. Although the detached GND symbol and net are trimmed during the implementation process, you can clean up the schematic by deleting them. To do this, click on the Delete button and then click on the GND symbol and net.

5. Write the updated stack schematic.

Removing the XC4000E Oscillator

If you are targeting the Calc design to an XC9500 or other device outside the XC4000 family, you must also remove the CLOCKGEN circuitry, which includes the OSC4 component, and replace it with an external source.

Note: The XC3000 and XC5200 families also have internal, on-chip oscillators. See the CLKGEN3K and CLKGEN5K components to see how these are used. You may choose to replace the CLOCKGEN component with one of these alternative macros with the "replace" command, instead of following the instructions below.

1. On the Calc schematic ("edit calc"), enter into Delete mode and select the CLOCKGEN component with the left mouse button.
2. Add components, nets, and labels as shown below. The IPAD symbol may be selected from the library xcepads, while the BUFG symbol may be selected from the xce4000e library.
3. Write the Calc schematic.

   Since the CLK signal is now sourced by a pad, it must be generated externally.

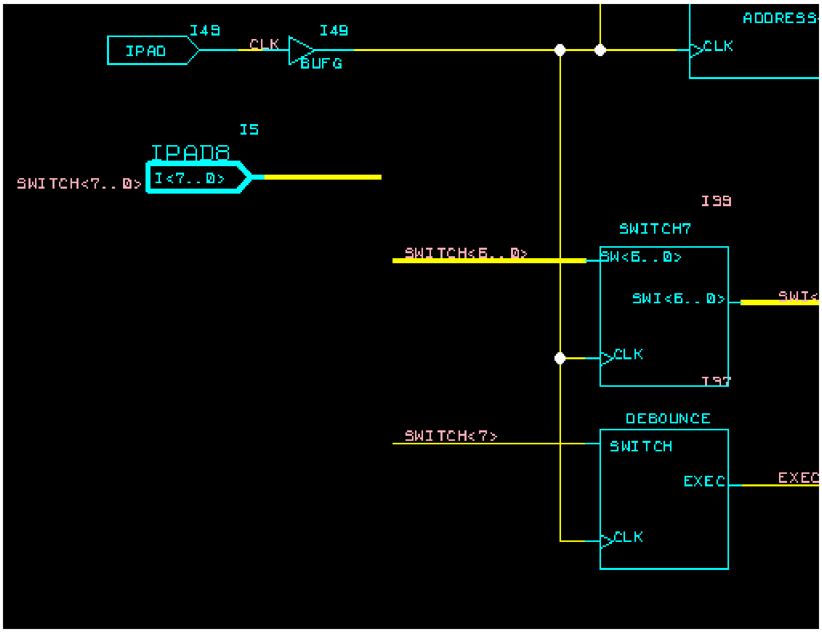


Figure 9-33 Device-Independent Clock Source

Using LogiBLOX

LogiBLOX is a tool that allows you to quickly synthesize modules for common functions such as adders, counters, and multiplexers. It allows you to create components of arbitrary bus width (*e.g.*, a 17-bit adder) and automatically uses the best architectural resources for a particular target device family. In this optional section, you replace the ADSU4 component in the ALU1 schematic with a LogiBLOX adder. If you choose to leave the ALU1 schematic in its original form, read this section but do not make any changes.

Note: With the M1 release of the Xilinx toolset, some special steps must be taken to integrate LogiBLOX symbols into the Concept schematic. In later releases, the flow for adding LogiBLOX symbols will be smoother and will not require all the steps that are described below.

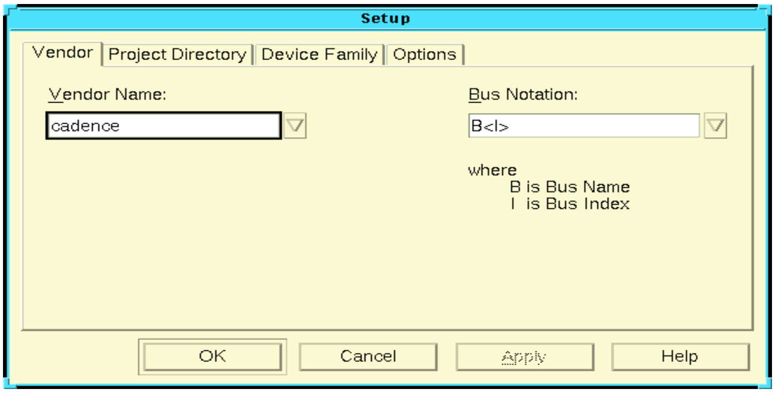
Creating a LogiBLOX Module

To replace the ADSU4 symbol with a LogiBLOX module:

1. Bring the ALU1 schematic into view (edit alu1).
2. Select Delete mode, then select the ADSU4 component to delete the symbol from the schematic.
3. From your UNIX prompt, with your M1 environment set up and within the design’s Project Directory, enter the command:

   ```
   lbgui &
   ```

   This should invoke the LogiBLOX GUI. If you are running LogiBLOX for the first time from this directory, then you will get a Setup menu, as shown in the figure below.



Vendor Name: cadence

Bus Notation: B<I>

where
B is Bus Name
I is Bus Index

Figure 9-34 LogiBLOX Setup Menu

4. Specify the parameters for the LogiBLOX module:

   a) Select “cadence” as the vendor and B<I> as the bus notation format.

   b) Select your Project Directory by clicking on the Project Directory tab on the Setup popup and either typing in the location of your project, or using the Browse button to navigate to the desired directory

   c) Select the Device Family (for example, XC4000E) by clicking on the Device Family tab, then on the arrow button to select the desired device family from the list of supported device families.

   d) Click on Options and select the desired simulation model (usually Structural Verilog). The Verilog netlist is generated to support functional simulation. It will also be used to generate the symbol body for the LogiBLOX module. Make sure that the NGO file is selected as the Implementation Netlist. The Component Declaration option is not used when you are doing a top level schematic.

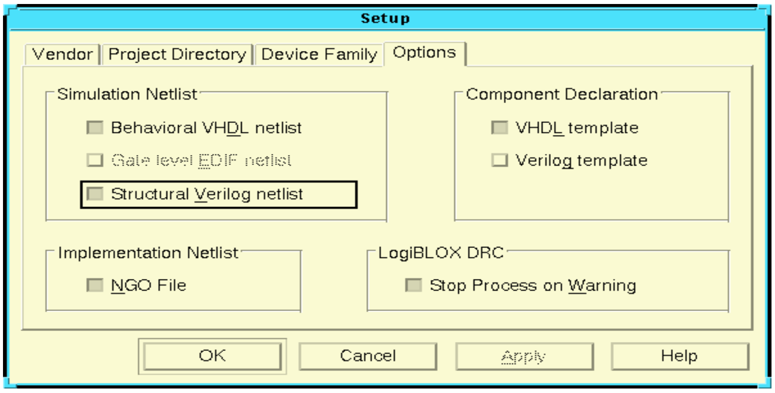
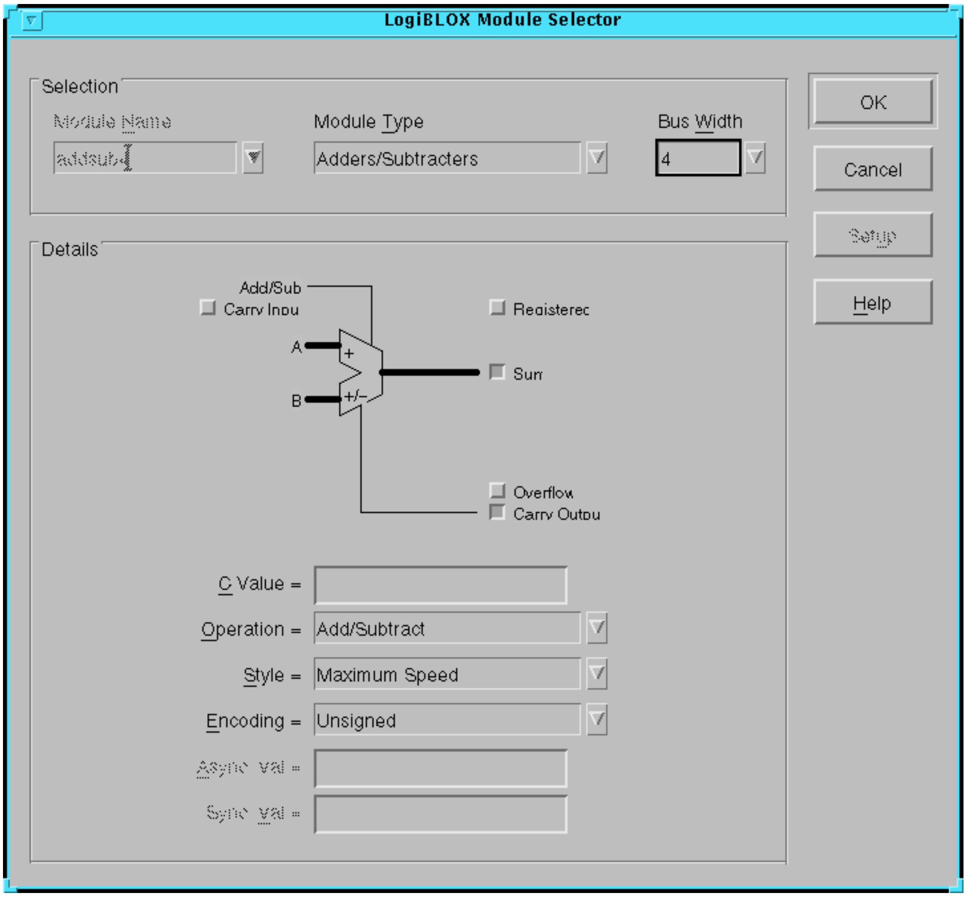


Figure 9-35 LogiBLOX setup

e) Click on OK to accept these settings. The LogiBLOX Module Selector appears.

5. The LogiBLOX Module Selector appears. Set the options in this dialog box as shown in the figure. You are making a non-registered adder/subtracter module of four bits. Name the component “addsub4”.



Module Name: addsub
Module Type: Adders/Subtracters
Bus Width: 4
Operation = Add/Subtract
Style = Maximum Speed
Encoding = Unsigned

Figure 9-36 Using the LogiBLOX Module Selector

6. Click OK. LogiBLOX will take a minute or so to generate a Verilog netlist and an NGO file for this new module. This NGO file will need to be present when you do the implementation in M1.

Creating a Symbol for the LogiBLOX Module

Since this tutorial is schematic based, it will be necessary to create a symbol for the LogiBLOX module. You must create the symbol manually from within Concept using the genview command.

1. Make sure the .v file for your LogiBLOX module is located in your Project directory. Enter the following command in the Concept message window to generate the body:

   genview -i *logiblox_module_name*.**v -v logic body verilog**

   Assuming you used the suggested name of addsub4, the command would be:

   ```
   genview -i  addsub4.v -v logic body verilog
   ```

   This tells Concept to generate a body view for a module named addsub4 from the Verilog netlist, and to put it in the logical view for this module.

2. Once the module has been generated, you must edit the resulting verilog.v file in the addsub4/logic/ subdirectory of the new module directory, and add the following directive after the module declaration:

   parameter cds_action = "ignore";

3. In Concept, in the ALU1 schematic, add the ADDSUB4 module ("add addsub4") and connect it as shown in the figure below.

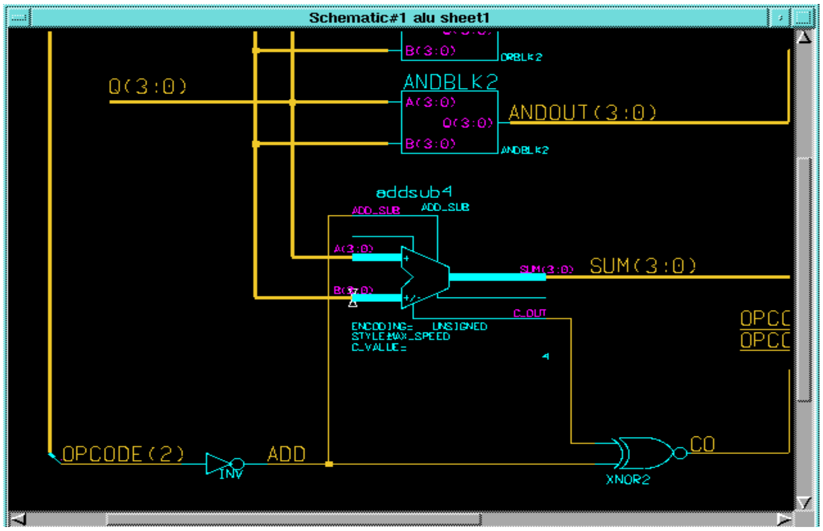


Figure 9-37 Adding the ADDSUB4 LogiBLOX Component

Other Special Components

To complete the Calc design, a STARTUP symbol is added to make the logic resetable. Also, adding a CONFIG symbol allows the Xilinx part number to be specified on the schematic.

The STARTUP Block (Optional, XC4000E/EX and XC5200 only)

The STARTUP block allows different aspects of a design to be controlled globally. In this example, STARTUP will be used to connect an external signal to the global set/reset net built into the XC4000 family and XC5200 architectures. This global net connects to all flip-flops in the device and sets or resets them asynchronously. (Set or reset is determined at the flip-flop level.) An advantage to using this built-in resource is that no routing resources are wasted tying a system-wide reset signal to all flip-flops in the design. For more information on STARTUP, see the Xilinx *Libraries Guide*.

The STARTUP symbol is used here to implement a system-wide reset signal called NOTGBLRESET. This signal is active-low; therefore, when NOTGBLRESET is low, the Calc circuitry is reset.

1. In the Calc schematic, add the components, nets, and labels as shown below. IPAD is in the xcepads library, and IBUF, INV, and STARTUP may be taken from the Xilinx device library (xce4000e or other family). Alternatively, you can enter the command “add ibuf” to add the IBUF, and similar commands to add the STARTUP, INV, and IPAD.

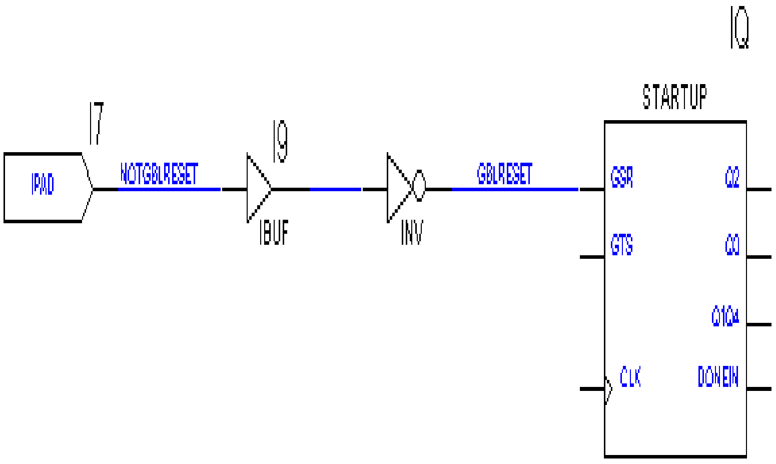


Figure 9-38 Adding the STARTUP Symbol

An inverter is added to the signal path since the GSR pin on STARTUP is active-high. Also, since GSR is *implicitly* connected to all reset logic throughout the device, GBLRESET is connected only to the GSR pin on the STARTUP block.

Note: If you target an XC5200 device, connect your chip-wide reset signal to the GR pin on the STARTUP module.

2. Write the Calc design.

Adding the CONFIG Symbol (Optional)

The CONFIG symbol is used to tell the place-and-route software how to process the design. In this example, it will be used to specify the part number for this device.

To add the CONFIG symbol:

1. From the Calc schematic, either enter the command “add config”, or select the part from the xce4000 library (or other Xilinx device library). Place this symbol in the lower-right hand corner of the Calc schematic.
2. Enter into Attribute mode and select the CONFIG symbol.

3. In the Attribute Form, click on Add, and enter the Name as “PART_TYPE”, and the Value as “XC4003E-4-PC84” and click OK. This specifies an XC4003E device with -4 speed grade (approximately 4 nanoseconds delay through a CLB) in an 84-pin PLCC.

