Foundation Series 2.1i User Guide

1- Introduction

2 - Project Toolset

3 - Design Methodologies - Schematic Flow

4 - Schematic Design Entry

5 - Design Methodologies - HDL Flow

6 - HDL Design Entry and Synthesis

7 - State Machine Designs

8 - LogiBLOX

9 - CORE Generator System

10 - Functional Simulation

11 - Design Implementation

12 - Verification and Programming

 Printed in U.S.A.

The Xilinx logo shown above is a registered trademark of Xilinx, Inc.

FPGA Architect, FPGA Foundry, NeoCAD, NeoCAD EPIC, NeoCAD PRISM, NeoROUTE, Timing Wizard, TRACE, XACT, XILINX, XC2064, XC3090, XC4005, XC5210, and XC-DS501 are registered trademarks of Xilinx, Inc.

The shadow X shown above is a trademark of Xilinx, Inc.

All XC-prefix product designations, A.K.A. Speed, Alliance Series, AllianceCORE, BITA, CLC, Configurable Logic Cell, CORE Generator, CoreGenerator, CoreLINX, Dual Block, EZTag, FastCLK, FastCONNECT, FastFLASH, FastMap, Foundation, HardWire, LCA, LogiBLOX, Logic Cell, LogiCORE, LogicProfessor, MicroVia, PLUSASM, PowerGuide, PowerMaze, QPro, RealPCI, RealPCI 64/66, SelectI/O, Select-RAM, Select-RAM+, Smartguide, Smart-IP, SmartSearch, Smartspec, SMARTSwitch, Spartan, TrueMap, UIM, VectorMaze, VersaBlock, VersaRing, Virtex, WebLINX, XABEL, XACT*step*, XACT*step* Advanced, XACT*step* Foundry, XACT-Floorplanner, XACT-Performance, XAM, XAPP, X-BLOX, X-BLOX plus, XChecker, XDM, XDS, XEPLD, Xilinx Foundation Series, XPP, XSI, and ZERO+ are trademarks of Xilinx, Inc. The Programmable Logic Company and The Programmable Gate Array Company are service marks of Xilinx, Inc.

All other trademarks are the property of their respective owners.

Xilinx, Inc. does not assume any liability arising out of the application or use of any product described or shown herein; nor does it convey any license under its patents, copyrights, or maskwork rights or any rights of others. Xilinx, Inc. reserves the right to make changes, at any time, in order to improve reliability, function or design and to supply the best product possible. Xilinx, Inc. will not assume responsibility for the use of any circuitry described herein other than circuitry entirely embodied in its products. Xilinx, Inc. devices and products are protected under one or more of the following U.S. Patents: 4,642,487; 4,695,740; 4,706,216; 4,713,557; 4,746,822; 4,750,155; 4,758,985; 4,820,937; 4,821,233; 4,835,418; 4,855,619; 4,855,669; 4,902,910; 4,940,909; 4,967,107; 5,012,135; 5,023,606; 5,028,821; 5,047,710; 5,068,603; 5,140,193; 5,148,390; 5,155,432; 5,166,858; 5,224,056; 5,243,238; 5,245,277; 5,267,187; 5,291,079; 5,295,090; 5,302,866; 5,319,252; 5,319,254; 5,321,704; 5,329,174; 5,329,181; 5,331,220; 5,331,226; 5,332,929; 5,337,255; 5,343,406; 5,349,248; 5,349,249; 5,349,250; 5,349,691; 5,357,153; 5,360,747; 5,361,229; 5,362,999; 5,365,125; 5,367,207; 5,386,154; 5,394,104; 5,399,924; 5,399,925; 5,410,189; 5,410,194; 5,414,377; 5,422,833; 5,426,378; 5,426,379; 5,430,687; 5,432,719; 5,448,181; 5,448,493; 5,450,021; 5,450,022; 5,453,706; 5,455,525; 5,466,117; 5,469,003; 5,475,253; 5,477,414; 5,481,206; 5,483,478; 5,486,707; 5,486,776; 5,488,316; 5,489,858; 5,489,866; 5,491,353; 5,495,196; 5,498,979; 5,498,989; 5,499,192; 5,500,608; 5,500,609; 5,502,000; 5,502,440; 5,504,439; 5,506,518; 5,506,523; 5,506,878; 5,513,124; 5,517,135; 5,521,835; 5,521,837; 5,523,963; 5,523,971; 5,524,097; 5,526,322; 5,528,169; 5,528,176; 5,530,378; 5,530,384; 5,546,018; 5,550,839; 5,550,843; 5,552,722; 5,553,001; 5,559,751; 5,561,367; 5,561,629; 5,561,631; 5,563,527; 5,563,528; 5,563,529; 5,563,827; 5,565,792; 5,566,123; 5,570,051; 5,574,634; 5,574,655; 5,578,946; 5,581,198; 5,581,199; 5,581,738; 5,583,450; 5,583,452; 5,592,105; 5,594,367; 5,598,424; 5,600,263; 5,600,264; 5,600,271; 5,600,597; 5,608,342; 5,610,536; 5,610,790; 5,610,829; 5,612,633; 5,617,021; 5,617,041; 5,617,327; 5,617,573; 5,623,387; 5,627,480; 5,629,637; 5,629,886; 5,631,577; 5,631,583; 5,635,851; 5,636,368; 5,640,106; 5,642,058; 5,646,545; 5,646,547; 5,646,564; 5,646,903; 5,648,732; 5,648,913; 5,650,672; 5,650,946; 5,652,904; 5,654,631; 5,656,950; 5,657,290; 5,659,484; 5,661,660; 5,661,685; 5,670,896; 5,670,897; 5,672,966; 5,673,198; 5,675,262; 5,675,270; 5,675,589; 5,677,638; 5,682,107; 5,689,133; 5,689,516; 5,691,907; 5,691,912; 5,694,047; 5,694,056; 5,724,276; 5,694,399; 5,696,454; 5,701,091; 5,701,441; 5,703,759; 5,705,932; 5,705,938; 5,708,597; 5,712,579; 5,715,197; 5,717,340; 5,719,506; 5,719,507; 5,724,276; 5,726,484; 5,726,584; 5,734,866; 5,734,868; 5,737,234; 5,737,235; 5,737,631; 5,742,178; 5,742,531; 5,744,974; 5,744,979; 5,744,995; 5,748,942; 5,748,979; 5,752,006; 5,752,035; 5,754,459; 5,758,192; 5,760,603; 5,760,604; 5,760,607; 5,761,483; 5,764,076; 5,764,534; 5,764,564; 5,768,179; 5,770,951; 5,773,993; 5,778,439; 5,781,756; 5,784,313; 5,784,577; 5,786,240; 5,787,007; 5,789,938; 5,790,479;

5,790,882; 5,795,068; 5,796,269; 5,798,656; 5,801,546; 5,801,547; 5,801,548; 5,811,985; 5,815,004; 5,815,016; 5,815,404; 5,815,405; 5,818,255; 5,818,730; 5,821,772; 5,821,774; 5,825,202; 5,825,662; 5,825,787; 5,828,230; 5,828,231; 5,828,236; 5,828,608; 5,831,448; 5,831,460; 5,831,845; 5,831,907; 5,835,402; 5,838,167; 5,838,901; 5,838,954; 5,841,296; 5,841,867; 5,844,422; 5,844,424; 5,844,829; 5,844,844; 5,847,577; 5,847,579; 5,847,580; 5,847,993; 5,852,323; Re. 34,363, Re. 34,444, and Re. 34,808. Other U.S. and foreign patents pending. Xilinx, Inc. does not represent that devices shown or products described herein are free from patent infringement or from any other third party right. Xilinx, Inc. assumes no obligation to correct any errors contained herein or to advise any user of this text of any correction if such be made. Xilinx, Inc. will not assume any liability for the accuracy or correctness of any engineering or software support or assistance provided to a user.

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

Copyright 1991-1999 Xilinx, Inc. All Rights Reserved.

Foundation Series 2.1i User Guide

Appendix A - Glossary

Appendix B - Foundation Constraints

Appendix C - Instantiated Components

Appendix D - File Processing Overview

About This Manual

This *Foundation Series 2.1i User Guide* provides a detailed description of the Foundation™ design methodologies, design entry tools, simulation (both functional and timing simulation). Information on synthesis is included for Foundation Express users.The manual also briefly describes the Xilinx design implementation tools. Detailed descriptions of the design implementation tools can be found in two other online books, *Design Manager/Flow Engine Guide* and *Development System Reference Guide*.

Before using this manual, you should be familiar with the operations that are common to all Xilinx 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 *Foundation Series 2.1i Quick Start Guide*. Consult the *Verilog Reference Guide* and the *VHDL Reference Guide* for detailed information on using Verilog and VHDL with Foundation Express.

Additional Resources

For additional information, go to http://support.xilinx.com. The following table lists some of the resources you can access from this page. You can also directly access some of these resources using the provided URLs.

Resource	Description/URL
Tutorial	Tutorials covering Xilinx design flows, from design entry to verification and debugging http://support.xilinx.com/support/techsup/tutorials/index.htm
Answers Database	Current listing of solution records for the Xilinx software tools Search this database using the search function at http://support.xilinx.com/support/searchtd.htm

Resource	Description/URL
Application Notes	Descriptions of device-specific design techniques and approaches http://support.xilinx.com/apps/appsweb.htm
Data Book	Pages from *The Programmable Logic Data Book*, which describe device-specific information on Xilinx device characteristics, including read-back, boundary scan, configuration, length count, and debugging http://support.xilinx.com/partinfo/databook.htm
Xcell Journals	Quarterly journals for Xilinx programmable logic users http://support.xilinx.com/xcell/xcell.htm
Tech Tips	Latest news, design tips, and patch information on the Xilinx design environment http://support.xilinx.com/support/techsup/journals/index.htm

Manual Contents

This guide covers the following topics:

- **Chapter 1**, "Introduction," lists supported architectures, platforms, and features. It also lists the available documentation and tutorials to help you get started with Foundation.
- **Chapter 2**, "Project Toolset," explains the two Foundation project types—Schematic Flow projects and HDL Flow projects—and how to access the various Foundation design tools from the Project Manager. It briefly describes each tool and its function.
- **Chapter 3**, "Design Methodologies - Schematic Flow," describes various design methodologies for top-level schematic designs and state machine designs in Schematic Flow projects.
- **Chapter 4**, "Schematic Design Entry," explains how to manage your schematic designs and how to create hierarchical schematic designs.
- **Chapter 5**, "Design Methodologies - HDL Flow," describes various design methodologies for HDL, schematic, and state machine designs in HDL Flow projects.
- **Chapter 6**, "HDL Design Entry and Synthesis," describes how to create top-level HDL designs, explains how to manage large designs, and discusses advanced design techniques.

- **Chapter 7**, “State Machine Designs,” explains the basic operations for creating state machine designs.
- **Chapter 8**, “LogiBLOX,” explains how to create LogiBLOX™ modules and how to use them in schematic and HDL designs.
- **Chapter 9**, “CORE Generator System” gives an overview of the Xilinx CORE Generator System.
- **Chapter 10**, “Functional Simulation,” describes the basic functional simulation process.
- **Chapter 11**, “Design Implementation,” briefly describes how to implement your design with the Xilinx Implementation Tools. The chapter also describes how to select various design options in the Implementation Options dialog box and describes the Implementation reports.
- **Chapter 12**, “Verification and Programming,” explains how to generate a timing-annotated netlist, how to perform a static timing analysis, and describes the basic timing simulation process. An overview of the device download tools is also included.
- **Appendix A**, “Glossary,” defines some of the commonly used terms in this manual.
- **Appendix B**, “Foundation Constraints,” discusses some of the more common constraints you can apply to your design to control the timing and layout of a Xilinx FPGA or CPLD. It describes how to use constraints at each stage of design processing.
- **Appendix C**, “Instantiated Components,” lists the components most frequently instantiated in synthesis designs.
- **Appendix D**, “File Processing Overview,” contains diagrams of the file manipulations for FPGAs and CPLDs during the design process.

Conventions

This manual uses the following typographical and online document conventions. An example illustrates each typographical convention.

Typographical

The following conventions are used for all documents.

- `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. However, braces "{ }" in Courier bold are not literal and square brackets "[ ]" in Courier bold are literal only in the case of bus specifications, such as bus [7:0].

  `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*

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

  `lowpwr` ={**`on`**|**`off`**}

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

  `lowpwr` ={**`on`**|**`off`**}

- 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

The following conventions are used for online documents.

- Red-underlined text indicates an interbook link, which is a cross-reference to another book. Click 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 the blue-underlined text to open the specified cross-reference.

Chapter 1

Introduction

This chapter contains the following sections.

- “Architecture Support”
- “Platform Support”
- “Foundation Demo”
- “Tutorials”
- “Online Help”
- “Books”

Architecture Support

Foundation supports the following Xilinx device families.

- XC3000A/L
- XC3100A/L
- XC4000E/L/EX/XL/XV/XLA
- XC5200
- XC9500, XC9500XL, XC9500XV
- Spartan, SpartanXL
- Virtex

The primary difference between these products lies in the number of gates and the architectural features of the individual devices.

For a detailed list of supported devices, see the “Device and Package Support” chapter in the *Foundation Series 2.1i Installation Guide and Release Notes.*

Platform Support

Foundation runs on Windows NT 4.0, Windows 95, and Windows 98.

Foundation Demo

After you install the Foundation Series 2.1i software, a multimedia demo of the Foundation product features is accessible from **Start** → **Programs** → **Xilinx Foundation Series 2.1i** → **Multimedia QuickStart (Requires CD)**.

Note: You must have the Foundation Series 2.1i Documentation CD in your PC's CD-ROM drive to run this demo.

Tutorials

The *Foundation Series 2.1i Quick Start Guide* contains a basic tutorial, JCOUNT, a simple 4-bit Johnson counter. This tutorial provides an overview of design entry, design implementation, and device programming for a Schematic Flow project. The HDL Flow version of the tutorial is also included for Base Express and Foundation Express users.

An in-depth tutorial, the Foundation Watch Tutorial, is available from the Education tab on the Xilinx support website (http://support.xilinx.com/support/techsup/tutorials/index.htm).

Online Help

Context-sensitive online help is available for Foundation applications. In addition, Foundation includes an "umbrella" help system called the Xilinx Foundation Series On-Line Help System. The umbrella help contains topics covering all of the design entry and implementation tools provided in the product plus additional information. It also contains in-depth information essential for designing with FPGAs and CPLDs, including the following topics:

- CPLD design techniques
- FPGA design techniques
- Application notes
- Several tutorials

- Reference information on the HDL languages, CPLD schematic library and attributes, and Foundation configurations

You can invoke the “umbrella” help system (shown in the following figure) by selecting **Help** → **Foundation Help Contents** from the Project Manager menu bar.

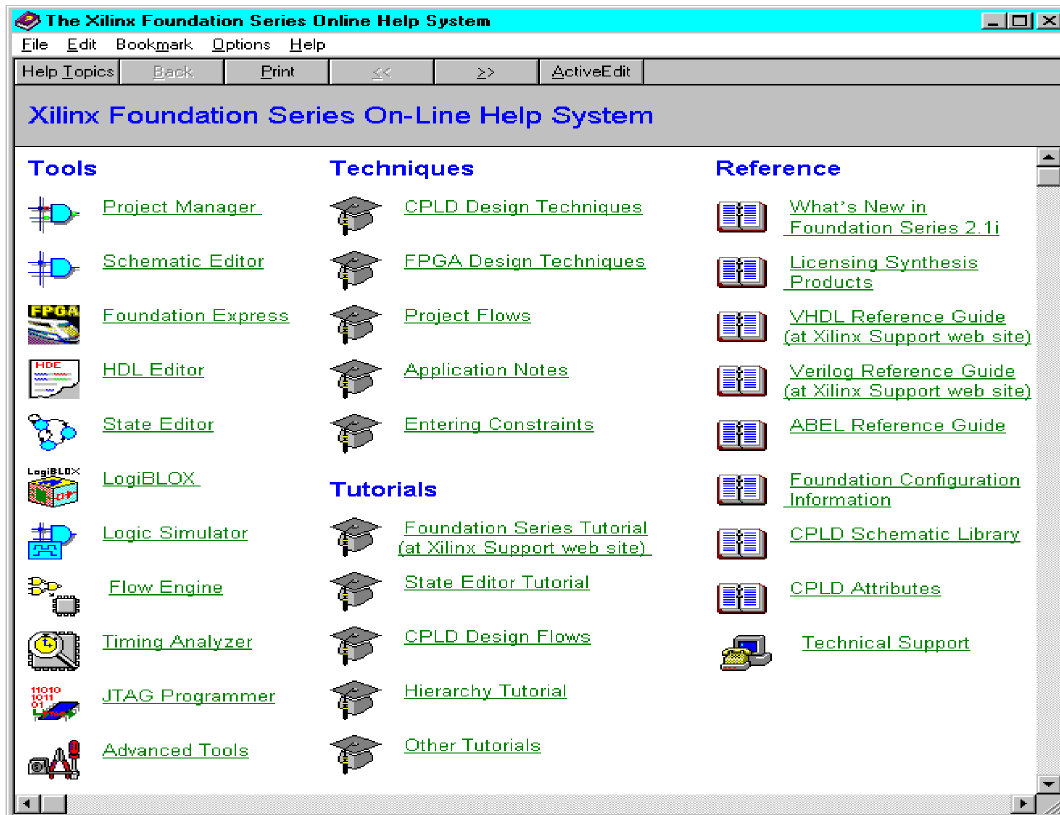


Figure 1-1 The Online “Umbrella” Help System

Books

Multiple printed and online books are available for the Foundation Series 2.1i product and the various tools included with it.

Printed Books

The *Foundation Series 2.1i Installation Guide and Release Notes* describes installation procedures, new features, supported devices, and the most critical known issues. It also includes information on the software license required for the Base Express and Foundation Express products.

The *Foundation Series 2.1i Quick Start Guide* provides an overview of the features and additions to Xilinx's 2.1i software. This book contains a tutorial overview of design entry tools and design implementation tools.

Adobe Acrobat PDF files for viewing and printing all of the Foundation Series 2.1i online books can be found in the print directory on the Documentation CD-ROM. Refer to the *Foundation Series 2.1i Installation Guide and Release Notes* for information on accessing and printing the PDF files. Or, click **`Help`** in the Document Viewer for instructions.

Online Books

The online Foundation Series 2.1i book collection is available from the Foundation Series 2.1i Documentation CD or from the Xilinx support page on the web at http://support.xilinx.com. You must use a Java-enabled HTML browser to view the Xilinx online books. If you do not already have an appropriate browser on your PC, you can install Netscape 4.0 from the Foundation Design Environment CD-ROM or the Foundation Documentation CD-ROM.

Document Viewer

The Document Viewer provided with Foundation Series 2.1i is powered by the Docsan™ indexing tool. This tool provides your HTML browser with optimal searching capabilities within the online book collection. Refer to the online help provided with the Document Viewer for detailed instructions on using this tool.

Foundation-Specific Online Books

The following online books contain information that applies only to the Xilinx Foundation Series products.

Title	Description
Foundation Series 2.1i Quick Start Guide	This guide gives an overview of the features and additions to Xilinx's Foundation 2.1i product. The primary focus of this guide is to show the relationship between the design entry tools and the design implementation tools. The guide also contains in-depth tutorials for a schematic-based and HDL-based stop watch design.
Foundation Series 2.1i User Guide	This guide provides a detailed description of the Foundation design methodologies, design entry tools, and both functional and timing simulation. The manual also briefly describes the Xilinx design implementation tools.
Verilog Reference Guide	This manual describes how to use Xilinx Foundation Express to translate and optimize a Verilog description into an internal gate-level equivalent.
VHDL Reference Guide	This manual describes how to use Xilinx Foundation Express to translate and optimize a VHDL description into an internal gate-level equivalent.

Design Entry Online Reference Books

The following books contain additional information not found in the Foundation-specific books regarding the Xilinx schematic library components (and constraints) and LogiBLOX.

Title	Description
Libraries Guide	This book describes the logic elements (primitives or macros), that you use to create your designs as well as the attributes and constraints used to process elements during logic implementation. It also discusses relationally placed macros (RPMs), which are macros that contain relative location constraints (RLOC) information. The Xilinx libraries enable you to convert designs easily from one family to another.
LogiBLOX Guide	This guide describes the high-level modules you can use to speed up design entry and the attributes that support logic synthesis, primarily for FPGA architectures. It also explains how to use the LogiBLOX program to create designs and the different types of logic synthesis completed by the LogiBLOX program.

Note: The CORE Generator User Guide is not currently part of the online book collection. It is an Adobe Acrobat file (.pdf) that can be accessed from the CORE Generator Help menu (**`Help`** → **`Online Documentation`**.)

Synthesis and Simulation Reference Book

The following book contains general information on Synthesis and Simulation.

Title	Description
Synthesis and Simulation Design Guide	This manual provides a general overview of designing FPGAs with Hardware Description Languages (HDLs). It includes design hints for the novice HDL user, as well as for the experienced user who is designing FPGAs for the first time.

Implementation-Related Online Books

The following books contain detailed information on the Xilinx implementation tools. Much of the information contained in these books is for the standalone or command line versions of the tool.

Title	Description
Constraints Editor Guide	This manual describes the Xilinx Constraints Editor GUI that can be used *after* the design has been implemented to modify or delete existing constraints or add new constraints to a design.
Design Manager/ Flow Engine Guide	This manual describes the Design Manager, a Xilinx Alliance Series tool for managing multiple implementations of the same design. This manual also explains the Xilinx Flow Engine, which implements designs, and explains how to interact with other programs that run in the Design Manager environment; namely, the Design Editor, the Timing Analyzer, the Hardware Debugger, the PROM File Formatter, and the PROM Programmer.
Development System Reference Guide	This book describes the Xilinx design implementation software, which includes programs to generate EDIF files, LCA files, and BIT files. The book covers all the program options and files that are generated by these programs. It also contains in-depth information on timing constraints.
FPGA Editor Guide	The FPGA Editor is a graphical editor used to display and configure FPGAs. The FPGA Editor enables you to place and route critical components before running automatic place and route tools on an entire design, modify placement and routing manually, interact with the physical constraints file (PCF) to create and modify constraints, and verify timing against constraints.
Floorplanner Guide	This book describes the Floorplanner, a graphical interface tool to help you improve performance and density of your design.

Title	Description
Hardware User Guide	This manual describes the Xilinx Demonstration hardware and its associated software interfaces. The hardware includes the FPGA and CPLD demonstration boards, which are used for design verification.
Timing Analyzer Guide	This manual describes Xilinx's Timing Analyzer program, a graphical user interface tool that performs static analysis of a mapped FPGA or CPLD design. The mapped design can be partially or completely placed, routed, or both.

Device Programming Online Books

Detailed information on the device programming process is included in the following books.

Title	Description
JTag Programmer Guide	This guide documents the graphical interface used for in-system programming and verification of CPLD and FPGA parts. The guide also describes how to set up and use JTAG download cables.
Hardware Debugger Guide	(FPGAs only) This manual describes how to program, verify, and debug FPGA devices. It describes the XChecker, MultiLINX, and Parallel III cables and explains how to connect the cable pins to your target device for various functions: downloading, verification, and debugging. It also includes a tutorial for debugging a design using the demonstration boards as target devices.
PROM File Formatter Guide	(FPGAs only) This manual explains how to use a Windows-based tool to format bitstream files into HEX format files compatible with Xilinx and third-party PROM programmers. You use the PROM files to program a PROM device, which is then used to configure daisy chains of one or more FPGAs for one application (configuration) or several applications (reconfiguration).

Contents

About This Manual

Additional Resources i
Manual Contents ii

Conventions

Typographical........ v
Online Document vi

Chapter 1 Introduction

Architecture Support 1-1
Platform Support 1-2
Foundation Demo........ 1-2
Tutorials 1-2
Online Help 1-2
Books 1-4
Printed Books........ 1-4
Online Books........ 1-4
Document Viewer 1-4
Foundation-Specific Online Books........ 1-5
Design Entry Online Reference Books 1-6
Synthesis and Simulation Reference Book........ 1-6
Implementation-Related Online Books 1-7
Device Programming Online Books........ 1-8

Chapter 2 Project Toolset

Creating Foundation 2.1i Projects........ 2-1
Schematic Flow Projects........ 2-2
HDL Flow Projects (Express Only) 2-5
Project Manager........ 2-7
Hierarchy Browser 2-8

Files Tab 2-9
Versions Tab.. 2-10
Project Flowchart Area.. 2-10
Flow Tab - Project Flowchart 2-10
Alternatives to Flowchart Buttons 2-11
Contents Tab 2-11
Reports Tab 2-11
Synthesis Tab (Schematic Flow Only) 2-11
Messages Area 2-12
Console Tab 2-12
HDL Errors Tab (HDL Flow Only) 2-12
HDL Warnings Tab (HDL Flow Only).. 2-12
HDL Messages Tab (HDL Flow Only) 2-12
Accessing LogiBLOX 2-12
Accessing the CORE Generator System 2-13
Documenting Your Design 2-13
Project Archiving 2-13
Design Entry Tools.. 2-14
Schematic Editor.. 2-14
State Editor 2-15
HDL Editor 2-15
Symbol Editor.. 2-16
Synthesis Tools.. 2-16
Synthesis Button (HDL Flow).. 2-16
Synthesis Tab (Schematic Flow) 2-17
Simulation/Verification.. 2-17
Logic Simulator 2-17
Timing Analyzer 2-17
Specialized Simulation Controls 2-18
HDL Behavioral Simulation Capabilities 2-18
Constraints Editors.. 2-19
Express Constraints Editor (HDL Flow) 2-19
Xilinx Constraints Editor.. 2-19
Implementation Tools.. 2-20
Control Files.. 2-20
User Constraints File 2-20
Implementation Guide File.. 2-20
Floorplanner File.. 2-21
Implementation Tools Menu.. 2-21
Constraints Editor 2-21
Flow Engine 2-21
Floorplanner.. 2-21

FPGA Editor 2-22
CPLD ChipViewer 2-22
Automatic Pin Locking 2-22
Device Programming 2-23
JTAG Programmer 2-23
PROM File Formatter 2-23
Hardware Debugger 2-23
Utilities 2-24
Schematic Symbol Library Manager 2-24
Command History 2-24
Project Notes 2-25
Implementation Template Manager 2-25
ABEL to VHDL/Verilog Converter 2-25
Altera HDL to VHDL/Verilog Converter 2-25

Chapter 3 Design Methodologies - Schematic Flow

Schematic Flow Processing Overview 3-1
Top-Level Designs 3-3
All-Schematic Designs 3-3
Creating the Schematic and Generating a Netlist 3-3
Performing Functional Simulation 3-4
Implementing the Design 3-5
Creating a New Revision 3-7
Creating a New Version 3-8
Editing Implementation Constraints 3-8
Verifying the Design 3-11
Performing a Static Timing Analysis (Optional) 3-11
Performing a Timing Simulation 3-11
Programming the Device 3-12
Schematic Designs with Instantiated HDL-Based Macros 3-13
Creating HDL Macros 3-13
Creating the Schematic and Generating a Netlist 3-14
Schematic Designs With Instantiated LogiBLOX Modules 3-15
Creating LogiBLOX Modules 3-15
Importing Existing LogiBLOX Modules 3-15
Schematic Designs With Instantiated CORE Generator Cores 3-16
Creating Core Symbols 3-16
Schematic Designs With Finite State Machine (FSM) Macros 3-18
Creating FSM Macros 3-18
Creating the Schematic and Generating a Netlist 3-19
Finite State Machine (FSM) Designs 3-20
Creating a State Editor Design 3-20

Defining States.. 3-21
Defining Transitions, Conditions, and Actions 3-22
Adding a Top-Level ABEL Design to the Project 3-22

Chapter 4 Schematic Design Entry

Managing Schematic Designs.. 4-1
Design Structure 4-2
Single Sheet Schematic... 4-2
Multi-sheet Flat Schematic... 4-3
Hierarchical Schematic 4-3
Adding New Sheets to the Project 4-5
Adding Existing Sheets to the Project...................................... 4-6
Opening Non-project Sheets.. 4-6
Removing Sheets from the Project 4-6
Renumbering Symbol References... 4-7
Copying a Section of a Schematic to Another Sheet................ 4-8
Troubleshooting Project Contents.. 4-8
Hierarchical Schematic Designs 4-8
Creating a Schematic Macro (Bottom-Up Methodology) 4-9
Recognizing Hierarchical Macros 4-10
Navigating the Project Hierarchy 4-10
Modifying Existing Macros 4-12
Difference between a Macro and a Schematic 4-13
Hierarchy Symbol Changes 4-13
Using a Top-down Methodology 4-13
Hierarchical Design Example... 4-14
Manually Exporting a Netlist... 4-18
Creating a Schematic from a Netlist... 4-19
Miscellaneous Tips for Using the Schematic Editor Tool 4-20
Color-coded Symbols.. 4-20
Using the Hierarchy Connector... 4-20
Using Input and Output Buffers... 4-21
Schematic Tabs 4-21
Simulate Current Macro 4-22

Chapter 5 Design Methodologies - HDL Flow

HDL Flow Processing Overview.. 5-1
Top-level Designs 5-3
All-HDL Designs.. 5-3
Creating the Design 5-3
Analyzing Design File Syntax 5-4

Performing HDL Behavioral Simulation (Optional).................... 5-5
Synthesizing the Design 5-5
Express Constraints Editor 5-8
Express Time Tracker... 5-10
Performing Functional Simulation 5-12
Implementing the Design 5-15
Editing Implementation Constraints 5-17
Verifying the Design.. 5-20
Performing a Static Timing Analysis 5-20
Performing a Timing Simulation... 5-20
Programming the Device 5-21
HDL Designs with State Machines... 5-21
Creating a State Machine Macro 5-21
HDL Designs with Instantiated Xilinx Unified Library Components 5-24
HDL Designs with Black Box Instantiation 5-25
LogiBLOX Modules in a VHDL or Verilog Design 5-26
VHDL Instantiation... 5-26
Verilog Instantiation 5-31
CORE Generator COREs in a VHDL or Verilog Design 5-36
VHDL Instantiation... 5-36
Verilog Instantiation 5-42
XNF file in a VHDL or Verilog Design 5-48
Schematic Designs in the HDL Flow.. 5-49
Adding a Schematic Library 5-49
Creating HDL Macros 5-50
Creating the Schematic and Generating a Netlist..................... 5-51
Selecting a Netlist Format... 5-52
Completing the design 5-52

Chapter 6 HDL Design Entry and Synthesis

HDL File Selection 6-1
Adding the File to the Project.. 6-3
Removing Files from the Project... 6-3
Getting Help with the Language.. 6-3
Synthesis of HDL Modules... 6-5
Schematic Flow Methodology 6-5
HDL Flow Methodology... 6-7
Managing Large Designs 6-9
Design Optimization.. 6-9
Setting Constraints Prior to Synthesis 6-10
Design Partitioning Guidelines... 6-10
User Libraries for HDL Flow Projects... 6-11

Creating a New Library 6-12
Declaring and Using User Libraries 6-12
Using Constraints in an HDL Design... 6-12
Express Constraints Editor 6-12
Xilinx Logical Constraints.. 6-14
Reading Instance Names from an XNF file for UCF Constraints 6-15
Instance Names for LogiBLOX RAM/ROM 6-16
Calculating Primitives for a LogiBLOX RAM/ROM Module.. 6-16
Naming Primitives in LogiBLOX RAM/ROM Modules.......... 6-16
Referencing LogiBLOX Entities 6-17

Chapter 7 State Machine Designs

State Machine Example 7-2
State Diagram 7-2
State Machine Implementation.. 7-3
Encoding Techniques... 7-4
Symbolic and Encoded State Machines 7-4
Compromises in State Machine Encoding 7-5
Binary Encoding.. 7-5
One-Hot Encoding 7-6
One-Hot Encoding in Xilinx FPGA Architecture.................. 7-6
Limitations... 7-6
Encoding for CPLDs 7-7

Chapter 8 LogiBLOX

Setting Up LogiBLOX on a PC 8-2
Starting LogiBLOX 8-2
Creating LogiBLOX Modules.. 8-4
LogiBLOX Modules 8-5
Using LogiBLOX for Schematic Designs....................................... 8-5
Using LogiBLOX for HDL Designs 8-6
Module-inferring Tools 8-6
Module-instantiation Tools 8-6
Documentation 8-6

Chapter 9 CORE Generator System

Setting Up the CORE Generator System on a PC......................... 9-1
Accessing the CORE Generator System 9-2
Instantiating CORE Generator Modules.. 9-6
Documentation 9-6

Chapter 10 Functional Simulation

Basic Functional Simulation Process 10-1
Invoking the Simulator 10-1
Attaching Probes (Schematic Editor Only)................................ 10-2
Adding Signals 10-2
Creating Buses 10-3
Applying Stimulus 10-3
Stimulator Selection Dialog.. 10-3
Waveform Test Vectors 10-4
Script File Macro.. 10-4
Running Simulation.. 10-4
HDL Top-down Methodology 10-5
HDL with Underlying Netlists... 10-6
Simulation Script Editor... 10-7
Waveform Editing Functions 10-7

Chapter 11 Design Implementation

Versions and Revisions.. 11-1
Schematic Flow Projects.. 11-1
Creating Versions 11-2
Creating Revisions.. 11-2
HDL Flow Projects 11-3
Creating Versions 11-3
Updating Versions 11-4
Creating Revisions.. 11-4
Creating a new Revision 11-7
Creating the First Version and Revision in One Step 11-8
Revision Control... 11-9
Implementing a Design 11-9
Setting Control Files.. 11-12
User Constraints File 11-12
Guide Files... 11-13
Guiding FPGA Designs... 11-13
Guiding CPLD Designs... 11-14
Setting Guide Files 11-14
Floorplan Files 11-16
Selecting Options.. 11-18
Place & Route Effort Level... 11-19
Program Options.. 11-19
Implementation Templates 11-20
Simulation Templates 11-20
Configuration Templates (FPGAs)...................................... 11-20
Template Manager... 11-21

Flow Engine 11-22
Translate 11-24
MAP (FPGAs) 11-24
Place and Route (FPGAs) 11-24
CPLD Fitter 11-25
Configure (FPGAs) 11-26
Bitstream (CPLDs) 11-26
Implementation Reports 11-26
Translation Report 11-28
Map Report (FPGAs) 11-28
Place and Route Report (FPGAs) 11-28
Pad Report (FPGAs) 11-29
Fitting Report (CPLDs) 11-29
Post Layout Timing Report 11-29
Additional Implementation Tools 11-29
Constraints Editor 11-29
Flow Engine Controls 11-30
Controlling Flow Engine Steps 11-30
Running Re-Entrant Routing on FPGAs 11-31
Configuring the Flow 11-33
Floorplanner 11-34
FPGA Editor 11-35
CPLD ChipViewer 11-35
Locking Device Pins 11-35

Chapter 12 Verification and Programming

Overview 12-1
Timing Simulation 12-2
Generating a Timing-annotated Netlist 12-3
Basic Timing Simulation Process 12-3
Timing Analyzer 12-4
Post Implementation Static Timing Analysis 12-5
Summary Timing Reports 12-5
Detailed Timing Analysis 12-6
In-Circuit Verification 12-7
Downloading a Design 12-7
JTAG Programmer 12-8
Hardware Debugger (FPGAs only) 12-9
PROM File Formatter 12-9

Appendix A Glossary

ABEL A-1

actions A-1
Aldec A-1
aliases A-1
analyze A-2
architecture A-2
attribute A-2
binary encoding A-2
BitGen A-2
Black Box Instantiation A-2
block A-2
breakpoint A-3
buffer A-3
bus A-3
CLB A-3
component A-3
condition A-3
constraint A-3
constraints editor A-4
constraints file A-4
CORE Generator A-4
CPLD A-4
CPLD fitter A-5
design entry tools A-5
design implementation tools A-5
Design Manager A-5
effort level A-6
elaborate A-6
Express Compiler A-6
Express Constraints Editor A-6
Express Time Tracker A-6
Finite State Machine Editor A-7
fitter A-7
floorplanning A-7
FPGA A-7
FPGA Editor A-7
FSM A-7
functional simulation A-8
guided design A-8
guided mapping A-8
HDL A-8
HDL Editor A-8
HDL Flow A-8

hierarchical designs A-9
Hierarchy Browser A-9
implementation A-9
Implementation Constraints Editor A-9
instantiation A-9
Language Assistant A-9
Library Manager A-9
locking A-10
LogiBLOX A-10
logic A-10
Logic Simulator A-10
macro A-10
MAP A-11
mapping A-11
MRP file A-11
NCD file A-11
net A-11
netlist A-12
NGA file A-12
NGDAnno A-12
NGDBuild A-12
NGD file A-12
NGM file A-12
one-hot encoding A-13
optimization A-13
optimize A-13
PAR (Place and Route) A-13
path delay A-13
PCF file A-13
PDF file A-14
physical Design Rule Check (DRC) A-14
physical macro A-14
pin A-14
pinwires A-14
project A-14
Project Flowchart A-15
Project Manager A-15
PROM File Formatter A-15
route A-15
route-through A-15
Schematic Editor A-15
Schematic Flow A-15

state diagram A-16
state machine A-16
state machine designs A-16
states A-16
static timing analysis A-16
static timing analyzer A-16
status bar A-17
stimulus information A-17
Symbol Editor A-17
Synopsys A-17
synthesis A-17
Time Tracker A-17
transitions A-17
TRCE A-18
TWR file A-18
UCF file A-18
verification A-18
Verilog A-18
VHDL A-18
Wire A-19
Xilinx Constraints Editor A-19

Appendix B Foundation Constraints

Constraint Entry Mechanisms B-2
Translating and Merging Logical Designs B-3
The Xilinx Constraints Editor B-4
Constraints File Overview B-4
- Netlist Constraints File (NCF) B-4
- User Constraints File (UCF) B-4
- Physical Constraints File (PCF) B-5
- Case Sensitivity B-6

Timing Constraints B-6
- The “From:To” Style Timespec B-6
- Using TPSYNC B-8
- The Period Style Timespec B-10
- The Offset Constraint B-11
- Ignoring Paths B-13
- Controlling Skew B-14
- Constraint Precedence B-14
  - Across Constraint Sources B-14
  - Within Constraint Sources B-15

Layout Constraints B-16

Converting a Logical Design to a Physical Design B-16
"Last One Wins" Resolution B-17
XC5200XL Constraints B-17
Efficient Use of Timespecs and Layout Constraints....................... B-18
The "Starter Set" of Timing Constraints B-18
Standard Block Delay Symbols B-21
Table of Supported Constraints B-22
Basic UCF Syntax Examples B-25
PERIOD Timespec.. B-25
FROM:TO Timespecs B-25
OFFSET Timespec B-26
Timing Ignore B-26
Path Exceptions B-27
Miscellaneous Examples B-28
User Constraint File Example B-29
Constraining LogiBLOX RAM/ROM with Synopsys B-33
Estimating the Number of Primitives Used B-33
How the RAM Primitives are Named B-33
Referencing a LogiBLOX Module/Component in the HDL Flow B-34
Referencing the Primitives of a LogiBLOX Module in the HDL Flow B-35
HDL Flow Verilog Example B-35
test.v: B-35
inside.v:.. B-36
test.ucf B-36
HDL Flow VHDL Example.. B-36
test.vhd B-36
inside.vhd... B-37
test.ucf B-38

Appendix C Instantiated Components

Library/Architecture Definitions C-2
XC3000 Library C-2
XC4000E Library.. C-2
XC4000X Library.. C-2
XC5200 Library C-2
XC9000 Library C-2
Spartan Library C-2
SpartanXL Library C-3
Virtex Library... C-3
STARTUP Component.. C-3
STARTBUF Component... C-3

BSCAN Component C-4
READBACK Component.. C-5
RAM and ROM.. C-6
Global Buffers C-8
Fast Output Primitives (XC4000X only) C-10
IOB Components.. C-11
Clock Delay Components.. C-13

Appendix D File Processing Overview

FPGAs.. D-1
CPLDs.. D-4

Chapter 2

Project Toolset

This chapter explains how to create Foundation projects and how to access the various Foundation tools that you use to complete the project. Each tool and its function is briefly described. This chapter contains the following sections.