   The PART value may take one of the following two formats:

   `[XC]` *part_number-speed-package*
   `[XC]` *part_number-package-speed*

   Therefore, the following values for the PART property are all legal:

   XC4003E-4-PC84 (recommended)
   XC4003E-PC84-4
   4003E-4-PC84
   4003E-PC84-4

Note: If using a different device, type that device number into the Property Value field instead, *e.g.*, XC95108-20-PC84

4. Move the property text to be within the CONFIG symbol as shown below.

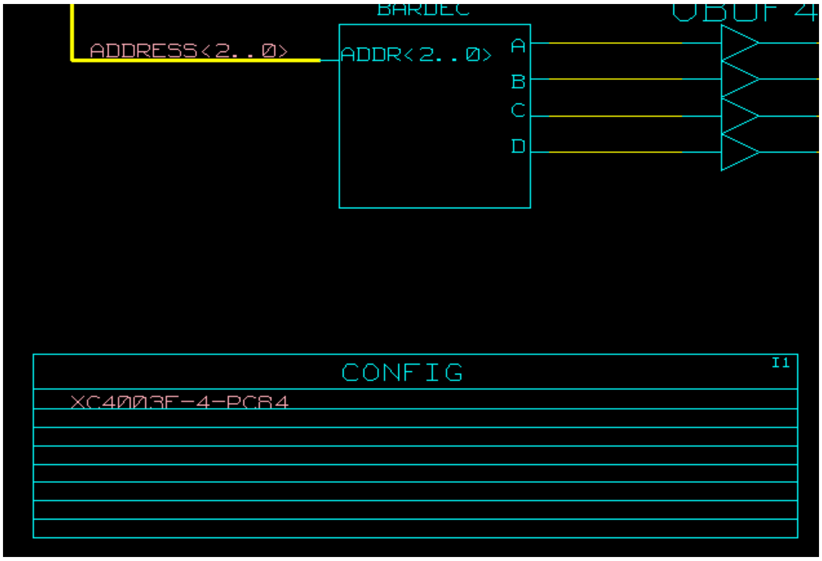


Figure 9-39 Adding the CONFIG Symbol

5. Write the Calc schematic.

Using a Constraints File

Using a constraints file, you can supply constraints information in a textual form rather than putting it on a schematic. Sometimes this method is more efficient than putting constraints on a schematic. An example of a constraints file is shown in the figure below. The figure shows the user constraints file, calc_4ke.ucf, that is supplied with this tutorial. The constraints file syntax is the same for all device families. Since you only specified one pin location for one of the many inputs and outputs on the Calc schematic, you must use a constraints file to place the rest.

The place and route software must be instructed to read and apply the .ucf file when the design is read into the Xilinx Design Manager. The procedure for doing this is detailed later in the "Using the Xilinx Design Manager" section.

```
# CALC_4KE.UCF
# User constraints file for CALC, XC4003E-PC84
# If the F pin is not constrained on the schematic,
# remove the comment (#) from NET F LOC=P50;

NET switch<7>   LOC=P19 ;
NET switch<6>   LOC=P20 ;
NET switch<5>   LOC=P23 ;
NET switch<4>   LOC=P24 ;
NET switch<3>   LOC=P25 ;
NET switch<2>   LOC=P26 ;
NET switch<1>   LOC=P27 ;
NET switch<0>   LOC=P28 ;


NET a           LOC=P49 ;
NET b           LOC=P48 ;
NET c           LOC=P47 ;
NET d           LOC=P46 ;
NET e           LOC=P45 ;
# NET f         LOC=P50 ;
NET g           LOC=P51 ;
NET ofl         LOC=P41 ;


NET gauge<3>    LOC=P61 ;
NET gauge<2>    LOC=P62 ;
NET gauge<1>    LOC=P65 ;
NET gauge<0>    LOC=P66 ;

NET stackled<3> LOC=P57 ;
NET stackled<2> LOC=P58 ;
NET stackled<1> LOC=P59 ;
NET stackled<0> LOC=P60 ;

# Remove the NOTGBLRESET line if STARTUP
# is not used in the schematic

NET notgblreset  LOC=P56;
```

Performing Functional Simulation

Functional simulation is performed before design implementation to verify that the schematic that you have designed is logically correct. All components in the Calc design, even the non-schematic LogiBLOX module, have built-in simulation models so little preprocessing is necessary.

Using CONCEPT2XIL

The CONCEPT2XIL program is used to both translate the design for implementation and to create a simulatable Verilog netlist. The latter is only possible after running CONCEPT2XIL if the design contains schematic components only. If the design has any “black boxes” with no underlying schematics, such as VHDL code that has been synthesized and compiled down to a .ngo file, then functional simulation is only possible after the program NGDBuild has been executed (NGDBuild will be executed during implementation in any case). The Calc schematic does not contain any non-schematic blocks, so proceed directly to simulation.

From the UNIX prompt, in the main design directory, enter the command:

```
concept2xil -sim_only -family xce4000e calc
```

If you are compiling this for another family, substitute the appropriate family name for “xce4000e” (i.e., xce9000, xce5200, xce3000, or xce4000x for the 9500,5200,3000A,or 4000EX, respectively).

The -sim_only option will cause CONCEPT2XIL not to create an EDIF file (.edf), which it normally would do so the Xilinx Design Implementation Tools can implement the design.

After CONCEPT2XIL has completed, a xilinx.run directory is created by default, which contains all the output files (except concept2xil.log). The generated output files are a .v file (Verilog file that declares the global GSR and GTS signals for a 4000E(X) part), and a .vf file (list of paths to all the Verilog files that comprise the design).

The Verilog files that the .vf file point to were either created by HDL Direct, or are part of the Xilinx-supplied Verilog library of components for all the Xilinx primitives and macros (located in $XILINX/cadence/data/*family*. Here are a few entries from a sample .vf file:

```
/home/jeremy/m1/cadence_m1tut/calc/logic/verilog.v
/home/jeremy/m1/cadence_m1tut/alu1/logic/verilog.v
/xilinxM1/cadence/data/xce4000e/adsu4/logic/verilog.v
/home/jeremy/m1/cadence_m1tut/andblk2/logic/verilog.v
/xilinxM1/cadence/data/xce4000e/fd4re/logic/verilog.v
```

Later, when Verilog-XL is executed, the “-f” option will be used to have Verilog-XL read in all the verilog.v files referenced by calc.vf.

Creating a Verilog Test Fixture

Now that CONCEPT2XIL has created the necessary files, a Verilog test fixture must be created to properly force the inputs for the simulation. A sample test fixture is included in the tutorial (calcf.stim) directory, which will be modified for the functional simulation. Here is an analysis of the various parts of the test fixture:

Timescale

A `timescale directive should be included near the top of the file. The `timescale directive will declare the time unit to be used, followed by the precision. Smaller precisions will result in longer runtimes, but can provide more accurate results. The following is a typical `timescale directive:

```
`timescale 1 ns/1 ps
```

Note: Using a smaller precision than 1 ps in a timing simulation is not usually necessary, since the delay numbers given in the .sdf file are not more precise than that. In functional simulation, it is essentially irrelevant (except for longer simulation run times) so long as it is more precise than the time unit (1 ns).

Test Fixture Module Declaration

Here is a test fixture module declaration that can be used with Calc:

```
module test;

  wire ofl;
  reg notgblreset;
  wire a, b, c, d, e, f, g;
  wire [3:0] gauge;
  wire [3:0] stackled;
  reg [7:0] switch;

calc uut ( .ofl (ofl) , .notgblreset (notgblreset) ,
.g (g) , .f (f), .e (e) , .d (d) , .c (c) , .b (b) ,
.a (a), .gauge (gauge) , .stackled (stackled) ,
.switch (switch) );
```

The test fixture itself is considered the top-level module (module test;), and the design is considered a component to be instantiated (calc uut (...)). All the inputs and outputs of the Calc module will be connected to signals in the test fixture with the same name (although it could be a different name). The inputs of the Calc module (switch and notgblreset) will be connected to signals in the test fixture of type “reg”, since they will be driven by the test fixture stimuli and should hold their values until driven differently. Outputs will go to signals of type “wire”.

The “calc uut” line is the instantiation of the Calc design. The string “uut” is an instance name (any name will do, so long as it conforms to Verilog naming conventions). The string .ofl (ofl) can be generalized to mean *.port_name* (*signal_name*), where *port_name* is the name of the I/O port in the Calc module, and *signal_name* is the name of the signal in the test fixture that it connects to.

Displaying Values

The output signal values of the Verilog-XL simulator can be displayed in the UNIX xterm window that you invoked Verilog-XL from, and/or it can be saved in a waveform database to be displayed in SimWave later on.

If you want to have the signal values displayed in the xterm window, then it is necessary to use a $monitor statement, which will define a list of signals to be displayed. For example,

```
$monitor("%t",$realtime,, "%b", uut.clk,,
notgblreset,, "%b", uut.stacken,, "%b",
uut.push,, "%b", uut.exec,, "%h",
uut.aluval[3:0],, "%h", uut.stackout[3:0],, ,,
"%b", switch[7],, "%b", switch[6:0]);
```

can be used for the Calc design. The $monitor statement will output the signal values whenever there is an event on one of the listed signals (one of the signals toggles). The $realtime function returns the system time as a real number; the way system time is displayed is via the $timeformat function. A typical $timeformat statement is:

```
$timeformat(-9,3,"ns",12);
```

which means to display the system time in units of 1E-9, to 3 decimal digits precision, to end with the string “ns”, and to display at least 12 digits.

The sample Calc test fixture file also uses several $display statements, to serve as a header for your output.

Opening a Waveform Database for SimWave

If you are using SimWaves to display the output waveforms, then you will need to have the test fixture write out a waveform database (.shm file), which SimWave can read in later on. The way this can be done is by the use of the $shm_open statement. The syntax is

```
$shm_open(“path_to_calc_dir/xilinx.run/
calc.shm”);
```

You may use any directory and filename you wish to write to. Also be sure to include the quotation marks.

In addition, you will need a $shm_probe statement, which indicates which signals will be written to the database. An example $shm_probe statement might be:

```
$shm_probe("AS")
```

The “AS” argument means that all signals in the test fixture and all instantiations below it (including Calc) will be included in the database, except for the internal signals of library cells.

Defining a Clock

Since most designs have an external clock signal coming in, it is useful to have a block in the test fixture that generates the pulses without having to manually toggle the clock signal back and forth.

Theoretically, it is not necessary to create a clock for the Calc schematic, since it already has the OSC4 component (inside CLOCKGEN) generating a 15Hz signal. However, one problem with this is that the OSC4 component also has a 8MHz pin, and therefore the OSC4 Verilog model will have to simulate the toggling of the 8MHz pin (even though the Calc schematic does not use it). What this means is that it will take an extraordinary amount of time for Verilog-XL to simulate a 15Hz clock signal (the osc4.v module, located in $XILINX/cadence/data/verilogxce4000e, has a ‘timescale precision of 100ps, so to make it to the first edge of the 15Hz clock, which is at 3.33E10 ps (.0333 seconds), would require 3.33E10 / 100 = 333 million simulation events).

Therefore, a clock will be defined in the test fixture that clocks much slower, and this clock will force its values onto the CLK net, overriding the OSC4 clock. The typical way to define a clock is to use a “always” block:

```
always begin
            #10 clock = ~clock;
```

Do not use this example in the Calc test fixture; it is only an general example. What this statement means is to assign “clock” the value of itself negated, but only after waiting 10 time units (assuming ‘timescale 1ns, this would mean 10 ns). So if clock is initially 0, then after 10ns clock will be assigned ~clock, which is 1. Since this is in an always block, it will continuously loop around until simulation is halted. Notice, however, the above analysis assumed the initial value is 0; you must include a statement inside an “initial” block that will define the value at time 0ns.

```
initial begin
  clock = 0;
```

The above assumes there is a port “clock” on the design, but that is not the case in our Calc design (because OSC4 generates the clock internal to the device). Because of this, you need the following in the test fixture:

```
always begin
               force uut.clk = 0;
            #50 force uut.clk = 1
            #50 ;
       end
```

The “always” block will immediately force uut.clk to 0, and then it will force uut.clk to 1 after 50ns. It will then pause for 50ns before looping around to force uut.clk back to 0. This will create a clock of period 100ns, that begins with 0 at 0ns and has its first rising edge at 50ns.

There are two points that need to be made about the statement “#50 force uut.clk = temp;”. The first is that the keyword “force” is necessary because “force” will always override the current value on a signal. Since OSC4 is still driving the signal, the “force” command will prevent contention on the signal. The second point to be made is the syntax “uut.clk” is required. If the signal to be referenced does not exist in the test fixture (or the module that the assignment is being made in), then you must specify the hierarchy of the signal by referencing the instance name it is contained in, followed by a period and the signal name.

Asserting the Global Set/Reset

In a netlist for a 4000E(X) design that uses STARTUP, the Global Set and Reset (“GSR”) net that leads to every flip-flop is connected to the STARTUP block implicitly. Toggling the signal that controls the GSR pin is needed to begin simulation and to simulate resetting the device. Even if your design does not utilize the STARTUP block, the GSR line should be pulsed once at the beginning of simulation to simulate the initial behavior of the device.

In the Unified Library functional simulation, you must connect the logic that controls the GSR pin to the underlying global GSR net by using a `define directive:

`define GSR_SIGNAL *testfixture*.*design*.*signal_on_GSR_pin*

Here *testfixture* is the name of the test fixture module, *design* is the instance name of the instantiated design, and *signal_on_GSR_pin* is the net name that sources the STARTUP GSR pin.

For the Calc design, we would have:

```
‘define GSR_SIGNAL test.uut.gblreset
```

Note we did not use “notgblreset”; the signal that actually hooks up to the STARTUP GSR *pin* must be used, not the signal that comes in from the IPAD. If you use the signal that comes in from the IPAD, the polarity of GSR in simulation will be reversed from what you would expect, since there is an inverter between the signal NOTGBLRESET and the GSR pin.

However, if the STARTUP block is not used, you must directly drive the GSR. You may again use the `define directive to define a GSR signal, even though it does not explicitly exist in the schematic or HDL code. To do this, you would define a dummy "reg test.GSR" (assuming the test fixture module name is "test", which is a name we recommend if you want to reuse the test fixture with post-NGDBuild simulation). You then need to use the `define to hook it up to the verilog models, and drive "test.GSR" in your stimulus.

```
‘define GSR_SIGNAL test.GSR
reg GSR

.....

.....

initial begin
             test.GSR=1;

        #300 test.GSR=0;

//assign inputs now
```

- For the 5200 family (which has a STARTUP symbol available), use

  `‘define GR_SIGNAL` *testfixture.design.signal_on_GR_pin*

  instead. If this is for the Calc design, and you are using the STARTUP as discussed in previous sections, then *signal_on_GR_pin* should be “gblreset”. If you are not using STARTUP, then you must define a dummy signal, as discussed before.

- For the 9500 family, use:

```
‘define PRLD_SIGNAL test.PRLD
reg PRLD; //no 9K STARTUP, so use this dummy
.....

.....

initial begin

           test.PRLD=1;

      #300 test.PRLD=0;

//assign inputs now
```

- For the 3000A family, use:

```
‘define GR_SIGNAL test.GR
reg test.GR;
```

and use a similar procedure as described above for the 9500. Note GR on a 3000A is active-low.

Assigning Values to the Inputs

In the test fixture, a series of stimuli must be added to all the inputs. Typically, this series is placed in an “initial” block, and timing controls are placed to specify when the inputs are to change. If one or more of the inputs is periodic (i.e., the clock), then you may instead put that signal in an “always” block, as described above for the clock signal.

The following is a series of assignments that will work with the Calc design (comments are in after “//”):

```
initial begin

      notgblreset = 0 ;
      switch[7]=1 ;
      switch[6:0]=0;

    #200                     //time=200

      notgblreset = 1 ;
      switch[6:0] = 7'b1100001;   //h61


    #500                     //time=700

      switch[7]=0;
      switch[6:0] = 7'b0001101;   //h0d
```

```
#200                        //time=900
  switch[7]=1;


#300                        //time=1200
  switch[7]=0;
  switch[6:0] = 7'b1111011;   //h7b


#200                        //time=1400
  switch[7]=1;


#400                        //time=1800
  switch[7]=0;
  switch[6:0] = 7'b0111111;   //h3f


#200                        //time=2000
  switch[7]=1;


#300                        //time=2300
  switch[7]=0;
  switch[6:0] = 7'b1111011;   //h7b


#200                        //time=2500
  switch[7]=1;


#300                        //time=2800
  switch[7]=0;
  switch[6:0] = 7'b1010000;   //h50


#200
  switch[7]=1;


#200 $finish;
```

```
        end
```

We begin by forcing the “notgblreset” signal low, then we assert it high once more after 200ns. Remember that in the Calc design the “notgblreset” net goes through an inverter, which makes the GSR active low, but in the 4000E(X) family GSR is normally active high.

The “switch” signal is the main input (it will input the opcode and the data), and it is assigned various values to test it various functions. Please see the *Hardware Debugger Guide* for a complete description of the Calc design, if you wish to understand what the function is of the “switch” settings. The signal “switch[7]” is specified independently only for clarity, since it has a separate function (execute).

Also, remember that time is cumulative when you use the (#xxx) notation (the sequence “ #20 statement1; #30 statement2; “ would mean “execute statement1 at 20ns, execute statement2 at 50ns).

At the end of the block, you can put a $finish or a $stop. A $finish will cause Verilog-XL to terminate. A $stop will cause Verilog-XL to terminate simulation from the test fixture, but it will go into “interactive mode”. See your Cadence documentation on using interactive mode.

Invoking the Verilog-XL simulator

Once the test fixture has been created and CONCEPT2XIL has been executed, you may go into the Verilog-XL simulator. From the xilinx.run directory, invoke the UNIX command line:

```
verilog calcf.stim calc.v -f calc.vf
```

The files calc.v and calc.vf should have been created by CONCEPT2XIL. The calcf.stim file must be user-generated (a copy exists in the tutorial design directory; copy it to the xilinx.run directory to use it).

Once the above command line is entered, the following should be displayed (only first portion of simulation is shown):

```
Highest level modules:

alias_vector

alias_bit

test
```

```
        T c n s p e a s  s sssssss
        i l o t u x l t  w wwwwwww
        m o t a s e u a  i iiiiiii
        e c g c h c v c  t ttttttt
          k b k     a k  c ccccccc
            l e     l o  h hhhhhhh
            r n     [ u  [ [[[[[[[
            e         t  7 6543210
            s       ] [  ] ]]]]]]]
            e
            t         ]
  0.000ns 0 0 0 0 0 0 x   1 0000000
 50.000ns 1 0 0 0 0 0 x   1 0000000
100.000ns 0 0 0 0 0 0 x   1 0000000
150.000ns 1 0 0 0 0 0 x   1 0000000
200.000ns 0 1 0 0 0 0 x   1 1100001
250.000ns 1 1 0 0 0 0 x   1 1100001
300.000ns 0 1 0 0 0 0 x   1 1100001
350.000ns 1 1 0 0 0 0 x   1 1100001
400.000ns 0 1 0 0 0 0 x   1 1100001
450.000ns 1 1 0 0 0 0 x   1 1100001
500.000ns 0 1 0 0 0 0 x   1 1100001
```

The header (Time, clock, notgblreset, etc.) was specified in the test fixture with $display statements. The “clock” corresponds to uut.clk, which is now being overridden with “force” statements in the test fixture. The simulation outputs are stacken, push, exec, aluval[3:0], and stackout[3:0]. Undefined states are marked with a “x”; some of the “x”s seen here are due to the RAM16X4 module being unwritten to. Values are written to the RAM16X4 later in the simulation.

Using SimWave

Cadence has shipped the cWaves waveform viewer up to and including its 9604 release; starting with the 9702 release, it will replace it with SimWave. This tutorial will use SimWave to display the waveforms.

Invoking SimWave

To invoke SimWave, enter at the UNIX prompt (in the xilinx.run directory, or wherever the .shm file is stored):

```
simwave &
```

This should bring up the SimWave GUI. To add signals to display:

1. Click on **`File`** → **`Database`** → **`Load`**.
2. Select "calc.shm" in the Directories field. If you executed SimWave from a different directory than xilinx.run, you should browse to the directory. Select OK.
3. From the SimWave GUI, select **`Edit`** → **`Add Signals`**.
4. Double-click the instance "test" in the Instance window. If you are using your own test fixture, this will be the instance name of your top-level test fixture module.
5. All the signals that occur inside the test fixture should now be displayed in the Signals window. Click on the "notgblreset" signal.
6. Click on the Display Signals button on the toolbar (it has a picture of a waveform, and is next to Close).You should see the waveform for this signal appear in the SimWave display. The label shown for this signal is "test.notgblreset", which is the full hierarchial name of the signal (note that part of the label may not be shown in the display). Highlight the "switch" bus in the Signals window, and again click on the Display Signals button. You should now see this signal displayed below "notgblreset".

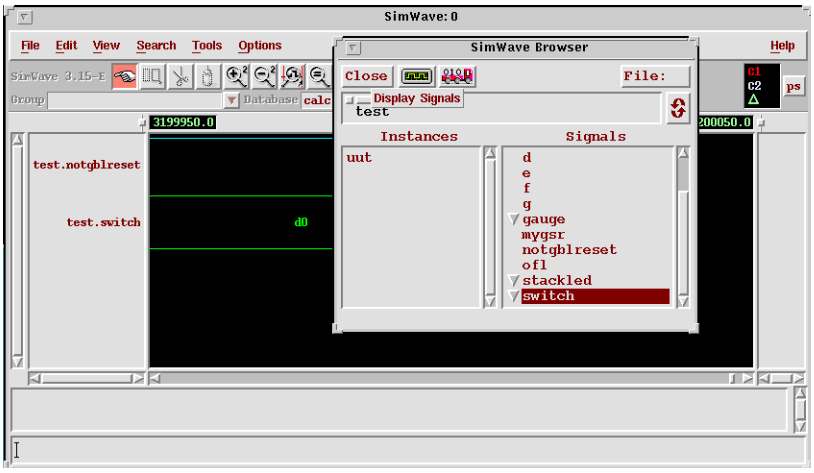


Figure 9-40 SimWave Browser/Display Tool

If you see only flat lines, as shown in the figure above, then do a **View** → **Zoom Fit** in the main SimWave GUI.

7. Double-click on “uut” in the Instances window (or whatever the instance name is of the “calc” module in the test fixture). All the signals inside the top-level schematic of Calc should be shown in the Signals window. To select several signals at once, hold down the Control key and highlight the following signals in the Signals window: aluval], stackout, clk, exec, push, and stacken. Click on Display Signal, as before. If you wish, you may add other signals.

8. Close the Browser / Display Tool by clicking on Close.

Changing the View in SimWave

The waveforms shown in the display may be difficult to read. For one thing, you might want to zoom out and see the entire waveform. Also, the timescale shown at the bottom of the display is now in pico-seconds, which is more difficult to deal with. It would also be easier to read the waveforms if the signals were arranged differently.

1. From the main SimWave GUI, select **View** → **Zoom Fit**. There should now be eight waveforms displayed, shown from 0 ps to approximately 3200000 ps (assuming you used the given test fixture).

2. To change the Time Scale, click on the button on the upper right corner of the GUI (under Help). Select “nanoseconds”.

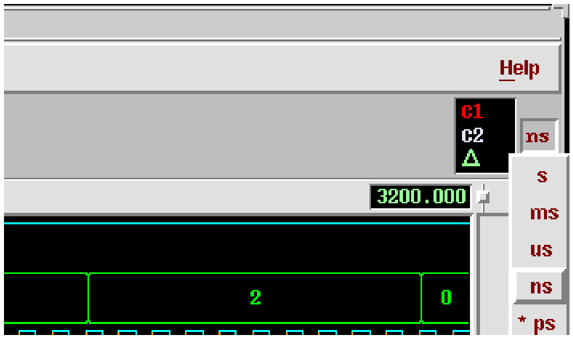


Figure 9-41 Changing the Displayed Time Scale in SimWave

3. To change the order in which the signals are displayed, you can select the waveform name, click on the right mouse button→Cut, select the waveform name you wish to place the signal above, and do a right mouse button→Paste.

   Order the waveforms in the following sequence, from top to bottom: clk, stacken, push, exec, aluval, stackout, and switc].

Splitting Up and Bundling a Bus in SimWave

The Calc design uses switch[6:0] as an opcode and data, but it uses switch[7] as an execute switch to be toggled. Therefore, it is easier to display switch[6:0] separately from switch[7].

1. In the SimWave Browser, again descend to test.uue as before. Select the down arrow beside “switch” in the Signals display to view the bus bits. Highlight “switch[6]”, and while holding down the Shift key, highlight “switch[0]”. “Switch[6]” through “switch[0]” should now be highlighted.

2. Click on the Create Bus icon on the toolbar (beside Display signals; has picture of truck with binary digits on it). The window should now have an expansion.

3. Click on the right arrow in the new window. Enter in the name on the Bus (“switch[6:0]” for instance), and select Create Bus. The SimWave GUI should now display the new bus. See the figure below.

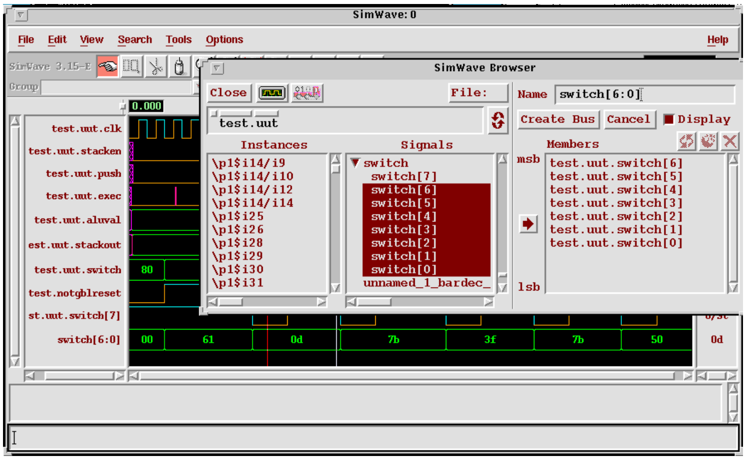


Figure 9-42 Splitting up the Switch[7:0] Bus in SimWave

4. Highlight the individual “switch[7]” bit, and add that to the display

5. To delete the original “switch[7:0]” bus from the display, highlight the waveform name in the main GUI and do a right mouse button→Delete.

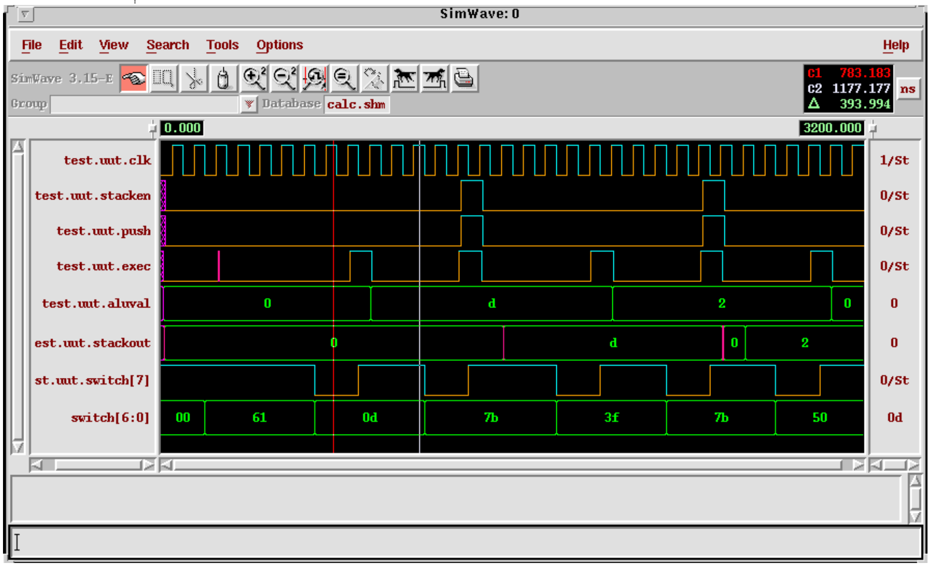


Figure 9-43 Final SimWave Display of Waveforms

6. Select **File** → **Database**→ **Save** to save your setup. This will not save the waveforms themselves; they are still in the .shm file.

Using CONCEPT2XIL for Implementation

Once your design is verified to be functionally correct, you use CONCEPT2XIL to translate your Concept design into a Xilinx-ready EDIF netlist. Running CONCEPT2XIL is always the first step in implementing a design. Whenever you make changes to your schematic, you must run CONCEPT2XIL again so that the Xilinx software can process those changes.

The command line for using CONCEPT2XIL is:

```
concept2xil -family target_technology design_name
```

where *target_technology* is xce4000e, xce4000x, xce3000, xce5200, or xce9000. Notice this command line is the same as the command line that is used for creating a functional simulation Verilog netlist, except that the -sim_only option is not present. You do not have to use the -sim_only option for CONCEPT2XIL even if you plan to simulate; this option merely stops CONCEPT2XIL from creating an EDIF (.edf) file, so it can execute faster.

CONCEPT2XIL will again put its output in xilinx.run by default (use the -rundir option to change), and it will use the files cds.lib and global.cmd to find the proper libraries (use the -cdslib and -gcmd options to change). For the Calc design, enter the following command in the main design directory:

```
concept2xil -family xce4000e calc
```

Upon completion, an .edf file will be created, along with the .v and .vf files needed for simulation (identical to what was created earlier for functional simulation). A log file called concept2xil.log will also be created in the design directory.

Using the Xilinx Design Manager

The Xilinx Design Manager is a graphical design-flow and project manager. The Xilinx Design Manager takes your design, represented by the EDIF file from CONCEPT2XIL, and implements it in an FPGA or CPLD. You can also use the Xilinx Design Manager to generate timing information that you can import into Verilog-XL.

This section gives a brief overview of the design implementation flow. For a more in-depth discussion of the flow, including advanced implementation options, see the *Development System Reference Guide*.

1. At the UNIX prompt, within the xilinx.run directory, enter the command:

   ```
   dsgnmgr &
   ```

2. Go to **File** → **New Project...**, and click on Browse to find your Input Design. Select the calc.edf file. Click on OK.

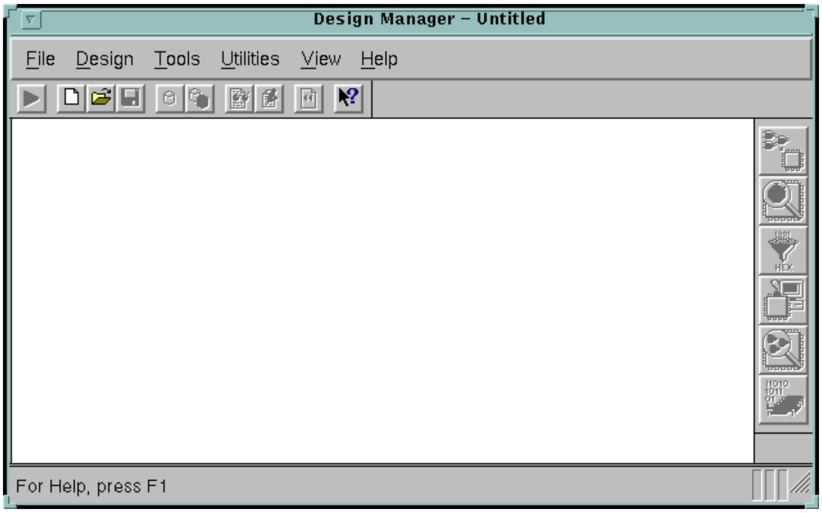


Figure 9-44 Xilinx Design Manager

Each project has associated with it objects known as “versions” and “revisions.” Versions represent logic changes in a design (for example, adding a new block of logic, replacing an AND gate with an OR gate, or adding a flip-flop); revisions represent different executions of the design flow on a single design version, usually with new implementation options (for example, higher place and route effort, a change in part type, or experimentation with new bitstream options). In the next stage, you make a new version and revision on which you run the implementation design flow.

3. In the Xilinx Design Manager, select **Utilities →Template** Manager. Select New..., and give a name such as “calc”. Select the Implementation Templates button. Select Customize in the Template Manager, and in the Program Name field, enter NGD2VER. In the Program Options field, enter -ul. The ul option will cause a `uselib directive to be written out in the output .v file, which will be needed in simulation. Select Close to exit the Template Manager.

4. Within the Xilinx Design Manager, select **Design → Implement**, which gives you the Implement dialog box, with fields for part type, design version, and revision as shown.

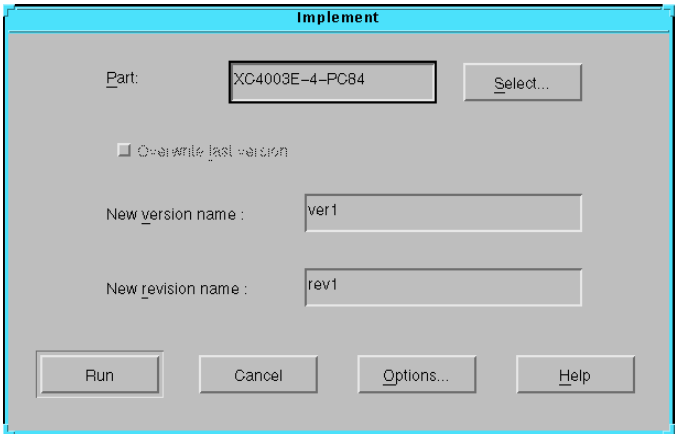


Figure 9-45 Implementation Dialog Box

5. If you chose not to use the CONFIG symbol in the earlier section, you will need to specify the part type manually.

   Click the Select button to display a pull-down listing of available devices. Choose a Family of XC4000E, a Device of XC4003E, a Package of PC84, and a Speed Grade of -4. Click OK. The part number is inserted into the Part field in the Implement dialog box.

6. Click on Options. The Options dialog box appears.

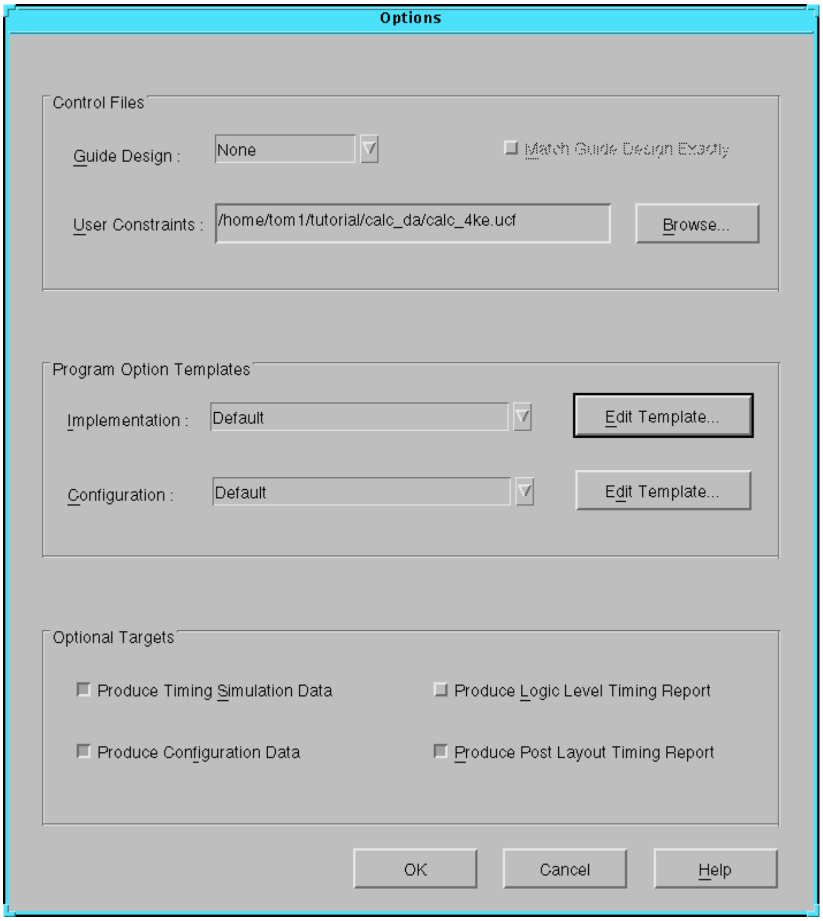


Figure 9-46 Options Dialog Box

7. Click Browse by the User Constraints field. Select the calc_4ke.ucf file from the design directory, then Click OK.

8. Under Program Option Templates, select “calc” from the pull-down menu for Implementation. This will cause the “calc” template options to be used in addition to the options selected in the GUI.