- "Creating Foundation 2.1i Projects"
- "Project Manager"
- "Accessing LogiBLOX"
- "Accessing the CORE Generator System"
- "Documenting Your Design"
- "Project Archiving"
- "Design Entry Tools"
- "Synthesis Tools"
- "Simulation/Verification"
- "Constraints Editors"
- "Implementation Tools"
- "Device Programming"
- "Utilities"

Creating Foundation 2.1i Projects

To organize your work, Foundation groups all related files into separate logical units called projects. Schematic, HDL, and Finite State Machine (FSM) designs must be defined as elements in a project. The associated libraries as well as netlists, bitstream files, reports, and configuration files are all part of the project.

Each project is stored in a separate directory called the project working directory. The location of the project working directory is specified when the project is created. The name of the project working directory is the same as the name of the project.

A Foundation Series 2.1i project can be either a Schematic Flow project or an HDL Flow project. If you are using the Base (DS-FND-BAS-PC) or Standard (DS-FND-STD-PC) products, only the Schematic Flow is available to you. Both flows are available to Base Express (DS-FND-BSX-PC) and Foundation Express (DS-FND-EXP-PC) users.

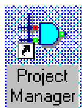
Schematic Flow Projects

A Schematic Flow project can have top-level schematic or ABEL files. Top-level schematic designs can contain underlying schematic, LogiBLOX, CORE Generator, or ABEL macros. The top-level ABEL files or underlying ABEL macros can be created with the Finite State Machine (FSM) Editor or a text editor. (Top-level ABEL files are not recommended for FPGA projects.)

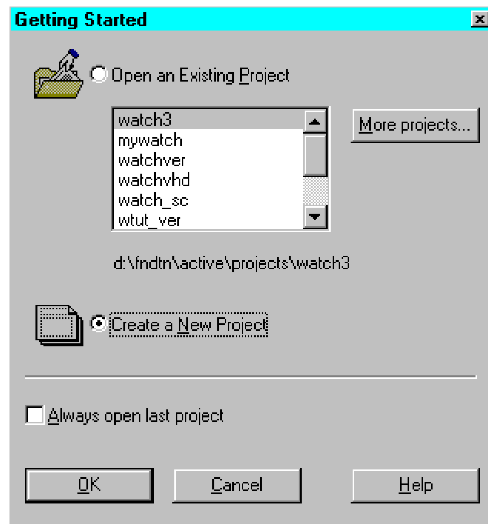
If you have Base Express or Foundation Express, a Schematic Flow project can also have underlying HDL, VHDL, or ABEL macros created with the HDL Editor, FSM Editor, or another text editor.

To create a Schematic Flow project, perform the following steps.

1. Open the Project Manager by clicking on the Project Manager icon (shown below) on your desktop or by **Start→ Programs→ Xilinx Foundation Series 2.1i→ Project Manager**.

2. Click the **Create a New Project** radio button on the Getting Started dialog box. Click **OK**. (To create new projects, you can also select **File→ New Project** from the Project Manager.)



3. Enter the project name, up to 8 characters, in the Name field of the New Project dialog box.
4. Select a location for the project in the Directory box.
5. Select **`F2.1i`** as the project type in the Type box.
6. Select the **`Schematic`** Flow.
7. Enter the device family, part, and speed of your target device.

8. Click **`OK`**.

The Project Manager screen for the new project appears (see the “Project Manager - Schematic Flow” figure). The Project Manager screen contains three main sections.

- On the left side is the Hierarchy Browser consisting of a hierarchy tree of the project files on the Files tab and of the project implementation versions on the Versions tab.
- The upper right area includes the Flow tab showing the design flowchart with the functions used for Schematic Flow projects. This section also contains Contents, Reports, and Synthesis tabs. If you create any lower-level HDL or FSM macros for the project, you can use functions on the Synthesis tab to list and update them. From the Contents tab, you can view information on the Files shown in the Hierarchy Browser area. You can access system-created reports from the Reports tab.
- The bottom console area displays errors, warnings, and messages.

Refer to the "Project Manager" section later in this chapter for more information on the Project Manager and the tools accessed from it.

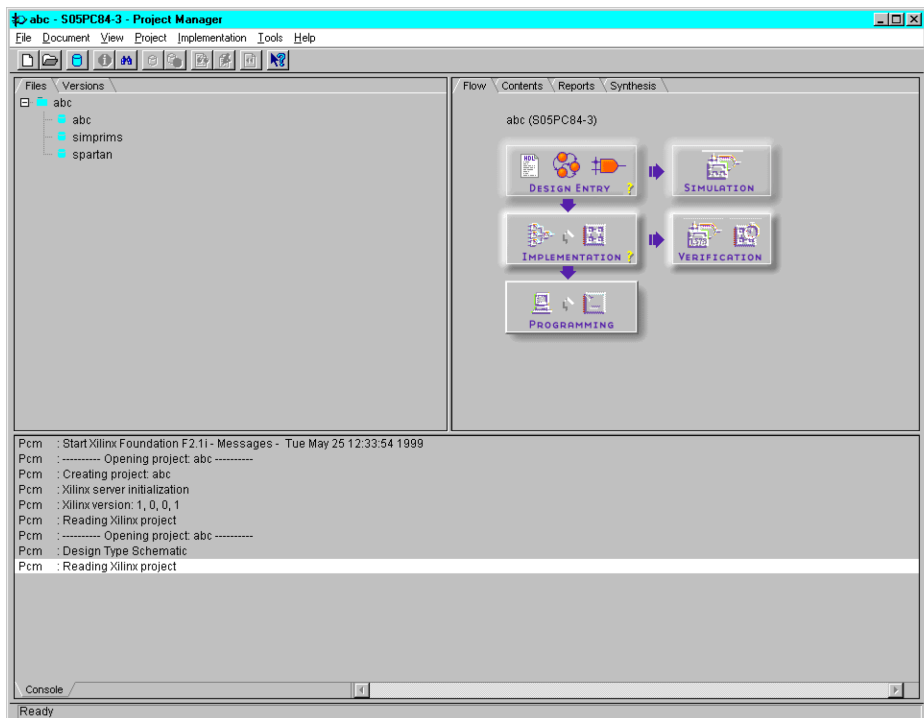


Figure 2-1 Project Manager - Schematic Flow

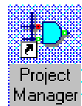
HDL Flow Projects (Express Only)

An HDL Flow project can contain VHDL, Verilog, or schematic top-level designs with underlying VHDL, Verilog, or schematic modules. HDL files can be created by the HDL Editor, Finite State Machine Editor, or other text editors.

LogiBLOX, CORE Generator, and ABEL modules as well as XNF files can be instantiated in the VHDL and Verilog code using the “black box instantiation” method.

To create an HDL Flow project, perform the following steps.

1. Open the Project Manager by clicking on the Project Manager icon (shown below) or by **Start** → **Programs**→ **Xilinx Foundation Series 2.1i**→ **Project Manager**.



2. Click the **Create a New Project** radio button on the Getting Started dialog box. Click **OK**. (To create new projects, you can also select **File**→ **New Project** from the Project Manager.)

3. Enter the project name in the Name box of the New Project dialog.

4. Select a location for the project in the Directory box.
5. Select **F2.1i** as the project type in the Type box.
6. Select the **HDL** Flow.

Note: When you select the HDL Flow button, the device family, part, and speed boxes for the target device are removed. You do not need to select a target device for HDL Flow projects until the design is synthesized.

7. Click **OK**.

The Project Manager screen for the new project appears. The Project Manager screen contains three sections.

- On the left side is the Hierarchy Browser consisting of a hierarchy tree of the project files on the Files tab and of the project versions on the Versions tab.
- The upper right area includes the Flow tab showing the design flowchart with the functions used for HDL Flow projects. This section also contains Contents and Reports tabs. From the Contents tab, you can view information on the Files and Versions shown in the Hierarchy Browser area. You can access system-created reports from the Reports tab.
- The bottom Console tab displays errors, warnings, and messages. The HDL Errors, HDL Warnings, and HDL Messages tabs display information about synthesis results when a specific version of the project is selected.

Refer to the “Project Manager” section later in this chapter for more information on the Project Manager and the tools accessed from it.

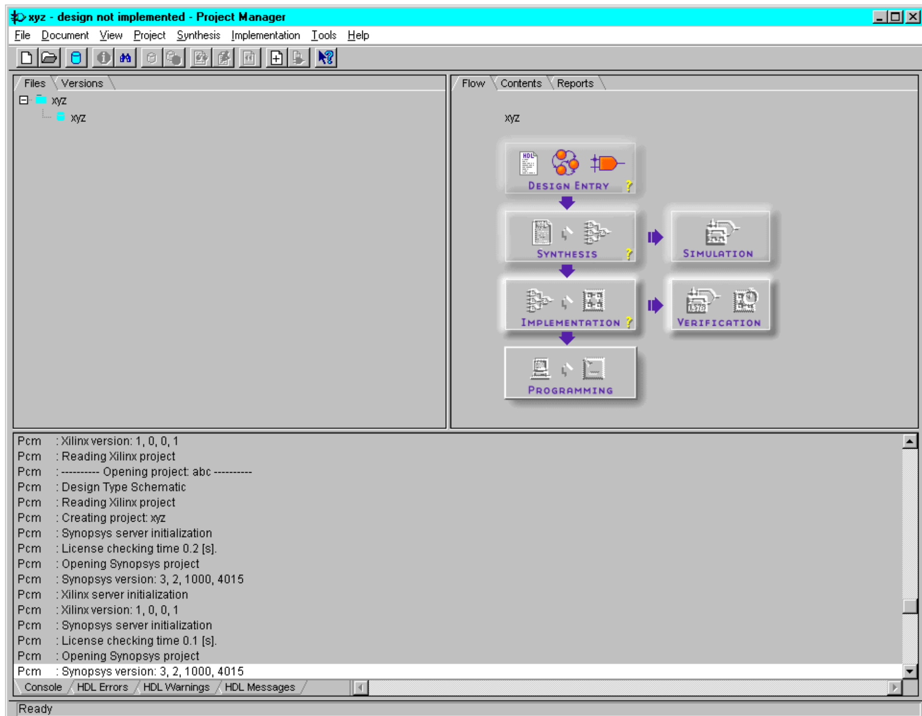


Figure 2-2 Project Manager - HDL Flow

Project Manager

The Project Manager, the overall project management tool, contains the Foundation Series tools used in the design process. The“Project Manager - Schematic Flow” figure and the“Project Manager - HDL Flow” figure illustrate the tools accessible for the two Foundation 2.1i project flow types. It is through the Project Manager that you access the tools for the design process from design entry tools to device programming.

The Project Manager performs the following functions:

- Automatically loads all design resources when opening a project
- Checks that all project resources are available and up-to-date
- Illustrates the design process flow

- Initiates applications used in the design process
- Displays error and status messages in the message window
- Provides automated data transfer between various Foundation design tools
- Displays design status information

The three main regions of the Project Manager are discussed in the following sections

Hierarchy Browser

Foundation organizes related files into a distinct logical unit called a *project*. Related files include the following:

- Project documents (schematics, HDL source files, and state diagram files)
- Project libraries
- Output and intermediate files (netlists, bitstreams, report and log files)
- Configuration files

Two tabs in the Hierarchy Browser area on the Project Manager window keep track of these files. The Hierarchy Browser is an interactive area in addition to a display area. You can open the listed files and versions/revisions by double clicking on them in the Hierarchy Browser—the application that is associated with the file type is invoked. For example, if you double click on a schematic file, the Schematic Editor displays the schematic file. You can also access menus listing the functions you can perform on the displayed items by right clicking on the item.

The Hierarchy Browser's Files and Version tabs are summarized in the following sections. To learn more about how to use the hierarchy browser, select **`Help`** → **`Foundation Help Contents`** → **`Project Manager`** → **`Hierarchy Browser`**.

Files Tab

The Files tab displays the hierarchy of the project files, project libraries, and external files. From this tab you can add, remove, or reorder the displayed files and libraries as well as open applications associated with them.

For new projects, the Project Manager automatically creates the following files:

- A configuration file called the Project Description File (PDF). The PDF file has the same name as the project plus the .pdf extension. The PDF file is stored at the top-level of the associated project directory.
- Three types of library files (project library, Simprims library, and device library). In HDL Flow projects, the Simprims library and device library are not added until the device is selected in the Synthesis phase.

A Foundation project always has one or more "top-level" design file(s). In a Schematic Flow project, you can see what the top-level designs in the project are by looking at the top level of the Hierarchy Browser. In a Schematic Flow project, all top-level files must be schematics, FSM (ABEL) diagrams, or ABEL files. In an HDL Flow project, you designate the top-level entity or module at the time of synthesis. The list of entities/modules is automatically generated from the list of HDL source files that have been added to the project. The added HDL design files are displayed in the File tab of the Hierarchy Browser and can be VHDL, Verilog, or schematic files.

The following table shows some the of common project files included in the Hierarchy Browser, their extensions, and the Foundation tool that creates them.

Extension	File Type	Created By
.pdf	Project description file	Project Manager
.sch	Schematic source file	Schematic Capture

Extension	File Type	Created By
.v	Verilog source file	HDL Editor
.vhd	VHDL source file	HDL Editor
.abl	ABEL source file	HDL Editor
.asf	Finite State Machine source file	FSM Editor
.ucf	User constraints file	Constraints Editor
.tve	Test vector file	Logic Simulator

For detailed information about the project files, libraries, and other project information, refer to the online help by selecting **`Help`** → **`Foundation Help Contents`**→ **`Foundation Configuration Information`**.

Versions Tab

The Versions tab displays the revisions and versions of the chip implementations of the design. For a newly created project, this tab is empty.

Project management consists of control over design versions and revisions. A version represents an input design netlist. Each time a change is made to the source design, such as logic being added to or removed from the schematic or the HDL source being modified, a new version may be created. A revision represents an implementation on a given version, usually with new implementation options such as different placement or router effort level.

Project Flowchart Area

The Foundation 2.1i Project Manager's project flowchart area contains four tabs that allow you to obtain current information about your current project and facilitate the design process.

Flow Tab - Project Flowchart

The Flow tab displays the project flowchart. You use the buttons on the flowchart to perform steps in the design flow, from design entry through device programming. The buttons included in the flowchart in this area depend on whether you have a Schematic Flow project or

an HDL Flow project (see the “Project Manager - Schematic Flow” figure and the “Project Manager - HDL Flow” figure).

When you start programs from the project flowchart, the Project Manager automatically controls the transfer of input and output data (files) between the applications. It performs the necessary steps to take the design to the point you requested.

Alternatives to Flowchart Buttons

In addition to the project flowchart, the Project Manager includes a number of alternative ways to run the Foundation application tools. You can access tools by right-clicking items listed in the Hierarchy Browser area. Or, you can use the Tools menus in the Project Manager Toolbar to access submenus for Design Entry, Simulation/Verification, Implementation, and Device Programming tools. It is also possible to start the Foundation applications directly from the Windows environment. The latter method is not recommended because, depending on the application, the Project Manager may not be started and would not be available to track the project properly.

Contents Tab

The Contents tab displays info related to the object currently selected (file, library, etc.) from the hierarchy tree on the Files tab. It displays the full pathname of the object selected as well as the date the object was last modified.

Reports Tab

Select this tab to access and display reports that have been generated in the design process.

Synthesis Tab (Schematic Flow Only)

Using the Synthesis tab, you can update or synthesize VHDL, Verilog, ABEL, and State Machine macros. Refer to the “Synthesis Tools” section later is this chapter for more information on this tab. (This tab is unnecessary in an HDL Flow project because the entire project is synthesized.)

Messages Area

The tabs included in the Messages area display general project messages and specific HDL processing messages.

Console Tab

The Console tab displays the contents of the project log.

HDL Errors Tab (HDL Flow Only)

This tab displays any errors encountered during HDL source file analysis, for the object selected in the Hierarchy Browser.

HDL Warnings Tab (HDL Flow Only)

This tab displays warnings generated during HDL source file processing, for the object selected in the Hierarchy Browser.

HDL Messages Tab (HDL Flow Only)

This tab displays messages other than errors or warnings generated during HDL source file processing, for the object selected in the Hierarchy Browser.

Accessing LogiBLOX

LogiBLOX is a graphical interactive tool for creating high-level modules, such as counters, shift registers, and multiplexers. LogiBLOX includes both a library of generic modules and a set of tools for customizing them. You can access LogiBLOX from the Project Manager by selecting **Tools** → **Design Entry** → **LogiBLOX module generator**, from the Schematic Editor by selecting **Tools** → **LogiBLOX module generator** or from the HDL Editor by selecting **Tools** → **LogiBLOX**. For details about creating LogiBLOX modules, refer to the “Creating LogiBLOX Modules” section of the “LogiBLOX” chapter.

Note: LogiBLOX supports all Xilinx architectures except Virtex.

Accessing the CORE Generator System

The Xilinx CORE Generator is a graphical interactive tool that generates and delivers parameterizable cores optimized for Xilinx FPGAs. You can access the CORE Generator system from the Project Manager by selecting **Tools** → **Design Entry** → **CORE Generator** or from the Schematic Editor or HDL Editor by selecting **Tools** → **CORE Generator**. For more information on the CORE Generator system, refer to the CORE Generator online help.

Documenting Your Design

To attach text files or other files to the Project, perform the following steps.

1. Select **Document** → **Add**.
2. In the Add Document dialog box, select the documents from the Files list box.
3. Click **OK**.

The files are then displayed in the Hierarchy Browser area. This is a convenient way to provide documentation for your design. Note that you can add almost any kind of file to the project.

Project Archiving

Foundation 2.1i supports automatic project archiving. Any or all of the following project components: project files, design source files, synthesis files, implementation files, or documentation files can be zipped into a single file or into multiple files. When you select **File** → **Archive Project** from the Project Manager, the Archive Project Wizard - Setup window appears. In this window, you can specify the location for the archive .zip file, add comments, provide a password, or modify the compression factor. A second window, the Project Components window, allows you to select the parts of the project to be archived. Likewise, the Foundation Project Manager contains a Restore Project option to automatically unzip archived projects. (**File** → **Restore Project**).

Project archiving maintains revision control. The resultant files from each implementation revision are archived in the project directory. The source design for each version is *not* archived, only the resulting

netlists and files for each revision. Therefore, if you want to save iterations of the source design (schematics, HDL files, for example), you must back those up yourself.

Foundation 2.1i also supports archiving of symbol libraries as well as any other user files (release notes, application notes, etc.) you want to save. To archive symbol libraries or other user files, perform the following steps:

1. Select **File** → **Archive Project**.
2. Select **Next** from the Archive Project Wizard - Setup window.
3. Select **Next** from the Project Components window to display the User Files window.
4. Select **Add Libraries** and then select the libraries from the list box that you want to archive. Or, select Add Files to select any additional files to archive.
5. Select **Start** to begin archiving.

Design Entry Tools

This section describes the design entry tools. Foundation includes a suite of tools for creating digital circuit designs. These tools provide the following design entry capabilities.

- Top-level schematic entry with the Xilinx Unified libraries components, LogiBLOX symbols, CORE Generator modules, HDL macros, and State Machine macros
- Top-level HDL design entry and synthesis
- Top-level HDL designs with state machine, CORE Generator, or LogiBLOX instantiated components
- Finite state machine diagram entry

Schematic Editor

With the Schematic Editor, you can create multi-sheet hierarchical schematics. The editor features include the following.

- Multiple sheet and hierarchical schematic support
- Viewlogic schematic import

- Board-level and PLD schematic support (requires the Active-CAD tool)
- Export of schematic netlists to XNF, EDIF, VHDL, and Verilog formats
- Integration with synthesis design tools (HDL Editor and State Diagram editor)
- Integration with the Logic Simulator

For detailed information about the Schematic Editor, select **`Help`** → **`Foundation Help Contents`** → **`Schematic Editor`**. Also, see the “Schematic Design Entry” chapter.

State Editor

State machine designs typically start with the translation of a concept into a “paper design,” usually in the form of a state diagram or a bubble diagram. The paper design is converted to a state table and finally into the source code itself. The State Editor, which allows you to create state machine designs, also supports the following functions:

- Generates behavioral VHDL, Verilog, or ABEL (Schematic Flow only) code from the state diagram
- Invokes the Express or XABEL compiler to convert the behavioral description into a gate-level netlist
- Simulates a state diagram macro graphically

For more information about how to use the State Editor, select **`Help`** → **`Foundation Help Contents`** → **`State Editor`**.

HDL Editor

The HDL Editor, a text editor, is designed to edit HDL source files created in the VHDL, Verilog, or ABEL (Schematic Flow only) languages. The HDL Editor utilizes syntax coloring for the VHDL, Verilog, and ABEL languages. The HDL Editor allows you to check HDL language syntax as well as create HDL macro symbols for placement on a schematic.

The Language Assistant tool (**`Tools`** → **`Language Assistant`** in the HDL Editor) furnishes the following templates with source code for VHDL, Verilog, and ABEL.

- Language templates with basic language constructs
- Synthesis templates of functional blocks such as counters, flip-flops, multiplexers, and Xilinx architectural features such as Boundary Scan and RAM

For detailed information about the HDL Editor, select **Help** → **Foundation Help Contents**→ **HDL Editor**. Also, refer to the "HDL Design Entry and Synthesis" chapter.

Symbol Editor

With the Symbol Editor, you can edit features of component symbols such as pin locations, pin names, pin numbers, pin shape, and pin descriptions.

From the Project Manager, you can access the Symbol Editor by selecting **Tools** → **Design Entry** → **Symbol Editor**.

For more details on how to use the Symbol Editor, select **Help** → **Foundation Help Contents** → **Advanced Tools** → **Symbol Editor**.

Synthesis Tools

Synthesis tools are available for both HDL Flow projects and Schematic Flow projects. If you are using the Base or Standard product, synthesis tools are available for Finite State Machine ABEL macros only.

Synthesis Button (HDL Flow)

For design synthesis, Base Express and Foundation Express users have access to FPGA Express from Synopsys, the industry-leading synthesis technology. The Express synthesis tools provide the following capabilities.

- Architecture-specific optimization
- Verilog, VHDL, or mixed HDL synthesis
- Automatic Finite State Machine extraction
- Automatic GSR and I/O insertion

- Graphical constraints editor. The Express Constraints Editor GUI is available to Foundation Express users only. It is used to set design constraints and view estimated design performance.

Synthesis Tab (Schematic Flow)

In a Schematic Flow project, the necessary synthesis of any underlying HDL macros in the design can be initiated in the various design entry tools.The Synthesis tab provides the capability to synthesize any or all of the HDL macros (FSM, ABEL, VHDL, or Verilog) in the current project and update the macro symbol and netlist without searching manually through the project and synthesizing/updating them individually.

Simulation/Verification

Simulation and verification tools are available for both Schematic and HDL Flow projects to determine if the timing requirements and functionality of your design have been met.

Logic Simulator

The Logic Simulator is a real-time interactive design tool for both functional and timing simulation of designs. You access the Logic Simulator from the project flowchart when you click the **`Simulation`** button or the Timing Simulation icon on the **`Verification`** button.

The Logic Simulator creates an electronic breadboard of your design directly from your design's netlist. The breadboard is tested with signals called test vectors. Each test vector lists logical states of all stimulus signals at a selected time interval. See the "Functional Simulation" chapter and the "Verification and Programming" chapter for more information on simulations. For details on how to use the Logic Simulator, select **`Help`** → **`Foundation Help Contents`**→ **`Logic Simulator`**.

Timing Analyzer

Select the Timing Analyzer icon on the **`Verification`** button on the project flowchart to access the Timing Analyzer for verification based on the post-layout timing netlist. The Timing Analyzer is used to verify that the delay along a given path or paths meets your specified

timing requirements. It creates timing analysis reports that you customize by applying filters. It organizes and displays data that allows you to analyze the critical paths in your circuit, the cycle time of the circuit, the delay along any specified paths, and the paths with the greatest delay. It also provide a quick analysis of the effect of different speed grades on the same design.

Specialized Simulation Controls

Typically, the Simulation and Verification functions are invoked from the project flowchart buttons. You can access the following individual functions from the Project Manager toolbar, if needed.

- Gate Simulator

  When you select **`Tools`**→ **`Simulation/Verification`**→ **`Gate Simulator`** from the Project Manager toolbar, you access three startup options for the simulator.

  - Opening the simulator with the netlist from the currently open Foundation project
  - Selecting the netlist manually
  - Opening the simulator without loading a netlist

- Checkpoint Gate Simulation Control

  Checkpoint simulation pulls simulation data from the current stage of the design database. If you want to select which netlist (hierarchical or flat NGA netlist) to use for timing simulation, you can access the Checkpoint Gate Simulation Control dialog by selecting **`Tools`** → **`Simulation/Verification`** → **`Check-point Gate Simulation Control`** on the Project Manager toolbar.

HDL Behavioral Simulation Capabilities

Foundation Series 2.1i allows you to add HDL behavioral simulation capabilities to all design flows. HDL simulators from Aldec, Incorporated, and from MTI can be added to your Xilinx software. Sale and support for Aldec's ACTIVE-VHDL Behavioral Simulator and for MTI's ModelSim product are handled directly by those vendors.

Constraints Editors

Two Constraints Editor GUIs are available in Foundation to assist with constraining elements of your design to obtain the desired performance.

Express Constraints Editor (HDL Flow)

The Express Constraints Editor is a feature available in the Foundation Express product only. The Express Constraints Editor is a GUI that allows you to set performance constraints, attributes, and optimization controls in the Synthesis phase before you start to optimize a design. Constraint entry is in the form of constraints tables for logically related groups (clocks, ports, paths, modules). Design-specific information based on the architecture specified for the selected version of the design is automatically extracted and displayed in the tables.

Xilinx Constraints Editor

The Xilinx Constraints Editor GUI allows you to create and edit certain constraints after the translation step in the Implementation phase of the design without directly editing the UCF (User Constraint File).

You can start the Constraints Editor from the Project Manager by selecting **`Tools`** → **`Implementation`** → **`Constraints Editor`**.

You can also invoke the Xilinx Constraints Editor by selecting **`Start`** → **`Programs`** → **`Xilinx Foundation Series 2.1i`** → **`Accessories`** → **`Constraints Editor`**.

The Xilinx Constraints Editor is not the same as the Express Constraints Editor available in the HDL Flow and is most useful for schematic and ABEL designs in Schematic Flow projects.

For more on the Constraints Editor, refer to the *Constraints Editor Guide*, an online book.

Implementation Tools

Once you have completed design entry and are ready for physical implementation of the design, you begin implementation processing by clicking the **Implementation** button on the project flowchart. All the steps needed to obtain the final results are invoked automatically. Refer to the "Design Implementation" chapter for more information.

Control Files

You can control the implementation of your design with a user constraints file, an implementation guide file, or a Floorplanner file. You can set these files by selecting **Implementation** → **Set Guide File(s)**, or **Set Floorplan File(s)**, or **Set Constraints File(s)** from the Project Manager. Or, you can access a dialog box to set the files by clicking the Control Files **SET** button in the Physical Implementation Settings section of the window that appears when you implement a new version or revision of your design.

User Constraints File

Constraints can be applied to control the implementation of a design. Location constraints, for example, can be used to control the mapping and positioning of logic elements in the target device. Timing constraints can be used to identify critical paths that need closer placement and faster routing. For a list of the constraints that can be applied for the various devices, refer to the "Attributes, Constraints, and Carry Logic" chapter of the *Libraries Guide*.

The User Constraints File (UCF) is a user-created ASCII file that holds the constraints. You can enter the constraints directly in the input design. However, putting them in the UCF separates them from the input design files and provides for easier modification and reduces re-synthesis of your design. You can create the UCF using a text editor or you can use the Xilinx Constraints Editor to produce the UCF for you. UCF files can also be reused from design to design.

Implementation Guide File

Guide files from a previous implementation can be used to speed up the current implementation. When an implementation guide file is

specified, only the sections of the current revision that are different from the specified guide file for the previous revision are processed.

Floorplanner File

The Floorplanner tool generates an MFP file that contains mapping and placement information. You can use this file as a guide for mapping an implementation revision for the XC4000, Spartan, and Virtex device families only. For Floorplanner guide files information, refer to the *Floorplanner Guide*, an online manual.

Implementation Tools Menu

Typically, designs are implemented by using the Implementation button on the project flowchart. However, you can access certain specialized functions from the Project Manager Tools menu.

Constraints Editor

The Constraints Editor accessed from the Project Manager by selecting **`Tools`** → **`Implementation`** → **`Constraints Editor`** is the Xilinx Constraints Editor. It becomes available for design implementation after the translation step in Flow Engine has completed. For more on the Constraints Editor, refer to the *Constraints Editor Guide*, an online book.

Flow Engine

The Flow Engine processes the design, controls the implementation of the design, and guides the implementation revisions. When initiated by selecting **`Tools`** → **`Implementation`** → **`Flow Engine`**, the Flow Engine is run as a standalone program. The project is not automatically brought up-to-date as it is when initiated by the Implementation button on the project flowchart. For more information, see the “Implementing a Design” section of the “Design Implementation” chapter.

Floorplanner

Selecting **`Tools`** → **`Implementation`** → **`Floor Planner`** from the Project Manager window, accesses the Floorplanner tool (for FPGAs only).The Floorplanner creates a file that contains mapping information, which can be used by the Flow Engine as a guide for mapping

an FPGA implementation revision. For more information on the Floorplanner, see the *Floorplanner Guide*, an online book.

FPGA Editor

Selecting **Tools** → **Implementation** → **FPGA Editor** from the Project Manager window opens the FPGA Editor. The FPGA Editor provides a graphic view of your placed and routed design, allowing you to make modifications. This option is supported for FPGAs only.

For more information on using the FPGA Editor, see the *FPGA Editor Guide*, an online book.

CPLD ChipViewer

Selecting **Tools** → **Implementation** → **CPLD ChipViewer** from the Project Manager window opens the ChipViewer. The ChipViewer provides a graphical view of the CPLD fitting report. With this tool you can examine inputs and outputs, macrocell details, equations, and pin assignments. You can examine both pre-fitting and post-fitting results.

More information on using the CPLD ChipViewer is available in that tool's online help or from the Umbrella Help menu accessed by **Help** → **Foundation Help Contents** → **Advanced Tools** → **Chip-Viewer**.

Automatic Pin Locking

I/O pins can be locked to a previous revision by clicking on the revision in the Versions tab of the Project Manger and selecting **Tools** → **Implementation** → **Lock Device Pins**. The Lock Pins Status dialog appears upon completion. You can click **View Lock Pins Report** from the Lock Pin Status dialog or select **Tools** → **Implementation** → **View Locked Pins Report** to access the Lock Pins Report. The Lock Pins Report contains information on any constraint conflicts between the pin locking constraints in the existing UCF file and the design file.

Device Programming

When the design meets your requirements, the last step in its processing is programming the target device. To initiate this step, click the `Programming` button in the project flowchart. The Select Programming dialog appears listing one or more of the following device programming tools: JTAG Programmer, Hardware Debugger, PROM File Formatter. For CPLD designs, you must use the JTAG Programmer.

JTAG Programmer

The JTAG Programmer downloads, reads back, and verifies FPGA and CPLD design configuration data. It can also perform functional tests on any device and probe the internal logic states of your design.

PROM File Formatter

The PROM File Formatter is available for FPGA designs only. The PROM File Formatter provides a graphical user interface that allows you to do the following.

- Format BIT files into a PROM file compatible with Xilinx and third-party PROM programmers
- Concatenate multiple bitstreams into a single PROM file for daisy chain applications
- Store several applications in the same PROM file

Hardware Debugger

The Hardware Debugger is a graphical interface that allows you to download an FPGA design to a device, verify the downloaded configuration, and display the internal states of the programmed device.

Utilities

Foundation contains multiple utilities to help you manage and organize your project. Those available from the Project Manager's **Tools** → **Utilities** menu are described below.

Schematic Symbol Library Manager

The Library Manager allows you to perform a variety of operations on the design entry tools libraries and their contents, such as copying macros from one project to another. These libraries contain the primitives and macros that you use to build your design.

The Foundation design entry tools contain two types of libraries: system libraries and user libraries.

- System libraries, which are supplied with the Foundation design entry tools, contain sets of components for each device family as well as for simulation. System library contents cannot be modified. The Foundation system libraries include: simprims, xabelsim, xc3000, xc4000e, xc4000x, xc5200, xc9500, spartan, spartanx, and virtex.
- User libraries contain user-defined components. Each project has at least one user library known as the project working library. The project working library is named the same as the project and is located in the LIB subdirectory of the project directory. The Library Manager automatically places any user-created macro in the current project's working library.

You can access the Library Manager from the Project Manager by selecting **Tools** → **Utilities**→ **Schematic Symbol Library Manager**. Refer to the online help accessed from the Library Manager window for details on how to use the Library Manager. Or, select **Help** → **Foundation Help Contents**→ **Advanced Tools**→ **Symbol Library Manager**.

Command History

Command History (**Tools** → **Utilities** → **Command History**) sequentially lists the processes that have been performed for the selected revision. You can select from two different modes: 1) Process, which displays the name of the process only, and 2) Command Line,

which displays the full command line of each process. An option to display the date and time for each command is also available.

Project Notes

Project Notes (**Tools** → **Utilities** → **Project Notes**) opens a standard text editor of your choice in which you can make notes for the current project. Specify the text editor in the Configuration dialog (**File** → **Preferences** → **Configuration**).

Implementation Template Manager

The Implementation Template Manager can create or modify three types of templates for a selected device: implementation, simulation, and configuration. Implementation templates control how an FPGA design is mapped, optimized, placed, and routed and how a CPLD design is fitted. Simulation templates control the creation of netlists for front- and back-end simulation. Configuration templates control the configuration startup, readback, and parameters for the device.

To access the Template Manager window, select **Tools**→ **Utilities**→ **Implementation Template Manager** from the Project Manager. For details on how to use the Implementation Templates refer to the online help available from the Template Manager window.

ABEL to VHDL/Verilog Converter

The ABEL2HDL utility accessed from **Tools**→ **Utilities**→ **ABEL2HDL** in the Project Manager allows you to select an ABEL (.abl) file and have it converted to a VHDL (.vhd) or Verilog (.v) file.

Altera HDL to VHDL/Verilog Converter

The AHDL2HDL utility accessed from **Tools**→ **Utilities**→ **AHDL2HDL** in the Project Manager allows you to select an Altera HDL (.tdf) file and have it converted to a VHDL (.vhd) or Verilog (.v) file.

Chapter 3

Design Methodologies - Schematic Flow

This chapter describes various design methodologies supported in the Schematic Flow project subtype.

This chapter contains the following sections.

- “Schematic Flow Processing Overview”
- “Top-Level Designs”
- “All-Schematic Designs”
- “Schematic Designs with Instantiated HDL-Based Macros”
- “Schematic Designs With Instantiated LogiBLOX Modules”
- “Schematic Designs With Instantiated CORE Generator Cores”
- “Schematic Designs With Finite State Machine (FSM) Macros”
- “Finite State Machine (FSM) Designs”

Schematic Flow Processing Overview

Refer to the“Project Toolset” chapter for information on how to create a Schematic Flow project and for an overview of the tools available for such projects.

The following figure illustrates the processing performed at the various stages of a Schematic Flow project.

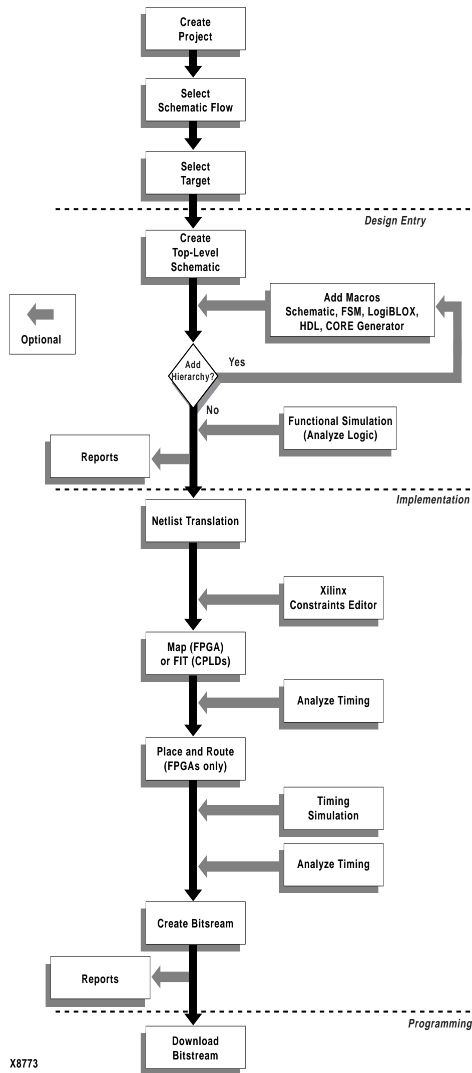


Figure 3-1 Schematic Flow Project Processing

Top-Level Designs

Schematic Flow projects can have top-level schematic or Finite State Machine (ABEL) designs. A top-level design can have any number of underlying schematic, HDL, LogiBLOX, CORE Generator, ABEL, or Finite State Machine (FSM) macros. Although individual modules may require some form of synthesis, the entire project is not synthesized and the netlist that is exported for implementation is not optimized across module boundaries as in an HDL Flow project.

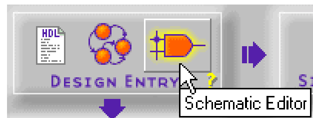
All-Schematic Designs

The following procedure describes how to create a top-level schematic design that contains schematics only, that is, there are no instantiated HDL or State Machine macros.

Creating the Schematic and Generating a Netlist

This section lists the basic steps for creating a schematic and generating a netlist from it.

1. Open the Schematic Editor by selecting the Schematic Editor icon from the Design Entry box on the Project Manager's Flow tab.

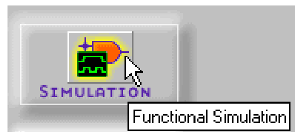


2. Select **Mode** → **Symbols** to add components to your new schematic. Select specific components from the SC Symbols window.
3. Complete your schematic by placing additional components from the Symbol toolbox including I/O ports, nets, buses, labels, and attributes.
4. Save your schematic by selecting **File** → **Save**.

For more information about schematic designs, see the "Schematic Design Entry" chapter or in the Schematic Editor window, select **Help** → **Schematic Editor Help Contents**.

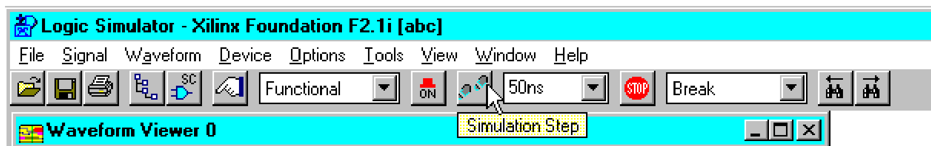
Performing Functional Simulation

1. Open the Logic Simulator by clicking the Functional Simulation icon in the Simulation box on the Project Manager's Flow tab.

   The design is automatically loaded into the simulator. The Waveform Viewer window displays on top of the Logic Simulator window.