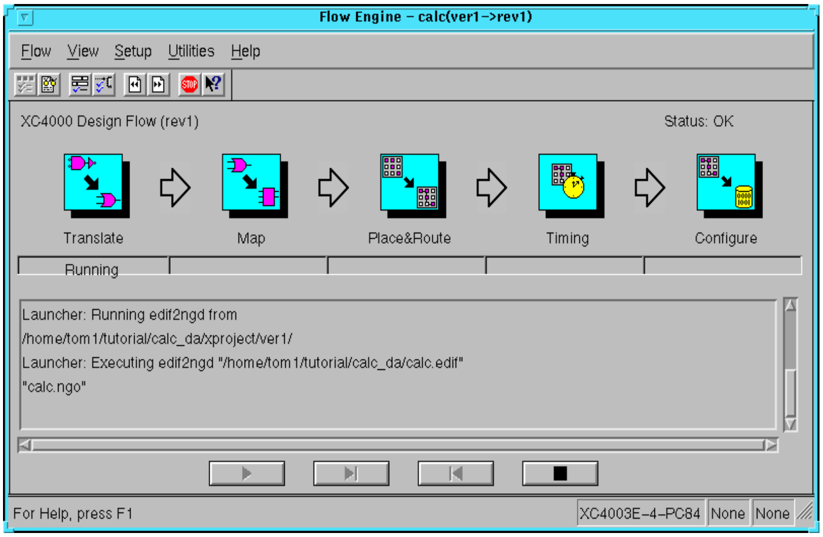
9. Under Optional Targets, make sure the following are selected:

- Produce Timing Simulation Data: This generates a back-annotated Verilog netlist that can be imported into the Cadence tools.
- Produce Configuration Data: This generates a programming bitstream suitable for downloading into the Xilinx device.
- Produce Post Layout Timing Report: This generates a timing report file based on how the design is actually routed.

You can also select the following option:

- Produce Logic Level Timing Report: This generates a preliminary (pre-place and route) timing report based on the number of logic levels in each signal path. Since it is generated before the place-and-route layout step, it does not contain information on device routing. Looking at this report before place and route can be useful for seeing how much "routing slack" you have in a design.

10. Under Program Option Templates Implementation, select Edit Template. The XC4000 Implementation Options dialog box appears.
11. Select the Interface tab. In the Interface pane, look under Simulation Data Options and verify that Format is set to Verilog and that Correlate Simulation Data to Input Design is selected.
12. Click OK to return to the Options window. Click OK to return to the Implementation dialog box.
13. Verify that the version is "ver1" and the revision is "rev1" then click Run. The Flow Engine comes up as shown in the following figure.



```
Launcher: Running edif2ngd from
/home/tom1/tutorial/calc_da/xproject/ver1/
Launcher: Executing edif2ngd "/home/tom1/tutorial/calc_da/calc.edif"
"calc.ngo"
```

Figure 9-47 The Xilinx Flow Engine

The status bar shows the progress of the implementation flow with the following stages:

- Translate: convert the design EDIF file into an NGD (Native Generic Design) file
- Map: group basic elements ("bels") such as flip-flops and gates into logic blocks ("comps"); also generate a logic-level timing report if desired
- Place & Route: place comps into the device, and route signals between them
- Timing: generate timing simulation data and an optional post-layout timing report
- Configure: generate a bitstream suitable for downloading into and configuring a device

14. When the implementation completes, an Implementation Status box appears with:

```
Implementing revision ver1->rev1 completed
successfully.
```

Click on View Logfile to display the logfile from Flow Engine. The report is displayed in vi. To exit the viewer, type **`:q!`** and press Return. Click OK in the Implementation Status dialog to return to the Xilinx Design Manager.

Note: To use another text editor, such as Emacs, as the report viewer, select **File** → **Preferences** from the Xilinx Design Manager.

Performing Timing Simulation

Timing simulation uses the block and routing delay information from the routed design to give a more accurate assessment of the behavior of the circuit under worst-case conditions. In this section, we will again invoke Verilog-XL and SimWave to display the timing data.

Invoking Verilog-XL for Timing Simulation

In your xilinx.run directory you should now see three files called “time_sim”. This .v file contains a complete Verilog netlist of the Calc design, which has been broken down to the Xilinx SIMPRIM library primitives. The .sdf file contains all the net delays, and is necessary if you want a non-unit delay timing simulation. NGD2VER (called by Design Manager) also created a .tv file, which is a test fixture template. Although you could modify this template to add your own input stimuli, in most cases you should able to re-use the functional simulation test fixture you created earlier (calcf.stim in this tutorial). However, before doing so, you must comment out the line that specifies the GSR_SIGNAL macro in your calcf.stim file. This must be done because the Verilog netlist for a post-NGDBuild or routed design already models the connection to the GSR net, and the definition will cause contention problems during your simulation. In the Calc tutorial design, the specific line in calcf.stim that needs to be commented out is:

```
`define GSR_SIGNAL test.uut.gblreset.
```

1. Enter in the UNIX command line:

   ```
   verilog calct.stim time_sim.v
   ```

   The .sdf file will be read in automatically, since there is a “$sdf_annotate” command inside the time_sim.v file.

2. The output should be similar to what was encountered in functional simulation, except that the displayed times will not always be on 50ns increments.

3. Invoke SimWave, and go to **File** → **Restore Setup**. Enter in the name of the .wrf file you created when you saved the setup under functional simulation (calc.wrf). The resulting waveforms should look similar, but there is now precise timing information.

Examining Routed Designs with EPIC

Note: This section applies only to FPGA designs. If you are targeting a CPLD such as an XC9500 device, skip to the "Making Incremental Design Changes" section

At this point in the tutorial, the design process is complete. If you would like to see how the design has been implemented by the Xilinx software, you can take a graphic look at your placed and routed design using the Editor for Programmable Integrated Circuits, or EPIC. You can access EPIC from the Xilinx Design Manager.

EPIC provides several useful functions, such as:

- Manual editing of a routed design
- Probe insertion during in-circuit verification (?)
- Static timing analysis

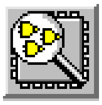


Figure 9-48 EPIC Icon

EPIC is explained in a separate tutorial. See the "EPIC Tutorial" chapter of the *EPIC Reference/User Guide*. Before starting this tutorial, be sure to select the ver1 → rev1 revision of the design in the project view.

Verifying the Design Using a Demonstration Board

Note: This section applies only to FPGA designs. If you are targeting a CPLD such as an XC9500 device, skip to the "Making Incremental Design Changes" section

Creating and Downloading the Bitstream

A bitstream has been created during the Configure stage in Flow Engine. At this point, you are ready to download the bitstream using a parallel download cable or the more versatile XChecker cable connected to your workstation. The XC4000E version of the Calc design is suitable for download into an FPGA demonstration board available from Xilinx.

Downloading is accomplished with the Hardware Debugger. To invoke Hardware Debugger, you select **Tools** → **Hardware Debugger** from the menu bar, or click the Hardware Debugger icon on the toolbar. If you are using an XChecker cable, you can also use the Hardware Debugger to read back information from the device to verify both the configuration as well as the state of memories and registers within the device.

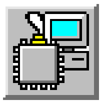


Figure 9-49 Hardware Debugger Icon

Hardware Debugger is explained in a separate tutorial. See the "CALC Tutorial" chapter of the *Hardware Debugger Reference/User Guide*. Before starting this tutorial, be sure to select the **ver1** → **rev1** revision of the design in the project view.

Making Incremental Design Changes

After initially placing and routing a design, it is often necessary to go back to the schematic and make slight modifications to the original design. When this situation occurs, much of the place and route information from the previous design iteration can be "recycled," as much of it is unchanged. This process is known as incremental design, and the NCD file (containing partition, placement, and routing information) from the prior place and route run is used as a guide.

Since much of the place and route information is extracted from the guide file, the place and route time is greatly reduced. The reuse of place and route information also results in more stable timing over a number of guided place and route iterations. Once a section of your design passes your timing requirements, guided design ensures that it will pass in the future, even if other parts of the design are modified.

In this section of the tutorial, you make a small change to the schematic and reprocess the design using the guide options available in the Xilinx Flow Engine.

Note: A small design change is the addition, removal, or replacement of only a small amount of logic in the design; the exact amount is dependent on the size of the design. If radical changes are made to a design, especially to existing portions of the design, it can be disadvantageous to guide the design.

Making an Incremental Schematic Change

Make a simple change to the Calc schematic that will be visible immediately on the demonstration board. For example, assume that the reset opcode is no longer needed and needs to be removed form the design. This can be done by grounding the ‘R’ pins that are inputs to the FDRE and FD4RE macros in the ALU1 schematic. The logic that generated the original reset signal, and the logic it drove, is automatically optimized out of the netlist by the MAP program.

Open Concept and load the Calc schematic.

1. Open the ALU1 schematic (edit alu1).
2. Zoom in on the lower right quadrant of the schematic.
3. Enter into Delete mode.
4. Delete the AND5B2 component that generates the GRESET net feeding the FDRE and FD4RE. Delete the dangling nets that are leftover, also.
5. Connect a ground symbol to the dangling QRESET net. The GND symbol can be found in the **Add Part** → **xce4000e** of the Xilinx Library menu. See the figure below.
6. Write the schematic.
7. Exit Concept, and retranslate using CONCEPT2XIL.

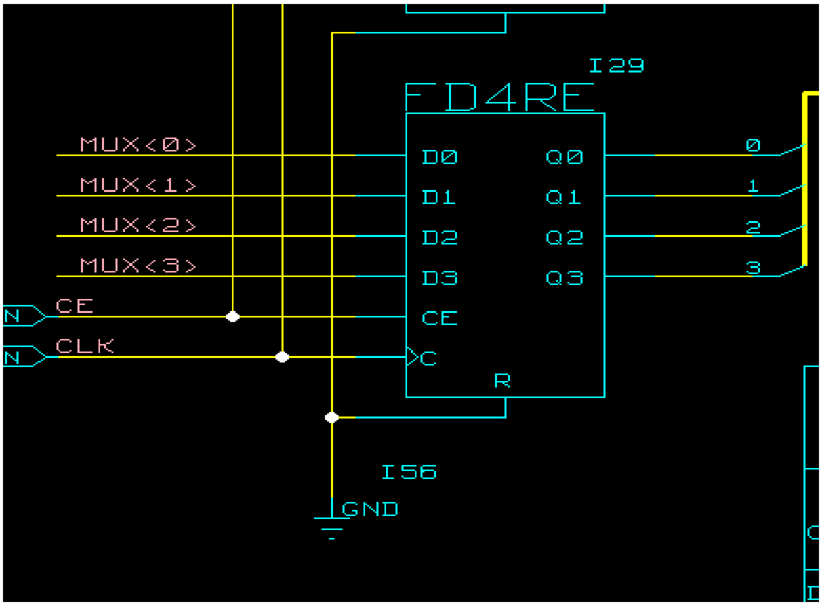


Figure 9-50 Grounding the Reset Logic

Translating the Incremental Design

Translate the guided Calc design by turning on the guide options in Flow Engine. The following instructions demonstrate an alternative method of running Flow Engine that offers more control over the implementation flow.

1. In the Xilinx Design Manager, select calc, then choose **Design** → **New Version**.
2. The New Version dialog box appears with the Name field automatically filled in as “ver2”. You may also add a comment to the new version. This comment appears in the project view next to the version number. Click OK.

Note: You can add a comment to any version or revision in the project view by selecting that version or revision, then selecting Right Mouse Button → **Properties**.

3. Select the newly created “ver2” in the project view, then select **Design** → **New Revision**.

4. The New Revision dialog box appears with the Name field automatically filled in as "rev1" and the Part field automatically filled in as "XC4003E-4-PC84". You may add a comment to the new revision if you wish. Click OK.

5. Select the newly created "rev1" in the project view, then select **Tools** → **Flow Engine**. Alternatively, you can click the Flow Engine icon in the Toolbox.

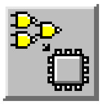


Figure 9-51 Flow Engine Icon

6. Flow Engine appears; however, unlike the procedure you used in the first revision, the implementation flow does not start automatically. This allows you to step forward and even backward through the implementation flow by individual stages, using the audio-player-like buttons at the bottom of the Flow Engine window, or the selections underneath the Flow menu.

   Select **Setup** → **Options** from the menu bar. The Options dialog box appears as before.

7. Go through the different options as before and verify that the settings you gave in the previous revision have been carried over into this revision.

8. In the Guide Design field, select Last. This sets the previous revision of the placed and routed design. In this case, it has the same effect as selecting **ver1** → **rev1**.

9. Click OK to return to Flow Engine.

10. Run the implementation as before by clicking the "play" button (on the far left) at the bottom on the Flow Engine window.

11. When all steps have completed successfully, select **Flow** → **Close** to exit Flow Engine.

Verifying the Change in the Demonstration Board

Verify that the change was performed by downloading the new bitstream to the demonstration board, as you did previously. As before, see the "CALC Tutorial" chapter of the *Hardware Debugger*

Reference/User Guide for more information. Before running through this tutorial, make sure that the **ver2** → **rev1** revision is selected in the project view.

Command Summaries

Although this tutorial uses the Cadence Design Manager and the Xilinx Design Manager to process the Calc design, you can also manually run the individual programs that these graphical tools run.

This section details command sequences that you can use to perform the translations the Xilinx Design Manager performs in this tutorial. The commands are written as you would type them at the system prompt or in a batch file. You may also see a summary of these system commands by using the **Utilities** → **Command History** and **Utilities** → **Command Preview** selections in Design Manager or Flow Engine. Once you are in the Command History or Command Preview dialog box, select the display Mode to Command Line to see the detailed system commands, including command-line options, used by Flow Engine. You can cut and paste from these Command dialog boxes into your text editor to create batch files.

Note: The commands listed here are slightly different from the commands in the Command History and Command Preview windows. The commands listed here show how you would typically execute the Xilinx programs from the system prompt, outside of the Xilinx Design Manager framework. The commands listed in the Command History and Command Preview windows reflect how Flow Engine executes Xilinx programs to fit into the Xilinx Design Manager framework.

Functional Simulation for XC4000E Family Designs

```
concept2xil -sim_only -family xce4000e calc

verilog +delay_mode_unit calcf.stim calc.v -f
calc.vf
```

Basic Translation for XC4000E Family Designs

```
ngdbuild -p XC4000E -uc calc_4ke.ucf calc.edf
  calc.ngd
map -p XC4003E-4-PC84 -o calc_map.ncd -oe normal
  calc.ngd calc.pcf
```

```
trce calc_map.ncd -a -o calc_map.twr
par -w -l 4 calc_map.ncd calc.ncd calc.pcf
trce calc.ncd -a -o calc.twr
ngdanno calc.ncd calc_map.ngm
ngd2ver -tf -w -ul calc.nga time_sim.v
bitgen calc.ncd -l -w -f bitgen.ut
```

Timing Simulation for XC4000E Family Designs

```
verilog calct.stim time_sim.v
simwave &
```

Incremental Translation for XC4000E Family Designs

```
mv calc.ncd calc_guide.ncd
mv calc.mdf calc_guide.mdf
ngdbuild -p XC4000E -uc calc_4ke.ucf calc.edf
  calc.ngd
map -p XC4003E-4-PC84 -o calc_map.ncd -oe normal
  -gf calc_guide.ncd calc.ngd calc.pcf
trce calc_map.ncd -a -o calc_map.twr
par -w -l 4 -gf calc_guide.ncd calc_map.ncd
  calc.ncd calc.pcf
trce calc.ncd -a -o calc.twr
ngdanno calc.ncd calc_map.ngm
ngd2ver -tf -w -ul calc.nga time_sim.v
bitgen calc.ncd -l -w -f bitgen.ut
```

Further Reading

The Concept tutorial is provided to give you the information necessary to begin a Xilinx design using Cadence software. It is important to note that a tool as broad and complex as Concept cannot be fully explained in a single tutorial. There are many different ways to use the commands in Concept, and there are also many ways to customize the application. It is strongly recommended that you read the Cadence Concept documentation as well as the Xilinx *Cadence Interface/Tutorial Guide.*

Appendix A

Glossary

This glossary describes the basic terminology for the Xilinx / Cadence interface.

body

A Concept symbol. The format of a body file name is body.*version.sheet_number*. Example: body.1.1 is version 1, sheet 1 of a Concept symbol.

cds_action = “ignore”;

Verilog parameter definition that must be added to the verilog.v file generated for a LogiBLOX or other non-schematic block to indicate to CONCEPT2XIL that there are no other underlying levels of hierarchy associated with a given block.

cds.lib

A library mapping file pointing to the VAN-compiled Verilog libraries used by CONCEPT2XIL and Concept.

chips_prt

Concept parts file. The file contains physical information about a board level part.