2. Add signals by selecting **Signal** → **Add Signals**.
3. From the Signals Selection portion of the Components Selection for Waveform Viewer window, select the signals that you want to see in the simulator.
4. Use **CTRL**-click to select multiple signals. Make sure you add output signals as well as input signals.
5. Click **Add** and then **Close**. The signals are added to the Waveform Viewer in the Logic Simulator screen.
6. Select **Signal** → **Add Stimulators** from the Logic Simulator menu. The Stimulator Selection window displays.
7. In the Stimulator Selection window, create the waveform stimulus by attaching stimulus to the inputs. For more details on how to use the Stimulus Selection window, click the Help button.
8. After the stimulus has been applied to all inputs, click the Simulation Step icon on the Logic Simulator toolbar to perform a simulation step. The length of the step can be changed in the Simulation Step Value pulldown menu to the right of the Simulation Step box. (If the Simulator window is not open, select **View** → **Main Toolbar**.)



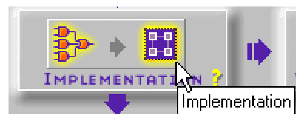
9. Verify that the output waveform is correct. Click the Step button repeatedly to continue simulating.

10. To save the stimulus for future viewing or reuse, select **`File`** → **`Save Waveform`**. Enter a file name with a .tve extension in the File name box of the Save Waveform window. Click **`OK`**.

    For more information about saving and loading test vectors, from the Logic Simulator window, select **`Help`** → **`Logic Simulator Help Contents`**. Then select **`Simulator Reference`** → **`Working With Waveforms`** → **`Saving and Loading Wave-forms`**.

Implementing the Design

1. Click the Implementation icon in the Implementation box on the Project Manager's Flow tab.



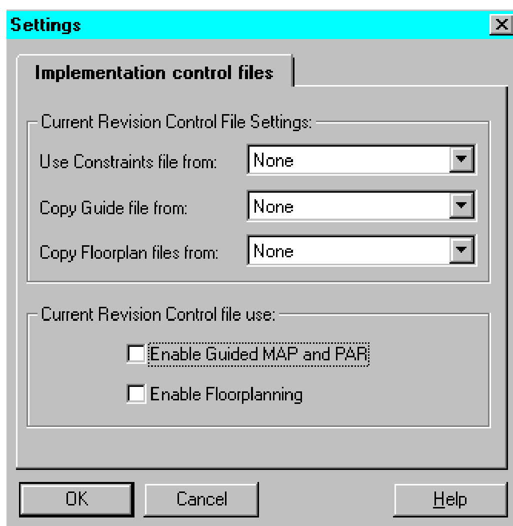
2. The Implement Design dialog box appears.

By default, the Implementation targets the device that was previously selected when you created the project. If you want to retarget the design to a different device, use the Implement Design dialog box. If you want to retarget to a new device family, you must first do so in the Foundation Project Manager by selecting **`File`** → **`Project Type`**.

The first time you implement the design, a new version of the design is created and given the default version and revision name

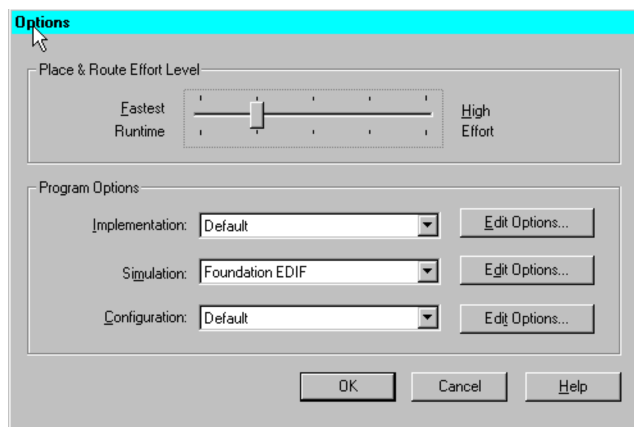
shown in the Implement Design dialog box. You can modify the version and revision names as desired.

3. In the Implement Designs dialog box, select **Set**. The Settings dialog box appears.



4. Specify control files if desired. Click **OK** to return to the Implement Design dialog box.

5. In the Implement Design dialog box, select **Options**. The Options dialog box displays.



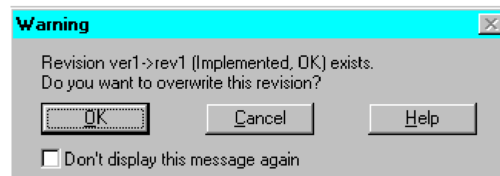
6. Choose any desired implementation options.
7. Click **OK** to return to the Implement Design dialog box.
8. Click **Run** to implement your design. The Flow Engine displays the progress of the implementation.

   When Implementation is complete, a dialog box appears indicating whether implementation was successful or not.

   For more information on the Flow Engine, refer to the “Design Implementation” chapter or select **Help**→ **Foundation Help Contents**→ **Flow Engine**.
9. Select the Reports tab on the Project Manager window and then double click the Implementation Report Files folder. Double click a report icon to review your design reports.

Creating a New Revision

If you modify the design, then click the Implementation button to re-implement the design after the first revision of a design version has been implemented, the existing revision is overwritten. A warning box appears to allow you to verify the overwrite operation.



You do not access the Implement Design dialog box for subsequent versions/revisions.

If you want to implement a new revision of the design (for any version), you must first create the new revision by selecting **Project** → **Create Revision**. This accesses the Create Revision dialog box that has the same fields as the Implement Design dialog box. The revision name is automatically entered. Modify the names, control files, and/or options and run the Flow Engine as described previously for the first version/revision.

Creating a New Version

If you want to implement a new version of the design (after the initial implementation), you must first create the new version by selecting **`Project`** → **`Create Version`**. This accesses the Create Version dialog box that has the same fields as the Implement Design dialog box. The version name is automatically entered. Modify the names, control files, and/or options and run the Flow Engine as described previously for the first version/revision.

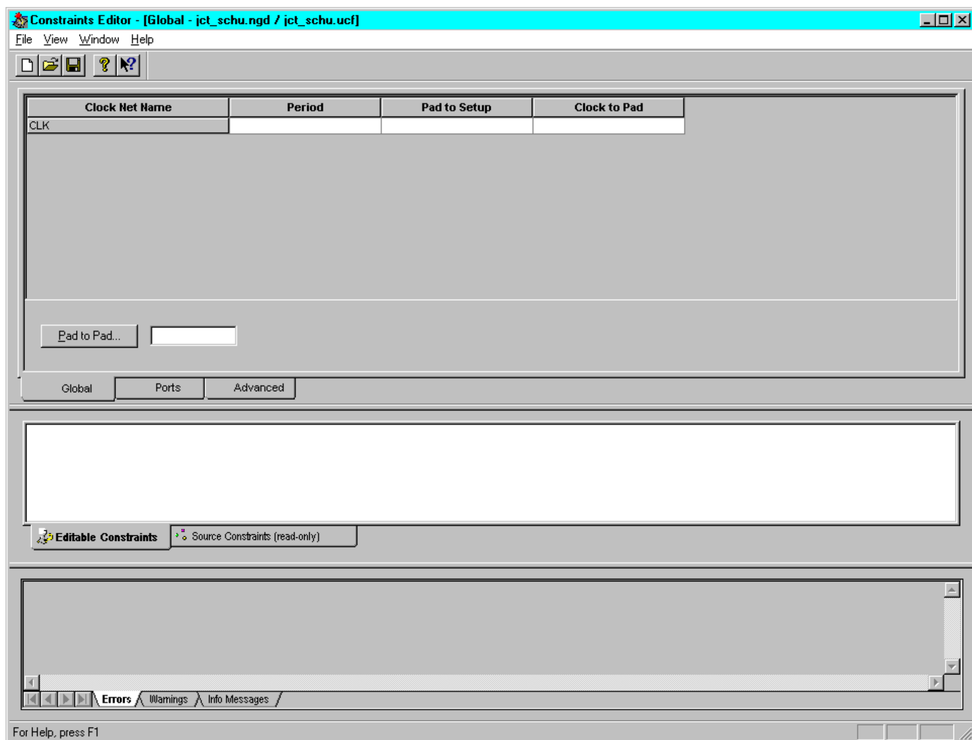
Editing Implementation Constraints

Constraints are instructions placed on symbols or nets in a schematic (or textual entry file such as VHDL or Verilog). They affect how the logical design is implemented in the target device. Applying constraints helps you to adapt your design's performance to expected worst-case conditions. The user constraint file (.ucf) is an ASCII file that holds timing and location constraints. It is read (by NGDBuild) during the translate process in the Flow Engine and is combined with an EDIF or XNF netlist into an NGD file.

In Foundation, a UCF file is automatically associated with a Revision. This UCF file is copied and used as your UCF file within a new revision. You can directly enter constraints in the UCF file or you can use the Xilinx Constraints Editor.

1. The Constraints Editor is a Graphical User Interface (GUI) that you can run after the Translate program to create new constraints in a UCF file. To access the Constraints Editor, select **`Tools`** → **`Implementation`** → **`Constraints Editor`** from the Project Manager.

   The following figure shows an example of the Global tab of the Implementation Constraints Editor.



2. Design-specific information is extracted from the design and displayed in device-specific spreadsheets. Click the tabs to access the various spreadsheets.

3. Right-click on an item in any of the spreadsheets to access a dialog box to edit the value. Use the online help in the dialog boxes to understand and enter specific constraints and options. Or, refer to the online software document, *Constraints Editor Guide* for detailed information.

   The following figure shows an example of the Pad to Setup dialog box accessed when you right click anywhere on the CLR Port row on the Ports tab of the Implementation Constraints Editor and then select `Pad to Setup`.

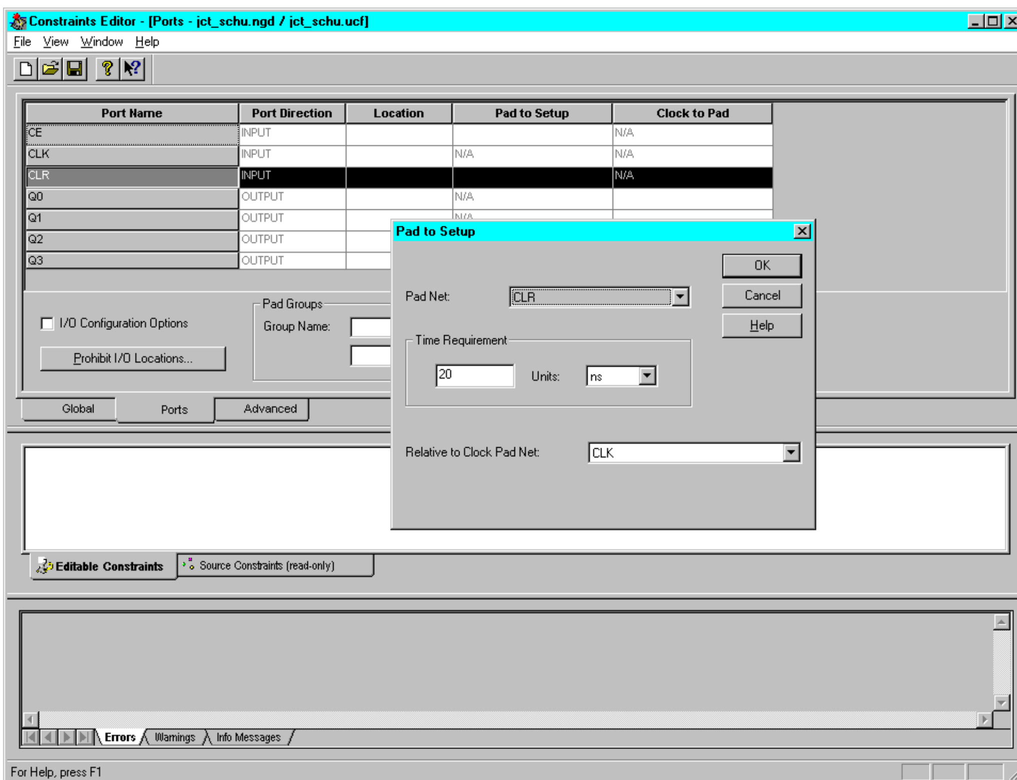


Figure 3-2 Implementation Constraints Editor - Ports Tab

4. After you finish editing the constraints, click `Save` to close the Constraints Editor window

5. You must rerun the Translate step in the Flow Engine to have your new constraints applied to the design.

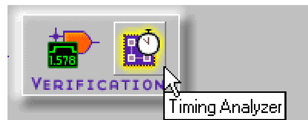
6. Click the Implementation icon on the Project Manager's Flow tab to rerun Translate (and the other phases).

   Or, to just rerun the Translate phase, select **`Tools`** → **`Implementation`** → **`Flow Engine`**. Click **`Yes`** to start at the Translate phase when prompted. Then click the Step button at the bottom of the Flow Engine Window window. Exit the Flow Engine when the Translate phase is Completed.

Verifying the Design

Performing a Static Timing Analysis (Optional)

1. Click the Timing Analyzer icon in the Verification box on the Project Manager's Flow tab.



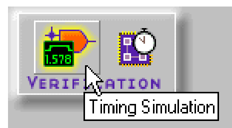
2. Perform a static timing analysis on mapped or placed and routed designs for FPGAs.

For FPGAs, you can perform a post-MAP or post-place timing analysis to obtain rough timing information before routing delays are added. You can also perform a post-implementation timing analysis on CPLDs after a design has been implemented using the CPLD fitter.

For details on how to use the Timing Analyzer, select **`Help`**→ **`Foundation Help Contents`**→ **`Timing Analyzer`**.

Performing a Timing Simulation

1. Open the Timing Simulator by clicking the Timing Simulation icon in the Verification box on the Project Managers's Flow tab. The implementation timing netlist will be loaded into the simulator.



2. The Waveform Viewer window displays on top of the Logic Simulator window.

   Refer to the “Performing Functional Simulation” section for instructions on simulating the design. (The operation of the simulator is the same for functional and timing simulation.)

3. If you have already saved test vectors (for instance, in the functional simulation), you may load these vectors into the timing simulator by selecting **`File`** → **`Load Waveform`**.

Programming the Device

1. Click the Device Programming icon in the Programming box on the Project Manager’s Flow tab.

2. From the Select Program box, choose the Hardware Debugger, the PROM File Formatter, or the JTAG Programmer.

   For CPLD designs, you must use the JTAG Programmer. For instructions, select **`Help`** → **`Foundation Help Contents`** → **`JTAG Programmer`**.

   For FPGA designs, use the JTAG Programmer, Hardware Debugger, or PROM File Formatter. For instructions, select **`Help`** → **`Foundation Help Contents`** → **`Advanced Tools`** and then select the desired tool.

Schematic Designs with Instantiated HDL-Based Macros

This section explains how to create HDL macros and then add them to a schematic design.

Creating HDL Macros

After you create an HDL macro, the macro is available from the SC Symbols window in the Schematic Editor. Following are the steps to create HDL macros.

1. Open the HDL Editor by clicking the HDL Editor icon in the Design Entry box on the Project Manager's Flow tab.

2. When the HDL Editor appears, you may select an existing HDL file or create a new one. The following steps describe creating a new HDL file with the Design Wizard.
3. In the HDL Editor dialog box, select **`Use HDL Design Wizard`**. Click **`OK`**.
4. From the Design Wizard window, select **`Next`** and then choose **`VHDL`**, **`Verilog`**, or **`ABEL`** and select **`Next`**. (You must have Base Express or Foundation Express to select VHDL or Verilog.)
5. Enter a name for your macro in the Design Wizard - Name window and then select **`Next`**.
6. Define your ports in the Design Wizard-Ports window.
7. Click **`Finish`**. The Wizard creates the ports and gives you a template in which you can enter your macro design.
8. Complete the design for your macro in the HDL Editor.
9. Create a macro symbol by selecting **`Project`** → **`Create Macro`** from the HDL Editor window.

   The synthesizer will not insert top level input and output pads for this macro. Instead the top level schematic, which contains the

macro, includes all top level input and output pads required for implementation.

For more information about creating HDL macros, from the Project Manager window, select **`Help`** → **`Foundation Help Contents`**→ **`HDL Editor`**.

Creating the Schematic and Generating a Netlist

1. Open the Schematic Editor by clicking the Schematic Editor icon in the Design Entry box on the Project Manager's Flow tab.
2. Select **`Mode`** → **`Symbols`** to add components to your new schematic.

   Any macros that you have created display in the SC Symbols toolbox under the project working library's heading.
3. Select the HDL macro that you created by clicking its name.
4. Move your cursor to the schematic sheet and place the macro symbol by clicking.
5. Complete your schematic by placing additional components from the Symbol toolbox including I/O ports, nets, buses, labels, and attributes.
6. Save your schematic by selecting **`File`** → **`Save`**.

For more information about schematic designs, see the"Schematic Design Entry" chapter or, in the Schematic Editor window, select **`Help`** → **`Schematic Editor Help Contents`**.

To complete the design, read the following sections in the order listed:

- "Performing Functional Simulation"
- "Implementing the Design"
- "Verifying the Design"
- "Programming the Device"

Schematic Designs With Instantiated LogiBLOX Modules

LogiBLOX modules can be used in schematic designs. First, the module must be created. The module can then be added to the schematic like any other library component.

Creating LogiBLOX Modules

To use the program in a schematic-based environment, follow these steps:

1. With a project open, invoke the LogiBLOX Module Selector from within the Schematic Editor (**`Tools`** → **`LogiBLOX Module Generator`**).
2. Select a name and a base module type (for example, counter, memory, or shift-register).
3. Customize the module by selecting pins and specifying attributes.
4. After completely specifying a module, click **`OK`**. Selecting **`OK`** initiates the generation of a schematic symbol and a simulation model for the selected module. The schematic symbol for the LogiBLOX component is incorporated into the project library and is automatically attached to the cursor for immediate placement.
5. Place the module on your schematic.
6. Connect the LogiBLOX module to the other components on your schematic using ordinary nets, buses, or both.
7. Complete your schematic by placing additional components from the symbol toolbox including I/O ports, nets, buses, labels, and attributes.
8. Save your schematic by selecting **`File`** → **`Save`**.

Importing Existing LogiBLOX Modules

You can also import LogiBLOX modules that already exist (for example, from another project).

To convert an existing LogiBLOX module to a binary netlist and save the component to the project working library, perform the following steps.

1. In the Schematic Editor, select **Tools** → **Import LogiBLOX**.
2. From the Import LogiBLOX from MOD File dialog box, select the MOD file for the LogiBLOX module that you want to import. Click **OK**.

   The schematic symbol for the LogiBLOX component is incorporated into the SC Symbols window in the Schematic Editor.
3. Follow Steps 5 through 8 in the previous section—"Creating LogiBLOX Modules"—to instantiate your module.

To complete the design, read the following sections in the order listed:

- "Performing Functional Simulation"
- "Implementing the Design"
- "Verifying the Design"
- "Programming the Device"

Schematic Designs With Instantiated CORE Generator Cores

Cores generated in the CORE Generator tool can be used in schematic designs. After the core is selected and customized, its schematic symbol is generated by the CORE Generator tool. The core can then be added to the schematic like any other library component.

Creating Core Symbols

To use the CORE Generator tool in a schematic-based environment, follow these steps:

1. With a project open, invoke the CORE Generator tool from within the Schematic Editor (**Tools** → **CORE Generator**).
2. Select **Project** → **Project Options**. Ensure that Design Entry is Schematic and that the Vendor is Foundation in the Project

Options dialog box. The Family entry should reflect the project's target device. Click **OK** to exit the Project Options dialog box.

3. To aid selection, the available Cores are categorized in folders on the View Mode section of the main CORE Generator window. Double click a folder to see its sub-categories. When you double click a sub-category folder, the available Cores are listed in the "Contents of" section of the main CORE Generator window.

   If you double click the name of the desired core, a new window opens to allow you to view its description or access its data sheet. (Acrobat Reader is required to view the data sheet.)

4. To select a core to instantiate into a schematic, highlight the core's name (click once) in the "Contents of" window and then select **Core** → **Customize** and enter a name for the core in the Component Name field.

   The name must begin with an alpha character. No extensions or uppercase letters are allowed. The name may include numbers and/or the underscore character.

5. Other available customization options are unique for each core. Customize the core as necessary.

6. Select **Generate** to create a schematic symbol and a simulation model for the selected core. The schematic symbol for the core is incorporated into the project library and can be selected from the SC Symbols list.

7. Select **File** → **Exit** to return to the Schematic Editor.

8. In the Schematic Editor, select the symbol from the SC Symbols list (**Mode** → **Symbols**) and place the core on your schematic.

9. Connect the core to the other components on your schematic using ordinary nets, buses, or both.

10. Complete your schematic by placing additional components from the symbol toolbox including I/O ports, nets, buses, labels, and attributes.

11. Save your schematic by selecting **File** → **Save**.

To complete the design, read the following sections in the order listed:

- "Performing Functional Simulation"

- “Implementing the Design”
- “Verifying the Design”
- “Programming the Device”

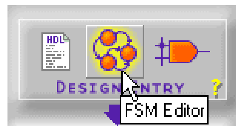
Schematic Designs With Finite State Machine (FSM) Macros

This section explains how to create state machine macros and instantiate them in schematic designs.

Creating FSM Macros

After a macro is created, it is available from the SC Symbols window in the Schematic Editor. These are the steps you follow to create State Machine macros.

1. Open the Finite State Machine (FSM) editor by clicking the FSM Editor icon in the Design Entry box on the Project Manager’s Flow tab.



2. When the State Editor window appears, you may select an existing FSM macro or create a new one. The following steps describe creating a new FSM macro with the Design Wizard.
3. Select **Use HDL Design Wizard**. Click **OK**.
4. From the Design Wizard window, select **Next**.
5. From the Design Wizard - Language window, choose **VHDL**, **Verilog**, or **ABEL**. Click **Next**. (You must have Base Express or Foundation Express to select VHDL or Verilog.)
6. Enter a name for your macro in the Design Wizard - Name window. Select **Next**.
7. Define your ports in the Design Wizard-Ports window. Select **Next**.

8. In the Design Wizards - Machines window, select the number of state machines that you want. Click **Finish**. The Wizard creates the ports and gives you a template in which you can enter your state machine design.
9. Create the design for your state machine in the State Editor.
10. When you are finished creating your state machine, create a macro symbol by selecting **Project** → **Create Macro**.

    The synthesizer will not insert top level input and output pads for this macro. Instead the top level schematic, which contains the macro, includes all top level input and output pads required for implementation.

    For more information about state machines, select **Help** → **Foundation Help Contents**→ **State Editor**.

Creating the Schematic and Generating a Netlist

1. Open the Schematic Editor.
2. Select **Mode** → **Symbols** to add components to your new schematic.

    Any macros that you have created display in the SC Symbols toolbox under the project working library's heading.
3. Select the state machine macro from the toolbox by clicking its name.
4. Move your cursor to the schematic sheet and place the macro symbol by clicking.
5. Complete your schematic by placing additional components from the SC Symbols toolbox including I/O ports, nets, buses, labels, and attributes.
6. Save your schematic by selecting **File** → **Save**.

For more information about schematic designs, see the "Schematic Design Entry" chapter or, in the Schematic Editor window, select **Help** → **Schematic Editor Help Contents**.

To complete the design, read the following sections in the order listed:

- "Performing Functional Simulation"

- “Implementing the Design”
- “Verifying the Design”
- “Programming the Device”

Finite State Machine (FSM) Designs

The FSM Editor allows you to specify functionality using the "bubble state diagram" concept. Once you have described the state machine (or machines) using the FSM Editor's available graphics objects, the State Editor generates behavioral VHDL, Verilog, or ABEL code (depending on which language type was selected when the state diagram was begun). This code can then be synthesized to a gate-level netlist.

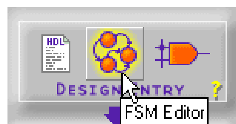
A schematic project can have a top-level ABEL design created with a text editor or with the FSM Editor. (Top-level ABEL designs are not recommended for FPGA projects.)

The Finite State Machine Editor can also generate ABEL, VHDL, or Verilog macros that can be included in top-level schematic or ABEL designs.

This section describes using the FSM Editor to produce a top-level ABEL design for a Schematic Flow project. It also includes information on using the FSM Editor to produce underlying ABEL, VHDL, or Verilog macros for inclusion into a top-level design in a Schematic Flow project.

Creating a State Editor Design

1. Click the FSM Editor icon in the Design Entry box on the Project Manager’s Flow tab.



2. When the State Editor window appears, you may select an existing FSM macro or create a new one. The following steps describe creating a new FSM macro with the Design Wizard.
3. From the Design Wizard window, select **Next**.

4. From the Design Wizard - Language window, choose **VHDL**, **Verilog**, or **ABEL** (Schematic Flow only) and select **Next**.
5. In the Design Wizard - Name window, enter a name for your design. Select **Next**.
6. Define your ports in the Design Wizard-Ports window. Select **Next**.
7. In the Design Wizards - Machines window, select the number of State Machines that you want. Click **Finish**. The Wizard creates the ports and gives you a template in which you can enter your macro design.
8. Define the states in the FSM Editor.

Defining States

1. From the State Editor window, select **FSM** → **State** or click on the State button in the vertical toolbar.
2. Place the state bubble. The default state name is S1.
3. Click on the state name to select it, then click again to edit the text.
4. Type the desired state name.
5. Click on the state bubble to select it. Click and drag the small squares to change the size and shape of the bubble. When the state bubble is large enough to hold the name, click and drag the state name to center it in the bubble.
6. Repeat steps 1-4 to create new states.

   To ensure that the state machine powers up in the correct state, you must define an asynchronous reset condition. This reset will not be connected in the schematic, but its presence directs the compiler to define the state encoding so that the machine will power up in the correct state.
7. Select **FSM** → **Reset**, or click **Reset** in the vertical toolbar.
8. Place the reset symbol in the state diagram. Click inside a state bubble to define this as the reset state.
9. To define the reset as asynchronous, right-click on the reset symbol and select **Asynchronous**.

10. Define the transition, conditions, and actions for the state diagram.
11. When you have completed the state diagram, select **`File`** → **`Save`**.

Defining Transitions, Conditions, and Actions

Transitions define the changes from one state to another. They are drawn as arrows between state bubbles.

If there is more than one transition leaving a state, you must associate a condition with each transition. A condition is a Boolean expression. When the condition is true, the machine moves along the transition arrow.

Actions are HDL statements that are used to make assignments to output ports or internal signals. Actions can be executed at several points in the state diagram. The most commonly used actions are state actions and transition actions. State actions are executed when the machine is in the associated state. Transition actions are executed when the machine goes through the associated transition.

Adding a Top-Level ABEL Design to the Project

ABEL FSM designs can be used as top-level designs in a Schematic Flow project. After you have created an ABEL macro using the FSM Editor, perform the following steps to add the design to the project.

1. From the State Editor window, select **`File`** → **`Save`** to save the ABEL state diagram.
2. Select **`Project`** → **`Add to project`**.
3. Select **`Synthesis`** → **`Synthesize`**.

To complete the design, read the following sections in the order listed:

- “Performing Functional Simulation”
- “Implementing the Design”
- “Verifying the Design”
- “Programming the Device”

Chapter 4

Schematic Design Entry

This chapter contains the following sections:

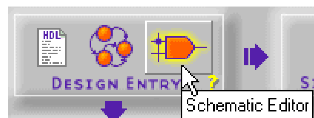
- "Managing Schematic Designs"
- "Hierarchical Schematic Designs"
- "Manually Exporting a Netlist"
- "Creating a Schematic from a Netlist"
- "Miscellaneous Tips for Using the Schematic Editor Tool"

Refer to the "Top-Level Designs" section of the "Design Methodologies - Schematic Flow" chapter for several examples of top-level schematic designs.

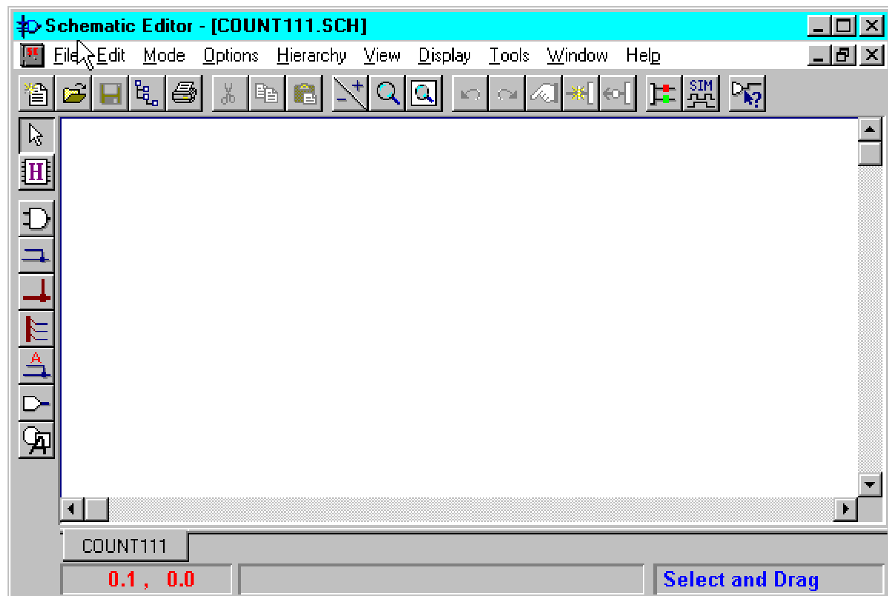
Managing Schematic Designs

The following subsections describe various features of the schematic design tool.

1. To access Schematic Editor, click the Schematic Editor icon in the Design Entry box on the Project Manager Flow tab.



2. The Schematic Editor window opens.



Design Structure

You can create Foundation schematic editor designs that have the following structures:

- Single sheet designs
- Multi-sheet designs
- Hierarchical designs

Selecting a structure depends on the design size (number of symbols and connections), purpose (board or chip design), and company standards. The following sections describe each of these design types.

Single Sheet Schematic

Single sheet designs are typically used for small designs. The largest page size is 44” x 34” (size E). The major advantage of a single sheet schematic is that you can use physical connections for an entire design, which makes tracking of the connections easier.

The disadvantages of using large pages are:

- Schematics redraw slowly. A schematic with many symbols may take a long time to scroll.
- Large schematics must be printed on plotters instead of laser printers.

Multi-sheet Flat Schematic

If a design is too large to print on a single sheet, you can use a multi-sheet design structure. When you create a new sheet, it is automatically added to the current project. To make connections between schematic sheets, you must make logical connections by using the same net names. For example, if you use the net CLOCK on sheet 1 and net name CLOCK on sheet 2, then both net segments are logically connected.

These connections can be confirmed by using the Query option. To activate the Query option, select **`Mode`** → **`Query`** from the Schematic Editor main window and then select items on the schematic. To find out more about Query options, select **`Help`** → **`Schematic Editor Help Contents`** from the Schematic Editor main window. Select the Index tab. Type **`Query`** in the search list box. Double click **`querying connections`**.

Following are the advantages of using the multi-sheet design structure.

- Small sheet sizes that print on laser printers
- Unlimited design sizes without condensing the schematics

Note: All reference designators for symbols in the multi-sheet schematics must be unique. The Foundation design entry tools automatically assign these unique numbers. If you manually assign the same reference numbers to two different devices, an error is reported when you create a netlist.

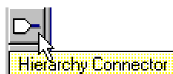
Hierarchical Schematic

Since large number of symbols are used in FPGA and CPLD designs, handling large designs using the multi-sheet design structure can become very difficult and complex. Large designs typically require thousands of simple primitives like gates and flip-flops. To simplify schematics, designers prefer to use high-level components that have

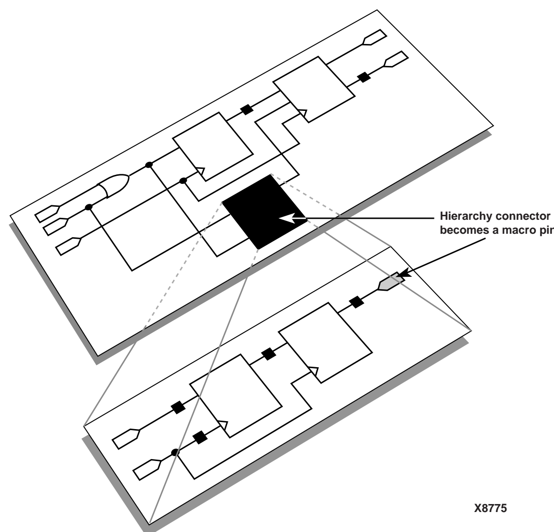
clear functionality. These high-level components are implemented using hierarchical macros. A hierarchical macro, a device in the library that looks like a standard component, is implemented as a symbol with an underlying schematic or netlist. For example, you can create an equivalent of a counter by drawing a macro schematic with only gates and flip-flops. This macro can then be saved and reused in your designs. All FPGA and CPLD libraries already contain many hierarchical macros.

Hierarchical designs are very effective with IC designs. In hierarchical macro schematics, all net names and reference names are local, which means that you can use the same signal names in different macros.

The connections between hierarchical macro symbols and the underlying schematic is made via hierarchy connectors. Use the Hierarchy Connector icon (shown below) in the Schematic Editor toolbar to place hierarchy connectors.



These connectors are converted into hierarchical symbol pins as illustrated in the following figure.



After the macro symbol is placed on the schematic sheet, you can connect wires to these pins on the macro. Only the signals shown as symbol pins can be connected.

Some advantages of hierarchical designs follow:

- The symbols in a hierarchical schematic library can represent large functional blocks implemented in detail on a lower level. By viewing the high level schematic, you can see the general design structure without being overwhelmed by the lower level details.
- Top-down or bottom-up methodology assists in team development by defining design sections for each designer. All conflicts between design sections are eliminated by allowing interfaces only to explicitly defined pins and bus pins.
- You can use multiple instances of the same macro. If you use a schematic sheet in a flat design, you must duplicate the macro for each instance. If you then make a correction to the macro, you must edit all instances. The hierarchical macro is modified once and all instances are then updated.
- Macros can be used in multiple projects. You can develop a set of reusable modules that are stored as hierarchical macros and used in several designs.

Following are some of the disadvantages of hierarchical designs:

- Netlist names can become very long because you must specify the complete hierarchical path. The method used to create unique reference identifiers adds the hierarchy reference name to each symbol reference. For example, a symbol U58 in a macro called H8 will be called H8/U58. In multilevel hierarchical designs, these names can become very long depending on the number of hierarchy levels.
- Updating macros often requires changing their symbols, which then means that you must correct all schematics that use that macro.

Adding New Sheets to the Project

To create a new empty sheet, select **`File`** → **`New Sheet`**. The new sheet receives the name of the project with the sequential sheet number assigned to it automatically. You can save the sheet with a different name by selecting **`File`** → **`Save As`**. Each new sheet is

automatically added to the project contents in the Hierarchy Browser. All schematic sheets that have been added to the project are visible in the Files tab of the Hierarchy Browser.

To open a sheet that does not belong to the project, select **Tools** → **Scratchpad**. To add a scratchpad sheet to the project, use the **File** → **Save As** option to define the schematic name. Then select **Hierarchy** → **Add Current Sheet to Project**.

Adding Existing Sheets to the Project

To add an existing schematic sheet to your project, select **Hierarchy** → **Add Sheets to Project**. In the Add to Project window, select the schematic file(s) you want to add and click the Add button. The schematic editor loads each added sheet and verifies that the symbols used in these schematics are available and that there are no duplicate reference numbers. The list of project sheets is then updated.

Note: The schematic editor automatically adds libraries used by its schematic sheets to the current project even if they are not listed in the project libraries. The libraries are added when you open a schematic file and symbols are not found in the current project libraries.

Opening Non-project Sheets

When you select **File** → **Open**, only the sheets that belong to the current project are shown. If you want to open a sheet that does not belong to the current project, use the Browse button to select a schematic file from any disk. The schematics opened with the Browse option display the Cannot Edit message in their title bar. These sheets can only be viewed. They cannot be edited.

To edit such schematics, select **Hierarchy** → **Add Current Sheet to Project**. The currently selected sheet, which is then added to the current project, can then be edited. The schematic is copied to the current project directory, so the changes do not affect other projects.

Removing Sheets from the Project

To remove a sheet from the project, from the Project Manager Files tab, select the schematic sheet that you want to remove and select **Document** → **Remove**. Click **Yes**.

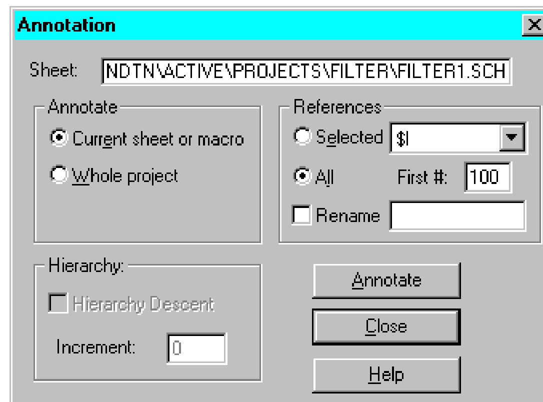
Note: Deleting the sheet from the project does not delete the schematic file from the disk. If you want to delete unwanted files, you can use the Windows Explorer and delete *.SCH files from the project directory.

Renumbering Symbol References

The reference numbers are assigned sequentially in the order that you place symbols on different sheets. As a result, the symbol reference numbers in the multi-sheet schematics can be random. To order the symbol numbers, you may want the symbol reference numbers to correspond to different sheets. For example, symbols on the first sheet may start with U100, and symbols on the second sheet may start with U200.

To renumber project sheets and the associated symbol reference numbers, from the Schematic Editor window proceed as follows:

1. Select **`Options`** → **`Annotate`**.
2. The Annotation dialog box appears.



3. From the Annotation window, click **`Whole Project`**.
4. In the References section, click **`All`** to apply the numbering to all symbols.
5. Enter **`100`** in the First # field.
6. Press the **`Annotate`** button.
7. Press the **`Close`** button.

Copying a Section of a Schematic to Another Sheet

If you want to move or duplicate a section of a schematic to another sheet, perform the following steps:

1. Place the cursor at the corner of the area to be copied, depress the mouse button and drag the cursor to outline a rectangular area for selection. All items within the selected area are selected when you release the mouse button.
2. To select additional objects on the schematic sheet without deselecting the currently selected object, use the Shift key.
3. Use the Copy or Cut options in the Edit menu. The Copy option copies the selected objects to the clipboard. The Cut option copies the selected block to the clipboard and deletes it from the schematic. The clipboard is a temporary sheet that stores the copied objects.
4. Go to the sheet where you want to paste the schematic objects and select the `Paste` option from the Edit menu. A rectangle is displayed at the cursor position. You can move it around the schematic to position the copied block to the desired location.
5. Press the mouse button to confirm the location of the pasted block.

Note: The selected schematic block contains all wires internal to the block, that is, between symbols or labels within the selected area. All wires connected to symbols outside the area are not copied to the clipboard.

Troubleshooting Project Contents

If a netlist creation error is reported, try removing one sheet at a time from the project until the netlist can be successfully created. Then analyze the last-removed sheet for any possible errors.

Hierarchical Schematic Designs

A design has a hierarchical structure if any of the symbols on the schematic sheet contain an underlying netlist or schematic. The hierarchical macros may be user-created or may already exist in a library. If you use one of these symbols, your design becomes hierarchical.

Creating a Schematic Macro (Bottom-Up Methodology)

To create a schematic macro using a bottom-up methodology, perform the following steps.

- Before you start drawing a schematic, make sure that the necessary libraries have been assigned to the project. You can view the currently attached project libraries in the Files tab of the Project Manager.
  - To add additional libraries, select **`File`** → **`Project Libraries`** from the Project Manager.
  - When the Project Libraries window displays, select the appropriate libraries and then click **`Add`**.

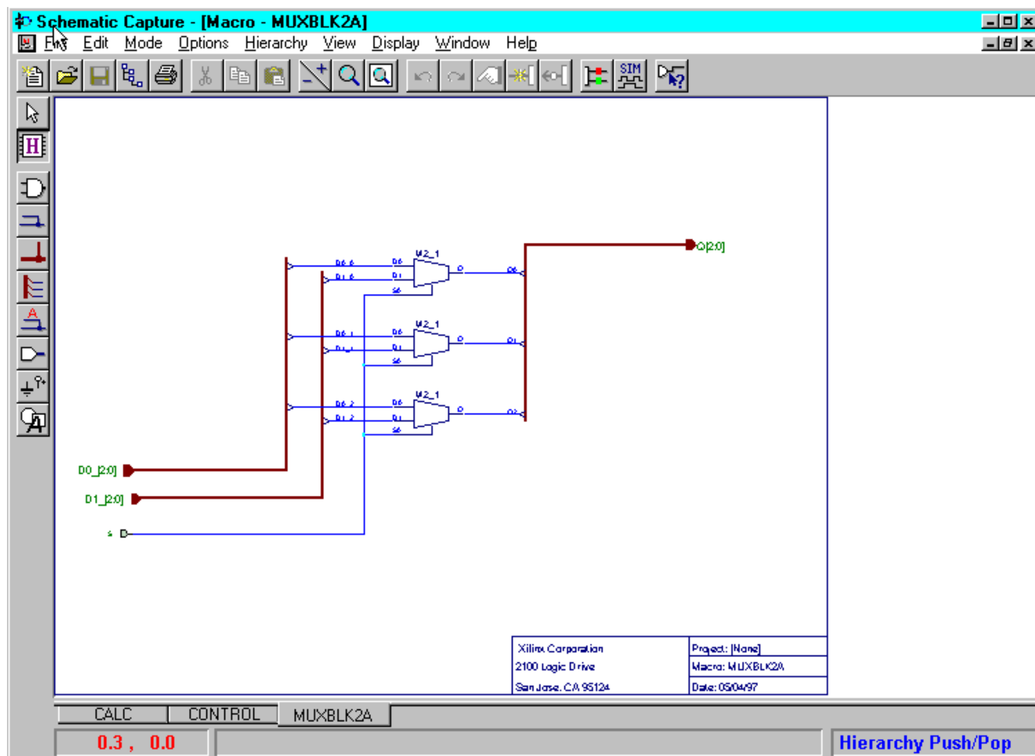
    This operation transfers the selected libraries from the Attached libraries window to the Project Libraries window, which makes these libraries available to the Schematic Editor.
- Enter your schematic design in the Schematic Editor, just like any other flat design, with the following constraints:
  - Each macro is a self-enclosed entity. Any connection to the top-level sheet can only be performed through hierarchy connectors.
  - The hierarchy connectors must be specified explicitly as Input, Output, or Bidirectional. This specification is important because the design entry tools automatically generate a symbol. The location of the pins on the symbol depends upon their schematic I/O definition (only inputs are on the left-side of the symbol outline). If needed, edit this symbol in the Symbol Editor.
- To create the macro symbol, select **`Hierarchy`** → **`Create Macro Symbol from Current Sheet`** from the Schematic Editor window. The new symbol is automatically placed in the current project's working library.

Recognizing Hierarchical Macros

You can recognize hierarchical macro symbols by their color. By default, the schematic-based macros are dark blue. The netlist-based macros are purple. You can change these default colors by selecting **View** → **Preferences** → **Colors**.

Navigating the Project Hierarchy

You can view the schematic of a hierarchical symbol by selecting **Hierarchy** → **Push**. When the H cursor is active, double click on a symbol to display its underlying schematic. You can use the tabs at the bottom of the Schematic Editor window to navigate between the top schematic (CALC in the following example) and its opened macro schematics (CONTROL and MUXBLK2A in the following example).

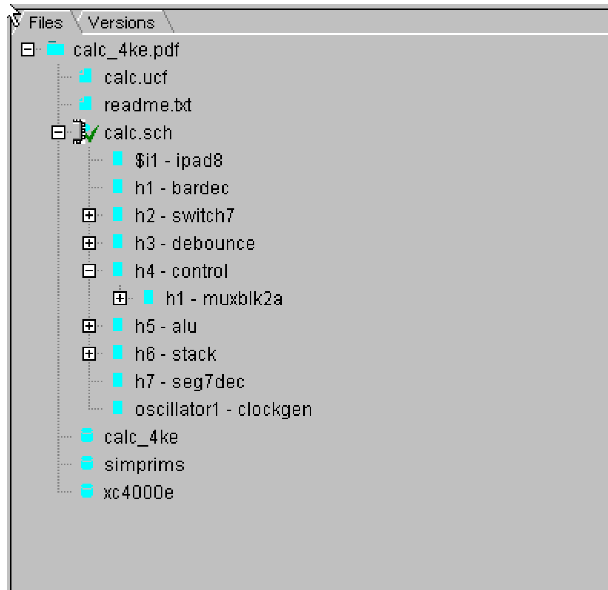


If you double click on a symbol that does not have an underlying schematic, HDL, or FSM file, the following message displays:

```
Symbol is a primitive cell.
```

To exit the H cursor mode, select **Hierarchy** → **Pop**.

You can also navigate the hierarchical structure of the design from the Files tab on the Project Manager window shown in the next figure.



Hierarchy sublevels can be expanded or collapsed by clicking on the + or - icons.

The Hierarchical Browser window shows the hierarchical design tree. A plus (+) designates a hierarchy with additional hierarchical sublevels. You can open them by single clicking on these icons.

A minus (-) denotes a hierarchy that already shows lower hierarchy levels. Clicking on the - symbols inside the icons reduces the hierarchy to the higher levels, which simplifies the viewing of very complex designs.

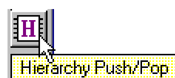
An icon with no symbol indicates that the given hierarchical level has no additional hierarchical sheets.

Note: Double clicking on the top schematic name or the name of any of its underlying schematic macros loads that schematic to the screen for viewing and editing.

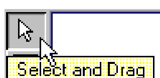
Modifying Existing Macros

If you want to make some changes to an existing macro schematic, perform the following steps:

1. Push into the schematic macro by clicking the Hierarchy Push/Pop icon and double clicking on the macro symbol.



2. Select the Select and Drag toolbar button to enter edit mode.



3. Make changes to the schematic.
4. Select **`File`** → **`Save`**.

When you change and save a hierarchical macro, you change all instances of this macro in the entire design. If the modified macro schematic has different I/O pins, its symbol changes and the pins may not match their previous locations on the schematics. If this happens, the wrong wires are automatically disconnected and will be marked with crossed circles. These wires must be manually reconnected by dragging the crossed circles over the target pins and then releasing the mouse button.

If you edit a macro from the system library that comes with the product, you cannot save it in the system library. You can only save it into a project library. For clarity, use a different name for the modified macro so that you can always be sure which symbols are currently used on the schematics.

Difference between a Macro and a Schematic

The following example explains what happens with the hierarchical schematic when you create a macro. Assume that the project TEST contains the schematic sheets TEST1 and TEST2. Create a macro for the schematic sheet TEST2 as follows:

1. Using the **`Hierarchy`** → **`Create Macro Symbol From Current Sheet`** option, convert the TEST2 schematic into a macro called MACRO1 in the TEST project library.

   The old schematic sheet TEST2 still resides in the project directory. You can open this schematic file, but there is no longer any relationship between the TEST2.SCH schematic file and the TEST project or MACRO1.

2. Use the Windows Explorer to delete the file TEST2.SCH from the project directory.

Hierarchy Symbol Changes

If you update a hierarchical macro, its symbol is not modified unless I/O pins change. When pin change, a new symbol is generated based on the new I/O pins, which may result in incorrect connections on the schematics that have previously used the symbol. The schematics that are currently open are automatically updated with the new symbol when you make the change. Other schematics are updated when you open them in the Schematic Editor.

When the symbols do not match the previous connections, the wires are automatically disconnected from that symbol, and a crossed circle is displayed at the end of these wires. To correct these connections, you can drag the circles and drop them at the appropriate pins.

You can edit the newly created symbol in the Symbol Editor program so that it matches the old pin locations.

Using a Top-down Methodology

To implement a top-down design, you first create a symbol for the hierarchical macro and then create the underlying schematic.

1. To create an empty symbol, select **`Tools`** → **`Symbol Wizard`** from the Schematic Editor.

2. Click **`Next`**.

3. In the Design Wizard - Contents dialog box, choose **Schematic** in the Contents section. Enter the Symbol Name and then select **Next**.
4. In the Design Wizard - Ports dialog box, select **New**.
5. Enter all ports including bus pins and power supply pins, if any. Select **Next**.
6. In the Design Wizard - Attributes, enter a reference for the new symbol. Select **Next**.
7. Click **Finish** in the Design Wizard - Contents window.
8. Place the symbol on a schematic sheet and make the required connections.
9. Push down into the symbol by clicking the Hierarchy Push/Pop function and double clicking on the macro symbol.

   An empty schematic sheet appears with the selected symbols' input pins located to the left and the output pins located to the right.
10. Enter the design and then select **File** → **Save**.

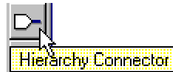
Hierarchical Design Example

This example explains how to create and use a hierarchical design.

1. Create a new project called MACROS.
   a) From the Project Manager, select **File** → **New Project**.
   b) In the Name field of the New Project window, enter **MACROS** and click **OK**.
2. Create the MACROS1 schematic.
   a) Click the Schematic Editor icon on the Design Entry button.
   b) Select the Symbol mode in the schematic toolbar and in the SC Symbols toolbox, select the **NAND2** symbol.

c) Draw the schematic shown in the "MACROS1 Schematic" figure including the input terminals, (INA, INB, INC) and output terminal (OUT).

d) Use the Hierarchy Connector icon (shown below) to draw the input and output terminals. Make sure you add the input and output buffers.



e) Connect the symbols (**`Mode`** → **`Draw Wires`**).

You can edit symbol dimensions and pins using the Symbol Editor. To open the Symbol Editor, select **`Tools`** → **`Symbol Editor`** from Schematic Editor, or double click on the placed symbol, and click the Symbol Editor button on the Symbol Properties dialog box. For more information on using the Symbol Editor, refer to the Symbol Editor's online help (**`Help`** → **`Contents`**).

f) Save the schematic using **`File`** → **`Save`**. The schematic is automatically named as MACROS1.

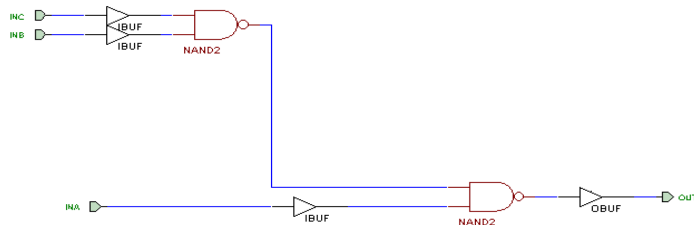
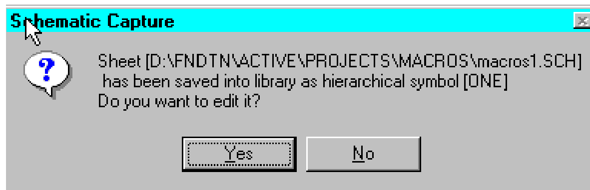


Figure 4-1 MACROS1 Schematic

3. Create the MACROS1 symbol.

a) Select **`Hierarchy`** → **`Create Macro Symbol From Current Sheet`**. The Create Symbol window should display the symbol name MACROS1. Click that name and change it to **`ONE`**, which will be the name of the hierarchical macro symbol. In the Input pins section, make sure that the following list of pins is entered: **`INA`**, **`INB`**, **`INC`**. Ensure that **`OUT`** is entered in the Output field.

b) Click **OK**. The netlist and the MACROS1 schematic are saved in the project library and a graphical symbol is automatically created. A verification message displays to allow you to edit the macro, if necessary, before continuing. Click **No**.



4. Create the second schematic (MACROS2).

a) Select **File** → **New Sheet**. A new schematic MACROS2 is automatically opened.

b) Select the Symbol mode in the schematic toolbar. In the SC Symbols toolbox, select the **FD** symbol.

c) Draw the schematic shown in the "MACROS2 Schematic" figure including the input terminals (D, CLK) and output terminal (OUT). Use the I/O Terminal icon to draw the terminals. Make sure you add the input and output buffers.

d) Connect the symbols (**Mode** → **Draw Wires**).

You can edit symbol dimensions and pins using the Symbol Editor. To open the Symbol Editor, select **Tools** → **Symbol Editor** from the Schematic Editor, or double click on the placed symbol, and click the Symbol Editor button on the Symbol Properties dialog box. For more information on using the Symbol Editor, refer to the Symbol Editor's online help (**Help** → **Contents**).

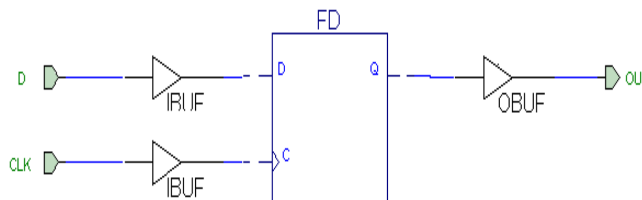


Figure 4-2 MACROS2 Schematic

e) Save the schematic using **File** → **Save**. The schematic is automatically named as MACROS2.

5. Create the MACROS2 symbol.

a) Select **Hierarchy** → **Create Macro Symbol From Current Sheet**. The Create Symbol window should display the symbol name MACROS2. Click that name and change it to **TWO**, which will be the name of the hierarchical macro symbol. In the Input pins section, enter pins: CLK, D. Enter **OUT** in the Output field. Note that only I/O terminals are recognized as pins for the symbol.

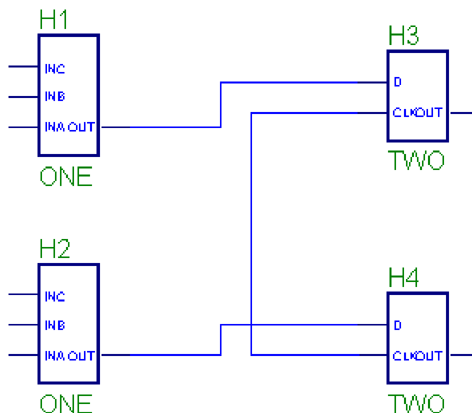
b) Click **OK**. The netlist and the MACROS2 schematic are saved in the project library and a graphical symbol is automatically created.

c) Save the schematic using the Save option.

6. Create a new sheet for the top level.

a) Select **File** → **New Sheet**. The empty sheet called MACROS3 opens.

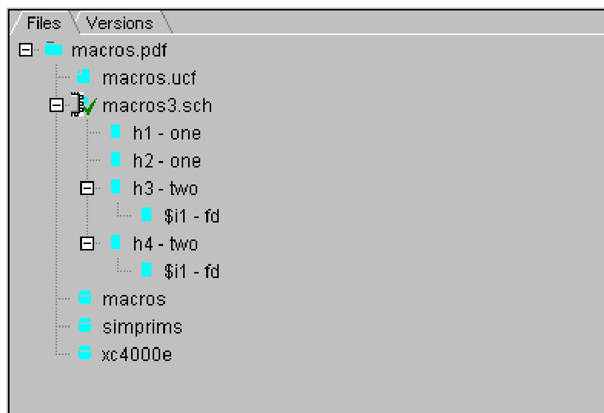
b) Select **Mode** → **Symbols** and find symbol ONE in the SC Symbols toolbox. Place two copies of that symbol, which are automatically called H1 and H2, on the schematic. Similarly, place two copies of the TWO symbol on the schematic. These symbols are automatically named H3 and H4. Refer to the following figure for placement details.



7. Use the Push/Pop option to view schematics.

   Select **Hierarchy** → **Hierarchy Push**. A cursor with the letter H displays. Point the cursor at the Symbol H1 and double click the mouse button. The schematic ONE opens showing you the schematic of the symbol H1.

8. View the project contents in the Files tab (shown in the following figure) of the Project Manager.

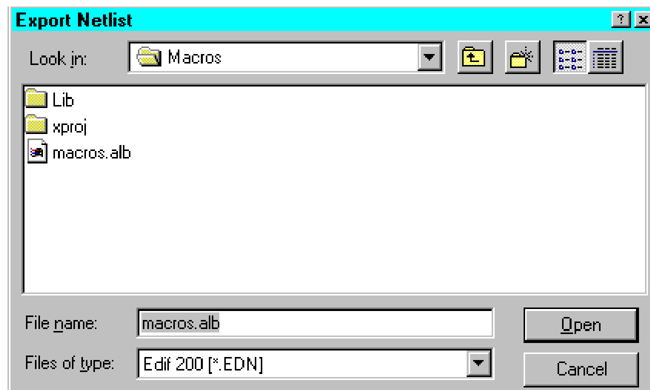


Manually Exporting a Netlist

External programs used in the Foundation Series software require netlist in proprietary text formats such as XNF, EDIF, and structural VHDL or Verilog.

To export the project netlist, perform the following steps from the Schematic Editor:

1. Select **Options** → **Export Netlist**. The Export Netlist dialog box displays.



2. From the File of Type pulldown menu, select the desired format.
3. Choose the source netlist ALB file. By default, the project netlist is automatically selected.
4. Click **OPEN** to start exporting.

Note: The EDIF netlist format is recommended for use with the Xilinx Design Implementation Tools.

Creating a Schematic from a Netlist

You can generate a schematic from an existing netlist. The Schematic Editor generates a schematic file and inserts it into the project directory as a non-project document. You can then use **File** → **Open** or add it to the project with **Hierarchy** → **Add Sheet to Project**. The names of automatically generated schematic files begin with the underline character (_). The underline character is followed by four initial letters of the project name and a three-digit suffix: 001 for the first file, 002 for the second, and so forth.

To generate a schematic from a netlist, perform the following steps:

1. Select **File** → **Generate Schematic from Netlist** from the Schematic Editor window. The Generating Schematic dialog box displays.
2. Select the desired netlist type from the List Files of Type list box. Then select the desired netlist file.
3. Click the **Options** button to display the Page Setup dialog box which allows you to select the desired page size and orientation.

4. Select the page size to be used for the generated schematics. The smaller the page size you select, the more numerous are the schematic files that are generated.
5. Select **Landscape** or **Portrait**.
6. Select **Wireless** to implement all connections using the connect-by-name method.
7. Click **OK**.

Miscellaneous Tips for Using the Schematic Editor Tool

This section describes various tips for creating schematic designs.

Color-coded Symbols

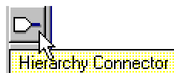
Symbols are color-coded to represent their type.

- Schematic user macros — blue
- Primitives and empty symbols — red
- HDL, State Editor, and netlist macros — purple
- State Editor macros — purple
- Library macros — black

These color codes are the default values. If you wish to change the defaults, select **View** → **Preferences** → **Colors** from the Schematic Editor.

Using the Hierarchy Connector

Only use the Hierarchy Connector when specifying pins for a schematic macro. Never use hierarchy connectors on top-level schematic sheets.



Using Input and Output Buffers

Xilinx schematics require that you use input and output buffers between input and output pads. The following figures illustrate incorrect and correct input and output port design.

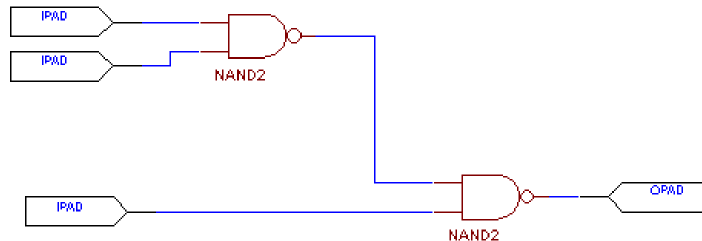


Figure 4-3 Incorrect Port Design (Without Buffers)

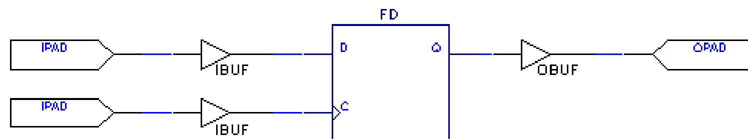
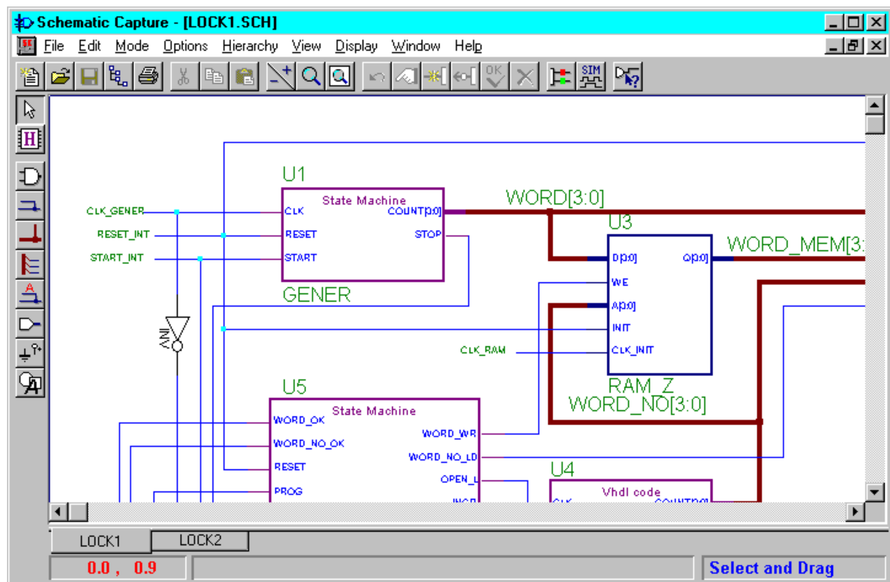


Figure 4-4 Correct Port Design (With Buffers)

Schematic Tabs

Tabs on a schematic sheet facilitate navigation between schematic sheets. The following example shows the tabs that display after opening the schematics for the “lock” project.



Note the LOCK1 and LOCK2 tabs in the lower left corner of the figure. Clicking on the **LOCK2** tab navigates to the LOCK2 schematic sheet. For every new schematic sheet added to the design, a new tab displays.

In addition, if you use **Hierarchy** → **Push** to display the schematic for a component or macro, a new tab also displays in the lower left corner.

Simulate Current Macro

In Foundation, you can simulate a macro in a schematic design:

1. Select the macro in your design.
2. Click **Hierarchy** → **Push** and then double click the design.
3. After the design displays, select **Options** → **Simulate Current Macro**. When the Logic Simulator window displays, you can perform a functional simulation of the macro. Refer to the "Functional Simulation" chapter for details.

Chapter 5

Design Methodologies - HDL Flow

This chapter describes various design methodologies supported in the HDL Flow project subtype.

This chapter contains the following sections.

- "HDL Flow Processing Overview"
- "Top-level Designs"
- "All-HDL Designs"
- "HDL Designs with State Machines"
- "HDL Designs with Instantiated Xilinx Unified Library Components"
- "HDL Designs with Black Box Instantiation"
- "Schematic Designs in the HDL Flow"

HDL Flow Processing Overview

Refer to the"Project Toolset" chapter for information on how to create an HDL Flow project and for an overview of the tools available for such projects.

The following figure illustrates the processing performed at the various stages of an HDL Flow project.

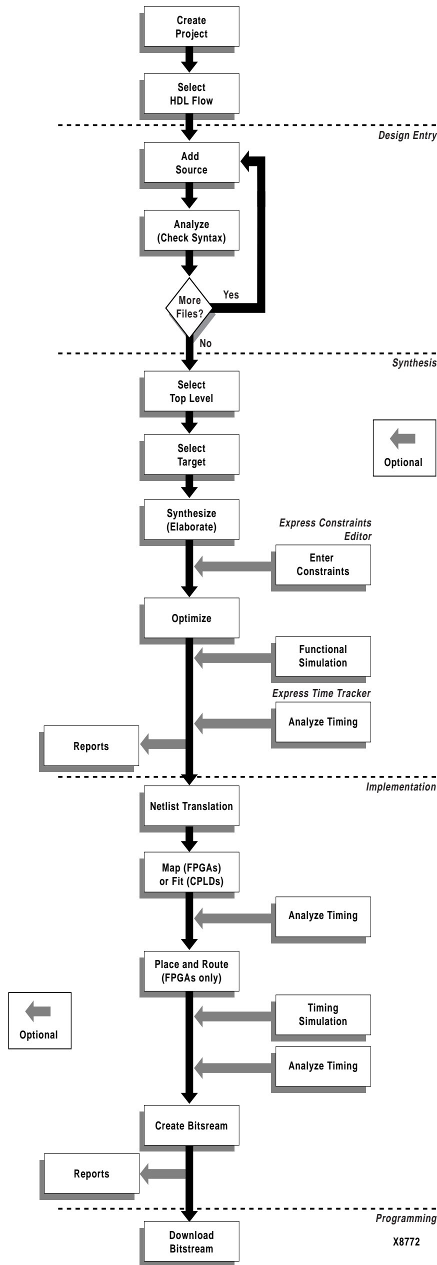


Figure 5-1 HDL Flow Project Processing

Top-level Designs

HDL Flow projects do not require the designation of a top-level design until synthesis. VHDL, Verilog, and schematic files can be added to an HDL Flow project. VHDL and Verilog source files can be created by the HDL Editor, Finite State Machine Editor, or other text editors. When you initiate the synthesis phase, you designate one of the project's entities (VHDL), modules (Verilog), or schematics as the top-level of the design. The list of entities, modules, and schematics is automatically extracted from all the source files added to the project. Synthesis processing starts at the designated top-level file. All modules below the top-level file are elaborated and optimized.

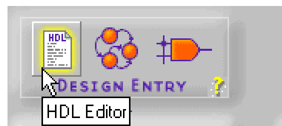
HDL designs can contain underlying LogiBLOXs, CORE Generator modules, and XNF/EDIF files that are instantiated in the VHDL and Verilog code as "black boxes." Black box modules are not elaborated and optimized during synthesis. (Refer to the "HDL Designs with Black Box Instantiation" section for more information on Black Boxes.)

All-HDL Designs

The following procedure describes the HDL flow for designs that are HDL only, that is, there are no schematics or instantiated LogiBLOX, netlist, or state machine macros.

Creating the Design

1. Open the HDL Editor by clicking the HDL Editor icon in the Design Entry box on the Project Manager's Flow tab.



2. When the HDL Editor window appears, you may select an existing HDL file or create a new one. The following steps describe creating a new HDL file with the Design Wizard.

3. When the HDL Editor dialog box displays, select **`Use HDL Design Wizard`**. Click **`OK`**.

4. Click **`Next`** in the Design Wizard window.

5. From the Design Wizard - Language window, select **VHDL** or **Verilog**. Click **Next**.

Note: For top-level ABEL designs, you must use the Schematic Flow.

6. In the Design Wizard - Name window, enter the name of your design file. Click **Next**.
7. Define your ports in the Design Wizard-Ports window by clicking **NEW**, entering the port name, and selecting its direction. Click **Finish**. The Wizard creates the ports and gives you a template (in VHDL or Verilog) in which you can enter your design.
8. Create the design in the HDL Editor. The Language Assistant is available to help with this step. It provides a number of language templates for basic language constructs and synthesis templates for synthesis-oriented implementation of basic functional blocks, such as multiplexers, counters, flip-flops, etc. Access the Language Assistant by selecting **Tools** → **Language Assistant**.
9. Add the design to the project by selecting **Project** → **Add to Project**.
10. Exit the HDL Editor.

For more information about HDL designs, see the“HDL Design Entry and Synthesis” chapter or, in the HDL Editor window, select **Help** → **Help Topics**.

Analyzing Design File Syntax

Syntax is checked automatically when the design is added to the project. You can initiate a syntax check in the HDL Editor by selecting **Synthesis** → **Check Syntax**. You can also analyze syntax by selecting **Project** → **Analyze All Sources** from the Project Manager.

Use the HDL Error and HDL Warnings tabs in the messages area at the bottom of the Project Manager to view any syntax errors or messages output during analysis.

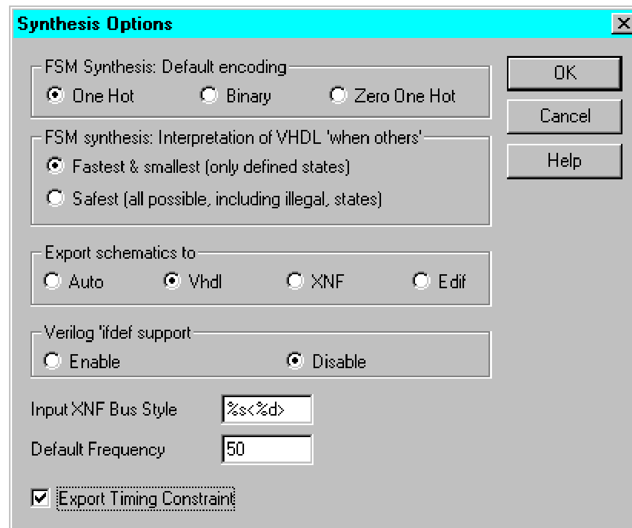
Performing HDL Behavioral Simulation (Optional)

If you installed an HDL simulation tool such as ACTIVE-VHDL or ModelSIM, you can perform a behavioral simulation of your HDL code. Please refer to the documentation provided with these tools for more information.

Synthesizing the Design

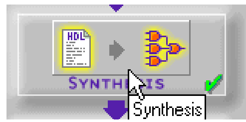
After the design files have been successfully analyzed, the next step is to translate the design into gates and optimize it for a target architecture. These steps are performed by running the Synthesis phase.

1. Set the global synthesis options by selecting **`Synthesis`** → **`Options`** from the Project Manager. In the Synthesis Options dialog, you can set the following defaults:
   - Default clock frequency
   - Export timing constraints to the place and route software
   - Input XNF bus style
   - FSM Encoding (One Hot or Binary)
   - FSM Synthesis Style

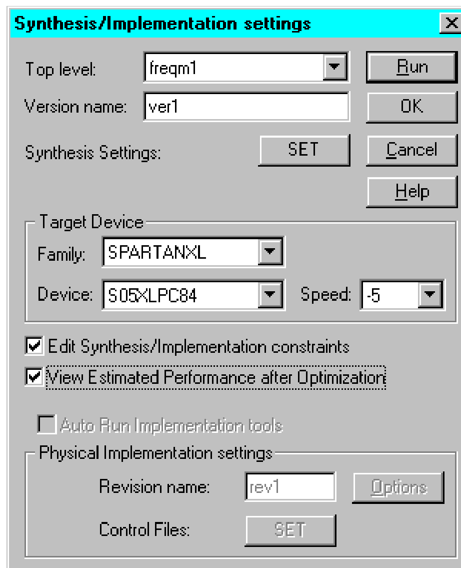


2. Click **`OK`** to close the Synthesis Options dialog

3. Click the Synthesis icon on the **Synthesis** button on the Flow tab.



4. The Synthesis/Implementation dialog box is displayed if this is the first version and revision of a project. (By default on subsequent runs, the same settings are used and the given version is overwritten. To create a new version, or to change settings, select **Project → Create Version**.)



5. Select the name of the top-level module. Processing will start from the file named here and proceed through all its underlying modules.

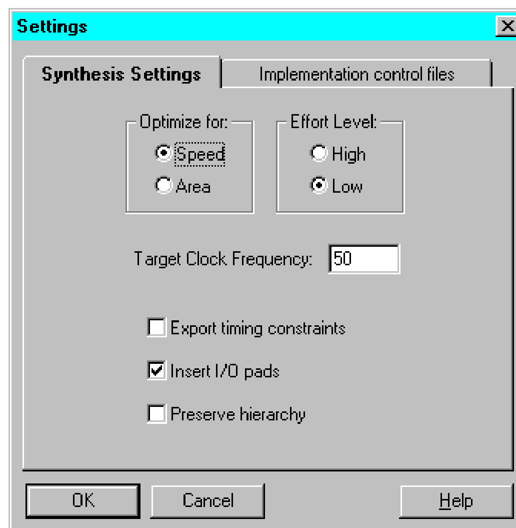
6. Enter a version name.

7. Select the target device.

8. If you have Foundation Express, you have the following two options.

- **`Edit Synthesis/Implementation Constraints`**. Selecting this options pauses synthesis processing after the elaboration phase to allow you to specify constraints for the design using the Express Constraints Editor GUI. For more information refer to the "Express Constraints Editor" section.
- **`View Estimated Performance after Optimization`**. Select this option to view the estimated performance results after design optimization using the Express Time Tracker GUI. For more information refer to the "Express Time Tracker" section.

9. Click **`Set`** to access the Settings dialog box containing Synthesis Setting for this version.



Modify the Synthesis Settings as desired.

- Modify the target clock frequency
- Select the optimization strategy as speed or area
- Select the effort level as high or low
- Select whether I/O pads should be inserted for the designated top-level module

Click **`OK`** to return to the Synthesis/Implementation Settings dialog box.

10. Click **`OK`** to synthesize the designated top-level design and its underlying modules. (Or, click **`Run`** to synthesis and implement the design.)

    The synthesis compiler automatically inserts top-level input and output pads required for implementation (unless instructed not to do so in the Synthesis Settings).

Express Constraints Editor

The Express Constraints Editor is available with the Foundation Express product only. It allows you to set performance constraints and attributes before optimization of FPGA designs.

1. The Express Constraints Editor window automatically displays during Synthesis processing if you checked the Edit Synthesis/Implementation Constraints box on the Synthesis/Implementation dialog.

   Alternatively, you can access the Express Constraints Editor via the Versions tab by right-clicking on the functional structure of a project version or functional structure in the Hierarchy Browser and then selecting **`Edit Synthesis Constraints`**.

   The following figure shows an example of the Clocks tab of the Express Constraints Editor.

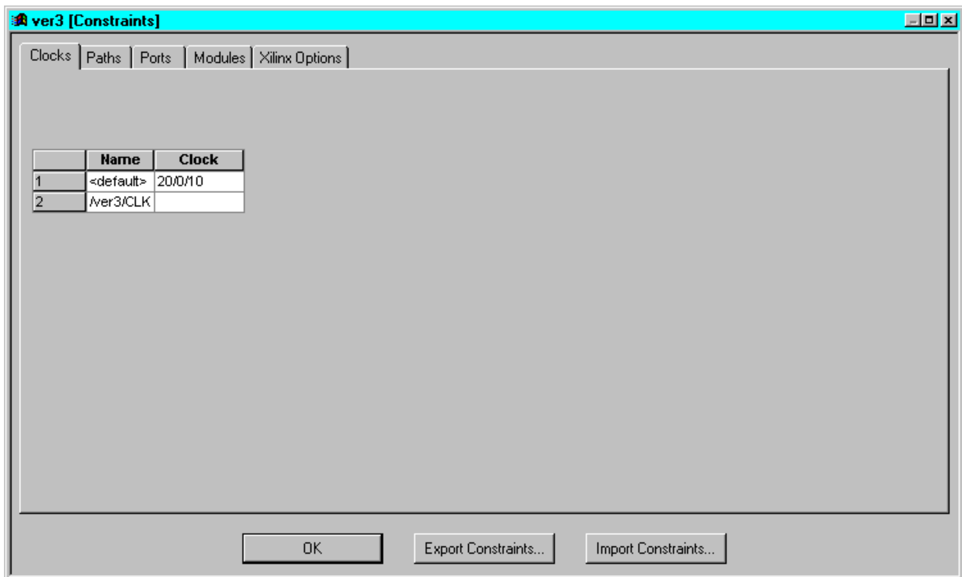


Figure 5-2 Express Constraints Editor - Clocks Tab

2. Design-specific information is extracted from the design and displayed in device-specific spreadsheets. Click the tabs to access the various spreadsheets.

   If you unchecked `Insert I/O pads` on the Synthesis/Implementation dialog, only the Modules and Xilinx Options tabs are shown. The Clocks, Ports, and Paths tabs apply only to top-level HDL designs.

3. Right-click on an item in any of the spreadsheets to edit the value, access a dialog box to edit the value, or access a pulldown menu to select a value. Use the online help in the dialog boxes to understand and enter specific constraints and options.

   The following figure shows an example of the dialog box accessed when you right click on an output delay value displayed on the Ports tab of the Express Constraints Editor.

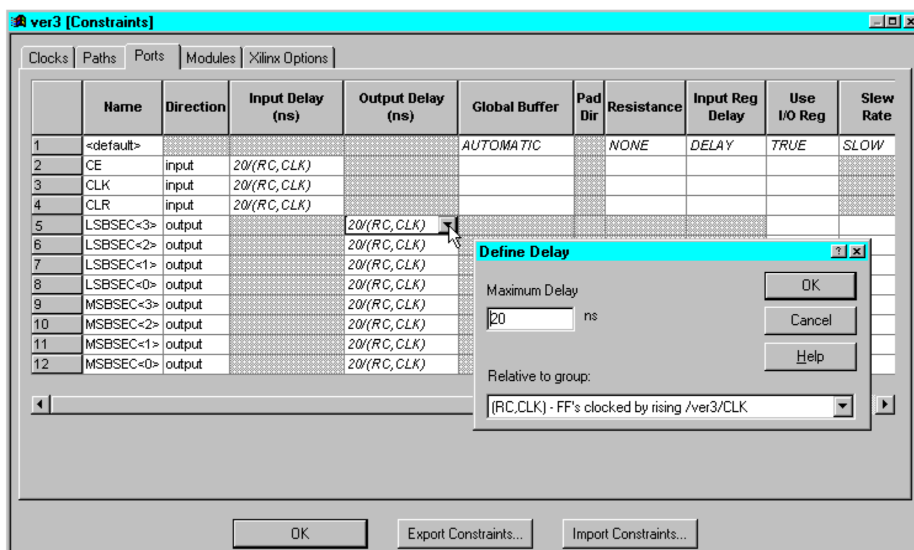


Figure 5-3 Express Constraints Editor - Ports Tab

4. Optionally, you can import a constraints file (.exc) to use now (click **`Import Constraints`**) or you can export the entered constraints to a constraints file (.exc) for reuse (click **`Export Constraints`**).

5. After you finish editing the constraints, click **`OK`** to close the Constraints window and continue the synthesis using the specified constraints.

Express Time Tracker

The Express Time Tracker is available with the Foundation Express product only. It allows you view estimated performance results after optimization of your design.

1. The Optimized (Constraints) window, shown in the figures at the end of this section, automatically displays after Synthesis processing if you checked the View Estimated Performance after Optimization box in the Synthesis/Implementation dialog window.

   Alternatively, you can access the Optimized (Constraints) window via the Versions tab by right-clicking on an optimized

structure in the Hierarchy Browser and then selecting **View Synthesis Results**.

2. Click the tabs to access the performance results in the various spreadsheets.

   If you unchecked **Insert I/O pads** on the Synthesis/Implementation dialog, only the Models and Xilinx Options tabs are shown. The Clocks, Ports, and Paths tabs apply only to top-level HDL designs.