CONCEPT2XIL

CONCEPT2XIL is the Concept schematic-to-EDIF netlister; it calls several other products, such as HDLConfig, VAN, and SIR2EDIF. CONCEPT2XIL is implemented as a C-shell script. CONCEPT2XIL uses Verilog as an intermediate format. The impact to Xilinx customers is that if a particular design construct is not supported by the Xilinx Verilog libraries, it will not get translated into the EDIF netlist.

Concept

Cadence schematic editor used mainly by board level designers.

Concept Setup Files

These four files include startup.concept, cds.lib, global.cmd, and master.local. These files set up your Concept environment. The *design*.wrk file is also necessary. The **use** command in the global.cmd file defines the .wrk file name.

Concept Unified Schematic Library

Xilinx supplies the Concept Unified Library for use with the Concept schematic design tool. The library contains the Xilinx device families and associated primitives and macros.

CPLD

A Complex Programmable Logic Device. Also the command “cpld” command that invokes the CPLD fitter. See the *CPLD Schematic Design Guide* for details.

EDIF

Electronic Design Interchange Format. An industry-standard netlist format.

EDIF2NGD

EDIF2NGD, a Xilinx translation tool, converts an EDIF 2.0.0 netlist to a Xilinx NGO file. The EDIF file includes the hierarchy of the input schematic. The output NGO file is a binary database describing the design in terms of the components and hierarchy specified in the input design file.

For a description of the EDIF2NGD syntax and options, see the *Development System Reference Guide*.

genview

A program that ships with the Cadence Concept schematic editor. Genview creates a symbol / body from a Concept schematic, a Verilog netlist, or a user-specified port list. This command is used for the Cadence / LogiBLOX interface for body generation using structural Verilog netlists from the LogiBLOX GUI.

global.cmd

A Concept setup file containing aliases to the Xilinx and Cadence libraries available for your design.

HDL

Hardware Description Language. A language that describes circuits in textual code. The two most widely accepted HDLs are VHDL and Verilog. HDL may be used to describe a design in a technology-independent manner using a high-level of abstraction. When used in this way, implementation of a design requires that you first synthesize the design to a gate-level description using a synthesis tool.

HDLConfig

Concept's HDLConfig traverses a design's hierarchy and generates a design configuration that points to the cellviews for all the blocks in a design. In the 97A release, HDLConfig reads the global.cmd and hdldirect.dat files.

HDL Direct

Cadence Concept methodology for generating simulatable Verilog code directly from schematics. HDL Direct creates a Verilog output file for a schematic design. Required methodology for the Xilinx Development System interface.

iterated instances

A Concept methodology for replicating bodies which involves adding an index range to the value of the PATH property for a given instance. Use this methodology to replicate bodies in designs.

logic drawing

A Concept schematic. The format of a logic drawing file name is logic.*version*.*sheet_number*. Example: logic.1.2 is version 1, sheet 2 of a Concept schematic.

LogiBLOX

LogiBLOX is a Xilinx tool that you can use to create high-level modules for insertion into a schematic or an HDL-based design. LogiBLOX is only supported in standalone mode for the Cadence interface. LogiBLOX is not supported for CPLDs.

MAP

MAP is a Xilinx tool that maps the logic in your design to the resources of the FPGA design.

For a description of the MAP syntax and options, see the *Development System Reference Guide*.

master.local

A SCALD library mapping file which list the explicit paths to user libraries. Aliases to each user library are defined in this file. Libraries defined in master.local are available to your design if you include a "master_library" directive in your global.cmd file pointing to master.local.

mixed mode design

Mixed Mode refers to designs that contain both schematic and non-schematic blocks.

NGDAnno

NGDAnno is Xilinx’s back-annotation utility.

For a description of the NGDAnno syntax and options, see the *Development System Reference Guide.*

NGDBuild

NGDBuild, a Xilinx utility, reads a file in EDIF or XNF format, reduces all the components in the design to Xilinx SIMPRIM primitives, runs a logical design rule check on the design, and writes an NGD file as output.

NGD2VER

NGD2VER translates your design into a Verilog HDL file that contains a netlist description of the design in terms of a generic, architecture-independent Xilinx simulation primitives. In other words, NGD2VER expands the design in terms of SIMPRIMs. The input files to NGD2VER can be an NGD file used for post-NGDBuild functional simulation or an NGA file that is used for timing simulation.

PAR

PAR is Xilinx’s place and route tool.

For a description of the PAR syntax and options, see the *Development System Reference Guide.*

SCALD

A Concept old-style design methodology describing the logical design of an electronic circuit. SCALD (Structured Computer Aided Logic Design) identifies a language for specification of a logic design around which Concept and its related tools work. The language originated in the Lawrence Livermore Lab.

Concept has been upgraded to work concurrently around an HDL-centric language (that is, Verilog or VHDL) through the HDL Direct tool, while still supporting the SCALD-based flow.

SIR2EDF

SIR2EDF is Cadence's generic SIR (Structural Intermediate Representation) to EDIF conversion tool. SIR2EDF is one of three subprograms invoked automatically when you invoke CONCEPT2XIL.

SIZE

A Concept property used for replicating symbols. Not supported in Xilinx Development System Verilog libraries.

Synergy

Cadence's synthesis engine. Currently supports 2K, 3K, 4K, 4KE, 5K, 7K device families. Supported by Cadence.

testbench file

A testbench file contains simulation commands that drive a simulation and exercise your design. Typically this file is separate from the actual design netlist. (The terms "testbench" and "test fixture" are used interchangeably.)

The Verilog-XL simulator uses a testbench file to conduct simulation. The Xilinx NGD2VER command -tf option creates a testbench template file that can be copied and edited for use as an actual testbench file. The testbench file contains test vectors to drive a simulation.

Unified Library

Xilinx library standard that emphasizes standardization of component naming and physical appearance of all schematic symbols across all FPGA and CPLD architectures.

VAN

VAN is the Cadence Verilog Analyzer. VAN is one of the 3 subprograms invoked automatically when you run CONCEPT2XIL. VAN parses and analyzes Verilog netlists.

Verilog

An industry-standard HDL developed by Cadence Design Systems. Recognizable as a file with a .v extension.

The term “verilog” is used in four contexts--to refer to an HDL, a command, the Verilog-XL simulator, or Verilog files.

Verilog is an HDL that is used both for design entry and simulation.

To invoke the Verilog-XL simulator, you typically type the command “verilog”.

The simulator is used for RTL / behaviorial functional and timing simulation.

With HDL Direct turned on, Concept generates Verilog files for use with the CONCEPT2XIL command.

Verilog SIMPRIM Library

Xilinx supplies SIMPRIM-based Verilog simulation libraries for Verilog timing simulation and post-NGDBuild functional simulation. The SIMPRIM library modules are generic, technology-independent. This library is located in the $XILINX / verilog / data tree.

Verilog Unified Simulation Library

Xilinx supplies the Verilog Unified Simulation Library for post-CONCEPT2XIL HDL Direct functional simulation.

Verilog-XL

Cadence’s Verilog HDL simulator.

VLOG2XIL

VLOG2XIL is an EDIF netlister that translates a Verilog netlist from Synergy into EDIF. This netlist is supported by Cadence Design Systems.

.wrk file

SCALD library mapping file for your design. Lists the design blocks in the project directory. Updated by Concept when a new block is created.

XIL2CDS

XIL2CDS is a Cadence back-end tool that generates data files used to integrate a Xilinx FPGA or CPLD into a board level schematic and board level simulation.

XIL2CDS translates the .PIN, .V, and .SDF files from NGD2VER to two other files: 1) a chips_prt file 2) a body file for the FPGA or CPLD.

The body or symbol can be instantiated into a board level schematic, while the chips_prt file can be used to document information about the FPGA / CPLD.

Note: Ensure that you use the most updated version of the Xilinx .PIN file for ball grid and pin grid arrays. Copy the latest version as follows:

```
cp $XILINX/cadence/data/xilinx.pga.pin
$CDS_INST_DIR/share/libary/xilinx/data/
xilinx.pga.pin
```

Xilinx Design Manager

The Xilinx Design Manager is Xilinx’s design implementation tool. It is invoked by entering **dsgnmgr** at the UNIX command line.

XNF

Xilinx Netlist Format.

Appendix B

Program Options

This appendix describes Xilinx and Cadence command line programs that pertain to the Xilinx/Cadence interface. The primary programs, such as NGD2VER, CONCEPT2XIL, XIL2CDS, and VERILOG are described in detail. Xilinx programs are referenced with interbook links.

CONCEPT2XIL

The command line program, CONCEPT2XIL, is the Cadence Concept EDIF netlister. The CONCEPT2XIL program converts the Verilog (.V) file produced by the Concept HDL Direct option to an EDIF (.EDF) file, which can then be input to the Xilinx design implementation tools. CONCEPT2XIL is shipped and supported by Cadence Design Systems.

Syntax

`concept2xil` [-sim_only] [**`-cdslib`** *filename*] [**`-gcmd`** *filename*] [**`-help`**] [**`-log`** *filename*] [**`-rundir`** *dirname*] **`-family`** *technology design_name*

Options

This subsection describes the options to the CONCEPT2XIL command.

-cdslib *filename*

The -cdslib filename option indicates the name of the library map file. The default is cds.lib. This parameter is optional.

-family *family_name*

This option specifies the target library, for example xce4000x. The xce4000x library can be used to target designs being implemented in the XC4000EX / XL / XV device architectures.

-gcmd *filename*

The -gcmd *filename* option specifies the name of the command file, if any. The default is global.cmd. This parameter is optional.

-help

This option prints a usage message. It allows you to obtain more information on CONCEPT2XIL and its options. It is optional.

-log *filename*

This option specifies the name of the log file. The default is concept2xil.log. This parameter is optional.

-rundir *dir_name*

The -rundir *dirname* option specifies the name of the directory in which CONCEPT2XIL is run. The default is xilinx.run. This parameter is optional.

-sim_only

This option generates a Verilog design configuration for Unified library functional simulation only; no EDIF is generated.

Files

This subsection describes CONCEPT2XIL input and output files.

Input Files

The input files are the structural Verilog netlist (.V) files that are generated by the HDL Direct option in Concept when each design block is saved.

Output Files

There are three output files from CONCEPT2XIL.

- Verilog (.V) file

  This Verilog file defines the following modules.

  alias_bit — defines a module that is used if you rename nets tapped off a bus. This is usually done in conjunction with a TAP, SLICE or MERGE body. Note that alias bits are *not* supported by the Xilinx Development System EDIF netlister. Because alias bits are not supported, if you rename a net (for instance when you tap a bit off a bus and do not keep the same bus name), you must keep the original net name and the new net name separate in your schematic by inserting a buffer component between the new and existing nets.

  alias_vector — defines a module that is used if you rename buses. This is usually done in conjunction with a SLICE or MERGE body. Alias vectors are NOT supported by the M1 EDIF netlister.

- Verilog configuration (.VF) file

  This configuration file contains information on the location of the Verilog netlists for each block of the design.

- EDIF file

  The EDIF file is used by the Xilinx NGDBuild program.

CONCEPT2XIL will overwrite any existing .v and .vf files with the same name in the destination directory. Ensure that you do not overwrite any existing files that you want to preserve.

Error and Warning Messages

This following subsections discuss error and warning messages generated by CONCEPT2XIL.

Error Message

```
No acceptable view exists for cell primitive_name in
library path_to_library.
```

If this is the only error message in the log file, ignore it. Any other error messages that display with this message will describe the problem.

Error Message

The following error message may display when you run CONCEPT2XIL.

```
. . . Architecture not found in your design library
```

This message should be preceded by a cell name.

Following is an example:

```
Occurrence p1$9p -> calc_lib.synonym.hdl:
Error! Architecture not found in your design library
```

The occurrence name (p1$9p) is the value of the PATH property in your Concept schematic. The expansion of the cell name, in terms of the analyzed Verilog, is *library.cell.view*. The netlister searches for the cell and view in the Concept Unified Schematic library. When the netlister cannot find the cell, it looks in your design library (calc_lib in the example error message).

If you get this type of error message, perform the following steps.

1. Verify that the libraries defined in your cds.lib physically exist at the path given in the cds.lib file.
2. Verify that the referenced cell exists in the library you are targeting and that the cell is defined in your cds.lib file.
3. Verify that the cell in the library you are targeting has an "hdl" view (subdirectory). If it does not, write the cell to generate this view with HDL Direct enabled. If it is a Xilinx library, contact Xilinx for a patch.
4. Verify that the .sir file in the hdl subdirectory of the library cell is the same version as those written to *rundir* / *design_lib* / *cell* / hdl.

   The .sir files appear as: vlog004u.sir

   The version is 004 in this case.

You will also receive this error message if you fail to enter the parameter cds_action=”ignore” statement in the verilog.v file in the logic/ subdirectory of a LogiBLOX module after generating the symbol for the module. See the “Creating a Symbol for the LogiBLOX module” section in the “Processing Designs with LogiBLOX Components” appendix for details.

Error Message

The following error message may also display when running CONCEPT2XIL.

```
Initializing environment ...
Error! Cell name not specified
```

This error message can be found in the CONCEPT2XIL.log file.

Example:

`concept2xil -family xce4000x` *mydesign*

For this example, the error message displays if *mydesign* is not present in the current working directory or if *mydesign* is not an entry in design.wrk (the work file specified by the global.cmd file).

CPLD

The cpld command invokes the CPLD design implementation software (the Fitter). The command is run in a UNIX command window. Your current working directory must be set to the project directory which contains your design source netlist files before invoking cpld. For a complete description, see the "Fitter Command and Option Summary" appendix in the *CPLD Schematic Design Guide.*

DSGNMGR

This command invokes the Xilinx Design Manager, Xilinx's design implementation interface. The dsgnmgr syntax can take the following three forms:

`dsgnmgr`

`dsgnmgr` *project*

`dsgnmgr -design` *design***`.edif`**

When you use the first form of the command, the Design Manager appears with no project loaded. In this context, a project means a Xilinx project.

When you use the second form of the command, the Design Manager appears but with the specified project loaded or opened. The project is a fully specified file name with a .prj extension. It is a file created by the Design Manager and contains the project information for a Xilinx project.

When you use the third form of the command, the Design Manager finds the specified design file. A design in this context is a netlist file such as an EDIF file. If the design does not already have a Xilinx project associated with it, the Design Manager creates a project and appears with this project loaded. If the design does already have a Xilinx project associated with it, the Design Manager appears with that project loaded.

For a complete description, see the "Getting Started" chapter in the *Design Manager/Flow Engine Reference/User Guide.*

LBGUI

This command invokes the LogiBLOX Graphical User Interface.

NGDAnno

The NGDAnno program distributes delays, setup and hold times, and pulse widths found in the physical NCD design file onto the logical design view represented in the NGD file. Physical component locations for PADs are also combined with the information in the NGD file. For a complete description, see the "NGDAnno" chapter in the *Development System Reference Guide.*

NGDBuild

The NGDBuild program performs all of the steps necessary to read a netlist file in XNF or EDIF format and create an NGD file describing the logical design (the design in terms of logic elements such as AND gates, OR gates, decoders, and RAMs). The NGD file resulting from an NGDBuild run contains both a logical description of the design reduced to Xilinx NGD (Native Generic Database) primitives and a description in terms of the original hierarchy expressed in the input netlist. The output NGD file can be mapped to the desired device family. For a complete description, see the "NGDBuild" chapter in the *Development System Reference Guide.*

NGD2VER

The NGD2VER program translates your design into a Verilog HDL file containing a netlist description of the design in terms of Xilinx simulation primitives. The Verilog file is then used to perform a simulation with the Verilog-XL Cadence simulator. For functional simulation and post-map timing simulation, you must use NGD2VER with the -tf and -ul options to create the appropriate files for use with the Cadence Verilog-XL simulator. NG2VER can be used to perform post-NGDBuild functional simulation, post-Map timing simulation, and post-route timing simulation. For post-implementation simulation, you must also use the -pf option to create a.PIN file if you wish to integrate your design into a Concept board level simulation.

Syntax

For post-NGDBuild functional simulation, use the following syntax.

`ngd2ver -tf -ul` *infile*.**ngd** *outfile*.**`v`**

For post-map timing simulation, use the following syntax.

`ngd2ver -tf -ul` *infile*.**nga** *outfile*.**`v`**

For post-implementation timing simulation, use the following syntax.

`ngd2ver -tf -ul -pf` *infile*.**nga** *outfile*.**`v`**

Options

This subsection describes the NGD2VER options.

-tf

The -tf option generates a test fixture file. The file has a .tv extension, and it is a ready-to-use template test fixture Verilog file based on the input NGD or NGA file. For examples of testbench files, see the annotated templates in the "Files" appendix.

-ul

The –ul option causes NGD2VER to write a Cadence `uselib directive pointing to the appropriate Verilog simulation library into the output Verilog file. The path will be written as shown below:

`uselib dir=*$XILINX* / verilog / data libext=.vmd

where *$XILINX* is the location of the Xilinx software.

This option is recommended if you are using the Cadence Verilog-XL simulator. If you do not enter a –ul option, the `uselib line will not be written into the Verilog file.

The alternative is to specify the path to your simulation library on the Verilog command line with the -y option.

-pf

The –pf option will write out a pin file—a Cadence signal-to-pin mapping file required by XIL2CDS. NGD2VER will generate a PIN file if the input file contains routed external pins and you have specified a –pf command line option. The file will have a .pin extension.