3. After you finish viewing the results, click **OK** to close the Optimized (Constraints) window.

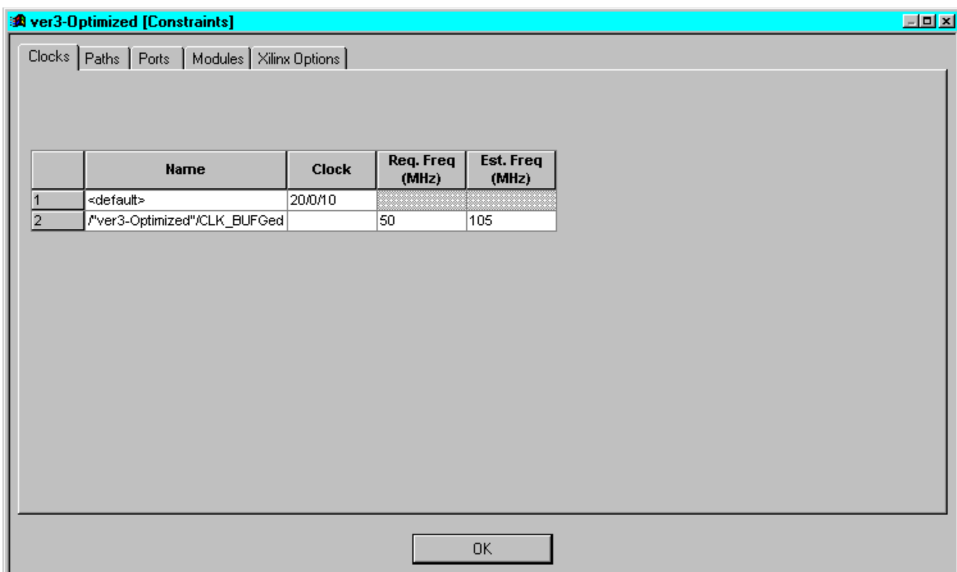


Figure 5-4 Express Time Tracker - Clocks Tab

ver3-Optimized [Constraints]

Clocks | Paths | Ports | Modules | Xilinx Options

	Name	Direction	Input Delay (ns)	Input Slack	Output Delay (ns)	Output Slack	Global Buffer	Pad Dir	Resistance	Input Reg Delay	Use I/O Reg	Slew Rate	Pad L
1	<default>						AUTOMATIC		NONE	DELAY	TRUE	SLOW	
2	CE	input	0/(O)	N/A									
3	CLK	input	0/(O)	N/A			BUFG						
4	CLR	input	0/(O)	N/A									
5	LSBSEC<3>	output			20/(RC,CLK_BUFGed)	11.0							
6	LSBSEC<2>	output			20/(RC,CLK_BUFGed)	10.7							
7	LSBSEC<1>	output			20/(RC,CLK_BUFGed)	10.7							
8	LSBSEC<0>	output			20/(RC,CLK_BUFGed)	10.5							
9	MSBSEC<3>	output			20/(RC,CLK_BUFGed)	11.0							
10	MSBSEC<2>	output			20/(RC,CLK_BUFGed)	10.7							
11	MSBSEC<1>	output			20/(RC,CLK_BUFGed)	10.7							
12	MSBSEC<0>	output			20/(RC,CLK_BUFGed)	10.5							

OK

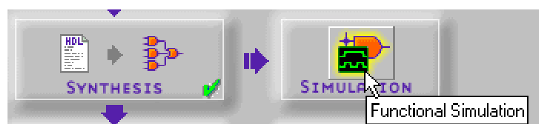
Figure 5-5 Express Time Tracker - Ports Tab

Performing Functional Simulation

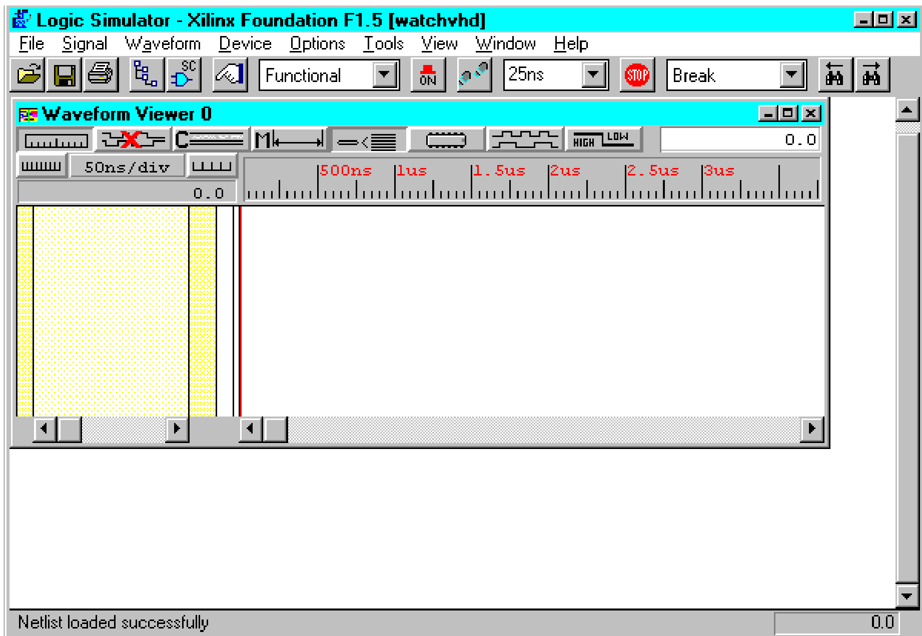
Functional Simulation may be performed to verify that the logic you created is correct. Gate-level functional simulation is performed after the design is synthesized.

Note: There are several ways to apply stimulus and simulate a design. This section discusses one way: using the stimulator dialog. For more information on using the simulator, refer to its online help.

1. Open the Logic Simulator by clicking the Functional Simulation icon in the Simulation box on the Project Manager's Flow tab.



2. The design is automatically loaded into the simulator. The Waveform Viewer window displays inside of the Logic Simulator window.



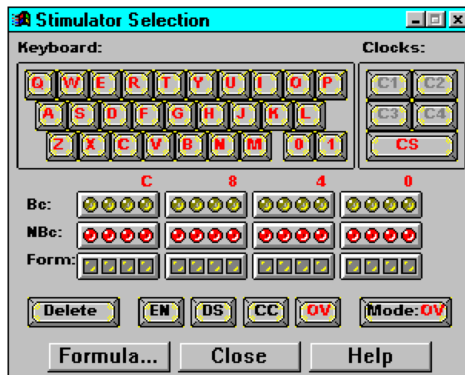
3. Add signals by selecting **`Signal`** → **`Add Signals`**.

4. From the Signals Selection portion of the Components Selection for Waveform Viewer window, select the signals that you want to see in the simulator.

5. Use CTRL-click to select multiple signals. Make sure you add output signals as well as input signals.

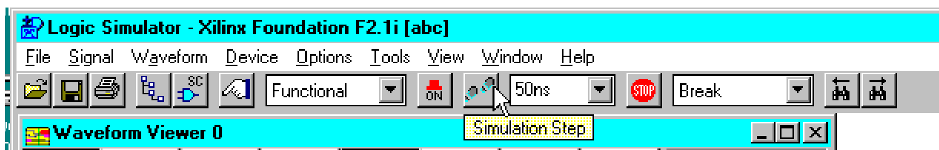
6. Click **`Add`** and then **`Close`**. The signals are added to the Waveform Viewer in the Logic Simulator screen.

7. Select **`Signal`** → **`Add Stimulators`** from the Logic Simulator menu. The Stimulator Selection window displays.



8. In the Stimulator Selection window, create the waveform stimulus by attaching stimulus to the inputs. For more details on how to use the Stimulus Selection window, click **`Help`**.

9. After the stimulus has been applied to all inputs, click the Simulator Step icon on the Logic Simulator toolbar to perform a simulation step. The length of the step can be changed in the Simulation Step Value box to the right of the Simulation Step box. (If the Simulator window is not open, select **`View`** → **`Main Toolbar`**.)



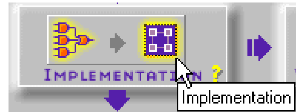
10. To save the stimulus for future viewing or reuse, select **`File`** → **`Save Waveform`**. Enter a file name with a .tve extension in the File name box of the Save Waveform window. Click **`OK`**.

    For more information about saving and loading test vectors, select **`Help`** → **`Logic Simulator Help Contents`** from the Logic Simulator window. From the Help Index, select **`Working With Waveforms`** → **`Saving and Loading Waveforms`**.

Implementing the Design

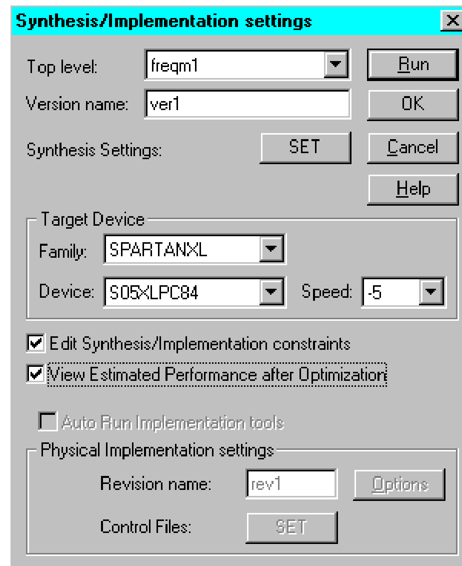
Design Implementation is the process of translating, mapping, placing, routing, and generating a Bit file for your design. Optionally, it can also generate post-implementation timing data.

1. Click the Implementation icon on the Implementation phase button on the Project Manager's Flow tab.

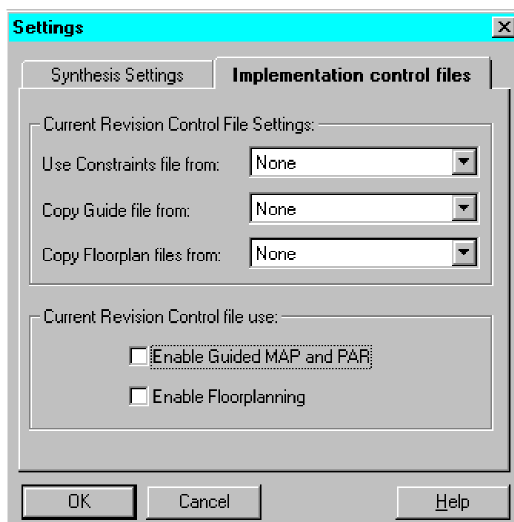


2. The Synthesis/Implementation dialog box appears if the implementation is out-of-date.

   A revision represents an implementation run on the selected-version. Modify the name in the Revision Name box, if desired. The synthesis settings are grayed out if synthesis has already been run.

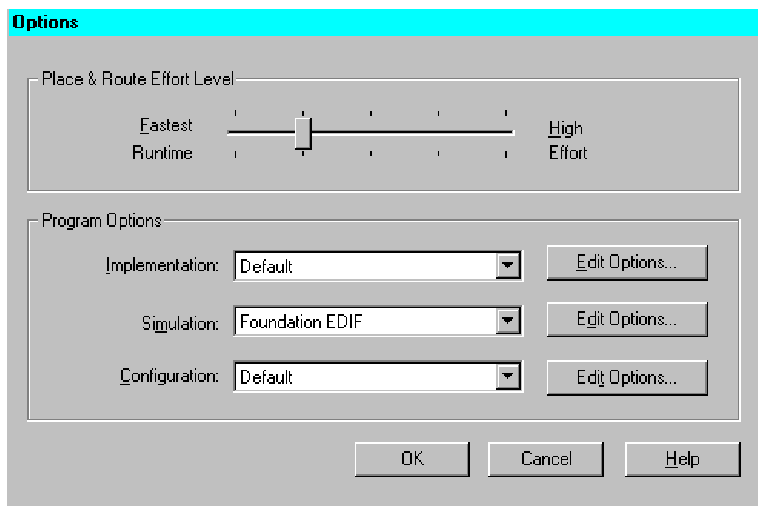


3. Click `Set` to access the Implementation control files dialog box. Identify any guide file, constraints file, or Floorplan file to use for this implementation.



Click **OK** to return to the Synthesis/Implementation Settings dialog box.

4. Click **Options** on the Synthesis/Implementation dialog box to set the Place and Route Effort level and edit implementation, simulation, or configuration options, if desired.



Click **OK** to return to the Settings/Implementation Settings dialog box.

5. Click **`Run`** to implement the design. The Flow Engine displays the progress of the implementation.

   The Project Manager displays a status message when Implementation is complete. View the Console tab on the Project Manager window for the results of all stages of the implementation. The Versions tab also indicates the status of the implemented revision.

6. To view design reports, select the desired revision in the Versions tab of the Project Manager. Then select the Reports tab in the Project Manager Flow window.

   Click on the Implementation Report Files icon to view the implementation reports. Click on the Implementation Log File icon to view the Flow Engine's processing log.

   For more information on the Flow Engine, select **`Help`** → **`Foundation Help Contents`** → **`Flow Engine`**.

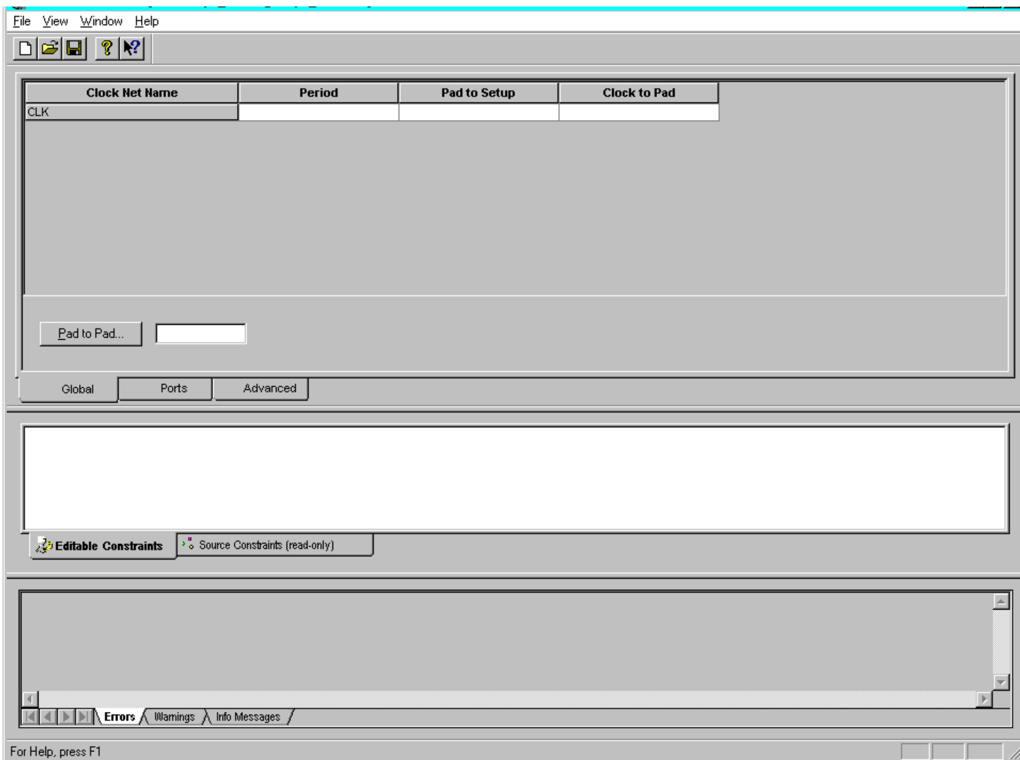
Editing Implementation Constraints

Design constraints affect how the logical design is implemented in the target device. Applying constraints helps you to adapt your design's performance to expected worst-case conditions. The user constraint file (.ucf) is an ASCII file that holds timing and location constraints. It is read (by NGDBuild) during the translate process in the Flow Engine and is combined with an EDIF or XNF netlist into an NGD file.

Each revision contains an associated UCF file. The UCF file may be a default (empty) UCF or one that you customize yourself. You can directly enter constraints in the UCF file through a text editor or you can use the Xilinx Constraints Editor.

1. The Constraints Editor is a Graphical User Interface (GUI) that you can run after the Translate program to create new constraints in a UCF file. To access the Constraints Editor, select **`Tools`** → **`Implementation`** → **`Constraints Editor`** from the Project Manager.

   The following figure shows an example of the Global tab of the Implementation Constraints Editor.



2. Design-specific information is extracted from the design and displayed in device-specific spreadsheets. Click the tabs to access the various spreadsheets.

3. Right-click on an item in any of the spreadsheets to access a dialog box to edit the value. Use the online help in the dialog boxes to understand and enter specific constraints and options. Or, refer to the online software document, *Constraints Editor Guide* for detailed information.

   The following figure shows an example of the Pad to Setup dialog box accessed when you right click anywhere on a Port row on the Ports tab of the Implementation Constraints Editor and then select `Pad to Setup`.

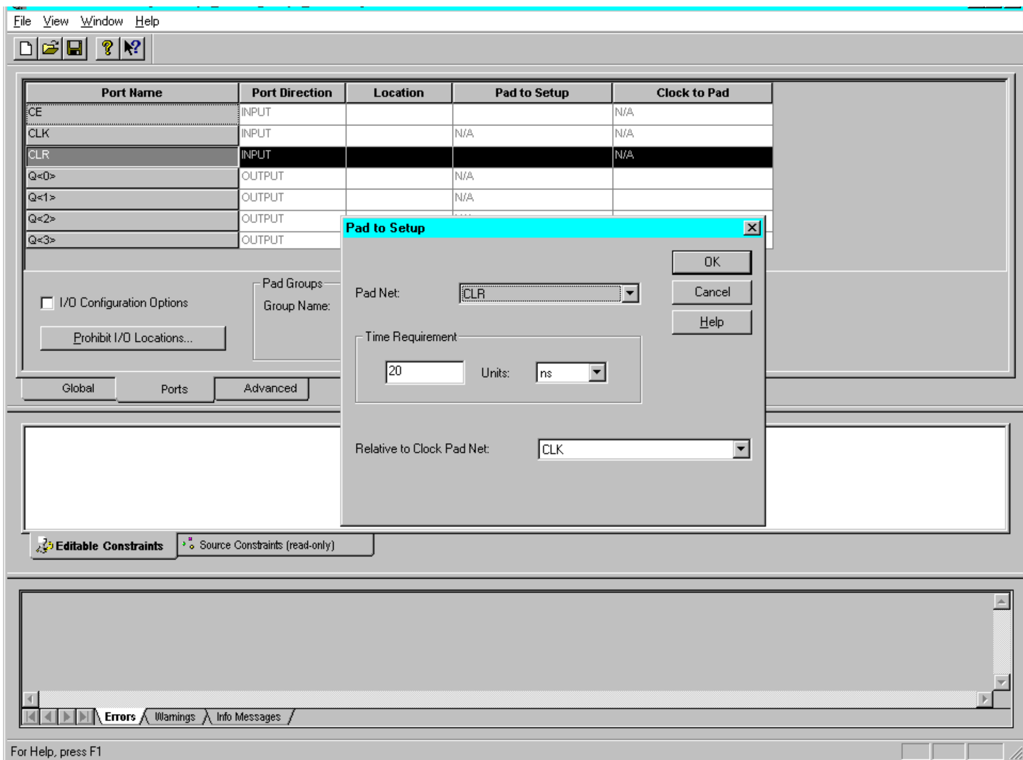


Figure 5-6 Implementation Constraints Editor - Ports Tab

4. After you finish editing the constraints, click `Save` to close the Constraints Editor window
5. You must rerun the Translate step in the Flow Engine to have your new constraints applied to the design.
6. Click the Implementation icon on the Project Manager's Flow tab to rerun Translate and the rest of the flow.

Verifying the Design

After the design has been implemented, the Timing Analyzer or the Timing Simulator can be used to verify the design. The Timing Analyzer performs a static timing analysis of the design. The Timing Simulator uses worst-case delays and user input stimulus to simulate the design.

Performing a Static Timing Analysis

1. Click the Timing Analyzer icon in the Verification box on the Project Manager's Flow tab to perform a static timing analysis.

2. For FPGAs, you can perform a post-MAP, post-place, or post-route timing analysis to obtain timing information at various stages of the design implementation. You can perform a post-implementation timing analysis on CPLDs after a design has been fitted.

For details on how to use the Timing Analyzer, select **Help** → **Foundation Help Contents** → **Timing Analyzer**.

Performing a Timing Simulation

1. Open the Timing Simulator by clicking the Timing Simulation icon in the Verification box on the Project Managers's Flow tab. The implementation timing netlist with worst-case delays will be loaded into the simulator.

   The Waveform Viewer window displays inside the Logic Simulator window.

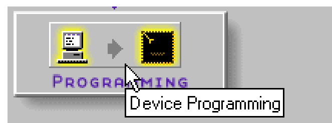
2. Refer to the "Performing Functional Simulation" section earlier in this chapter for instructions on simulating the design. (The

operation of the simulator is the same for functional and timing simulation.)

3. If you have already saved test vectors (for instance, in the functional simulation), you may load these vectors into the timing simulator by selecting **File** → **Load Waveform**.

Programming the Device

1. Click the Device Programming icon in the Programming box on the Project Manager's Flow tab.



2. From the Select Program box, choose the Hardware Debugger, the PROM File Formatter, or the JTAG Programmer. For CPLD designs, use the JTAG Programmer. For instructions, select **Help** → **Foundation Help Contents** → **Advanced Tools** → **JTAG Programmer**. For FPGA designs, use the JTAG Programmer, Hardware Debugger, or PROM File Formatter. For instructions, select **Help** → **Foundation Help Contents** → **Advanced Tools** and then select the desired tool.

HDL Designs with State Machines

This section explains how to create a state machine and add it into a HDL Flow project.

The Files tab in the Hierarchy Browser displays the state machine name. HDL code is automatically generated from the FSM diagram. The module (VHDL) or entity (Verilog) name is automatically added to the top-level selection list.

Creating a State Machine Macro

1. Open the State Editor by clicking the FSM icon in the Design Entry box on the Project Manager's Flow tab.
2. Select **Use the HDL Design Wizard**. Click **OK**.
3. From the Design Wizard window, select **Next**.

4. From the Design Wizard - Language window, choose **`VHDL`** or **`Verilog`** and select **`Next`**.
5. In the Design Wizard - Name window, enter a name for your macro. Select **`Next`**.
6. Define your ports in the Design Wizard-Ports window. Select **`Next`**.
7. In the Design Wizards - Machines window, select the number of state machines that you want. Click **`Finish`**. The Wizard creates the ports and gives you a template in which you can enter your macro design.
8. Complete the design for your FSM in the State Editor.
9. Add the macro to the project by selecting **`Project`** → **`Add to Project`** from the Project Manager.

   You will see the FSM module listed in the Files tab of the Project Manager.

   Following is an example of VHDL code (my_fsm.vhd) generated from the State Editor for a state machine macro.

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

entity my_fsm is
  port (clk: in STD_LOGIC;
        in_a: in STD_LOGIC;
        in_b: in STD_LOGIC;
        in_c: in STD_LOGIC;
        reset: in STD_LOGIC;
        out_a: out STD_LOGIC;
        out_b: out STD_LOGIC;
        out_c: out STD_LOGIC);
end;

architecture my_fsm_arch of my_fsm is

-- SYMBOLIC ENCODED state machine: Sreg0
type Sreg0_type is (S1, S2, S3);
signal Sreg0: Sreg0_type;
```

```
begin
--concurrent signal assignments
--diagram ACTIONS

process (clk)

begin

if clk'event and clk = '1' then
 if reset='1' then
  Sreg0 <= S1;
 else
 case Sreg0 is
  when S1 =>
   if in_a = '1' then
    Sreg0 <= S2;
   end if;
  when S2 =>
   if in_b = '1' then
    Sreg0 <= S3;
   end if;
  when S3 =>
   if in_c = '1' then
    Sreg0 <= S1;
   end if;
  when others =>
   null;
 end case;
 end if;
end if;
end process;

-- signal assignment statements for combinatorial
-- outputs
out_c <= '0' when (Sreg0 = S2) else
         '0' when (Sreg0 = S3) else
         '1';

out_a <= '1' when (Sreg0 = S2) else
         '0' when (Sreg0 = S3) else
         '0';

out_b <= '0' when (Sreg0 = S2) else
         '1' when (Sreg0 = S3) else
         '0';

end my_fsm_arch;
```

For more information about creating state machine modules, refer to the"State Machine Designs" chapter. Or, select **Help** → **Foundation Help Contents** and then Click **State Editor**.

HDL Designs with Instantiated Xilinx Unified Library Components

It is possible to instantiate certain Xilinx Unified Library components directly into your VHDL or Verilog code. In general, you will find this most useful for components that the Express compiler is unable to infer, such as BSCAN, RAM, and certain types of special Xilinx components. The "Instantiated Components" appendix lists the most commonly instantiated components, including descriptions of their function and pins.

When instantiating Unified Library components, the component must first be declared before the `begin` keyword in VHDL the architecture and then may be instantiated multiple times in the body of the architecture.

The following example shows how to instantiate the STARTUP component in a VHDL file, which in turn allows use of the dedicated GSR (global set/reset) net.

The following sample written in VHDL shows an example of an instantiated Xilinx Unified Library component, STARTUP.

library IEEE;
use IEEE.std_logic_1164.all;

```
entity gsr_test is
   port (
      CLK: in STD_LOGIC;
      D_IN: in STD_LOGIC;
      RESET: in STD_LOGIC;
      Q_OUT: out STD_LOGIC
   );
end gsr_test;

architecture gsr_test_arch of gsr_test is
component STARTUP
   port (GSR: in std_logic);
end component;
```

```
begin

U1: STARTUP  port map (GSR=>RESET);

process (CLK)
begin
   if (CLK event and CLK='1') then
      Q_OUT <= D_IN;
   end if;
end process;

end gsr_test_arch;
```

1. The HDL code must be added to the project. Select **Project** → **Add to Project** from the HDL Editor or select **Document** → **Add** from the Project Manager.
2. Synthesize the design by selecting the **Synthesis** button on the Project Manager Flow tab.The synthesizer will automatically include top level input and output pads for the designated top-level design.

   For more information about HDL designs, see the "HDL Design Entry and Synthesis" chapter or, in the HDL Editor window, select **Help** → **Help Topics**.
3. To complete the design, refer to the "Synthesizing the Design"through the "Programming the Device" sections under the "All-HDL Designs" section.

HDL Designs with Black Box Instantiation

LogiBLOXs, CORE Generator modules, ABEL modules, and EDIF and XNF files can be instantiated in the VHDL and Verilog code using the "black box instantiation" method.

The Files tab in the Hierarchy Browser does *not* display the black box module name under the HDL file(s) in which it is instantiated. The Express compiler does not synthesize the black box. It is left as an unlinked cell and resolved in the Translate phase of the implementation.

This section describes how to create HDL designs that instantiate black boxes.

LogiBLOX Modules in a VHDL or Verilog Design

LogiBLOX modules may be generated in Foundation and then instantiated in the VHDL or Verilog code. This flow may be used for any LogiBLOX component, but it is especially useful for memory components such as RAM. Never describe RAM behaviorally in the HDL code, because combinatorial feedback paths will be inferred.

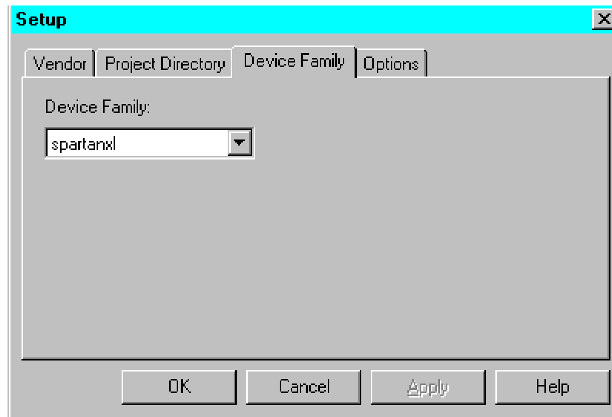
The module being instantiated must be located in the HDL project directory (that is, the directory where the top-level HDL file resides). Running LogiBLOX from the Foundation project ensures this condition is met.

LogiBLOX provides a template tool for generating the VHDL or Verilog component declaration statement.

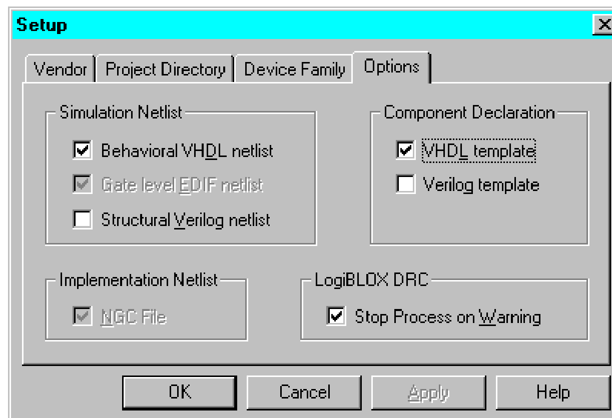
VHDL Instantiation

This section explains how to instantiate a LogiBLOX module into a VHDL design using Foundation. The example described below creates a RAM48X4S using LogiBLOX.

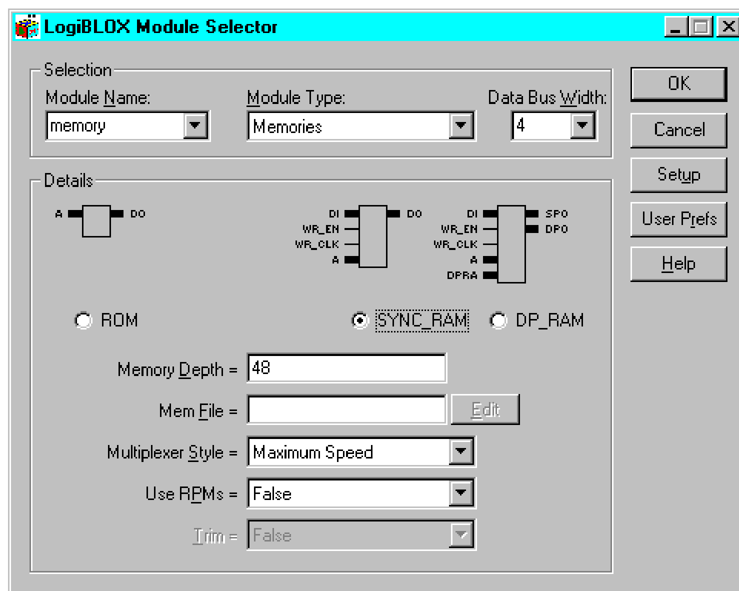
1. Access the LogiBLOX Module Selector window using one of the following methods. Its operation is the same regardless of where it is invoked.
   - From the Project Manger, select **`Tools`** → **`Design Entry`** → **`LogiBLOX module generator`**
   - From the HDL Editor, select **`Tools`** → **`LogiBLOX`**
   - From Schematic Editor, select **`Tools`** → **`LogiBLOX Module Generator`**
2. Click **`Setup`** on the LogiBLOX Module Selector screen. (The first time LogiBLOX is invoked, the Setup screen appears automatically.)
3. In the Setup window, enter the following items.
   - Under the Device Family tab, use the pulldown list to select the target device family (SpartanXL, for example).



- Under the Options tab, select the Simulation Netlist and Component Declaration template.To instantiate the LogiBLOX module in VHDL code, select **`VHDL template`** in the Component Declaration area. If you plan to perform a behavioral simulation, select **`Behavioral VHDL netlist`** in the Simulation Netlist area, as shown below. Click **`OK`**.



4. In the LogiBLOX Module Selector window, define the type of LogiBLOX module and its attributes. The Module Name specified here is used as the name of the instantiation in the VHDL code.



5. When you click **OK**, the LogiBLOX module is created automatically and added to the project library.

   The LogiBLOX module is a collection of several files including those listed below. The files are located in your Xilinx project directory for the current project.

component_name.ngc	Netlist used during the Translate phase of Implementation
component_name.vhi	Instantiation template used to add a LogiBLOX module into your VHDL source code
component_name.vhd	VHDL file used for functional simulation
component_name.mod	Configuration information for the module
logiblox.ini	LogiBLOX configuration for the project

   The component name is the name given to the LogiBLOX module in the GUI. The port names are the names provided in the .vhi file.

6. In the HDL Editor, open the LogiBLOX-created .vhi file (memory.vhi) located under the current project. The .vhi file for the memory component created in the previous steps is shown below.

```
-----------------------------------------------
-- LogiBLOX SYNC_RAM Module "memory"
-- Created by LogiBLOX version C.16
--    on Tue Jun 01 16:46:04 1999
-- Attributes
--    MODTYPE = SYNC_RAM
--    BUS_WIDTH = 4
--    DEPTH = 48
--    STYLE = MAX_SPEED
--    USE_RPM = FALSE
-----------------------------------------------
-----------------------------------------------
-- Component Declaration
-----------------------------------------------
component memory
PORT(
A: IN std_logic_vector(5 DOWNTO 0);
DO: OUT std_logic_vector(3 DOWNTO 0);
DI: IN std_logic_vector(3 DOWNTO 0);
WR_EN: IN std_logic;
WR_CLK: IN std_logic);
end component;

-----------------------------------------------
-- Component Instantiation
-----------------------------------------------
instance_name : memory port map
(A => ,
DO => ,
DI => ,
WR_EN => ,
WR_CLK => );
```

7. Open a second session of the HDL Editor. In the second HDL Editor window, open the VHDL file in which the LogiBLOX component is to be instantiated.

Note: Instead of opening a second sesssion, you could use **`Edit`** → **`Insert File`** from the HDL Editor tool bar to insert the file into the current HDL Editor session.

Cut and paste the Component Declaration from the LogiBLOX component's .vhi file to your project's VHDL code, placing it after the architecture statement in the VHDL code.

Cut and past the Component Instantiation from the LogiBLOX component's .vhi file to your VHDL design code after the "begin" line. Give the inserted code an instance name. Edit the code to connect the signals in the design to the ports of the LogiBLOX module.

The VHDL design code with the LogiBLOX instantiation for the component named *memory* is shown below. For each .ngc file from LogiBLOX, you may have one or more VHDL files with the .ngc file instantiated. In this example, there is only one black box instantiation of memory, but multiple calls to the same module may be done.

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

entity top is
port (          D: in STD_LOGIC; CE: in STD_LOGIC;
              CLK: in STD_LOGIC; Q: out STD_LOGIC;
             Atop: in STD_LOGIC_VECTOR (5 downto 0);
            DOtop: out STD_LOGIC_VECTOR (3 downto 0);
            DItop: in STD_LOGIC_VECTOR (3 downto 0);
         WR_ENtop: in STD_LOGIC;
        WR_CLKtop: in STD_LOGIC);
end top;

architecture inside of top is

component userff
port (    D: in STD_LOGIC; CE: in STD_LOGIC;
        CLK: in STD_LOGIC; Q:  out STD_LOGIC);
end component;

component memory
port (    A:  in STD_LOGIC_VECTOR (5 downto 0);
          DI:  in STD_LOGIC_VECTOR (3 downto 0);
          WR_EN:  in STD_LOGIC;
          WR_CLK:   in STD_LOGIC;
          DO:  out STD_LOGIC_VECTOR (3 downto 0));
end component;
```

```
begin

UO:userff port map (D=>D, CE=>CE, CLK=>CLK, Q=>Q);

U1:memory port map(A=>Atop,DI=>DItop,WR_EN=>WR_ENtop,
                   WR_CLK=>WR_CLKtop, DO=>DOtop);
end inside;
```

8. Check the syntax of the VHDL design code by selecting **`Synthesis`** → **`Check Syntax`** in the HDL Editor. Correct any errors. Then save the design and close the HDL Editor.
9. The design with the instantiated LogiBLOX module can then be synthesized (click the **`Synthesis`** button on the Flow tab).

Note: When the design is synthesized, a warning is generated that the LogiBLOX module is unlinked. Modules instantiated as black boxes are not elaborated and optimized. The warning message is just reflecting the black box instantiation.

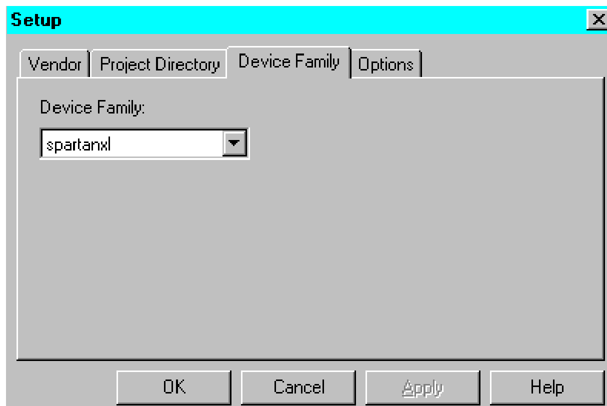
10. To complete the design, refer to the “Synthesizing the Design”through the “Programming the Device” sections under the “All-HDL Designs” section.

Verilog Instantiation

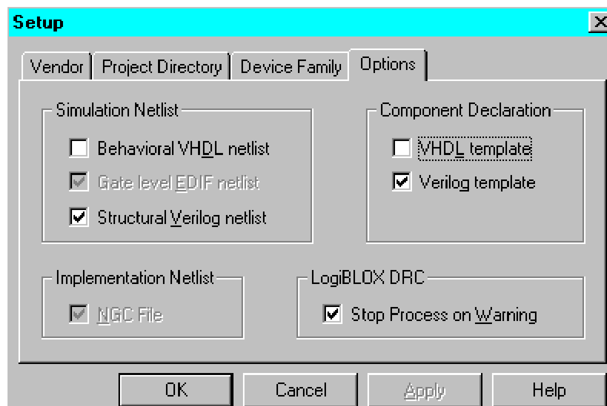
This section explains how to instantiate a LogiBLOX module into a Verilog design using Foundation. The example described below creates a RAM48X4S using LogiBLOX.

1. Access the LogiBLOX Module Selector window using one of the following methods. Its operation is the same regardless of where it is invoked.
   - From the Project Manger, select **`Tools`** → **`Design Entry`** → **`LogiBLOX module generator`**.
   - From the HDL Editor, select **`Tools`** → **`LogiBLOX`**.
   - From Schematic Editor, select **`Tools`** → **`LogiBLOX Module Generator`**.
2. Click **`Setup`** on the LogiBLOX Module Selector screen. (The first time LogiBLOX is invoked, the Setup screen appears automatically.)
3. In the Setup window, enter the following items.

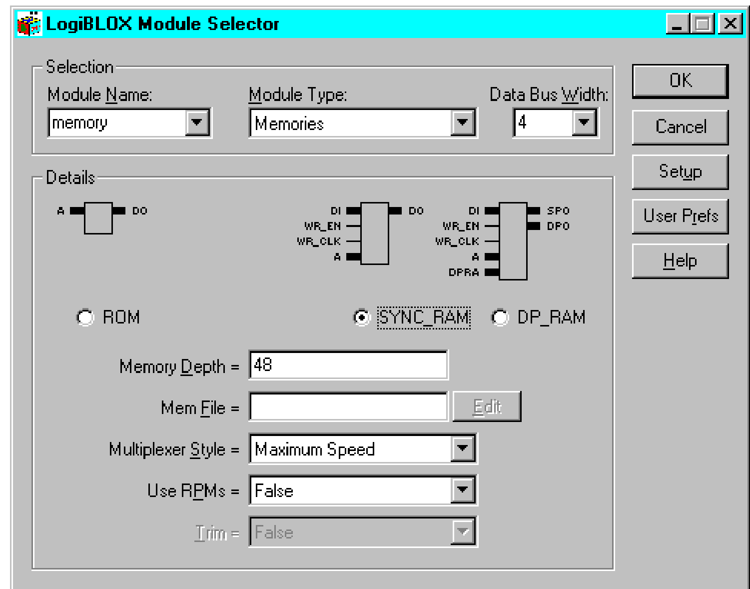
- Under the Device Family tab, use the pulldown list to select the target device family (XC4000E, for example).



- Under the Options tab, select **`Verilog template`** in the Component Declaration area. If you plan to perform a behavioral simulation, select **`Structural Verilog netlist`** in the Simulation Netlist area, as shown below. Click **`OK`**.



4. In the LogiBLOX Module Selector window, define the type of LogiBLOX module and its attributes. The Module Name specified here is used as the name of the instantiation in the Verilog code.



5. When you click **OK**, the LogiBLOX module is created automatically and added to the project library.

   The LogiBLOX module is a collection of several files including those listed below. The files are located in your Xilinx project directory for the current project.

component_name.ngc	Netlist used during the Translate phase of Implementation
component_name.vei	Instantiation template used to add LogiBLOX module into your Verilog source code
component_name.v	Verilog file used for functional simulation
component_name.mod	Configuration information for the module
logiblox.ini	LogiBLOX configuration for the project

   The component name is the name given to the LogiBLOX module in the GUI. The port names are the names provided in the .vei file.

6. In the HDL Editor, open the LogiBLOX- created .vei file (memory.vei) located under the current project. The .vei file for the memory component created in the previous steps is shown below.

```
//----------------------------------------------------
// LogiBLOX SYNC_RAM Module "memory"
// Created by LogiBLOX version C.16
//    on Wed Jun 01 10:40:25 1999
// Attributes
//    MODTYPE = SYNC_RAM
//    BUS_WIDTH = 4
//    DEPTH = 48
//    STYLE = MAX_SPEED
//    USE_RPM = FALSE
//----------------------------------------------------
memory instance_name
( .A(),
  .DO(),
  .DI(),
  .WR_EN(),
  .WR_CLK());

module memory(A, DO, DI, WR_EN, WR_CLK);
input [5:0] A;
output [3:0] DO;
input [3:0] DI;
input WR_EN;
input WR_CLK;
endmodule
```

7. Open a second session of the HDL Editor. In the second HDL Editor window, open the Verilog design file in which the Logi-BLOX component is to be instantiated.

Note: Instead of opening a second sesssion, you could use **`Edit`** → **`Insert File`** from the HDL Editor tool bar to insert the file into the current HDL Editor session.

Cut and paste the module declaration from the LogiBLOX component's .vei file into the Verilog design code, placing it after the "endmodule" line within the architecture section or the Verilog design code.

Cut and paste the component instantiation from the .vei file into the design code. Give the added code an instance name and edit it to connect the ports to the signals.

The Verilog design code with the LogiBLOX instantiation for the component named *memory* is shown below. For each .ngc file from LogiBLOX, you may have one or more VHDL files with the .ngc file instantiated. In this example, there is only one black box instantiation of memory, but multiple calls to the same module may be done.

```
module top (D,CE,CLK,Q,
            Atop, DOtop, DItop, WR_ENtop, WR_CLKtop);

input D;
input CE;
input CLK;
output Q;

input  [5:0] Atop;
output  [3:0] DOtop;
input  [3:0] DItop;
input WR_ENtop;
input WR_CLKtop;

userff U0  (.D(D),.CE(CE),.CLK(CLK),.Q(Q));

memory U1  (   .A(Atop),
               .DO (DOtop),
               .DI (DItop),
               .WR_EN (WR_ENtop),
               .WR_CLK (WR_CLKtop));
endmodule
```

Note: An alternate method is to place the module declaration from the .vei file into a new, empty Verilog file (MEMORY.V) and add the new file (shown below) to the project.

```
//-----------------------------------------------------
// LogiBLOX SYNC_RAM Module "memory"
// Created by LogiBLOX version C.16
//    on Wed Jun 01 10:40:25 1999
// Attributes
//    MODTYPE = SYNC_RAM
//    BUS_WIDTH = 4
//    DEPTH = 48
```

```
//    STYLE = MAX_SPEED
//    USE_RPM = FALSE
//-----------------------------------------------------
module MEMORY (A, DO, DI, WR_EN, WR_CLK);
input [5:0] A;
output [3:0] DO;
input [3:0] DI;
input WR_EN;
input WR_CLK;
endmodule
```

8. Check the syntax of the Verilog design code by selecting **Synthesis→ Check Syntax** in the HDL Editor. Correct any errors and then save the design and close the HDL Editor.
9. The design with the instantiated LogiBLOX module can then be synthesized (click the **Synthesis** button on the Flow tab).

Note: When the design is synthesized, a warning is generated that the LogiBLOX module is unlinked. Modules instantiated as black boxes are not elaborated and optimized. The warning message is just reflecting the black box instantiation.

10. To complete the design, refer to the "Synthesizing the Design" section through the "Programming the Device" section under the "All-HDL Designs" section in this chapter.

CORE Generator COREs in a VHDL or Verilog Design

CORE Generator COREs may be generated in Foundation and then instantiated in VHDL or Verilog code. COREs can be generated for valid Foundation projects only.

This flow may be used for any CORE Generator CORE. The CORE being instantiated must be located in the HDL project directory (that is, the directory where the top-level HDL file resides). Running Logi-BLOX from the Foundation project ensures this condition is met.

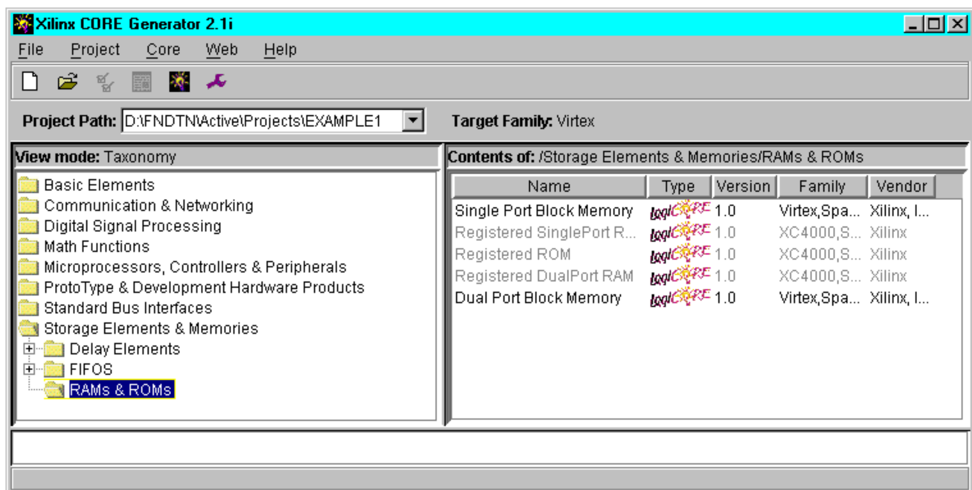
VHDL Instantiation

This section explains how to instantiate a CORE component into a VHDL design using Foundation.

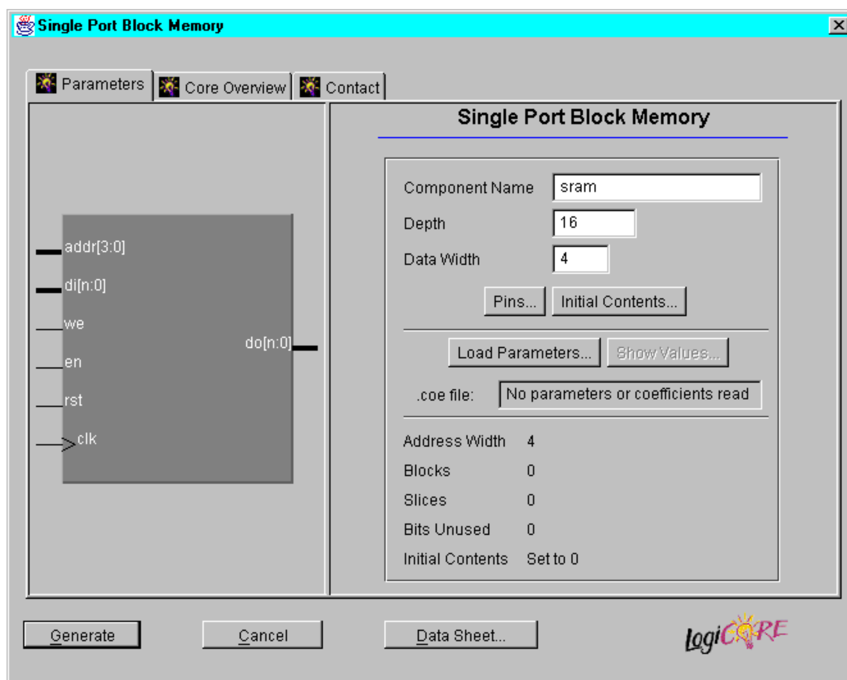
1. With a valid Foundation project open, access the CORE Generator window using one of the following methods. Its operation is the same regardless of where it is invoked.

- From the Project Manger, select **Tools** → **Design Entry** → **CORE Generator**
- From the HDL Editor or Schematic Editor, select **Tools** → **CORE Generator**

2. Select **Project** → **Project Options**. In the Project Options dialog box, ensure that Design Entry is VHDL, that Behavioral Simulation is VHDL, and that the Vendor is Foundation. The Family entry should reflect the project's target device. Click **OK** to exit the Project Options dialog box.
3. To aid selection, the available COREs are categorized in folders on the View Mode section of the main CORE Generator window. Double click a folder to see its sub-categories. When you double click a sub-category folder, the available COREs are listed in the "Contents of" section of the main CORE Generator window.



4. To select a CORE, double click on the CORE's name in the "Contents of" window. A new window opens to allow you to view a description of the CORE or its data sheet, to customize the CORE for your application, and to generate the customized CORE. (Acrobat Reader is required to view the data sheet.)



5. When the CORE's window appears, enter a name for the component in the Component Name field.

   The name must begin with an alpha character. No extensions or uppercase letters are allowed. After the first character, the name may include numbers and/or the underscore character.

6. Other available customization options are unique for each CORE. Customize the CORE as necessary.

7. Select **`Generate`** to create the customized CORE and add its files to the project directory.

The customized CORE component is a collection of several files including those listed below. The files are located in your Xilinx project directory for the current project.

component_name.coe	ASCII data file defining the coefficient values for FIR filters and initialization values for memory modules
component_name.xco	CORE Generator file containing the parameters used to generate the customized CORE
component_name.edn	EDIF implementation netlist for the CORE
component_name.vho	VHDL template file
component_name.mif	Memory Initialization Module for Virtex Block RAM modules

The component name is the name given to the CORE in the customization window. The port names are the names provided in the .vho file.

An example .vho file is shown below.

```
----------------------------------------------------------------------
-- This file was created by the Xilinx CORE Generator tool, and     --
-- is (c) Xilinx, Inc. 1998, 1999. No part of this file may be      --
-- transmitted to any third party (other than intended by Xilinx)   --
-- or used without a Xilinx programmable or hardwire device without --
-- Xilinx's prior written permission.                               --
----------------------------------------------------------------------

-- The following code must appear in the VHDL architecture header:

------------- Begin Cut here for COMPONENT Declaration ------ COMP_TAG
component sram
port (
addr: IN std_logic_VECTOR(3 downto 0);
clk: IN std_logic;
di: IN std_logic_VECTOR(3 downto 0);
we: IN std_logic;
en: IN std_logic;
rst: IN std_logic;
do: OUT std_logic_VECTOR(3 downto 0));
end component;
```

```
-- COMP_TAG_END ------ End COMPONENT Declaration ------------

-- The following code must appear in the VHDL architecture
-- body. Substitute your own instance name and net names.

------------- Begin Cut here for INSTANTIATION Template ----- INST_TAG
your_instance_name : sram
port map (
addr => addr,
clk => clk,
di => di,
we => we,
en => en,
rst => rst,
do => do);
-- INST_TAG_END ------ End INSTANTIATION Template ------------

-- The following code must appear above the VHDL configuration
-- declaration. An example is given at the end of this file.

------------- Begin Cut here for LIBRARY Declaration -------- LIB_TAG

-- synopsys translate_off

Library XilinxCoreLib;

-- synopsys translate_on

-- LIB_TAG_END ------- End LIBRARY Declaration ------------

-- The following code must appear within the VHDL top-level
-- configuration declaration. Ensure that the translate_off/on
-- compiler directives are correct for your synthesis tool(s).

------------- Begin Cut here for CONFIGURATION snippet ------ CONF_TAG

-- synopsys translate_off

for all : sram use entity XilinxCoreLib.C_MEM_SP_BLOCK_V1_0(behavioral)
generic map(
c_has_en => 1,
c_rst_polarity => 1,
c_clk_polarity => 1,
c_width => 4,
c_has_do => 1,
```

```
c_has_di => 1,
c_en_polarity => 1,
c_has_we => 1,
c_has_rst => 1,
c_address_width => 4,
c_read_mif => 0,
c_depth => 16,
c_pipe_stages => 0,
c_mem_init_radix => 16,
c_default_data => "0",
c_mem_init_file => "sram.mif",
c_we_polarity => 1,
c_generate_mif => 0);
end for;

-- synopsys translate_on

-- CONF_TAG_END ------ End CONFIGURATION snippet ------------

-------------------------------------------------------------
-- Example of configuration declaration...
-------------------------------------------------------------
--
-- <Insert LIBRARY Declaration here>
--
-- configuration <cfg_my_design> of <my_design> is
--     for <my_arch_name>
--         <Insert CONFIGURATION Declaration here>
--     end for;
-- end <cfg_my_design>;
--
-- If this is not the top-level design then in the next level up, the
following text
-- should appear at the end of that file:
--
-- configuration <cfg> of <next_level> is
--     for <arch_name>
--           for all : <my_design> use configuration <cfg_my_design>;
--           end for;
--     end for;
-- end <cfg>;
--
```

8. Select **File** → **Exit** to close the CORE Generator.

9. In the HDL Editor, open the CORE's .vho file (component_name.vho) located under the current project.
10. Open a second session of the HDL Editor. In the second HDL Editor window, open the VHDL file in which the CORE component is to be instantiated.

Note: Instead of opening a second sesssion, you could use **`Edit`** → **`Insert File`** from the HDL Editor tool bar to insert the file into the current HDL Editor session.

11. Cut and paste the Component Declaration from the CORE component's .vho file to your project's VHDL code, placing it after the architecture statement in the VHDL code.

    Cut and past the Component Instantiation from the CORE component's .vho file to your VHDL design code after the "begin" line. Give the inserted code an instance name. Edit the code to connect the signals in the design to the ports of the CORE component.
12. Check the syntax of the VHDL design code by selecting **`Synthesis`** → **`Check Syntax`** in the HDL Editor. Correct any errors. Then save the design and close the HDL Editor.
13. The design with the instantiated CORE module can then be synthesized (click the **`Synthesis`** button on the Flow tab).

Note: When the design is synthesized, a warning is generated that the CORE module is unexpanded. Modules instantiated as black boxes are not elaborated and optimized. The warning message is just reflecting the black box instantiation.

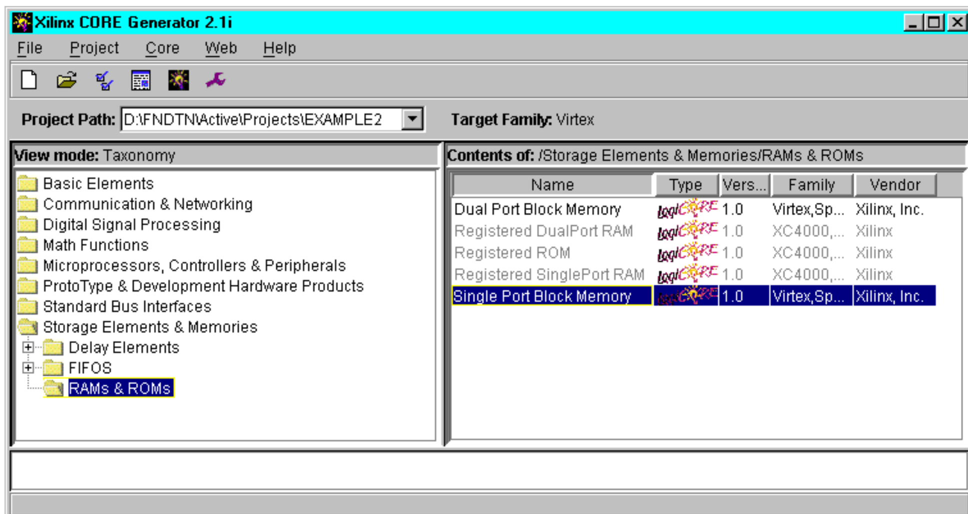
14. To complete the design, refer to the "Synthesizing the Design"through the "Programming the Device" sections under the "All-HDL Designs" section.

Note: The instantiated module must be in the same directory as the HDL code in which it is instantiated.

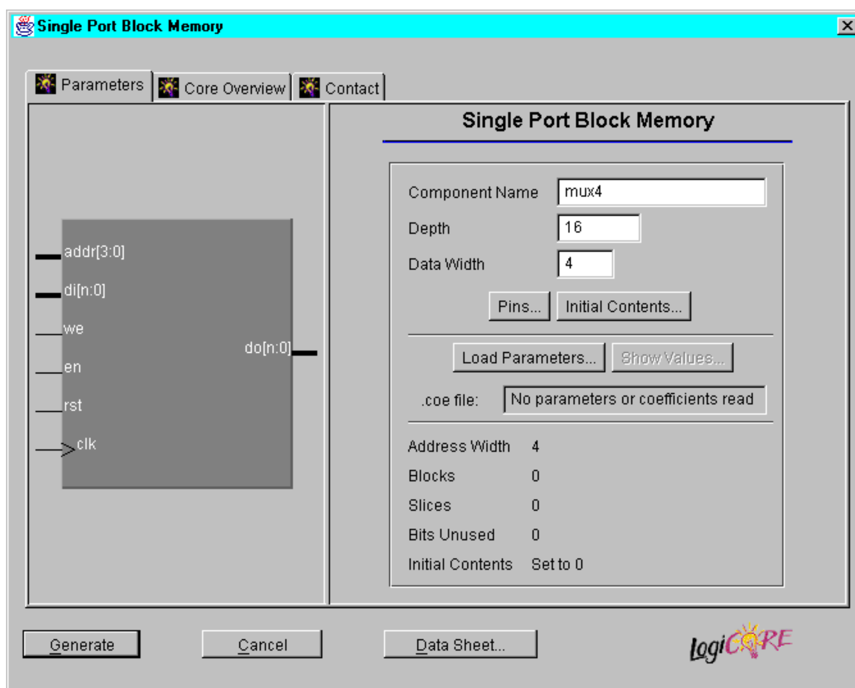
Verilog Instantiation

This section explains how to instantiate a CORE component into a Verilog design using Foundation.

1. With a valid Foundation project open, access the CORE Generator window using one of the following methods. Its operation is the same regardless of where it is invoked.
   - From the Project Manger, select **Tools** → **Design Entry** → **CORE Generator**
   - From the HDL Editor, select **Tools** → **CORE Generator**
2. Select **Project** → **Project Options**. In the Project Options dialog box, ensure that Design Entry is Verilog, that Behavioral Simulation is Verilog, and that the Vendor is Foundation. The Family entry should reflect the project's target device. Click **OK** to exit the Project Options dialog box.
3. To aid selection, the available COREs are categorized in folders on the View Mode section of the main CORE Generator window. Double click a folder to see its sub-categories. When you double click a sub-category folder, the available COREs are listed in the "Contents of" section of the main CORE Generator window.



4. To select a CORE, double click on the CORE's name in the "Contents of" window. A new window opens to allow you to view a description of the CORE or its data sheet, to customize the CORE for your application, and to generate the customized CORE. (Acrobat Reader is required to view the data sheet.)



5. When the CORE's window appears, enter a name for the component in the Component Name field.

   The name must begin with an alpha character. No extensions or uppercase letters are allowed. After the first character, the name may include numbers and/or the underscore character.

6. Other available customization options are unique for each CORE. Customize the CORE as necessary.

7. Select **Generate** to create the customized CORE and add its files to the project directory.

   The customized CORE component is a collection of several files including those listed below. The files are located in your Xilinx project directory for the current project.

component_name.coe	ASCII data file defining the coefficient values for FIR filters and initialization values for memory modules
component_name.xco	CORE Generator file containing the parameters used to generate the customized CORE
component_name.edn	EDIF implementation netlist for the CORE
component_name.veo	Verilog template file
component_name.mif	Memory Initialization Module for Virtex Block RAM modules

The component name is the name given to the CORE in the customization window. The port names are the names provided in the .veo file.

An example .veo file produced by the CORE Generator system follows.

```
/*******************************************************************
* This file was created by the Xilinx CORE Generator tool, and     *
* is (c) Xilinx, Inc. 1998, 1999. No part of this file may be      *
* transmitted to any third party (other than intended by Xilinx)   *
* or used without a Xilinx programmable or hardwire device without *
* Xilinx's prior written permission.                               *
*******************************************************************/

// The following line must appear at the top of the file in which
// the core instantiation will be made. Ensure that the translate_off/_on
// compiler directives are correct for your synthesis tool(s)

//----------- Begin Cut here for LIBRARY inclusion --------// LIB_TAG

// synopsys translate_off

`include "XilinxCoreLib/C_MEM_SP_BLOCK_V1_0.v"

// synopsys translate_on

// LIB_TAG_END ------- End LIBRARY inclusion --------------
```

```
// The following code must appear after the module in which it
// is to be instantiated. Ensure that the translate_off/_on compiler
// directives are correct for your synthesis tool(s).

//----------- Begin Cut here for MODULE Declaration -------// MOD_TAG
module mux4 (
ADDR,
CLK,
DI,
WE,
EN,
RST,
DO);

input [3 : 0] ADDR;
input CLK;
input [3 : 0] DI;
input WE;
input EN;
input RST;
output [3 : 0] DO;

// synopsys translate_off

C_MEM_SP_BLOCK_V1_0 #(
4,
1,
"0",
16,
1,
0,
1,
1,
1,
1,
1,
"mux4.mif",
16,
0,
0,
1,
1,
4)
inst (
.ADDR(ADDR),
```

```
.CLK(CLK),
.DI(DI),
.WE(WE),
.EN(EN),
.RST(RST),
.DO(DO));

// synopsys translate_on

endmodule
// MOD_TAG_END ------- End MODULE Declaration -------------

// The following must be inserted into your Verilog file for this
// core to be instantiated. Change the instance name and port connections
// (in parentheses) to your own signal names.

//----------- Begin Cut here for INSTANTIATION Template ---// INST_TAG
mux4 YourInstanceName (
.ADDR(ADDR),
.CLK(CLK),
.DI(DI),
.WE(WE),
.EN(EN),
.RST(RST),
.DO(DO));
// INST_TAG_END ------ End INSTANTIATION Template ---------
```

8. Select **`File`** → **`Exit`** to close the CORE Generator.
9. In the HDL Editor, open the CORE's .veo file (*component_name*.veo) located under the current project.
10. Open a second session of the HDL Editor. In the second HDL Editor window, open the Verilog file in which the CORE component is to be instantiated.

Note: Instead of opening a second sesssion, you could use **`Edit`** → **`Insert File`** from the HDL Editor tool bar to insert the file into the current HDL Editor session.

11. Cut and paste the Component Declaration from the CORE component's .veo file to your project's Verilog code, placing it after the architecture statement in the Verilog code.

    Cut and past the Component Instantiation from the CORE component's .veo file to your Verilog design code after the

"begin" line. Give the inserted code an instance name. Edit the code to connect the signals in the design to the ports of the LogiBLOX module.

12. Check the syntax of the VHDL design code by selecting **Synthesis** → **Check Syntax** in the HDL Editor. Correct any errors. Then save the design and close the HDL Editor.
13. The design with the instantiated CORE module can then be synthesized (click the **Synthesis** button on the Flow tab).

Note: When the design is synthesized, a warning is generated that the CORE module is unexpanded. Modules instantiated as black boxes are not elaborated and optimized. The warning message is just reflecting the black box instantiation.

14. To complete the design, refer to the "Synthesizing the Design"through the "Programming the Device" sections under the "All-HDL Designs" section.

Note: The instantiated module must be in the same directory as the HDL code in which it is instantiated.

XNF file in a VHDL or Verilog Design

This section explains how to instantiate an XNF file as a black box in a VHDL or Verilog design.

1. To attach an XNF module in the VHDL or Verilog code, use the nets named in the PIN records and/or SIG records in the XNF file as the port names of the component instantiation. The following is an example XNF file with PIN and SIG records.

```
SYM, current_state_reg<4>, DFF, LIBVER=2.0.0
PIN, D, I, next_state<4>, ,
PIN, C, I, N10, ,
PIN, Q, O, current_state<4>, ,
END
SIG, current_state<4>
SIG, CLK, I, ,
SIG, DATA, I, ,
SIG, SYNCFLG, O, ,
```

To reference buses in the instantiation of XNF modules, the nets named in PIN records and/or SIG records must be of the form.

netname<number>

This designation allows the bus to be referenced in the VHDL component as a vector data type.

2. Using the filename of the XNF file as the name of the component and the name of nets in the XNF file as port names, instantiate the XNF file in the VHDL or Verilog code.
3. The design with the instantiated XNF black box can then be synthesized (click the `Synthesis` button on the Flow tab).

Note: When the design is synthesized, a warning is generated that the XNF module is unexpanded. Modules instantiated as black boxes are not elaborated and optimized. The warning message is just reflecting the black box instantiation. Expansion of the XNF module takes place during the Translation stage of the Implementation phase.

4. To complete the design, refer to the “Synthesizing the Design”through the “Programming the Device” sections under the “All-HDL Designs” section.

The instantiated module must be in the same directory as the HDL code in which it is instantiated.

Schematic Designs in the HDL Flow

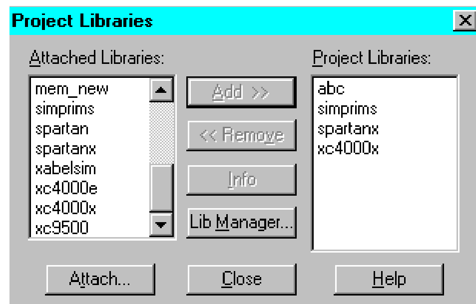
To take advantage of cross-boundary optimization and top-down synthesis methodology, you can use the HDL flow instead of the Schematic flow for top-level schematic designs with underlying HDL macros. In the HDL flow, your entire design is synthesized and optimized resulting in overall design performance improvement. The HDL flow is recommended for schematic top-level designs that contain underlying HDL macros. Used in this way, the tool behaves like an HDL block level diagram editor.

The following sections describe the HDL flow procedure for top-level schematic designs containing underlying HDL macros.

Adding a Schematic Library

In an HDL flow project, the device family is not selected until the design is synthesized. Therefore, you need to add a Xilinx library manually to make the Xilinx components available for schematic entry.

1. From the Project Manager window, select **`File`** → **`Project Libraries`**.



2. Select the target library for the desired device in the Attached Libraries window.
3. Click **`Add`** to add the library to your project. The library name will appear in the Files tab.

Note: If you want to create a top-level schematic to act only as a block diagram for your HDL designs, you do not need to add a schematic library.

Creating HDL Macros

Use the following procedure to create a macro from an HDL file (or State Machine) for use in a schematic.

1. Click the HDL Editor on the Design Entry button on the Project Manager's Flow tab.
2. Create or open an HDL file in the HDL Editor.
3. To create a symbol for the HDL file after you have finished editing or creating the file, select **`Project`** → **`Create Macro`** from the HDL Editor. The symbol is created and added to the SC Symbols list.

   (If you are asked for an initial target device when the macro is being created, enter any device. The synthesis that is done here is only necessary to create the symbol.)

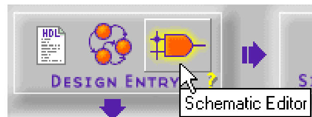
Note: If the Create Macro or Update Macro option is not available, check whether the HDL file has already been "added" to the project. If it is listed in the Files tab of the Project Manager, it is currently "added" to the project. Remove the file from the project by selecting it

and choosing **Document** → **Remove**. You can now create the macro. The file will be automatically added to the project when the entire design is analyzed later.

Creating the Schematic and Generating a Netlist

This section lists the basic steps for creating a schematic and generating a netlist from it.

1. Open the Schematic Editor by selecting the Schematic Editor icon from the Design Entry box on the Project Manager's Flow tab.



2. Select **Mode** → **Symbols** to add components to your new schematic. Select specific components from the SC Symbols window.

3. To define the ports, use Hierarchy Connectors.

Do *not* use pad components (IPAD, OPAD, etc.) from the Xilinx Unified Libraries. Foundation will synthesize the design from the top down and will add ports as well as buffers, if necessary.

Care must be taken when adding attributes to the schematic as follows:

- Pin locations, slew rates, and certain other design constraints may be added to the design using the Express Constraints Editor or a UCF file.
- Pin location or slew rate constraints may be placed on the I/O buffer (or flip-flop or latch) on the schematic. Do not place them on the net or the hierarchy connector.

4. Save your schematic by selecting **File** → **Save**.

5. Add the schematic to your project by selecting **Hierarchy** → **Add Current Sheet to Project**. The schematic is netlisted

and added to the project. The schematic (as well as any underlying HDL files) appear in the Files tab.

Note: If the HDL macros in the schematic have lower levels of hierarchy or use user-defined libraries, you must add the files for the lower levels to your project manually. Select **`Document`** → **`Add`** from the Project Manager to add the files. Foundation needs access to all the design files before synthesis can occur.

Selecting a Netlist Format

When a schematic is added to the project or when Foundation analyzes the schematic portion of the design, the schematic is netlisted into one of three formats: VHDL, XNF, or EDIF. (By default, VHDL is used.)

From the Project Manager, select **`Synthesis`** → **`Options`** and choose a netlist format in the “Export schematics to” section based on the following criteria.

- If the design is only a block diagram (there are no Unified Library components), use VHDL.
- If no attributes are passed from the schematic (including within Xilinx macros), use VHDL
- If the schematic includes XNF macros that contain RLOCs, use VHDL or select the Preserve Hierarchy option (in the Synthesis Settings dialog box).
- If any attributes have been applied within the schematic, then select XNF or EDIF.
- If the design targets a Virtex device, XNF may not be used.

Completing the design

Synthesize the design in the same manner you would a top-level HDL design.

1. Click the Synthesis (or Implementation) button on the Flow tab.
2. Select the schematic as the top-level in the Synthesis/Implementation settings dialog box.
3. In the Target Device section, be sure to select the device family that matches the schematic library you added to the project.

4. Click **Run**.

Foundation links all the project files and synthesizes the design using the top-down methodology.

HDL files from the schematic are added to the project when the schematic is analyzed. All HDL and State Machine files for which schematic macros were created are added to the Files tab. You may open and edit these files by double clicking on them in the Files tab. However, you can only update the HDL macros by opening them from the Schematic Editor and then selecting **Project** → **Update Macro**.

For more information on completing an HDL flow project, refer to the "Synthesizing the Design"through the "Programming the Device" sections under the "All-HDL Designs" section.

Chapter 6

HDL Design Entry and Synthesis

This chapter give an overview of HDL file selection for projects, compares synthesis of HDL modules in Schematic Flow projects and HDL Flow projects, explains how to manage large designs, and discusses advanced design techniques.

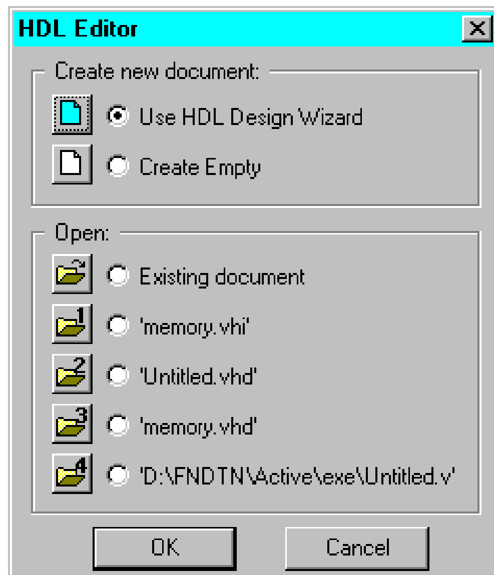
This chapter contains the following sections:

- “HDL File Selection”
- “Synthesis of HDL Modules”
- “Managing Large Designs”
- “Design Partitioning Guidelines”
- “User Libraries for HDL Flow Projects”
- “Using Constraints in an HDL Design”

Refer to the“Design Methodologies - HDL Flow” chapter for several examples of HDL designs.

HDL File Selection

To begin entering or editing a design in HDL, click the HDL Editor icon, which is part of the Design Entry button on the Project Manager’s Flow tab. The Editor dialog box displays and presents options for a design file, as shown in the following figure.



- Create new document
  - Use HDL Design Wizard

    Use this option for new designs. The Wizard includes dialogs for you to select the HDL language (VHDL or Verilog), enter the design name, and create ports. When finished, “skeleton” code pops up, complete with the library, entity, ports, and architecture already declared.

  - Create Empty

    Use this option for new designs. This option starts the HDL Editor and displays a blank page.

- Open
  - Existing Document

    Use this option to select an already existing HDL file.

  - Active document

    Use this option to select from the list of up to the last four active documents.

Adding the File to the Project

After creating an HDL file for an HDL Flow project, you must “add” the HDL file to the project. You can do this from within the HDL Editor by choosing **`Project`** → **`Add to Project`**. Alternatively, you can add files to the project by selecting **`Synthesis`** → **`Add Source File(s)`** or **`Document`** → **`Add`** from the Project Manager.

In an HDL Flow project, the top level of the design is chosen prior to design “elaboration” in the Synthesis phase. For Verilog, it is not necessary to add files in a specific order. For VHDL, it is important to add the files in the order in which they must be analyzed. Any files depending on the successful analysis of another must appear below that file in the Files tab.

Removing Files from the Project

You can remove files from a project by clicking on the file and selecting **`Document`** → **`Remove`** from the Project Manger.

Note: Removing a file from a project does not erase the file from the disk. It merely removes it from the project.

Getting Help with the Language

The Foundation HDL Editor provides HDL language assistance through both the Language Assistant and the Online Synthesis Documentation. The Language Assistant, shown in the“VHDL Language Assistant” figure, provides templates to aid you in common VHDL logic functions, and architecture-specific features. The “Verilog Language Assistant” figure shows the Verilog Language Assistant that provides templates to aid in for editing Verilog files. The Language Assistant also includes CORE Generator Instantiation templates (see the “CORE Generator Templates in Language Assistant” figure) for modules created with the CORE Generator tool.

To access the Language Assistant, open the HDL Editor, and select **`Tools`** → **`Language Assistant`**.

The HDL Editor also checks syntax. From the HDL Editor, select **`Synthesis`** → **`Check Syntax`** to analyze the file.

Refer to the HDL Editor’s online help for more information on the Language Assistant.

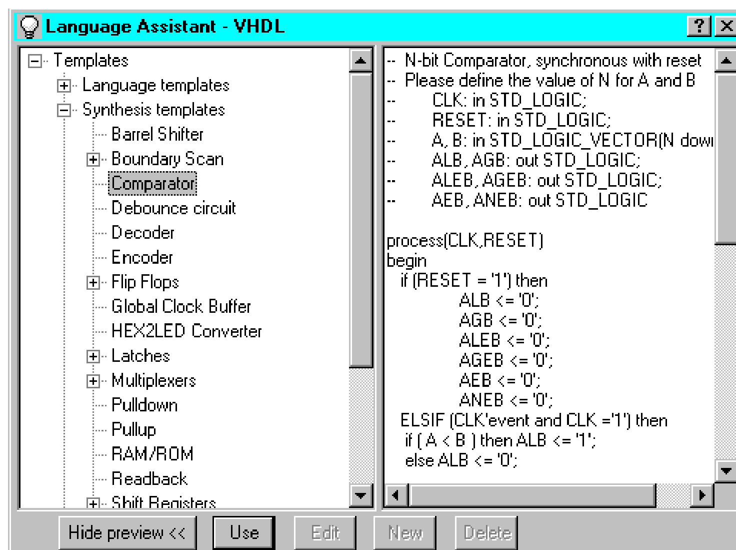


Figure 6-1 VHDL Language Assistant

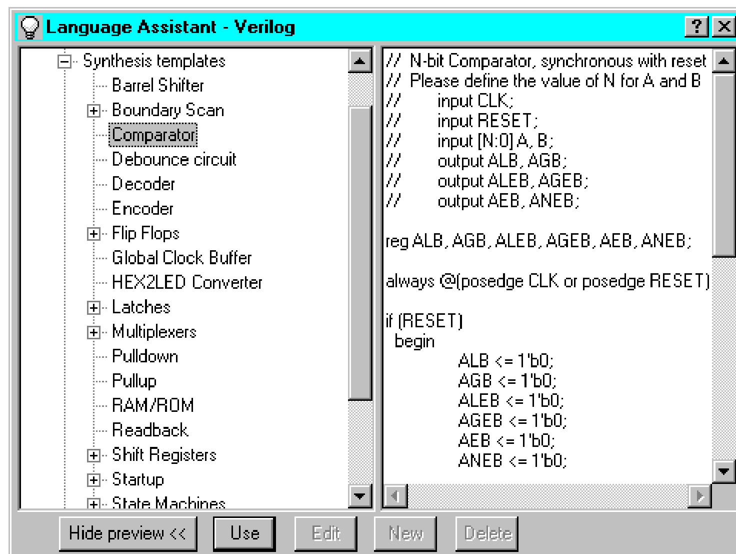


Figure 6-2 Verilog Language Assistant

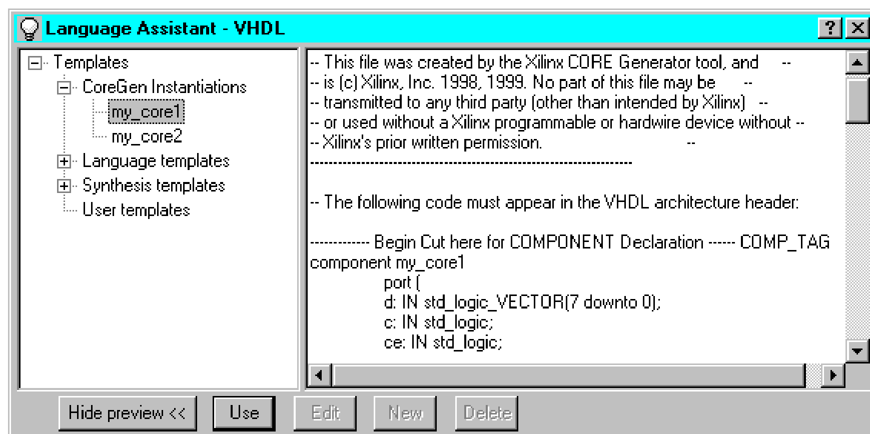


Figure 6-3 CORE Generator Templates in Language Assistant

Synthesis of HDL Modules

Foundation projects can be either Schematic Flow or HDL Flow projects. Many of the HDL editing and synthesis operations described in this section are the same for both flows; however, differences do exist and are noted where appropriate. This section describes how to synthesize your design without also continuing through implementation.

Schematic Flow Methodology

In a Schematic Flow project, VHDL and Verilog modules can only be underlying modules in a top-level schematic design. Each HDL file is synthesized and optimized separately. Top-level ABEL designs and ABEL State Machine designs are only supported in the Schematic Flow.

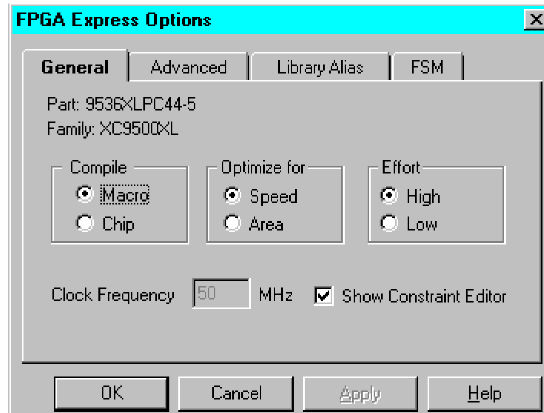
The Schematic Flow methodology can be beneficial if you have a few HDL blocks in an otherwise schematic environment. In this case, you synthesize each individual HDL module separately.