Files

This subsection describes NGD2VER input and output files.

Input Files

Input to NGD2VER can be either of the following files:

- NGA—a back-annotated logical design file containing Xilinx simulation primitives. If you enter a file name with no extension, NGD2VER will look for a file with an .nga extension and the name you specified. An NGA file is the required input for a post-map or post-implementation timing simulation.
- NGD—a logical design file containing Xilinx simulation primitives. An NGD file is the required input for a SIMPRIM-based functional simulation.

Output Files

Output from NGD2VER consists of the following files:

- Verilog file—a structural Verilog netlist containing the netlist information obtained from the input NGD or NGA file.
- SDF file—an SDF (Standard Delay Format) file containing delays obtained from the input file. NGD2VER will only generate an SDF file if the input is an NGA file which

contains timing information. The SDF file generated by NGD2VER is based on SDF version 2.1.

The SDF file produced is intended solely for use with the Verilog file generated along with it by NGD2VER.

- TV file—a testbench template file created if you enter the –tf option on the NGD2VER command line. This option is recommended when using NGD2VER with the Xilinx/ Cadence interface.
- PIN file—a Cadence signal-to-pin mapping file. NGD2VER will generate a PIN file if the input file contains routed external pins and you have specified a –pf command line option.

All output files will have the same root name as the NGD or NGA file unless you specify otherwise.

PAR

PAR takes an NCD file, places and routes the design, and outputs an NCD file which is used by the bitstream generator (BitGen). For a complete description, see the "PAR—Place and Route" chapter in the *Development System Reference Guide.*

VERILOG

This command invokes the Verilog-XL simulator to perform functional and timing simulation. For a complete description of this command, see the Cadence manual, *Verilog-XL User Guide*.

Syntax

The syntax varies depending on which simulation is run.

For Unified Library based functional simulation, use the following syntax:

verilog +gui +delay_mode_unit \
*design***.stim** *full_path_to_design***.v \**
-f *full_path_to_design***.vf**

For SIMPRIM-based functional simulation, use the following syntax:

verilog +delay_mode_unit *designf***.stim** *full_path_to_designf***.v**

For timing simulation, use the following syntax:

`verilog` *full_path_to_designt***`.stim`** *full_path_to_designt***`.v`**

(The "\"at the end of a line indicates that the line following the current one can be typed on the same command line.)

Options

This subsection describes the Verilog command options.

+delay_mode_unit

Xilinx recommends that you specify the +delay_mode_unit option when performing a functional simulation to specify all delays as unit delays in your simulation. This prevents race conditions if there are any feedback loops in your design. This option is not used for timing simulations.

-f *full_path_to_verilog_configuration_file*

The -f option reads the Verilog configuration file (.vf). This option is only used for Unified Library based functional simulation.

-y *full_path_to_library_name*

The -y option requires the full path name to the directory of the simulation library you want to use. The +libtext+ extension must be specified also when you use the -y option. This extension specifies a library in the simulation library module. If you need to explicitly specify the SIMPRM Verilog library modules, the appropriate extension is .vmd. This option is not required.

+libtext+

This option selects the library file extensions that will be used. For the functional and timing simulations described in this manual, this option is either **`.v`** or **`.vmd`**.

+gui

Optional. This option invokes the Verilog Environment.

Files

This subsection describes Verilog-XL input and output files.

Input Files

Input files include the following files for these simulations:

Unified Library Based Functional Simulation

- Verilog (.V) file—this file is required for Unified Library based functional simulation. The file, which is generated using the -sim_only option with CONCEPT2XIL, defines the following modules:

  alias_bit — defines a module that is used if you rename nets tapped off a bus. This is usually done in conjunction with a TAP, SLICE or MERGE body. Note that alias bits are *not* supported by the M1 EDIF netlister. Because alias bits are not supported, if you rename a net (for instance when you tap a bit off a bus and do not keep the same bus name), you must keep the original net name and the new net name separate in your schematic by inserting a buffer component between the new and existing nets.

  alias_vector — defines a module that is used if you rename buses. This is usually done in conjunction with a SLICE or MERGE body. Alias vectors are NOT supported by the M1 EDIF netlister.

- Verilog configuration file — the configuration file, *design*.vf, contains information on the location of the Verilog netlist for each block of the design. The file is generated by the -sim_only option of the CONCEPT2XIL command. The file is used when conducting Unified Library based functional simulation in conjunction with HDL Direct.
- Testbench file (.tv)—a user-specified test fixture file that you can create from a testbench template. A copy of the testbench file (.stim) is used during functional and timing simulations.

SIMPRIM-Based Functional Simulation

- A structural Verilog HDL file containing the netlist information obtained from the input NGA file. This file is generated by NGD2VER and used in SIMPRIM-based functional simulation, post-map timing simulation, and post-implementation timing simulation.

- Testbench file (.tv)—a user-specified test fixture file that you can create from the testbench template generated from NGD2VER. A copy of the testbench file (.stim) is used during functional and timing simulations.

Timing Simulation

- A structural Verilog HDL file containing the netlist information obtained from the input NGA file. This file is generated by NGD2VER and used in SIMPRIM-based functional simulation, post-map timing simulation, and post-implementation timing simulation.
- SDF file—an SDF (Standard Delay Format) file containing the delays for the design. This file is required for timing simulation only. NGD2VER automatically writes a directive, $sdf_annotate, to your Verilog netlist specifying the appropriate .sdf file to use in conjunction with that netlist. The SDF file is based on SDF version 2.1.

  The SDF file is intended solely for use with the Verilog file generated along with it by NGD2VER.
- Testbench file (.tv)—a user-specified test fixture file that you can create from the testbench template generated from NGD2VER. A copy of the testbench file (.stim) is used during functional and timing simulations.

Output File

The output file name is verilog.log. Verilog-XL writes a record of all simulation commands and outputs associated with the $display and $monitor commands specified in your test fixture file. Any error or warning messages issued by Verilog-XL are also written to the log.

XIL2CDS

XIL2CDS is a Cadence-supported utility that integrates a Xilinx FPGA or CPLD into a board level schematic for board level simulation and routing. The command generates a symbol body for the

FPGA / CPLD that can be instantiated into a board level schematic, and a chips_prt file that can be used to document information about the FPGA / CPLD. For a complete description of this command, contact Cadence Design Systems.

Ensure that you use the most updated version of the Xilinx .PIN file for ball grid and pin grid arrays. Copy the latest version as follows:

```
cp $XILINX/cadence/data/xilinx.pga.pin \
$CDS_INST_DIR/share/libary/xilinx/data/ \
xilinx.pga.pin
```

Syntax

The following syntax illustrates how to create the chips_prt and body output files.

xil2cds *verilog_filename* **-family** *architecture* **-mode** *mode_type* **-pkg** *pkg_type* **-lwbverilog**

Options

This section describes the main XIL2CDS options.

-family *architecture*

This option instructs XIL2CDS to use the package and pin files for the family specified by the architecture name. This option is useful for block mode designs that do not have a PART property specifying the target device.

Valid architecture values are: 3000, 4000E, 4000X, 5200, 7000, 9000. The Xilinx XC4000EX / XL / XV are considered as a 4000X component. For example, enter “4000X” for the XC4000XL. The 4000L architecture is part of the 4000E architecture.

-mode *mode_type*

This option specifies the type of package pins to include in the output files. Values that you can use are as follows:

pkg - Include only the user pins and programming pins.

sp - Include user, programming and special mode pins.

all - Include all pins.

user - Include only user pins.

The default is **pkg**.

-pkg *pkg_file*

This option specifies which package file XIL2CDS should use. The default is to use the package file corresponding to the target device you have specified. Package files are located in the $XILINX/cadence/data directory.

-lwbverilog

This option creates the output files, chips_prt and body.1.1 for the FPGA or CPLD.

Files

This subsection describes XIL2CDS input and output files.

Input Files

The input files for the XIL2CDS command are as follows.

- Routed Verilog netlist—the .V file generated by NGD2VER
- Cadence XIL2CDS package file—this file contains data that is used to create a body. The package files are located in the $XILINX/cadence/data directory
- PIN files —a Cadence signal-to-pin mapping file which may be generated by NGD2VER.
- SDF File —Standard Delay Format file. Contains the design delay data.

Output Files

The output files for the XIL2CDS command are as follows.

- Chips_PRT—the physical part file for the Xilinx device
- Body—the symbolic representation of the Xilinx device

Appendix C

Processing Designs with LogiBLOX Components

LogiBLOX is a utility that gives designers access to a library of automatically generated, high level functional modules to use in schematic-based or HDL designs. For Cadence designs, LogiBLOX generates both a functional simulation model for the block, as well as an NGO module that is ready to be implemented as part of the overall design. Possible modules include adders, counters, accumulators, and memories. Currently LogiBLOX supports only the XC3000A/L, XC3100A/L, XC4000/E/EX/L/XL/XV and XC5200 architectures.

Because LogiBLOX is not integrated into the Concept schematic editor, it must be run standalone. After the LogiBLOX module has been generated, you must create a symbol/body for the LogiBLOX module to integrate it into a schematic. For Verilog designs, LogiBLOX can be directed to generate a Verilog template that you can follow to instantiate the LogiBLOX into your Verilog design.

This chapter has the following sections.

- "Generating the LogiBLOX module" section
- "Creating a Symbol for the LogiBLOX module" section
- "Netlisting the Design for the Functional Simulation" section
- "Functional Simulation" section

Generating the LogiBLOX module

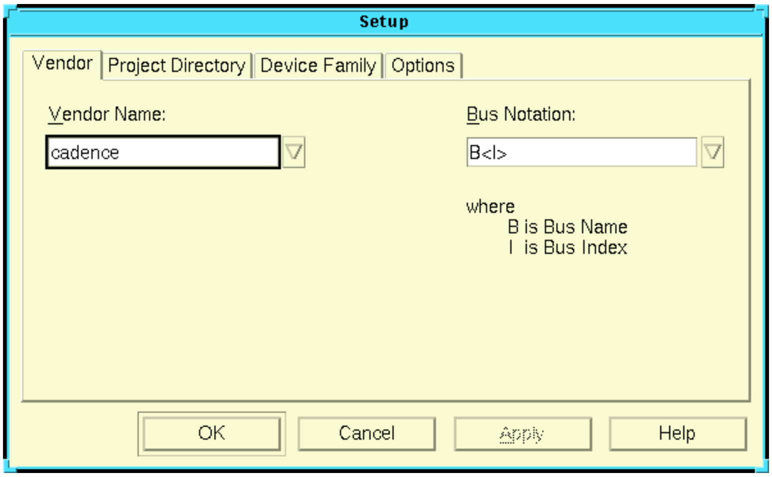
For schematic based designs that also contain LogiBLOX components, you currently must run LogiBLOX in standalone mode to generate the required .MOD and .v (behavioral Verilog) files. You can then use the Concept genview command to generate a symbol body for the LogiBLOX module, and add the module to your schematic.

To generate the LogiBLOX module:

1. Start up LogiBLOX in standalone mode.

   `lbgui`

   A Setup popup appears if you are running LogiBLOX for the first time.



Vendor Name: cadence

Bus Notation: B<I>

where
B is Bus Name
I is Bus Index

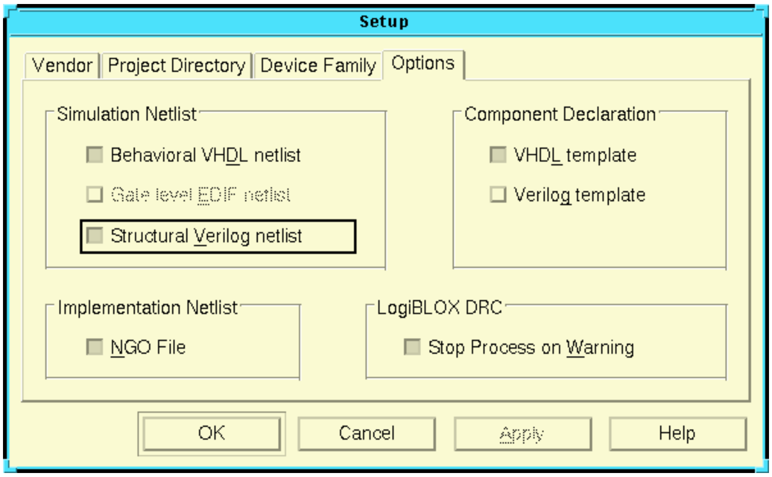
2. Specify the parameters for the LogiBLOX module:

   a) Select "cadence" as the vendor and B<1> as the bus notation format.

   b) Select your Project Directory (usually your working design directory) by clicking on the Project Directory tab on the Setup popup and either typing in the location of your project, or using the Browse button to navigate to the desired directory

   c) Select the Device Family (for example, XC4000EX) by clicking on the Device Family tab, then on the arrow button to select the desired device family from the list of supported device families.

   d) Click on Options and select the desired simulation model (usually Structural Verilog). The Verilog netlist is generated to support functional simulation. It will also be used to generate the symbol body for the LogiBLOX module. Make sure that the NGO file is selected as the Implementation Netlist. The Component Declaration option is not used when you are doing a top level schematic.



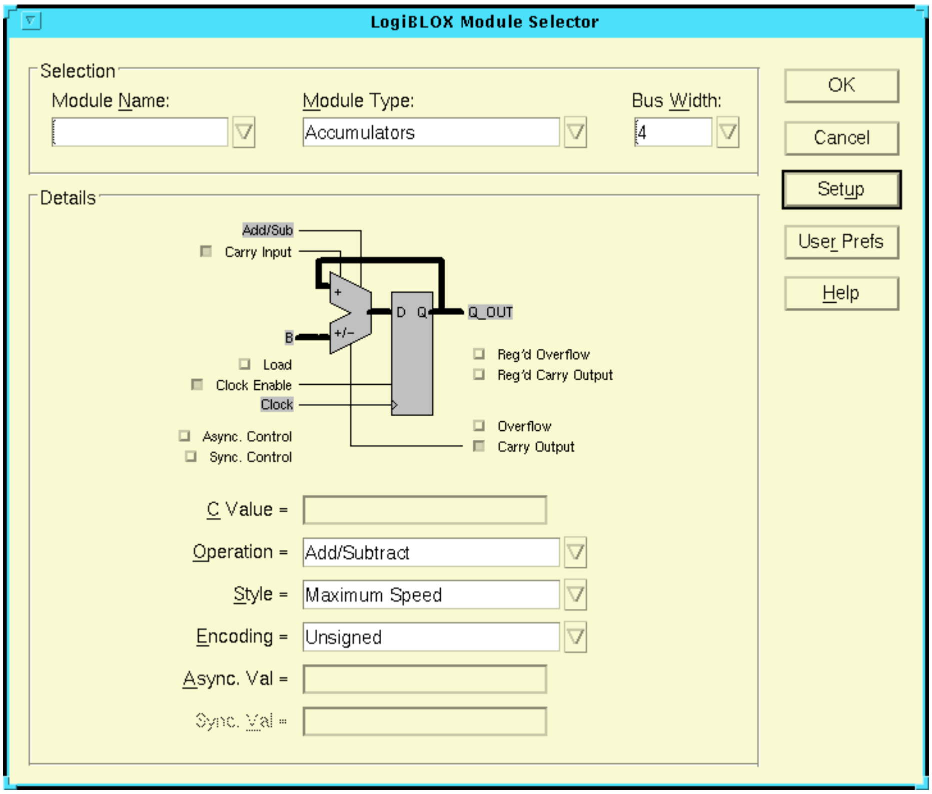
Simulation Netlist
- Behavioral VHDL netlist
- Gate level EDIF netlist
- Structural Verilog netlist

Component Declaration
- VHDL template
- Verilog template

Implementation Netlist
- NGO File

LogiBLOX DRC
- Stop Process on Warning

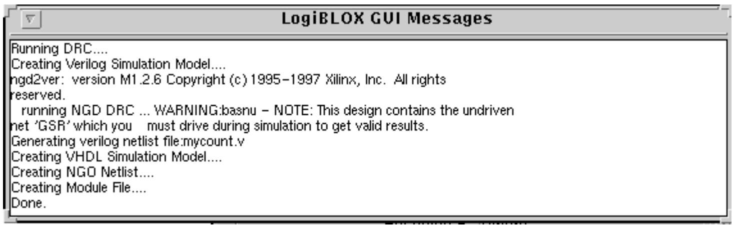
e) Click on OK to accept these settings. The LogiBLOX Module Selector appears.



3. Enter a name for your module in the Module Name field. Select the Module Type by clicking on the down arrow button next to the Module Type field and selecting a module type from the list.

4. Select a bus width by clicking on the down arrow button next to the Bus Width field.

5. Set the other parameters as desired, then click OK to generate the block and associated files. The LogiBLOX GUI Messages window appears.



```
Running DRC....
Creating Verilog Simulation Model....
ngd2ver:  version M1.2.6 Copyright (c) 1995-1997 Xilinx, Inc.  All rights
reserved.
  running NGD DRC ... WARNING:basnu - NOTE: This design contains the undriven
net 'GSR' which you    must drive during simulation to get valid results.
Generating verilog netlist file:mycount.v
Creating VHDL Simulation Model....
Creating NGO Netlist....
Creating Module File....
Done.
```

6. Click on Cancel when LogiBLOX has completed its run.

Creating a Symbol for the LogiBLOX module

If you are doing a schematic-based design, you will need to generate a symbol for your LogiBLOX module. You must create the symbol manually from within Concept using the genview command.