Following is the general procedure to synthesize HDL Modules in Schematic Flow Projects.

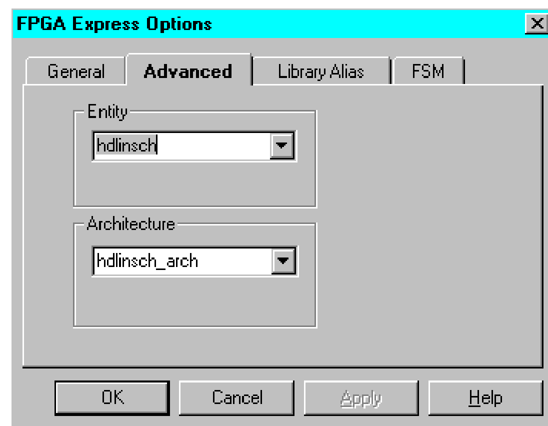
1. Open the HDL file in the HDL Editor. This can be done by the methods listed in the “HDL File Selection” section or by double

clicking on the .vhd (VHDL) or .v (Verilog) file in the Project Manager.

2. Select **`Synthesis`** → **`Options`** to access the FPGA Express Options window. In the General tab, select the optimization options for the module.



3. Click on the Advanced tab. Select the top-level entity and architecture, and click **`OK`**.



4. To synthesize the module and create a symbol, choose **`Project`** → **`Create Macro`** from the HDL Editor window.

5. Repeat step 4 for each HDL module.

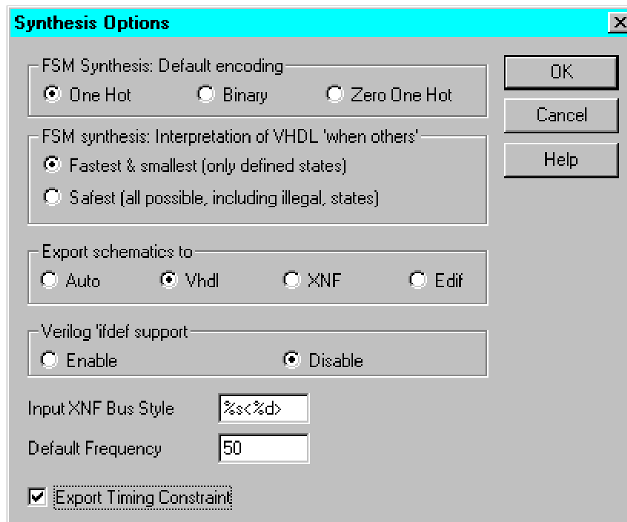
HDL Flow Methodology

In an HDL Flow project, all top-level VHDL and Verilog files and schematics are exported to the synthesis tool and optimized. Pre-Implementation constraint editing, cross-boundary optimization, and auto I/O buffer insertion are only available in an HDL Flow Project.

The HDL Flow approach provides an easier method of compilation. It requires only a single synthesis action for all HDL modules. In addition, this method includes optional cross-boundary optimization of the entire design, editing of constraints prior to implementation, and auto I/O buffer insertion.

Following is the general procedure to synthesize HDL modules in HDL Flow Projects.

1. Be sure that all HDL files are added to the project. See the “Adding the File to the Project” section for instructions on adding files to a project. Underlying HDL macros in top-level schematics in HDL projects are an exception to this; files for those HDL macros are added automatically during synthesis.
2. From the Project Manager window, set the global synthesis options by selecting **Synthesis** → **Options** to open the Synthesis Options dialog box.



In the Synthesis Options dialog box, set the Default FSM Encoding style, XNF Bus Style, and Default Frequency. Check the **Export Timing Constraint** box if you want to have timing and pin location constraints entered after the elaboration step to be automatically exported to place and route tools.

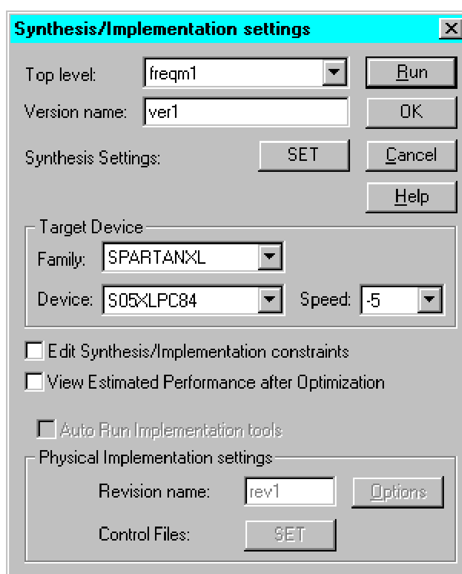
For FSM Encoding style, use the following guidelines for best results.

- If your target device is an FPGA, choose One Hot.
- If your target device is a CPLD, choose Binary.

Refer to the “Selecting a Netlist Format” section of the “Design Methodologies - HDL Flow” chapter for information on setting the “Export schematic to” option.

Click **OK** when all desired options are set.

3. To synthesize the design, click the **Synthesis** button on the Flow tab. This opens the Synthesis/Implementation dialog box.



4. In the Synthesis/Implementation dialog box, you can do the following.

- Select the name of the top-level entity or module from which processing of the design hierarchy should begin

- Enter a version name
- Select the target device
- Choose to edit constraints after elaboration. This option opens the Express Constraints Editor before the design is optimized by the synthesis engine.
- Choose to view the estimated performance after optimization spreadsheets. This opens the Express Time Tracker and displays the design's pre-implementation timing estimates.
- Click **SET** to access the Synthesis Setting and modify the synthesis settings as desired

When ready, click **Run** to synthesize the design.

Managing Large Designs

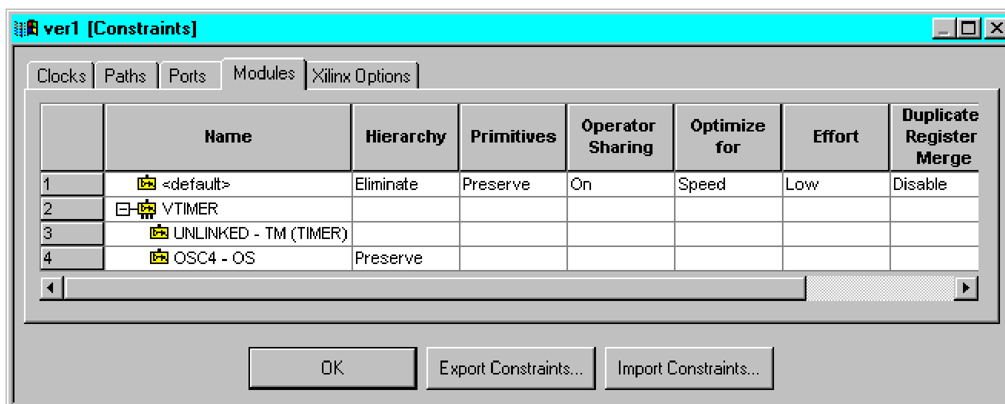
The following subsections explain how to manage large designs.

Design Optimization

With Foundation, you can control optimization of the design on a module-by-module basis. This means that you have the ability to, for example, optimize certain modules of your design for speed and some for area. In addition, an effort level for the optimization engine can be set to either high or low.

For the Schematic Flow projects, the optimization goals may be set in the HDL Editor, by selecting **Synthesis** → **Options**.

For Foundation HDL Flow projects, the optimization goals are set for individual modules in the "module" tab of the Express Constraints Editor. (The module tab is shown in the following figure.)



Setting Constraints Prior to Synthesis

With the Foundation Express product you can set performance constraints and attributes to guide the optimization process on a module-by-module basis. Select **`Edit Synthesis/Implementation Constraints`** in the Synthesis/Implementation settings dialog box to access the Express Constraints Editor window. This window contains tabs with spreadsheets and dialog boxes specific to the target architecture. You need to select **`View Estimated Performance after Optimization`** in the Synthesis/Implementation settings dialog box to view spreadsheets containing the results obtained as a result of setting the constraints. Refer to the "Using Constraints in an HDL Design" section for more information on constraints in HDL designs.

Design Partitioning Guidelines

The way in which a design is partitioned can affect how well the optimizer can optimize the combinatorial logic. If a design is poorly partitioned in the entry phase, logic optimization can suffer. Here are some HDL coding and partitioning guidelines that will help improve logic optimization.

- Avoid imposing boundaries on combinatorial paths.

  If parts of a combinatorial logic path are compiled in separate modules, no logic optimization can be performed across the block boundaries.

Instead, partition the design so that combinatorial paths are not split across multiple modules. This gives the software the best opportunity to optimize combinatorial logic on the path.

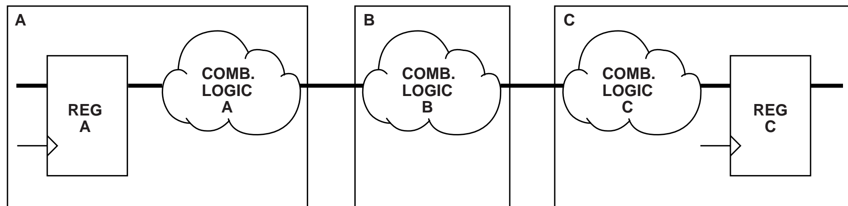


Figure 6-4 Combinatorial Logic Path Split Across Boundaries (Inefficient Use of Design Resources)

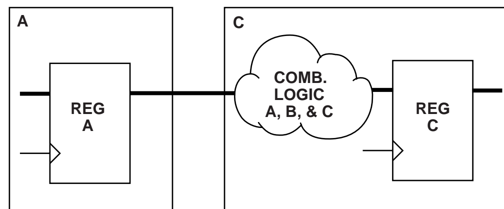


Figure 6-5 Combinatorial Logic Path Grouped Into One Block (Efficient use of Design Resources)

- Register all block outputs.

  Partition the design into modules in such a way that all block outputs are registered. This guarantees that no boundaries are imposed on any combinatorial paths, as discussed previously.

User Libraries for HDL Flow Projects

In the Foundation Express environment, a user library is an HDL file which is referenced by another file through a LIBRARY statement. A user library can contain packages and/or entities.

Creating a New Library

User libraries are stored as part of the Foundation project. Following are the basic steps to create new libraries in HDL Flow projects.

1. Select **Synthesis** → **New Library** from the Project Manger.
2. Enter a name for the new library and click **OK**. The new library is added to the list of project files on the Files tab.
3. To add files to the new library, right click on the library name in the Files tab list.
4. Select the **Add Source Files to "*library_name*"** option to access the Add Document dialog window where you can select the files to be added to the library. The files are analyzed automatically as they are added.

Declaring and Using User Libraries

In the VHDL or Verilog code, user libraries for Foundation projects are declared and used just like system libraries such as IEEE. For example, to access the entities defined in the library mylib.vhd, use the following VHDL syntax:

```
library MYLIB
use MYLIB.all;
```

User library directories that are part of a project are automatically searched when referenced in VHDL.

Using Constraints in an HDL Design

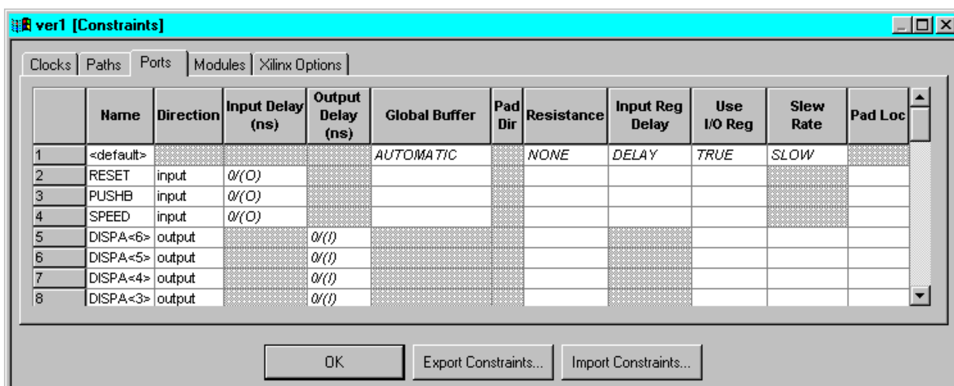
The following sections provide information on adding constraints to HDL designs.

Express Constraints Editor

Foundation Express users have access to the Express Constraints Editor. The Express Constraints Editor includes a window with five different tabs. The following three tabs represent constraints that can be applied to the design prior to synthesis: Clock, Paths, and Ports.

- The Clocks tab allows you to specify overall speeds for the clocks in a design.

- The Paths tab allows you precise control of point-to-point timing in a design.
- The Ports tab allows OFFSETS, pullups/pulldowns, and pin locations to be specified in a design.



The timing constraints specified in the Express Constraints Editor tabs are translated into FROM:TO or PERIOD timespecs and placed in an NCF file. Following is an example:

```
TIMESPEC TS_CLK = PERIOD "CLK" 20 ns HIGH 10;
```

Currently, Express cannot apply all Xilinx constraints. Express can apply the following constraints:

- PERIOD
- FROM:TO timespecs which use FFS, LATCHES, and PADS
- Pin location constraints
- Slew rate
- TNM_NET
- PULLUP / PULLDOWN
- OFFSET:IN:BEFORE
- OFFSET:OUT:AFTER

Express cannot apply the constraints listed below:

- TPSYNC
- TPTHRU

- TIG
- user-RLOCs, RLOC_ORIGIN, RLOC_RANGE
- non-I/O LOCs
- KEEP
- U_SET, H_SET, HU_SET
- user-BLKNM and user-HBLKNM
- PROHIBIT

Express can create its own timegroups by grouping logic with common clocks and clock enables. In addition, you can form user-created timing subgroups by right clicking on an existing timing path and choosing New Sub Path.

Xilinx Logical Constraints

For constraints that cannot be applied using the Express Constraint Editor, a UCF file can be used to specify logical constraints. Constraints or attributes that can be applied within a schematic, netlist, or UCF file are known as logical constraints. Logical constraints ignore timing paths, prohibit pin locations, or constrain placement of elements in an FPGA or CPLD design. In order to use a logical constraint correctly, the "instance" name of the logic in a design must be used. Instance names are XNF SYM record names, XNF SIG record names, XNF net names, and EXT record names. For examples of reading these instance names out of a XNF file from Express, refer to the following figure.

```
SYM, current_state_reg<4>, DFF, LIBVER=2.0.0
PIN, D, I, next_state<4>, ,
PIN, C, I, N10, ,
PIN, Q, O, current_state<4>, ,
END
SIG, current_state<4>
EXT, CLK, I, ,
EXT, DATA, I, ,
EXT, SYNCFLG, O, ,
```

Figure 6-6 XNF example

In the preceding figure, the SYM record name can be referenced by a logical constraint by using the instance name, current_state_reg<4>. A net called N10 or current_state<4> can also be used in a logical

constraint. EXT records correspond to pins used on a package. The EXT records named CLK, DATA, and SYNCFLG can be referenced in a pin locking constraint.

For more information on Xilinx constraints, refer to the "Attributes, Constraints, and Carry Logic" chapter in the *Libraries Guide*.

Reading Instance Names from an XNF file for UCF Constraints

UCF constraints are applied by referencing instance names that are found in the XNF file. Instance names for logic in a design can be found by reading the XNF file. Refer to XNF syntax in the "XNF example" figure for the examples in this section. The following examples illustrate valid entries within a UCF file.

- A TNM constraint can be applied to an FF by using the instance name from the XNF file. Similarly, a LOC/RLOC can be applied:

  ```
  INST "current_state_reg<4>" TNM=group1;
  INST "current_state_reg<4>" LOC=CLB_R5C5;
  ```

  By attaching a TNM to this flip-flop instance name, this flip-flop can be referenced in a FROM:TO timing specification. Any symbol that can have an M1 constraint applied is referenced by using the string following the keyword: SYM.

- A pin on a device may be locked to a package-specific position by referencing the EXT record name and adding the .PAD string:

  ```
  INST "DATA.PAD" LOC=P124;
  ```

- An attribute which can be placed on a net, like KEEP or TNM, can be referenced by referencing the netname on the PIN record or SIG record:

  ```
  NET "current_state<4>" KEEP;
  NET "current_state<4>" TNM=group2;
  ```

A final note on referencing instance names from a XNF file: match the case; names are case-sensitive. If the case of names in the XNF file is not followed exactly, the implementation software may not be able to find (or may incorrectly find) an instance name for a constraint.

Instance Names for LogiBLOX RAM/ROM

In the Foundation Express methodology, whenever large blocks of RAM/ROM are needed, LogiBLOX RAM/ROM modules should be instantiated by the user in the HDL code. With LogiBLOX RAM/ROM modules instantiated in the HDL code, timing and/or placement constraints on these RAM/ROM modules and the RAM/ROM primitives that comprise these modules, are specified in a .ucf file.

To create timing and/or placement constraints for RAM/ROM LogiBLOX modules, you must know how many primitives are used and how the primitives inside the RAM/ROM LogiBLOX modules are named.

Note: LogiBLOX does not support Virtex. You can get a Virtex RAM from the CORE Generator system.

Calculating Primitives for a LogiBLOX RAM/ROM Module

When a RAM/ROM is specified with LogiBLOX, the RAM/ROM depth and width are specified. If the RAM/ROM depth is divisible by 32, then 32x1 primitives are used. If the RAM/ROM depth is not divisible by 32, then 16x1 primitives are used instead. In the case of dual-port RAMs, 16x1 primitives are always used. Based on whether 32x1 or 16x1 primitives are used, the number of RAM/ROMs primitives can be calculated.

For example, if a RAM48x4 was required for a design, RAM16x1 primitives would be used. Based on the width, there would be four banks of RAM16x1's. Based on the depth, each bank would have three RAM16x1's.

Naming Primitives in LogiBLOX RAM/ROM Modules

Using the example of a RAM48x4, the RAM primitives inside the LogiBLOX would be named as follows:

```
MEM0_0    MEM1_0    MEM2_0    MEM3_0
MEM0_1    MEM1_1    MEM2_1    MEM3_1
MEM0_2    MEM1_2    MEM2_2    MEM3_2
```

Each primitive in a LogiBLOX RAM/ROM module has an instance name of MEMx_y, where y represents the primitive position in the bank of memory, and where x represents the bit position of the RAM/ROM output.

Referencing LogiBLOX Entities

This section is written in terms of the Verilog example, using the files illustrated in Figures 6-6 through 6-9. This section also applies to the VHDL example in Figures 6-10 through 6-13.

LogiBLOX RAM/ROM modules in an HDL Flow project are constrained using a UCF file.

LogiBLOX RAM/ROM modules instantiated in the HDL code can be referenced by the complete hierarchical instance name. If a LogiBLOX RAM/ROM module is at the top-level of the HDL code, then the instance name of the LogiBLOX RAM/ROM module is just the instantiated instance name. In the case of a LogiBLOX RAM/ROM that is instantiated within the hierarchy of the design, the instance name of the LogiBLOX RAM/ROM module is the full hierarchical path to the LogiBLOX RAM/ROM. The hierarchy level names are listed from the top level down and are separated by a "_".

In the Verilog example, the RAM32X1S is named "memory". The memory module is instantiated in the Verilog module "inside" with an instance name "U1". "inside" is instantiated in the top-level module "test" with an instance name "U0". Therefore, the RAM32X1S can be referenced in a UCF file as "U0_U1". For example, to attach a TNM to this block of RAM, the following line could be used in the UCF file:

```
INST "U0_U1" TNM=block1;
```

Since U0_U1 is composed of two RAM primitives, a timegroup called block1 is created; the block1 TNM can be used throughout the UCF file as a timespec end/start point, and/or U0_U1 could have a LOC area constraint applied to it. If the RAM32X1S has been instantiated in the top-level file and the instance name used in the instantiation is U1, then this block of RAM can just be referenced by U1.

Sometimes it is necessary to apply constraints to the primitives that compose the LogiBLOX RAM/ROM module. For example, if you choose a floorplanning strategy to implement your design, it may be necessary to apply LOC constraints to one or more primitives inside a

LogiBLOX RAM/ROM module. Consider the RAM32X2S example. Suppose that each of the RAM primitives needs to be constrained to a particular CLB location.

Based on the rules for determining the MEMx_y instance names, using the example from above, each of the RAM primitives can be referenced by concatenating the full-hierarchical name to each of the MEMx_y names. The RAM32x2S created by LogiBLOX will have primitives named MEM0_0 and MEM1_0. So, CLB constraints in a .ucf file for each of these two items would be:

```
INST "U0_U1/MEM0_0" LOC=CLB_R10C10;
INST "U0_U1/MEM0_1" LOC=CLB_R11C11;
```

In the following figure, the LogiBLOX module is contained in the "inside UO" component.

```
test.v:

module test(DATA,DATAOUT,ADDR,C,ENB);
input   [3:0] DATA;
output  [3:0] DATAOUT;
input   [5:0] ADDR;
input   C;
input ENB;
wire [3:0] dataoutreg;
reg [3:0] datareg;
reg [3:0] DATAOUT;
reg [5:0] addrreg;

inside U0 (.MDATA(datareg),.MDATAOUT(dataoutreg),.MADDR(addrreg),.C(C),.WE(ENB));

always@(posedge C)
        datareg = DATA;

always@(posedge C)
               DATAOUT = dataoutreg;

always@(posedge C)
        addrreg = ADDR;
endmodule
```

Figure 6-7 Top-level Verilog File

The following figure illustrates the instantiated LogiBLOX module, "memory U1".

```
inside.v:

module inside(MDATA,MDATAOUT,MADDR,C,WE);

input   [3:0] MDATA;
output  [3:0] MDATAOUT;
input   [5:0] MADDR;
input   C;
input WE;

memory  U1
( .A(MADDR),
 .DO(MDATAOUT),
 .DI(MDATA),
 .WR_EN(WE),
 .WR_CLK(C));

endmodule
```

Figure 6-8 Verilog File with Instantiated LogiBLOX Module

When the LogiBLOX module is created, a .vei file is created, which is used as an instantiation reference.

```
//----------------------------------------------------
// LogiBLOX SYNC_RAM Module "memory"
// Created by LogiBLOX version M1.4.12
//    on Fri Dec 19 14:56:42 1997
// Attributes
//     MODTYPES = SYNC_RAM
//     BUS_WIDTH = 4
//     DEPTH = 48
//----------------------------------------------------
memory instance_name
( .A () ,
 .DO () ,
.DI () ,
.WR_EN () ,
.WRCLK () );

module memory (A, DO, DI, WR_EN, WR_CLK);
input  [5:0] A;
output  [3:0] DO;
input  WR_EN;
input WR_CLK;
endmodule
```

Figure 6-9 VEI File Created by LogiBLOX

```
test.ucf:

INST U0_U1 TNM = usermem;
TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;
INST U0_U1/mem0_0 LOC=CLB_R7C2;
```

Figure 6-10 UCF File for Verilog Example

```
test.vhd:

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity test is
        port(   DATA: in STD_LOGIC_VECTOR(3 downto 0);
                DATAOUT: out STD_LOGIC_VECTOR(3 downto 0);
                ADDR: in STD_LOGIC_VECTOR(5 downto 0);
                C, ENB: in STD_LOGIC);
end test;

architecture details of test is
signal dataoutreg,datareg: STD_LOGIC_VECTOR(3 downto 0);
signal addrreg: STD_LOGIC_VECTOR(5 downto 0);

component inside
port(   MDATA: in STD_LOGIC_VECTOR(3 downto 0);
        MDATAOUT: out STD_LOGIC_VECTOR(3 downto 0);
        MADDR: in STD_LOGIC_VECTOR(5 downto 0);
        C,WE: in STD_LOGIC);
end component;
begin
U0: inside port
map(MDATA=>datareg.,MDATAOUT=>dataoutreg.,MADDR=>addrreg,C=>C,WE=>ENB); map

process( C )
begin
        if(C'event and C='1') then
        datareg <= DATA;
        end if;
end process;

process( C )
begin
        if(C'event and C='1') then
        DATAOUT <= dataoutreg;
        end if;
end process;

process( C )
begin
        if(C'event and C='1') then
        addrreg <= ADDR;
        end if;
end process;

end details;
```

Figure 6-11 Top-level VHDL Example File

```
inside.vhd:

entity inside is
        port(   MDATA: in STD_LOGIC_VECTOR(3 downto 0);
                MDATAOUT: out STD_LOGIC_VECTOR(3 downto 0);
                MADDR: in STD_LOGIC_VECTOR(5 downto 0);
                C,WE: in STD_LOGIC);
end inside;

architecture details of inside is
component memory
port(   A: in STD_LOGIC_VECTOR(5 downto 0);
        DO: out STD_LOGIC_VECTOR(3 downto 0);
        DI: in STD_LOGIC_VECTOR(3 downto 0);
        WR_EN,WR_CLK: in STD_LOGIC);
end component;

begin
U1: memory port map(A=>MADDR,DO=>MDATAOUT,DI=>MDATA,WR_EN=>WE,WR_CLK=>C);
end details;
```

Figure 6-12 VHDL File with Instantiated LogiBLOX Module

```
LogiBLOX SYNC_RAM Module "memory"
Created by LogiBLOX version C.16
    on Tue Jun 22 12:57:06 1999
Attributes
   MODTYPE = SYNC_RAM
   BUS_WIDTH = 4
   DEPTH = 16
   STYLE = MAX_SPEED
   USE_RPM = FALSE

component memory
  PORT(
    A: IN std_logic_vector(3 DOWNTO 0);
    DO: OUT std_logic_vector(3 DOWNTO 0);
    DI: IN std_logic_vector(3 DOWNTO 0);
    WR_EN: IN std_logic;
    WR_CLK: IN std_logic);
end component;

instance_name : memory port map
(A => ,
 DO => ,
 DI => ,
 WR_EN => ,
 WR_CLK => );
```

Figure 6-13 VHI File Created By LogiBLOX

```
test.ucf:

INST U0_U1 TNM = usermem;
TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;
INST U0_U1/mem0_0 LOC=CLB_R7C2;
```

Figure 6-14 UCF File for VHDL Example

Chapter 7

State Machine Designs

This chapter explains the basic operations used to create state machine designs.

State machine design typically starts with the translation of a concept into a "paper design," usually in the form of a state diagram or a bubble diagram. The paper design is converted to a state table and, finally, into the source code itself. To illustrate the process of developing state machines, this chapter discusses an example in which a state machine repetitively sequences through the five numbers 9, 5, 1, 2, and 4.

This chapter contains the following sections.

- "State Machine Example"
- "State Diagram"
- "State Machine Implementation"
- "Encoding Techniques"

Refer to the"Finite State Machine (FSM) Designs" section of the "Design Methodologies - Schematic Flow" chapter for a detailed procedure on creating a state machine design. For an example of how to create a state machine, refer to the Foundation WATCH tutorial accessed from the Xilinx Support web site at http:\\support.xilinx.com.

For additional information, select **`Help`** → **`Foundation Help Contents`**. Click **`State Editor`** under Tools or **`The State Editor`** under Tutorials in the Xilinx Foundation Series On-Line Help System menu.

For information on creating state machine macros, refer to the "Schematic Designs With Finite State Machine (FSM) Macros" section of the "Design Methodologies - Schematic Flow" chapter and to the

"HDL Designs with State Machines" section of the "Design Methodologies - HDL Flow" chapter.

State Machine Example

The state machine in this example has four modes, which can be selected by two inputs: DIR (direction) and SEQ (sequence). DIR reverses the sequence direction; SEQ alters the sequence by swapping the position of two of the numbers in the sequence. When the machine is turned on, it starts in the initial state and displays the number 9. It then sequences to the next number shown, depending on the input. This sequence is summarized in the following table.

Table 7-1 State Relationships

SEQ	DIR	Sequence of Displayed Number
1	1	$9 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 9$ . . .
1	0	$9 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 5 \rightarrow 9$ . . .
0	1	$9 \rightarrow 5 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 9$ . . .
0	0	$9 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 5 \rightarrow 9$ . . .

Conceptual descriptions show the state progression and controlling modes, but they do not clearly show how change conditions result.

State Diagram

The state diagram is a pictorial description of state relationships. The "State Diagram" figure gives an example. Even though a state diagram provides no extra information, it is generally easier to translate this type of diagram into a state table. Each circle contains the name of the state, while arrows to and from the circles show the transitions between states and the input conditions that cause state transitions. These conditions are written next to each arrow.

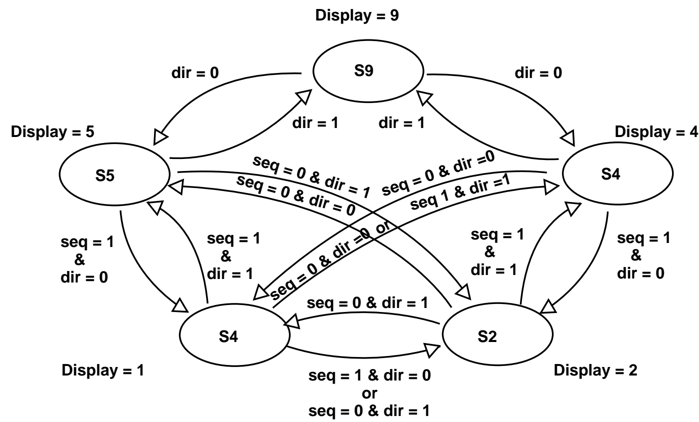


Figure 7-1 State Diagram

State Machine Implementation

A state machine requires memory and the ability to make decisions. The actual hardware used to implement a state machine consists of state registers (flip-flops) and combinatorial logic (gates). State registers store the current state until the next state is calculated, and a logic network performs functions that calculate the next state on the basis of the present state and the state machine inputs. The following figure shows the logic transitioning through the state registers to the output decoder logic.

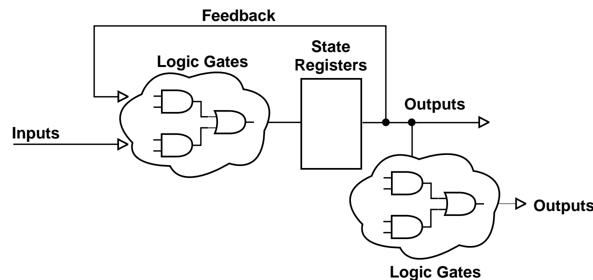


Figure 7-2 Parts of a State Machine

The amount of logic used to calculate the next state varies according to the type of state machine you are implementing. You must choose the most efficient design approach, depending on the hardware in which the design will be implemented.

Encoding Techniques

The states in a state machine are represented by setting certain values in the set of state registers. This process is called state assignment or state encoding.

There are many ways to arrange, or encode, state machines. For example, for a state machine of five states, you can use three flip-flops set to values for states 000, 001, 010, 011, 100, which results in a highly encoded state machine implementation. You can also use five flip-flops set to values 00001, 00010, 00100, 01000, 10000, that is, one flip-flop per state, which results in a one-hot-encoded state machine implementation. State encoding has a substantial influence on the size and performance of the final state machine implementation.

Symbolic and Encoded State Machines

A symbolic state machine makes no reference to the actual values stored in the state register for the different states in the state table. Therefore, the software determines what these values should be; it can implement the most efficient scheme for the architecture being targeted or for the size of the machine being produced.

All that is defined in a symbolic state machine is the relationship among the states in terms of how input signals affect transitions between them, the values of the outputs during each state, and in some cases, the initial state.

An encoded state machine requires the same definition information as a symbolic machine, but in addition, it requires you to define the value of the state register for each state.

Symbolic state machines are supported for CPLDs, but they are less efficient than encoded state machines.

Compromises in State Machine Encoding

A good state machine design must optimize the amount of combinatorial logic, the fanin to each register, the number of registers, and the propagation delay between registers. However, these factors are interrelated, and compromises between them may be necessary. For example, to increase speed, levels of logic must be reduced. However, fewer levels of logic result in wider combinatorial logic, creating a higher fanin than can be efficiently implemented given the limited number of fanins imposed by the FPGA architecture.

As another example, you must factor out the logic to decrease the gate count; that is, you must extract and implement shared terms using separate logic. Factoring reduces the amount of logic but increases the levels of logic between registers, which slows down the circuit. In general, the performance of a highly encoded state machine implemented in an FPGA device drops as the number of states grows because of the wider and deeper decoding that is required for each additional state. CPLDs are less sensitive to this problem because they allow a higher fanin.

Binary Encoding

Using the minimum number of registers to encode the machine is called binary, or maximal, encoding, because the registers are used to their maximum capacity. Each register represents one bit of a binary number. The example discussed earlier in this chapter has five states, which can be represented by three bits in a binary-encoded state machine.

Although binary encoding keeps the number of registers to a minimum, it generally increases the amount of combinatorial logic

because more combinatorial logic is required to decode each state. Given this compromise, binary encoding works well when implemented in Xilinx CPLD devices, where gates are wide and registers are few.

One-Hot Encoding

In one-hot encoding, an individual state register is dedicated to one state. Only one flip-flop is active, or hot, at any one time. There are two ways that one-hot encoding can significantly reduce the amount of combinatorial logic used to implement a state machine.

As noted in the "Compromises in State Machine Encoding" section, highly encoded designs tend to require many high fanin logic functions to interpret the inputs. One-hot encoding simplifies this interpretation process because each state has its own register, or flip-flop. As a result, the state machine is already "decoded," so the state of the machine is determined simply by finding out which flip-flop is active. One-hot encoding reduces the width of the combinatorial logic and, as a result, the state machine requires fewer levels of logic between registers, reducing its complexity and increasing its speed.

Although one-hot encoding can be used for CPLDs and FPGAs, it is better suited to FPGAs.

One-Hot Encoding in Xilinx FPGA Architecture

One-hot encoding is well-suited to Xilinx FPGAs because the Xilinx architecture is rich in registers, while each configurable logic block (CLB) has a limited number of inputs. As a result, state machine designs that require few registers, many combinatorial elements, and large fanin do not take full advantage of these resources. In general, a one-hot state machine implemented in a Xilinx FPGA minimizes both the number of CLBs and the levels of logic used.

Limitations

In some cases, the one-hot method may not be the best encoding technique for a state machine implemented in a Xilinx device. For example, if the number of states is small, the speed advantages of using the minimum amount of combinatorial logic may be offset by delays resulting from inefficient CLB use.

Encoding for CPLDs

CPLD devices generally implement binary-encoded state machines more efficiently. Binary encoding uses the minimum number of registers. Each state is represented by a binary number stored in the registers. Using as few registers as possible usually increases the amount of combinatorial logic needed to interpret each state.

CPLD devices have wide gates and a large amount of combinatorial logic per register, so it is best to start with binary encoding. If the complexity of the state machine logic is such that binary encoding exhausts all product term resources of a CPLD, try a slightly less fully encoded state machine.

The syntax used to specify one-hot encoded state machines for FPGAs is also supported for CPLD designs.

Chapter 8

LogiBLOX

LogiBLOX is an on-screen design tool for creating high-level modules such as counters, shift registers, and multiplexers for FPGA and CPLD designs. LogiBLOX includes both a library of generic modules and a set of tools for customizing these modules. LogiBLOX modules are pre-optimized to take advantage of Xilinx architectural features such as Fast Carry Logic for arithmetic functions and on-chip RAM for dual-port and synchronous RAM. With LogiBLOX, high-level LogiBLOX modules that will fit into your schematic-based design or HDL-based design can be created and processed.

This chapter contains the following sections.

- "Setting Up LogiBLOX on a PC"
- "Starting LogiBLOX"
- "Creating LogiBLOX Modules"
- "LogiBLOX Modules"
- "Using LogiBLOX for Schematic Designs"
- "Using LogiBLOX for HDL Designs"
- "Documentation"

Note: LogiBLOX supports all Xilinx architectures except Virtex.

For information about instantiating LogiBLOX into designs, refer to the "Schematic Designs With Instantiated LogiBLOX Modules" section of the "Design Methodologies - Schematic Flow" chapter and the"HDL Designs with Black Box Instantiation" section of the "Design Methodologies - HDL Flow" chapter.

For an example of how to use a LogiBLOX module, refer to the in-depth Foundation Watch tutorial available via the Xilinx web site at http://support.xilinx.com.

Setting Up LogiBLOX on a PC

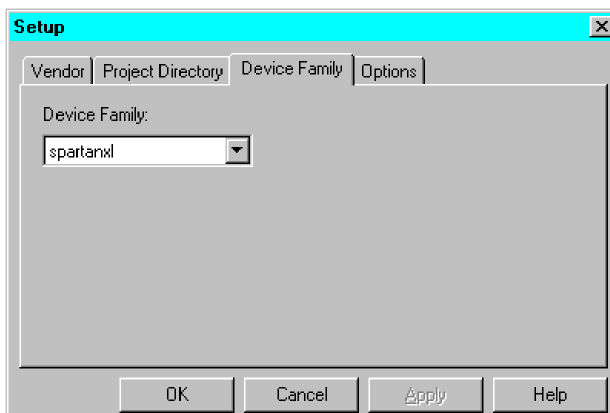
LogiBLOX is automatically installed with the Xilinx design implementation tools and is ready to use from the Foundation Project Manager interface when you start the product.

Starting LogiBLOX

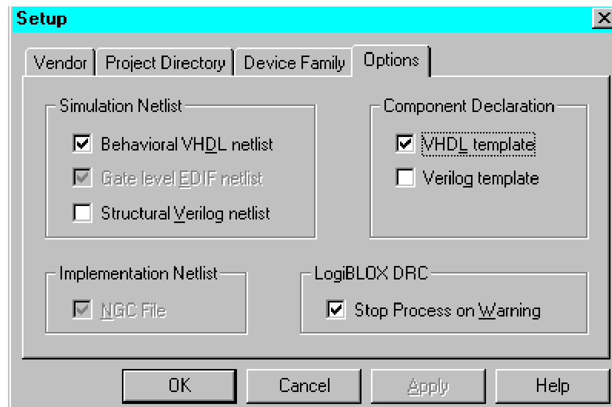
LogiBLOX can be started from the Project Manager window using **`Tools`** → **`Design Entry`** → **`LogiBLOX module generator`**. LogiBLOX can also be started within Schematic Capture by selecting **`Options`** → **`LogiBLOX`** or in the HDL Editor by selecting **`Synthesis`** → **`LogiBLOX`**. The LogiBLOX Module Selector dialog box then opens. See the "LogiBlox Module Selector - Accumulators" figure for an example.

The first time you access LogiBLOX, a Setup dialog appears. Or, you can click **`Setup`** on the LogiBLOX Module Selector dialog box to access the Setup dialog box.

Use the Device Family tab (shown below) to select a Device Family.



You can instantiate a LogiBLOX module in VHDL or Verilog code. Use the Options tab to select appropriate Simulation Netlist and Component Declaration template. For VHDL, select **`VHDL template`** and **`Behavioral VHDL netlist`** (shown below). For Verilog, select **`Verilog template`** and **`Structural Verilog netlist`**.



You can use LogiBLOX components in schematics and HDL designs for FPGAs and CPLDs. Once you are in the LogiBLOX GUI, you can customize standard modules and process them for insertion into your design.

Note: Once a LogiBLOX module is created, do not change parameters for the module on the schematic. Any changes to the module parameters must be made through the LogiBLOX GUI and a new module created.

You can also import an existing LogiBLOX module from another directory or project into the current project library by selecting **`Options`** → **`Import LogiBLOX`** from the Schematic Capture window and choosing the MOD file of the module you want to import. For details, see the “Importing Existing LogiBLOX Modules” section of the “Design Methodologies - Schematic Flow” chapter.