1. Make sure the .v file for your LogiBLOX module is located in your current directory. Start up Concept, then enter the following command in the Concept command window to generate the body:

   genview -i *logiblox_module_name*.**v -v logic body \ verilog**

   This tells Concept to generate a body view for a module named *logiblox_module_name* from the Verilog netlist, and to put it in the logical view for this module.

2. Once the module has been generated, you must edit the resulting verilog.v file in the logic/ subdirectory of the new module directory, and add the following directive after the module declaration:

   parameter cds_action = “ignore”;

   Example:

   If your LogiBLOX module is named “mycount”, genview would do the following:

   a) Create an entry in your.wrk file for the module.

   b) Create a directory called “mycount” in your project directory.

c) Create a subdirectory called “logic” under mycount and copy the LogiBLOX-generated mycount.v to mycount/logic/verilog.v.

d) Create a symbol for the LogiBLOX module, mycount/body.1.1. Instantiate and connect up the new symbol in your design and save your design.

```
module mycount (load, up_dn, clk_en, clock,
async_ctrl, term_cnt, d_in, q_out);

parameter cds_action=”ignore”; // <---- Add this line

    input load;

    input up_dn;

    input clk_en;

    input clock;

    input async_ctrl;

    output term_cnt;

    input [3:0] d_in;

    output [3:0] q_out;

....
```

3. Copy the .NGO file for the Logiblox module to your design implementation directory (usually xilinx.run).

Netlisting the Design for the Functional Simulation

To generate a netlist for functional simulation, type the following command line:

`concept2xil -family` *family_name* **`-sim_only`** *design_name*

Family_name is the architecture.

Functional Simulation

To conduct HDL Direct Verilog Unified Library simulation manually, create a testbench file as described in the “Functional Simulation” chapter with the desired test vectors. Then invoke the simulation as follows:

```
verilog +delay_mode_unit calcf.stim calc.v -f
calcf.vf
```

The testbench file

After Verilog-XL has finished compiling your simulation commands you can view the waveforms for the signals in your design. Your simulation waveforms can be displayed by a waveform display application that is separate from Verilog-XL. Given the appropriate directives, Verilog-XL writes the waveform data to a simulation history directory (design.shm). The waveform viewer application (usually Simwave) reads this data from the simulation history database and displays the waveforms.

If you wish to view your simulation waveforms graphically while performing your functional simulation, you must add an “initial” block to the testbench file containing directives to create a simulation history database for the waveform viewer.

Note: NGD2VER will add the “initial” block to a stimulus template file (design.tv) by default. For pre-NGDBuild functional simulation flows, this block must be added to your testbench manually.

Sample “initial” block adding Simulation History Manager support:

```
initial

begin

    $shm_open(“/home/user/cadence/calc/xilinx.run/ \

      calc.shm”);

    $shm_probe(“AS”);

end
```

The $shm_open command creates the database directory, “calc.shm”. $shm_probe(“AS”) directs the simulation history manager to probe *all* signals, thus making them available for viewing in the waveform viewer.

Global Reset

Global reset should always be toggled at the beginning of a simulation to ensure that all flip-flops and latches initialize to a known state. See the “Setting Global Set / Reset and Tri-state Signals (FPGAs)” section of the “Design and Simulation Techniques” chapter for information on toggling global reset for XC3000A / L, XC3100A / L, XC4000 / E / EX / L / XL / XV, XC5200, and XC9500 devices.

Appendix D

Synopsys/Verilog Design Flow

With Xilinx design tools, you can translate Synopsys synthesized gate-level netlists for use with Cadence's Verilog-XL simulator. See the "Synopsys/Verilog Design Flow" figure.

NGDBuild will accept two types of Synopsys netlists, SXNF and SEDIF. The .SXNF netlist is generated by the Synopsys FPGA compiler and the .SEDIF netlist is created by the Synopsys Design Compiler. Once you have created one of the Synopsys netlists from a behavioral design, NGDBuild can translate the netlist into an NGD file or NGA file for use with the NGD2VER command. This command with its options generates the required files to run Verilog-XL simulation.

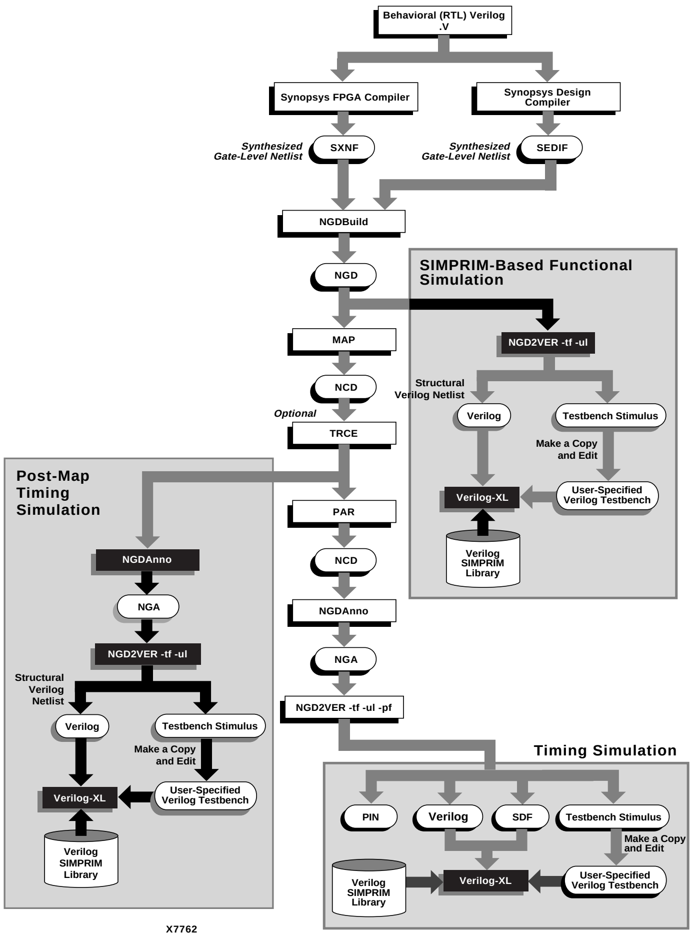


Figure D-1 Synopsys/Verilog Design Flow

Appendix E

Files

This section contains an annotated test fixture template as well as three sample test fixtures illustrating how the global GSR and GTS signals are specified and exercised in 1) Unified Library functional simulation, 2) post-NGDBuild SIMPRIM functional simulation, 3) post-map timing simulation, and 4) post-implementation SIMPRIM timing simulation.

Note: The terms “testbench” and “test fixture” are used synonymously throughout this manual.

Testbench Template

```
// NGD2VER VERILOG Test Fixture Template
// Design file: calc.ngd
// Date:Sat May 17 00:07:34 1997


// ATTENTION: This file was created by NGD2VER and may therefore be
// overwritten by subsequent runs of NGD2VER. Xilinx recommends that you
// copy this file to a new name, or 'paste' this text into another file,
// to avoid accidental loss of data.


// The timescale directive specifies the default time unit used for the
// simulation. In this case, it is 1ns, with a precision of 1 ps (.001ns)

`timescale 1 ns/1 ps


// The module, "test", is the testbench module.  Within it, we first see
// a listing of all the OUTPUTs of the design, which are declared as
// "wires" (ofl, a, b, c, d, e, f, g, gauge[3:0], stackled[3:0], and
// switch[7:0]. These are followed by the inputs, which are declared with
// data type, "reg", for "register".  In this case, the only input is an
// 8-bit bus named "switch[7:0]".

// An instance of the design, "calc", is also instantiated within the
// "test" module, with an instance name of "uut".


module test;
  wire ofl;
  reg notgblreset;
  wire g;
  wire f;
  wire e;
  wire d;
  wire c;
  wire b;
  wire a;
  wire [3:0] gauge;
  wire [3:0] stackled;
  reg [7:0] switch;

 // To properly simulate your design containing a Startup component,
  // be sure to do the following at the beginning of your simulation:
  //
  // 1. Toggle your GSR port to initialize all registers.
  //
  // 2. Deactivate your global tri-state (GTS) control signal.
  //


  calc uut ( .ofl (ofl) , .notgblreset (notgblreset) , .g (g) , .f (f)
, .e (e) , .d (d) , .c (c) , .b (b) , .a (a) , .gauge (gauge)
, .stackled (stackled) , .switch (switch) );


// The "timeformat" system task specifies how time information for the
// $display and $monitor commands is to be formatted when displayed by
// Verilog-XL. The first number represents the "unit" value, the second,
// the precision number, followed by the suffix string to be displayed (in
// this case, "ns"),and lastly, the minimum width of the field for the
// string (12 characters, in this example).Reference:  Verilog-XL
// Reference Manual.


  initial begin
     $timeformat(-9,3,"ns",12);
```

```
// Support for the Cadence waveform viewer is added here by calling the
// $shm_open task, which directs Verilog-XL to create a Simulation History
// Manager (SHM) database called "routed.shm".  The waveform viewer
// (SIMWAVES) displays the waveform information that has been
// written to .shm database.

// The $shm_probe ("AS") system task tells the Verilog-XL simulator
// which signal changes are to be recorded in the Simulation History
// Manager database.  In this case, "AS" specifies that ALL signal changes
// are to be recorded.

$shm_open("/export/vol1/m1.0/testing/m1tutorial/newsch/xilinx.run/
calcf.shm");
     $shm_probe("AS");
  end

// The "initial" block here displays the transitions of all the external
// signals of the design in vertical columnar format.

  initial begin
   $display("           T ongfedcbaggggsssssssssss");
   $display("           i fo       aaaattttwwwwwwww");
   $display("           m lt       uuuuaaaaiiiiiiii");
   $display("           e  g       ggggcccctttttttt");
   $display("              b       eeeekkkkcccccccc");
   $display("              l       [[[[llllhhhhhhhh");
   $display("              r       3210eeee[[[[[[[[");
   $display("              e       ]]]]dddd76543210");
   $display("              s           [[[[]]]]]]]]");
   $display("              e           3210        ");
   $display(“              t           ]]]]        “);


// The $monitor system task specifies the format by which all signal
// changes will appear when displayed in text format.  In this example,
// "gauge", "stackled", and "switch" are displayed in bus format.


    $monitor("%t",$realtime,, ofl, notgblresetg, f, e, d, c, b, a, "%b",
gauge, "%b", stackled, "%b", switch );

  end

  initial begin
    #100
      notgblreset = 0 ;
      switch = 0 ;
```

```
// ########## USER-SPECIFIED test vectors can be added here  #############
// ########## USER-SPECIFIED test vectors can be added here  #############
// ########## USER-SPECIFIED test vectors can be added here  #############
// ######### USER-SPECIFIED test vectors can be added here  #############

   #1000 $stop;

    #1000 finish;
   end

endmodule
```

Sample Test Fixture - XC4000EX Unified Library Functional Simulation (GSR and GTS simulation)

This text fixture is for a design called *count_invstartup* containing a STARTUP block with only the GSR pin connected to a pad via a signal called “mygsr”. The net connected to the GSR pin is “gsrin”.

```
// NGD2VER VERILOG TestFixture Template
// Design file: count_startup.ngd
// Date:Thu May 15 21:15:11 1997


// ATTENTION: This file was created by NGD2VER and may therefore be
// overwritten by subsequent runs of NGD2VER. Xilinx recommends that you
// copy this file to a new name, or 'paste' this text into another file,
// to avoid accidental loss of data.

`timescale 1 ns/1 ps

module test;

  reg mygsr;
  reg clock;
  wire [15:0] q;
  wire [7:0] qpp;

// To properly simulate your design containing a Startup component,
// be sure to do the following at the beginning of your simulation:

// 1. Toggle your GSR port to initialize all registers.
// 2. Deactivate your global tri-state (GTS) control signal.


`define GSR_SIGNAL test.uut.gsrin
```

```
// For HDL Direct / Unified Library simulation, you may optionally create
// a GTS control signal and connect it to the GTS_SIGNAL declared
// in the Unified Library simulation models.

wire GTS;
  `define GTS_SIGNAL test.GTS


  count_invstartup uut ( .mygsr (mygsr) , .clock (clock) , .q (q) , .qpp
(qpp)

);

initial begin

    $timeformat(-9,3,"ns",12);
    $shm_open("count_star.shm");
    $shm_probe("AS");

  end


  initial begin

    $display("           T mGGcqqqqqqqqqqqqqqqqqqqqqqqq");

    $display("           i ySTl[[[[[[[[[[[[[[[[pppppppp");

    $display("           m gRSo1111119876543210pppppppp");

    $display("           e s  c543210]]]]]]]]]][[[[[[[[");

    $display("             r  k]]]]]]          76543210");

    $display("                               ]]]]]]]]");

    $monitor("%t",$realtime,, mygsr, `GSR_SIGNAL, `GSR_SIGNAL,  clock,
"%b", q, "%b", qpp );

   // $monitor("%t",$realtime,, mygsr, `GSR_SIGNAL, `GTS_SIGNAL,  clock,
"%b", q, "%b", qpp );

  end

always #20 clock = ~clock;

  initial begin

      mygsr = 0 ;
      clock = 1 ;
      force `GTS_SIGNAL = 0;

    #100
```

```
      mygsr = 1 ;
    #3000 $stop;
    // #1000 $finish;
  end
endmodule
```

Sample Test Fixture - XC4000EX Post-NGDBuild Simulation (GSR and GTS simulation)

This text fixture is for the design, *count_invstartup*, containing a STARTUP block with only the GSR pin connected to a pad via a signal called “mygsr”. The net connected to the GSR pin is “gsrin”.

In this example, the GTS pin on the STARTUP block is not connected.

This sample test fixture can be used for the following types of simulations.

- post-NGDBuild SIMPRIM library based functional simulation
- post-map functional simulation
- post-implementation timing simulation

```
// NGD2VER VERILOG TestFixture Template
// Design file: count_invstartup.ngd
// Date:Thu May 15 21:15:11 1997


// ATTENTION: This file was created by NGD2VER and may therefore be
// overwritten by subsequent runs of NGD2VER. Xilinx recommends that you
// copy this file to a new name, or 'paste' this text into another file,
// to avoid accidental loss of data.


`timescale 1 ns/1 ps


module test;
  reg mygsr;
  reg clock;
  wire [15:0] q;

  wire [7:0] qpp;


  // To properly simulate your design containing a Startup component,
  // be sure to do the following at the beginning of your simulation:

  // 1. Toggle your GSR port to initialize all registers.

  //

  // 2. Deactivate your global tri-state (GTS) control signal.

  //

//`define GSR_SIGNAL test.uut.gsrin
// Comment out this definition


// wire GTS;
// `define GTS_SIGNAL test.GTS

 `define GTS_SIGNAL test.uut.GTS


  count_invstartup uut ( .mygsr (mygsr) , .clock (clock) , .q (q) , .qpp
(qpp)

);


  initial begin

    $timeformat(-9,3,"ns",12);

    $shm_open("count_star.shm");

    $shm_probe("AS");

  end


  initial begin

    $display("          T mGGcqqqqqqqqqqqqqqqqqqqqqqqq");

    $display("          i ySTl[[[[[[[[[[[[[[[[pppppppp");

    $display("          m gRSo1111119876543210pppppppp");

    $display("          e s  c543210]]]]]]]]]][[[[[[[[");
    $display("             r  k]]]]]]          76543210");
    $display("                               ]]]]]]]]");
    $monitor("%t",$realtime,, mygsr, test.uut.GSR, `GTS_SIGNAL,  clock,
"%b", q, "%b", qpp );
  end


  always #20 clock = ~clock;


  initial begin
      mygsr = 0 ;
      clock = 1 ;
      force `GTS_SIGNAL = 0;
    #100
      mygsr = 1 ;
    #3000 $stop;
    // #1000 $finish;
  end


endmodule
```

Sample Test Fixture, No Startup Block in the Design

When a design does not contain a STARTUP block, the same test fixture can usually be used for Unified Library functional simulation, post-NGDBuild functional simulation, as well as timing simulation.

```
// NGD2VER VERILOG TestFixture Template
// Design file: count_top.ngd
// Date:Fri May 16 18:59:39 1997

// ATTENTION: This file was created by NGD2VER and may therefore be
// overwritten by subsequent runs of NGD2VER. Xilinx recommends that you
// copy this file to a new name, or 'paste' this text into another file,
// to avoid accidental loss of data.

`timescale 1 ns/1 ps
```

```
module test;
  reg clock;
  wire [15:0] q;
  wire [7:0] qpp;

  reg GSR;
  `define GSR_SIGNAL test.GSR

  reg GTS;
  `define GTS_SIGNAL test.GTS

  count_top uut ( .clock (clock) , .q (q) , .qpp (qpp) );

  initial begin
    $timeformat(-9,3,"ns",12);
    $shm_open("count_topf.shm");
    $shm_probe("AS");
  end

  initial begin
    $display("           T GGcqqqqqqqqqqqqqqqqqqqqqqqq");
    $display("           i STl[[[[[[[[[[[[[[[[pppppppp");
    $display("           m RSo1111119876543210pppppppp");
    $display("           e   c543210]]]]]]]]]][[[[[[[[");
    $display("               k]]]]]]          76543210");
    $display("                              ]]]]]]]]");
    $monitor("%t",$realtime,, `GSR_SIGNAL, `GTS_SIGNAL, clock, "%b", q,
"%b", qpp );
  end
  always #20 clock = ~clock;

  initial begin
      clock = 0 ;
      `GSR_SIGNAL = 1;
      `GTS_SIGNAL = 0;
    #100
      `GSR_SIGNAL = 0;
    #3000 $stop;
    // #1000 $finish;
  end

endmodule
```

Appendix F

XILINX.PFF Property Filter File Format

The xilinx.pff property filter file, located in $XILINX/cadence/data, specifies what properties SIR2EDF (called by CONCEPT2XIL) should write out to the EDIF netlist from your Concept schematic. The xilinx.pff file also specifies what the format of these properties should look like when they are written out to the EDIF netlist.

The xilinx.pff file has sections for symbol, pin, and net properties. Each section begins with a property section header record of the form, $*xxx*_PROP.

For symbol properties this header record is:

```
$SYM_PROP
```

This header is followed by a list of properties that may be legally placed on symbols in Concept.

Similarly, for pin properties, the header record for the pin property section is called:

```
$PIN_PROP
```

For net properties:

```
$NET_PROP
```

Each section also ends with a property section END record:

```
$END.
```

Comments are specified with curly braces:

{ This is a comment in the property format file. }

You must provide a property specification record for every property that you want SIR2EDF to recognize and write out to the output EDIF file. The syntax of the property specification record is as follows:

```
property_name:   "property_format"   property_cast   property_type;
```

property_name— is the name of the property as it will be used in your design;

property_format— depicts the appearance and format of the property when it is written to the EDIF file (this specification is case-sensitive);

property_cast— is the data type of the property value. Valid cast types are:

- Boolean (valid property values are TRUE or FALSE)
- Integer (valid property values are integers)
- Real (valid property values are real numbers)
- String (valid property values are alphanumeric strings);

property_type— the SIR2EDF type of the property, which is used by SIR2EDF to determine how the property should be handled. The only valid SIR2EDF property type for Xilinx properties is NORMAL.

```
Example:
loc:             “LOC”          String  NORMAL;
```

In the example, the “loc” property (found in the $SYM_PROP symbol properties section) has a property_name of “loc”. When written out to the EDIF file, it will be written out to the EDIF file as “LOC=*value*”, where “LOC” is in upper case, and ”*value”* is some alphanumeric string.

As with all Xilinx properties, the SIR2EDF type is “NORMAL”.

```
Example :
keep:             “KEEP”        Boolean  NORMAL;
```

In this example, the property, “keep” from the $NET_PROPS section is a Boolean property which can be attached to a net. Its value must be specified as either TRUE or FALSE in the input design. When written out to the EDIF file, it will appear as “KEEP=TRUE”.

The standard xilinx.pff property filter file shipped with the M1 Cadence Concept interface is located in $XILINX/cadence/data and is listed below.

Note: The properties which are commented with the notation, “{For lblox_syn}” are only provided to support netlisting of Cadence Synergy Verilog designs which target LogiBLOX library components.

Note: Properties labelled as “{Not entered by user}” are generated by internally by Xilinx tools only.

```
XILINX.PFF File:
{Concept2xil property formatfile-4/19,1996,}
{Referenced rev1.2 of UVC and Attribute Description}

$SYM_PROP
ASYNC_VAL:      “ASYNC_VAL”     String  NORMAL; {For lblox_syn}
ASYNC_COUNT:    “ASYNC_COUNT”   String  NORMAL; {For lblox_syn}
SYNC_COUNT:     “SYNC_COUNT”    String  NORMAL; {For lblox_syn}
BUS_WIDTH:      “BUS_WIDTH”     String  NORMAL; {For lblox_syn}
COUNT_TO:       “COUNT_TO”      String  NORMAL; {For lblox_syn}
C_VALUE:        “C_VALUE”       String  NORMAL; {For lblox_syn}
DECODEMASK:     “DECODEMASK”    String  NORMAL; {For lblox_syn}
DEF:            “DEF”           String  NORMAL; {For lblox_syn}
DELAY:          “DELAY”         String  NORMAL; {For lblox_syn}
DEPTH:          “DEPTH”         String  NORMAL; {For lblox_syn}
DIVIDE_BY:      “DIVIDE_BY”     String  NORMAL; {For lblox_syn}
DUTY_CYCLE:     “DUTY_CYCLE”    String  NORMAL; {For lblox_syn}
ENCODING:       “ENCODING”      String  NORMAL; {For lblox_syn}
FLOAT_VAL:      “FLOAT_VAL”     String  NORMAL; {For lblox_syn}
INPUT_BUSSES:   “INPUT_BUSSES”  String  NORMAL; {For lblox_syn}
IN_TYPE:        “IN_TYPE”       String  NORMAL; {For lblox_syn}
INVMASK:        “INVMASK”       String  NORMAL; {For lblox_syn}
PAD_LOC:        “PAD_LOC”       String  NORMAL; {For lblox_syn}
MEMFILE:        “MEMFILE”       String  NORMAL; {For lblox_syn}
```

```
MODTYPE:        “MODTYPE”       String  NORMAL; {For lblox_syn}
OPTYPE:         “OPTYPE”        String  NORMAL; {For lblox_syn}
OUT_TYPE:       “OUT_TYPE”      String  NORMAL; {For lblox_syn}
REGISTERED:     “REGISTERED”    String  NORMAL; {For lblox_syn}
REG_OUT:        “REG_OUT”       String  NORMAL; {For lblox_syn}
SHIFT_TYPE:     “SHIFT_TYPE”    String  NORMAL; {For lblox_syn}
SLEWRATE:       “SLEWRATE”      String  NORMAL; {For lblox_syn}
STYLE:          “STYLE”         String  NORMAL; {For lblox_syn}
SYNC_VAL:       “SYNC_VAL”      String  NORMAL; {For lblox_syn}
add:            “ADD”           String  NORMAL; {Not entered by user.}
alu:            “ALU”           String  NORMAL; {Not entered by user.}
base:           “BASE”          String  NORMAL; {3k only}
blknm:          “BLKNM”         String  NORMAL;
bufg:           “BUFG”          String  NORMAL; {7k,9k}
clock_out:      “CLOCK_OUTPUT”  Boolean NORMAL; {4KXV only}
config:         “CONFIG”        String  NORMAL;
cymode:         “CYMODE”        String  NORMAL; {Not entered by user.}
decode:         “DECODE”        Boolean NORMAL;
d_invert:       “D_INVERT”      String  NORMAL; {Not entered by user.}
divide1_by:     “DIVIDE1_BY”    Integer NORMAL;
divide2_by:     “DIVIDE2_BY”    Integer NORMAL;
double:         “DOUBLE”        Boolean NORMAL;
drive:          “DRIVE”         Integer NORMAL; {4KXV only}
eqn:            “EQN”           String  NORMAL; {Not entered by user.}
equate_f:       “EQUATE_F”      String  NORMAL;
equate_g:       “EQUATE_G”      String  NORMAL;
fast:           “FAST”          Boolean NORMAL;
file:           “FILE”          String  NORMAL;
grp01:          “GRP01”         String  NORMAL; {Timegroup}
grp02:          “GRP02”         String  NORMAL;
```

```
grp03:          “GRP03”         String  NORMAL;
grp04:          “GRP04”         String  NORMAL;
grp05:          “GRP05”         String  NORMAL;
grp06:          “GRP06”         String  NORMAL;
grp07:          “GRP07”         String  NORMAL;
grp08:          “GRP08”         String  NORMAL;
grp09:          “GRP09”         String  NORMAL;
grp010:         “GRP010”        String  NORMAL;
hblknm:         “HBLKNM”        String  NORMAL;
hu_set:         “HU_SET”        String  NORMAL;
init:           “INIT”          String  NORMAL;
libver:         “LIBVER”        String  NORMAL; {Not entered by user.}
io:             “IO”            Boolean NORMAL;
loc:            “LOC”           String  NORMAL;
lowpwr:         “LOWPWR”        String  NORMAL;
map:            “MAP”           String  NORMAL;
meddelay:       “MEDDELAY”      Boolean NORMAL;
minim:          “MINIM”         String  NORMAL;
nodelay:        “NODELAY”       Boolean NORMAL;
opt:            “OPT”           String  NORMAL;
optimize:       “OPTIMIZE”      String  NORMAL;
opt_effort:     “OPT_EFFORT”    String  NORMAL;
osc:            “OSC”           String  NORMAL; {5k only}
part:           “PART”          String  NORMAL;
prohibit:       “PROHIBIT”      String  NORMAL;
pwr_mode:       “PWR_MODE”      String  NORMAL;
reg:            “REG”           String  NORMAL; {Not entered by user.}
rloc:           “RLOC”          String  NORMAL;
rloc_origin:    “RLOC_ORIGIN”   String  NORMAL;
rloc_range:     “RLOC_RANGE”    String  NORMAL;
```

```
slow:          "SLOW"         Boolean NORMAL;
tig:           "TIG"          String  NORMAL;
tnm:           "TNM"          String  NORMAL;
ts01:          "TS01"         String  NORMAL; {Timespec}
ts02:          "TS02"         String  NORMAL;
ts03:          "TS03"         String  NORMAL;
ts04:          "TS04"         String  NORMAL;
ts05:          "TS05"         String  NORMAL;
ts06:          "TS06"         String  NORMAL;
ts07:          "TS07"         String  NORMAL;
ts08:          "TS08"         String  NORMAL;
ts09:          "TS09"         String  NORMAL;
ts10:          "TS10"         String  NORMAL;
u_set:         "U_SET"        String  NORMAL;
use_rloc:      "USE_RLOC"     Boolean NORMAL;
wireand:       "WIREAND"      Boolean NORMAL;
inreg:         "INREG"        String  NORMAL; {5k ONLY}
outreg:        "OUTREG"       String  NORMAL; {5k ONLY}
bsreadback:    "BSREADBACK"   String  NORMAL; {For Config Symbol Only}
bsreconfig:    "BSRECONFIG"   String  NORMAL; {For Config Symbol Only}
configrate:    "CONFIGRATE"   String  NORMAL; {For Config Symbol Only}
crc:           "CRC"          String  NORMAL; {For Config Symbol Only}
donepin:       "DONEPIN"      String  NORMAL; {For Config Symbol Only}
doneactive:    "DONEACTIVE"   String  NORMAL; {For Config Symbol Only}
gsrinactive:   "GSRINACTIVE"  String  NORMAL; {For Config Symbol Only}
inputs:        "INPUTS"       String  NORMAL; {For Config Symbol Only}
lengthcount:   "LENGTHCOUNT"  String  NORMAL; {For Config Symbol Only}
m0pin:         "M0PIN"        String  NORMAL; {For Config Symbol Only}
m1pin:         "M1PIN"        String  NORMAL; {For Config Symbol Only}
m2pin:         "M2PIN"        String  NORMAL; {For Config Symbol Only}
```

```
oscillator:     “OSCILLATOR”    String  NORMAL; {For Config Symbol Only}
oscclk:         “OSCCLK”        String  NORMAL; {For Config Symbol Only}
outputs:        “OUTPUTS”       String  NORMAL; {For Config Symbol Only}
outputsactive:  “OUTPUTSACTIVE” String  NORMAL; {For Config Symbol Only}
progmode:       “PROGMODE”      String  NORMAL; {For Config Symbol Only}
progpin:        “PROGPIN”       String  NORMAL; {For Config Symbol Only}
readabort:      “READABORT”     String  NORMAL; {For Config Symbol Only}
readback:       “READBACK”      String  NORMAL; {For Config Symbol Only}
readcapture:    “READCAPTURE”   String  NORMAL; {For Config Symbol Only}
readclk:        “READCLK”       String  NORMAL; {For Config Symbol Only}
startupclk:     “STARTUPCLK”    String  NORMAL; {For Config Symbol Only}
startupmode:    “STARTUPMODE”   String  NORMAL; {For Config Symbol Only}
synctodone:     “SYNCTODONE”    String  NORMAL; {For Config Symbol Only}
userstring:     “USERSTRING”    String  NORMAL; {For Config Symbol Only}
$END.
$PIN_PROP
tig:            “TIG”           String  NORMAL;
tnm:            “TNM”           String  NORMAL;
tspec:          “TSPEC”         String  NORMAL;
$END.
$NET_PROP
bufg:           “BUFG”          String  NORMAL; {7k,9k}
collapse:       “COLLAPSE”      Boolean NORMAL;
cymode:         “CYMODE”        String  NORMAL;
decode:         “DECODE”        Boolean NORMAL;
divide1_by:     “DIVIDE1_BY”    Integer NORMAL;
divide2_by:     “DIVIDE2_BY”    Integer NORMAL;
double:         “DOUBLE”        Boolean NORMAL;
fast:           “FAST”          Boolean NORMAL;
hblknm:         “HBLKNM”        String  NORMAL;
```

```
hu_set:        “HU_SET”        String  NORMAL;
init:          “INIT”          String  NORMAL;
io:            “IO”            Boolean NORMAL;
keep:          “KEEP”          Boolean NORMAL;
loc:           “LOC”           String  NORMAL;
maxdelay:      “MAXDELAY”      String  NORMAL;
maxskew:       “MAXSKEW”       String  NORMAL;
noreduce:      “NOREDUCE”      Boolean NORMAL;
offset:        “OFFSET”        String  NORMAL;
opt:           “OPT”           String  NORMAL;
period:        “PERIOD”        String  NORMAL;
pwr_mode:      “PWR_MODE”      String  NORMAL;
slow:          “SLOW”          Boolean NORMAL;
tig:           “TIG”           String  NORMAL;
tpsync:        “TPSYNC”        String  NORMAL;
tpthru:        “TPTHRU”        String  NORMAL;
tspec:         “TSPEC”         String  NORMAL;
tnm:           “TNM”           String  NORMAL;
wireand:       “WIREAND”       Boolean NORMAL;
f:             “F”             Boolean NORMAL; {Net Flag Attributes}
s:             “S”             Boolean NORMAL; {Net Flag Attributes}
h:             “H”             Boolean NORMAL; {Net Flag Attributes}
p:             “P”             Boolean NORMAL; {Net Flag Attributes}
x:             “X”             Boolean NORMAL; {Net Flag Attributes}
$END.
```

figures/x7425.epsi 2
figures/x7426.epsi 2
figures/figicon.rs @ 72 dpi vii
figures/tblicon.rs @ 150 dpi vii
figures/cpyicon.rs @ 150 dpi vii
figures/footnote.rs @ 150 dpi vii
figures/x7747.epsi 1-8
figures/x8069.epsi 1-9
figures/x8062.epsi 3-3
figures/properties.rs @ 150 dpi 3-10
figures/x7839.epsi 3-16
figures/x7840.epsi 3-17
figures/x7926.epsi 3-17
figures/x7927.epsi 3-18
figures/x8064.epsi 3-23
figures/x8063.epsi 4-2
figures/veriloggui.rs @ 120 dpi 4-6
figures/veriloggui.rs @ 120 dpi 4-10
figures/x8066.epsi 4-12
figures/x8065.epsi 5-2
figures/x7834.epsi 5-3
figures/x8077.epsi 5-4
figures/x7760.epsi 6-3
figures/veriloggui.rs @ 120 dpi 6-7
figures/x8067.epsi 6-9
figures/gsrstartup.rs @ 150 dpi 7-11
figures/mygsr.ras @ 150 dpi 7-12
figures/startup5k.rs @ 150 dpi 7-14
figures/startupgts.rs @ 150 dpi 7-19
figures/calc.rs 9-11
figures/sted0.rs 9-12
figures/sted1.rs 9-12
figures/sted2.rs 9-13
figures/compbrws.rs 9-17
figures/place1.rs 9-18
figures/place4.rs 9-19

figures/addbus.rs 9-21
figures/addnet.rs 9-22
figures/allnets.rs 9-23
figures/attslice.rs 9-24
figures/bustap.rs 9-25
figures/addports.rs 9-26
figures/andblk2.rs 9-27
figures/hdldir.rs 9-28
figures/orblk2s.rs 9-29
figures/symcompl.rs 9-31
figures/addblock.rs 9-33
figures/addfd4re.rs 9-34
figures/netfd4re.rs 9-35
figures/netandor.rs 9-37
figures/attform.rs 9-38
figures/addlabel.rs 9-39
figures/alu.rs 9-40
figures/fd4re.rs 9-41
figures/adsu4.rs 9-43
figures/clockgen.rs 9-44
figures/attloc.rs 9-46
figures/addloc.rs 9-47
figures/addfast.rs 9-48
figures/switch7.rs 9-49
figures/ram16x4s.rs 9-52
figures/replram.rs 9-53
figures/newclock.rs 9-54
figures/lbgui.rs 9-56
figures/lbgui_options.rs 9-57
figures/modsel.rs 9-58
figures/placeblx.rs 9-59
figures/startup.rs 9-61
figures/config.rs 9-62
figures/swbrowser.rs @ 120 dpi 9-76
figures/swtscle.rs @ 120 dpi 9-77
figures/swbbus.rs @ 120 dpi 9-78

figures/swwave.rs @ 120 dpi 9-79
figures/dsgnmgr.rs 9-81
figures/impldia.rs 9-82
figures/feopt.rs 9-83
figures/floweng.rs 9-85
figures/epicon.rs 9-87
figures/hdicon.rs 9-88
figures/gndreset.rs 9-90
figures/flowicon.rs 9-91
figures/lbgui.rs @ 150 dpi C-2
figures/lbgui_options.rs @ 150 dpi C-3
figures/lbgui_modsel.rs @ 150 dpi C-4
figures/lbgui_msgs.rs @ 150 dpi C-5
figures/x7762.epsi D-2