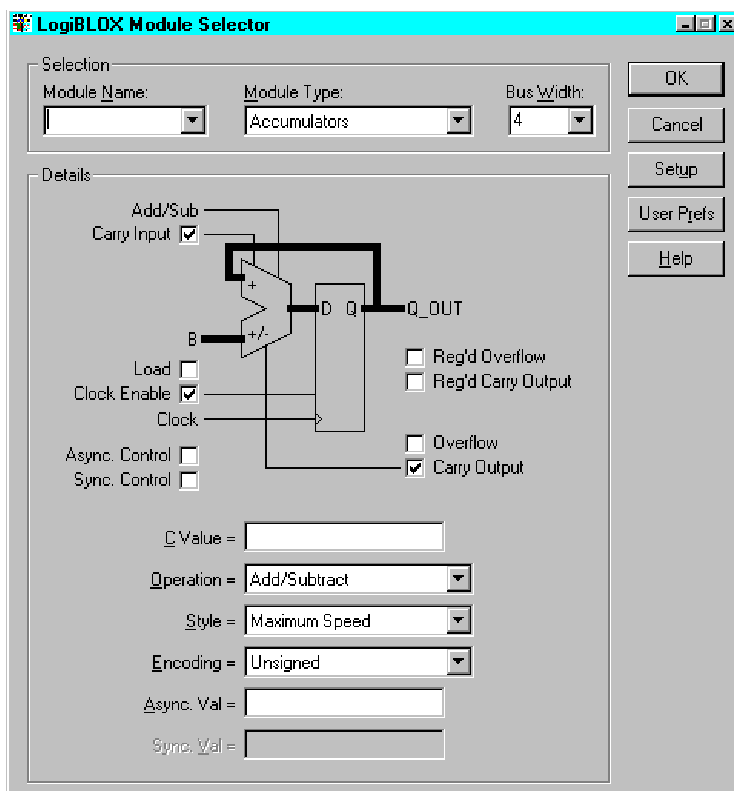


Figure 8-1 LogiBlox Module Selector - Accumulators

Creating LogiBLOX Modules

Once you have opened LogiBLOX, create a module as follows:

1. Enter the name of the module you want to create in the Module Name field, or select an existing one from the list box.
2. Select the type of module from the Module Type list box.
3. Select the bus width for the module from the Bus width list box.
4. Select or deselect optional pins of the module symbol displayed in the Details box by clicking the appropriate check boxes.
5. Click OK. LogiBLOX automatically creates the MOD file, which contains symbol pins and a template for each module, and an EDIF netlist for simulation.

The Project Manager automatically converts the EDIF netlist and reads the generic module file from the \fndtn\active\config\logiblox directory and the MOD file to customize the module symbol. The Project Manager then generates the ALR and ASX files containing the module's binary netlist and ports description and saves the module to the project working library. The module is then ready to use in your project.

LogiBLOX Modules

LogiBLOX has many different modules that you can use in a schematic or HDL synthesis design. The following is a list of the LogiBLOX modules.

Accumulator	Adder/Subtracter	Clock Divider
Comparator	Constant	Counter
Data Register	Decoder	Input/Output (schematic only)
Memory	Multiplexer	Pad (schematic only)
Shift Register	Simple Gates	Tristate Buffers

Using LogiBLOX for Schematic Designs

LogiBLOX modules can be created for use in schematic designs. First, the module is created. Then, the module is added to the schematic like any other library component. For details on this procedure, refer to the "Schematic Designs With Instantiated LogiBLOX Modules" section of the "Design Methodologies - Schematic Flow" chapter.

Using LogiBLOX for HDL Designs

The tools for synthesis-based designs are described in the following subsections.

Module-inferring Tools

Base Express and Foundation Express infer LogiBLOX components where appropriate. Use the HDL Editor to create the HDL file; the Design Wizard can help you with this process.

Module-instantiation Tools

You can instantiate the LogiBLOX components in your HDL code to take advantage of their high-level functionality. Define each LogiBLOX module in HDL code with a component declaration, which describes the module type, and a component instantiation, which describes how the module is connected to the other design elements. For more information, refer to the“HDL Designs with Black Box Instantiation” section of the “Design Methodologies - HDL Flow” chapter.

Documentation

The following documentation is available for the LogiBLOX program:

- The *LogiBLOX Guide* is available with the Xilinx online book collection on the CD-ROM supplied with your software or from the Xilinx web site at http://support.xilinx.com.
- You can access LogiBLOX online help from LogiBLOX or from the Foundation online help system.
- The *Xilinx Software Conversion Guide from XACTstep v5.X.X to XACTstep vM1.X.X* compares XBLOX and LogiBLOX. It describes how to convert an XBLOX design to LogiBLOX. This document is available on the Xilinx web site at http://support.xilinx.com.

Chapter 9

CORE Generator System

The Xilinx CORE Generator System is a design tool that delivers parameterizable COREs optimized for Xilinx FPGAs. It provides the user with a catalog of ready-made functions ranging in complexity frorm simple arithmetic operators such as adders, accumulators, and multipliers, to system-level building blocks including filters, transforms, memories.

This chapter contains the following sections:

- “Setting Up the CORE Generator System on a PC”
- “Accessing the CORE Generator System”
- “Instantiating CORE Generator Modules”
- “Documentation”

Setting Up the CORE Generator System on a PC

The CORE Generator tool can be selected from the setup menu during installation of the Foundation Series 2.1i Design Environment. If you select to install it, it is ready to use from the Foundation Project Manager interface when you start the product.

New COREs can be downloaded from the Xilinx web site and added to the CORE Generator System. The URL for downloading CORES is

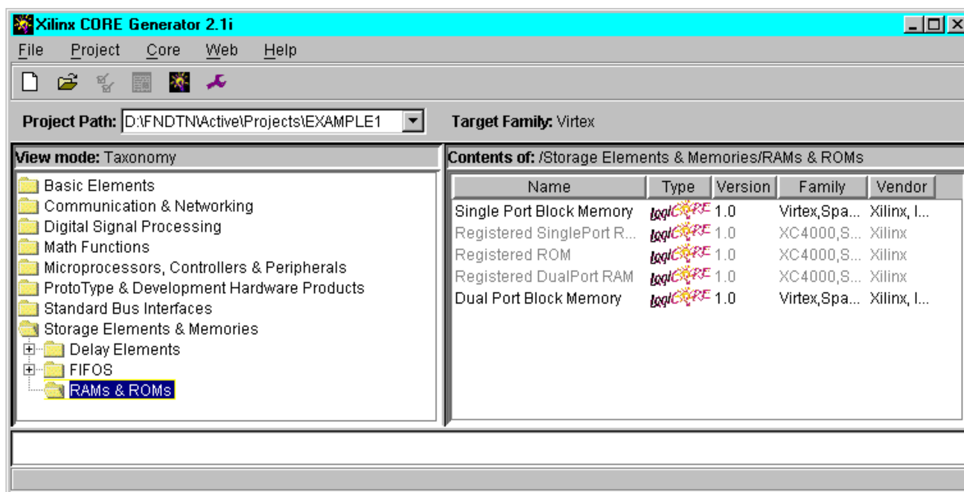
http:/www.xilinx.com/products/logicore/coregen

You can check this web site to verify you have the latest version of each CORE and CORE data sheet.

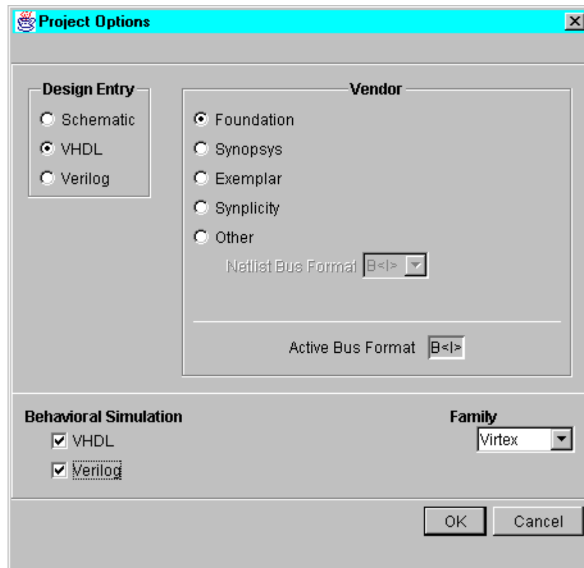
Accessing the CORE Generator System

In the Foundation Series 2.1i software, the CORE Generator System must be started within a valid Foundation project. Within an open project, it can be started from the Project Manager window using **Tools → Design Entry → CORE Generator**. It can also be started within the HDL Editor or the Schematic Editor by selecting **Tools → CORE Generator**.

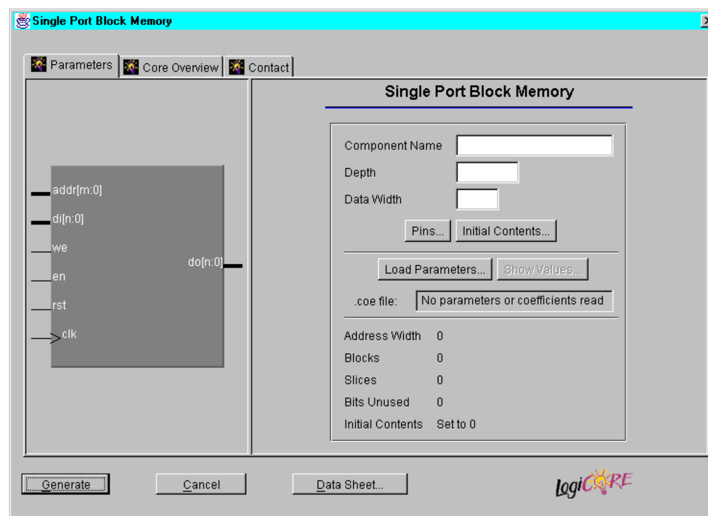
The Xilinx CORE Generator dialog box (an example is shown below) then opens to allow selection of the available COREs. The COREs are categorized on the left side of the window. The specific COREs are selected in the "Contents of" section of the window.



You can select **`Project`** → **`Project Options`** to access the project setup options. However, the Foundation Series software automatically sets the Project Options (shown in the following figure) to the appropriate values for the project. You do not need to set them manually.



You select a CORE by clicking on its name in the "Contents of" section of the CORE Generator window. This opens a new window where you can customize the CORE for your use, view its data sheet, and get other information concerng the CORE. The items that can be customized for a particular CORE depend on what the CORE is. The following figure shows that window that opens when you select a single port block memory core for a Virtex project.

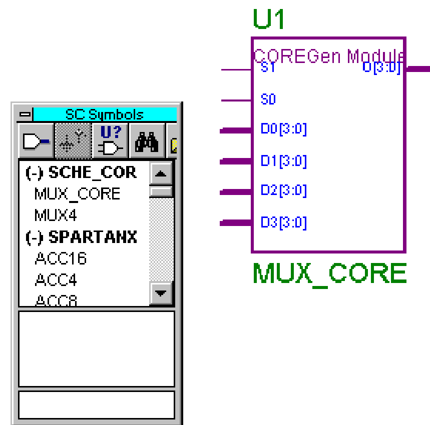


Click the Data Sheet button to view detailed information on the CORE. You must have the Adobe Acrobat Reader installed on your PC to view the data sheet.

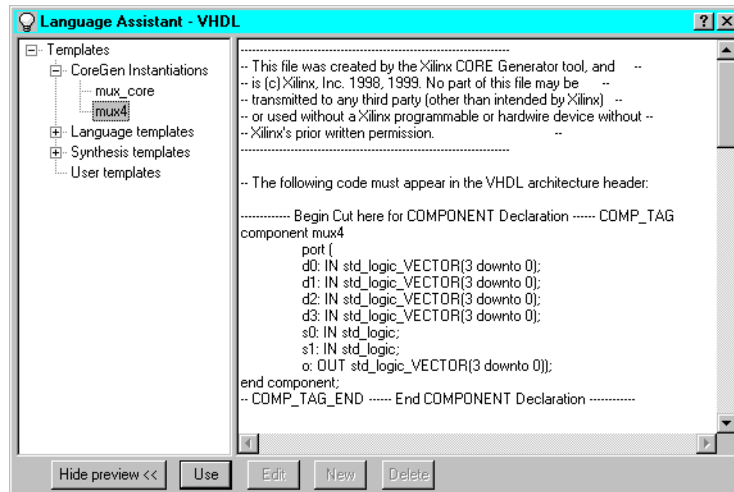
After you customize the CORE for your project, you need to generate the new CORE.

After the CORE has been successfully generated, the new CORE and its related files are placed in the current Foundation project directory for use in a schematic or HDL file.

You can select a schematic CORE from the SC Symbols menu in the Schematic Editor. An example of a schematic CORE is shown in the following figure.



As shown in the figure below, the Language Assistant in the HDL Editor (**Tools** → **Language Assistant**) includes CORE Generator Modules. You can get assistance with instantiating them in VHDL or Verilog.



Instantiating CORE Generator Modules

For information on using COREs in schematic designs, refer to the "Schematic Designs With Instantiated CORE Generator Cores" section of the "Design Methodologies - Schematic Flow" chapter.

For information on using COREs in HDL designs, refer to the "CORE Generator COREs in a VHDL or Verilog Design" section of the "Design Methodologies - HDL Flow" chapter.

Documentation

The following documentation is available for the CORE Generator System:

- The *CORE Generator System 2.1i User Guide* is available from the CORE Generator's help menu by selecting **Help** → **Online Documentation**. This book is in PDF format and requires the Adobe Acrobat Reader to view it.
- You can access the CORE Generator Home Page and other web resources from the CORE Generator's help menu by selecting **Help** → **Help on the Web**.

Chapter 10

Functional Simulation

For schematic and HDL designs, functional simulation is performed before design implementation to verify that the logic you created is correct. Your design methodology determines when you perform functional simulation. Generally, for Schematic Flow projects, you can perform functional simulation directly after you have completed your design within the design entry tools. For HDL Flow projects, you perform functional simulation after the design has been entered and synthesized. However, if your design contains underlying netlists (XNF or EDIF), the design must first be "translated" in the Implementation phase in order to merge these additional netlists.

This chapter contains the following sections:

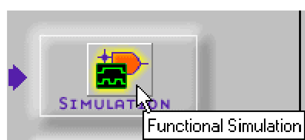
- "Basic Functional Simulation Process"
- "HDL Top-down Methodology"
- "HDL with Underlying Netlists"
- "Simulation Script Editor"
- "Waveform Editing Functions"

Basic Functional Simulation Process

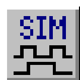
This section describes the basic process for performing simulation.

Invoking the Simulator

You can invoke the simulator from either the Project Manager or directly from the Schematic Editor. To invoke the simulator (for functional simulation) from the Project Manager, click on the Functional Simulation icon in the Simulation button on the Flow tab.



Note: For a schematic design, you can invoke the simulator (for functional simulation) from the Schematic Editor by clicking on the Simulator toolbar button.



Attaching Probes (Schematic Editor Only)

Prior to opening the Simulator, you can attach probes to signals in the Schematic Editor to allow those signals to be automatically loaded into the Simulator Waveform Viewer. Select **`Mode`** → **`Test Points.`** The SC probes toolbox displays. You can select both input and output test points.

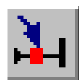


Figure 10-1 Input Test Points

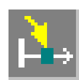
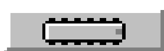


Figure 10-2 Output Test Points

A gray box appears next to the signal name, indicating the placement of the probe. You can add probes at any point during the simulation to add signals to the Waveform Viewer.

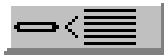
Adding Signals

Once in the Simulator, you can add signals by selecting the Add Signals toolbar button.



Creating Buses

You can create buses by combining any set of signals. Highlight the desired signals and then selecting **`Signal`** → **`Bus`** → **`Combine`**. This same menu may be obtained by right-mouse-clicking in the signal list area of the Waveform Viewer. To expand or collapse the bus, click on the Bus Expansion toolbar button.

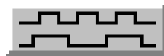


Applying Stimulus

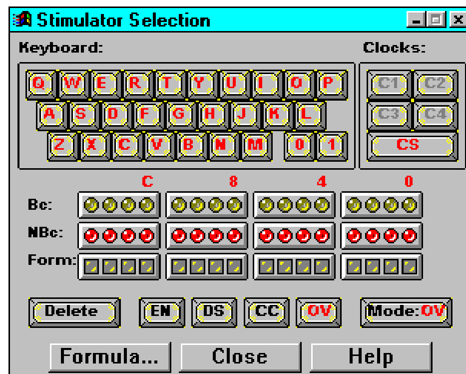
You can apply stimulus in a number of various ways.

Stimulator Selection Dialog

Click the **`Stimulator Selection`** toolbar button (below) to access the Stimulator Selection dialog.



Using Stimulator Selection dialog box, you can add stimulus using keyboard keys, formulas, or output signals of an internal software-generated 16- bit binary counter. For more information on these methods, click **`Help`** in the Stimulator Selection dialog box.



Waveform Test Vectors

A second method of applying stimulus is by editing and using waveform test vectors. Test vectors may be edited and/or created using the **Waveform → Edit...** menu selection. Additionally, test vectors and/or simulation results may be saved by selecting **File → Save Waveform**. These test vector waveforms may then be loaded into the simulator at any time by selecting **File → Load Waveform**.

For more information on using and saving waveforms, refer to the online Help at **Help → Logic Simulator Help Contents → Simulator Reference → Working with Waveforms**.

Script File Macro

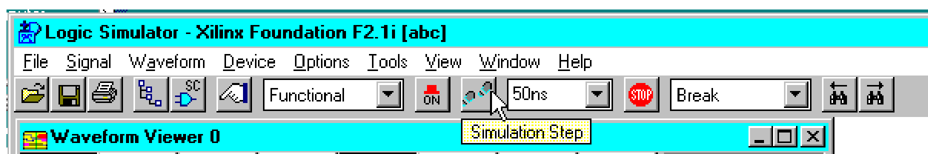
A third method of applying stimulus is through a script file macro. Stimulus is entered through commands in the script file (.cmd) and the simulator displays the input and output response in the Waveform Viewer when the script is run.

Note: Foundation contains a Macro Editor for creating simulation scripts. See the “Simulation Script Editor” section.

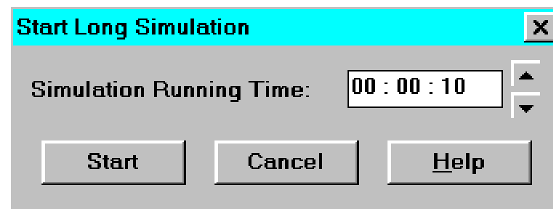
Proper script syntax is documented in the online Help at **Help → Logic Simulator Help Contents → Simulator Reference → Simulation Scripts**. To run a command script, select **File → Run Script File**, and choose the appropriate .cmd file. Additionally, you can edit the .cmd file by selecting **Tools → Script Editor**.

Running Simulation

Click the Simulator Step icon on the Logic Simulator toolbar to perform a simulation step. The length of the step can be changed in the Simulation Step Value box to the right of the Simulation Step box. (If the Simulator window is not open, select **View → Main Toolbar**.)



To start a simulation for an extended amount of time, select **`Options`** → **`Start Long Simulation`**. In the dialog box, enter the desired length of simulation.



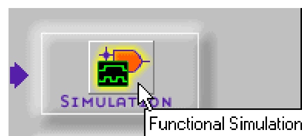
To interrupt the simulation while it is running, click the Stop button in the toolbar.

Save Simulator results by selecting **`File`** → **`Save Simulation State`** and **`File`** → **`Save Waveform`**. Choosing Save Simulation State saves the simulation results and current state of the simulation only. On the other hand, Save Waveforms saves the waveforms in test vector format, allowing you to resimulate the saved waveforms at a later time.

For more information about simulator options and features, refer to the online Help by selecting **`Help`** → **`Logic Simulator Help Contents`**.

HDL Top-down Methodology

If your design has been created and synthesized as a top-level design, then click the Functional Simulation icon on the Simulation button in the Foundation Project Manager to automatically invoke the simulator and load the netlist. The Functional Simulation icon is shown in the following figure.

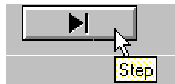


For a description of how to select signals, choose stimulators, and run the simulation, refer to the online help tutorial by selecting **`Help`** → **`Foundation Help Contents`**. Then, under Tools, click on **`Logic Simulator`**. Double click on the **`Getting Started Tutorial`**.

HDL with Underlying Netlists

If your design includes underlying netlists (XNF or EDIF), the design must first be “translated” with the Xilinx Implementation tools in order to merge these additional netlists. Follow the steps below to successfully combine all of the individual modules into one netlist for simulation by “translating” the design in the Xilinx Implementation tools.

1. From the Project Manager, select **Project** → **Create Version**. The Synthesis/Implementation dialog appears. The new version is given the default name shown in the Version Name box unless you change it. Click **OK** and the new version is created.
2. From the Project Manager, select **Project** → **Create Revision**. The New Revision dialog appears. The new revision is given the default name shown in the Name box unless you change it. Click **OK** and the new revision is added to the newly created version from step 1.
3. From the Versions tab, right click on the newly created revision and select **Invoke interactive Flow Engine**.
4. From within the Flow Engine, select the Step button to translate the design.



5. After Translate is complete, go back to the Foundation Project Manager, and select **Tools** → **Simulation/Verification** → **Checkpoint Gate Simulation Control**.
6. Choose the appropriate NGD file from the Revision which was just created, and click **OK**. This invokes the simulator and loads the netlist.

   For a description of how to select signals, choose stimulators, and run the simulation, refer to Steps 2 through 10 in the“Performing Functional Simulation” section of the “Design Methodologies - HDL Flow” chapter.

   Detailed information can also be found in the online help tutorial by selecting **Help** → **Foundation Help Contents**. Then click on **Logic Simulator**. Double click on the **Getting Started**

Tutorial. Another very detailed source can be found by selecting **Help** → **Foundation Help Contents**. Click **CPLD Design Flows**. Scroll down and click **The Functional Simulation Tutorial**. You can also click **Creating a New Test Vector File** to find out detailed information about creating stimuli.

Simulation Script Editor

The Simulation Script Editor facilitates script creation. To access this editor, select **Tools** → **Script Editor** from the Logic Simulator. The Script Editor includes the following features:

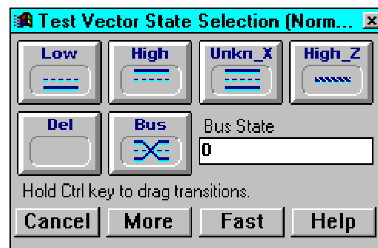
- A Script Wizard for creating new simulation script files
- Syntax highlighting of simulation commands
- Simulation scripts in Macro Assistant (**Tools** → **Macro Assistant**). The Macro Assistant contains examples of Viewsim-compatible macros as well as Aldec® proprietary macros.
- Script command reference (**Help** → **SIM Macros Help**)
- Debugging capabilities
- An online link to the simulator which allows single stepping through command sequences and support for breakpoints

For a description of the Macro Editor and commands, select **Help** → **SIM Macros Help**.

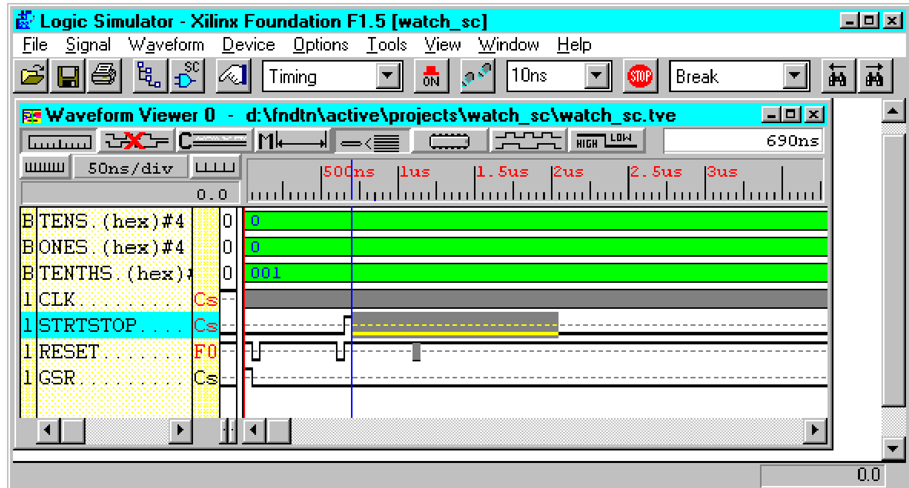
Waveform Editing Functions

Foundation supports dragging of signal transitions within the Waveform Editor. Following is an example.

1. Open the “watch_sc” project in the Project Manager.
2. Click the Functional Simulation icon in the Simulation button.
3. In the Logic Simulator, select **File** → **Load Waveform**.
4. Double click “watch_sc.tve” in the Load Waveform list box.
5. Right click the mouse button. Select Edit from the menu. The Test Vector State Selection box displays.



6. After the Test Vector State Selection box displays, press and hold the left mouse button at the point of the signal that you want to begin altering the signal transition. Drag the mouse to the desired endpoint. The following figure displays an example selection for the STRTSTOP signal.



7. Select High from the Test Vector State Selection box. The low signal transforms to high.

Chapter 11

Design Implementation

This chapter contains the following sections.

- “Versions and Revisions”
- “Implementing a Design”
- “Setting Control Files”
- “Selecting Options”
- “Flow Engine”
- “Implementation Reports”
- “Additional Implementation Tools”

Versions and Revisions

Each project may have multiple versions and revisions. You have complete control over the creation of versions and revisions. They may be used to create snapshots of the project. A generally accepted project structure is to have versions represent logic changes in a design and revisions to represent different implementations on a single design version. The Project Manager graphically displays information about versions and revisions in the Versions tab of the Hierarchy Browser.

Schematic Flow Projects

In Schematic Flow projects, new versions of the design and revisions on each version are associated with the Implementation phase. You determine when to create a new version or revision. For example, versions may represent logic changes in a design such as replacing an AND gate with an OR gate. Revisions may represent different executions of the design flow on a single design version with new imple-

mentation options (for example, changing to a different device in the same device family).

Creating Versions

When you click the **`Implementation`** phase button, the current version/revision is overwritten by default. If you want your changes implemented in a new version, you must explicitly create the new version. This is done by selecting **`Project`** → **`Create Version`** to access the Create Version dialog box shown in the following figure.

Creating Revisions

Revisions represent different implementations of a single design version. You can create a new revision for a version by selecting **`Project`** → **`Create Revision`** to access the Create Revision dialog box shown in the following figure.

In either the Create Version or the Create Revision dialog box, you can select a new device (in the same device family), a new speed for the device, name the version, name the revision, or enter comments.

Click **OK** to create the new revision and/or version. When you are ready to implement the new revision/version, click the **Implementation** phase button.

Or, Click **Run** to create the new revision and/or version and run implementation immediately.

HDL Flow Projects

In HDL Flow projects, new versions of the design are associated with the Synthesis phase. Whenever you change the logic in the design or select a new target device, you must synthesize the design. Revisions on each version are associated with the Implementation phase. As in the Schematic Flow, versions and revisions of a design are overwritten unless you explicitly create a new version or revision.

Creating Versions

You can create a new version of the design by selecting **Project** → **Create Version**. This accesses the Synthesis/Implementation settings dialog box (see the“Synthesis/Implementation Dialog Box” figure) where you can select the Top Level design, name the version, select a target device.

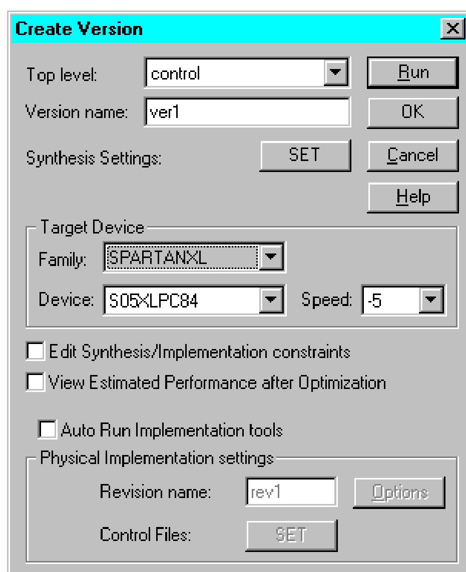


Figure 11-1 Synthesis/Implementation Dialog Box

Updating Versions

If you click the Synthesis phase button to synthesize the design for the first time, the Synthesis/Implementation settings dialog box also appears. However, the Physical Implementation Settings at the bottom of the screen are *not* available. In this case, the design will be synthesized only, not implemented.

Clicking the Synthesis phase button after making changes to an existing version, automatically updates the existing version. No new version is created. You can also update an existing, synthesized version by right-clicking on the functional structure or on the optimized structure in the Versions tab and then selecting **`Update`**.

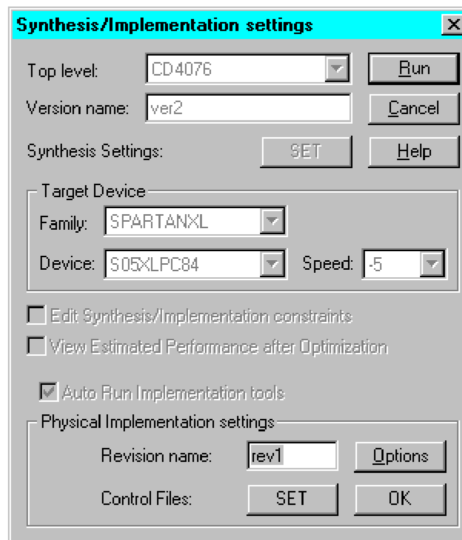
Creating Revisions

R*evisions* of HDL Flow projects represent different implementations of a design version.

You implement the design and create a new revision by clicking the **`Implementation`** phase button. What happens after you click the

Implementation phase button depends on whether this is the first revision for the version or if there are existing revisions of the version.

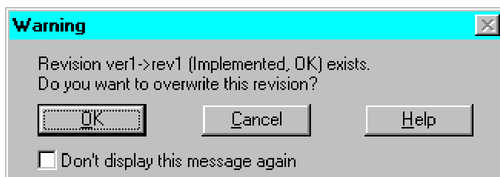
- If this is the first revision and the design has already been synthesized, the Synthesis/Implementation dialog box shown in the following figure appears. Only the Physical Implementation Settings at the bottom of the screen are available at this point. You can name the revision and/or click **`Options`** to access the Options dialog box. When you click **`Run`**, the Flow Engine starts.



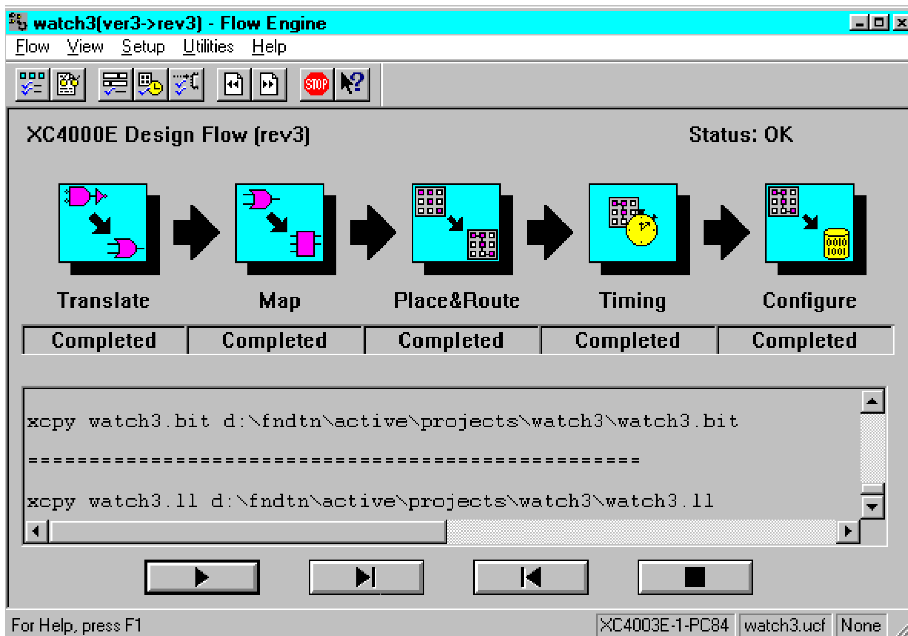
- For later revisions, if you change the design and then click the **`Implementation`** phase button, the design can be automatically updated, synthesized, and implemented. The Project Manager displays a dialog box (shown in the following figure) to inform you that the current version will be overwritten. Select **`OK`** to overwrite the current version or **`Cancel`** to abort the update.

- After you select **`OK`**, the current version is updated and another dialog box (shown in the following figure) appears to inform you

that the current revision will be overwritten. Select **OK** to overwrite the current revision or **Cancel** to abort the update.

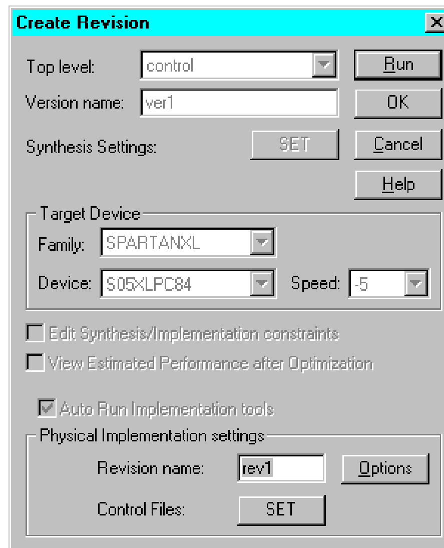


- After you select OK, the Flow Engine appears. If you want to modify the implementation options, you must select **Implementation** → **Options** from the Project Manager menu bar *before* clicking the **Implementation** phase button.
- If you click the **Implementation** phase button and you have made no changes to the design, the completed Flow Engine appears. You can then choose to re-start the Flow Engine in interactive mode.



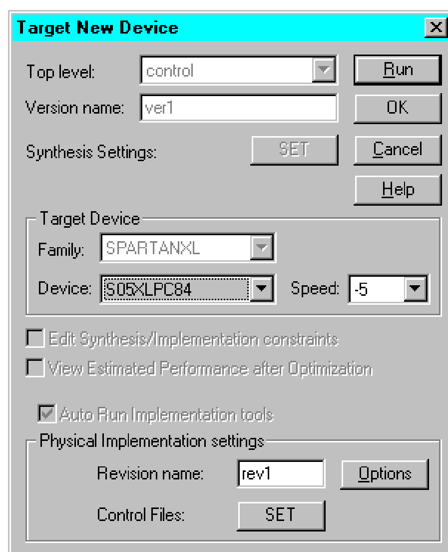
Creating a new Revision

After the design has been synthesized, you can manually create a new revision for a version by selecting **`Project`** → **`Create Revision`**. This accesses the Create Revision dialog box (shown below) where you can name the revision, set the implementation options, or choose to use a Guide or Floorplan file from a previous revision.



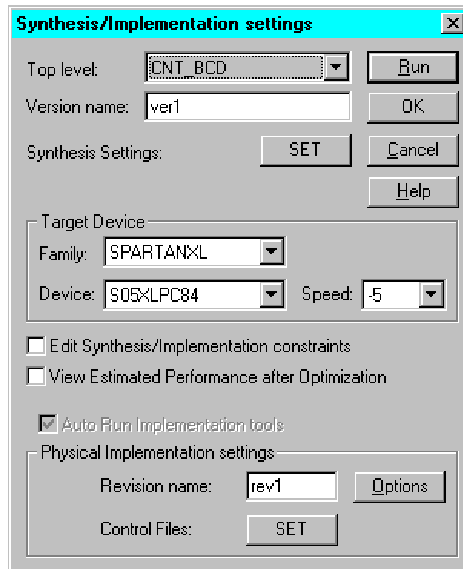
Click **`OK`** to create the revision only. Click **`RUN`** to create the revision and to start the Flow Engine to implement the newly created revision.

Note: You can also right click on an optimized structure in the Versions tab and select **`Target New Device`** to access the Target New Device dialog box shown in the following figure. You may select a new device in the same family or a new speed grade. If you want to target a new device family, you must create a new version and resynthesize the design.



Creating the First Version and Revision in One Step

If the design has not been synthesized, you can create the first version and revision automatically in one step by selecting the **`Implementation`** phase button immediately after design entry. When you click the **`Implementation`** phase button without first synthesizing the design, the Synthesis/Implementation dialog box shown in the following figure appears. All fields are available—the Target Device and Synthesis Settings associated with the synthesis phase as well as the Physical Implementation Setting associated with the implementation phase. You can enter the version and revision information and then click **`OK`**. The Project Manager performs all the necessary processing to synthesis and implement the design to create the first version and revision.



Revision Control

Foundation maintains revision control, meaning that the resultant files from each implementation revision are archived in the project directory. Note that the source design for each version is not archived, only the resulting netlists and files for each revision. Therefore, if you wish to save iterations of the source design (Schematic, HDL files, for example), you should use the project archive functions to archive the appropriate files.

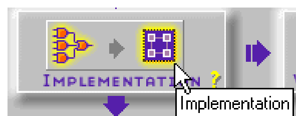
See the "Project Archiving" section in the "Project Toolset" chapter for more information on the Foundation archiving feature.

Implementing a Design

You can implement your design automatically using the Implementation phase button on the Project Manager's Flow tab or you can implement your design by executing the Flow Engine steps separately. The Implementation phase button method is described in this section. Refer to "Flow Engine Controls" section under the "Additional Implementation Tools" section for information on controlling the Flow Engine manually.

When you implement your design using the Implementation phase button, the Project Manager invokes the Flow Engine and automatically performs all steps needed to update your design for implementation.

1. From the Project Manager, click the **`Implementation`** phase button on the project flowchart.



2. The implementation window that appears now depends on whether your project is a Schematic Flow project or an HDL Flow project.

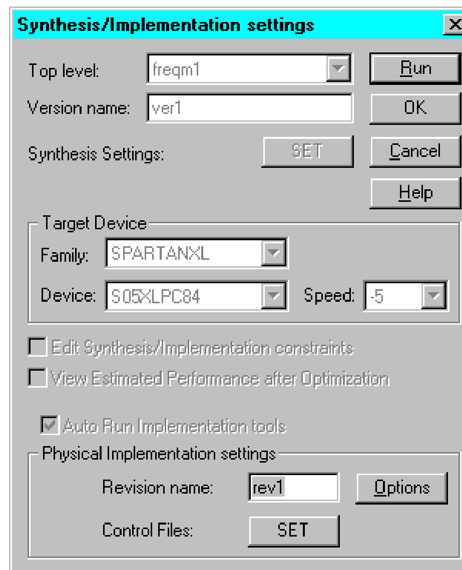
   a) If your project is a Schematic Flow project, the Implement Design dialog box shown in the following figure appears.

   By default, the implementation targets the device selected when the project was created. You can specify a different device within the same family and a new speed grade. If you want to target a device in a different family, you must use **`File`** → **`Project Type`** to select a new family before you click the **`Implement`** phase button.

   Availability of fields in the Implement dialog box depend on whether the design has been implemented before. After the first implementation, only the revision name is available for editing.

   b) If your project is an HDL Flow project, the Synthesis/Implementation dialog box shown in the following figure appears

if the design has been synthesized and no revisions exist for the current synthesized version. (Refer to the “HDL Flow Projects” in the Versions and Revisions section for a description of the various paths available in HDL Flow projects for creating new revisions and updating existing ones for implementation.)



3. Select **`Options`** in the Implement Design dialog box or in the Synthesis/Implementation dialog box to access the Options dialog box. (For HDL Flow projects, you may need to select **`Implementation`** → **`Options`** from the Project Manager menu bar to access the Options dialog box.) Use the Options dialog box to set important implementation options such as selecting a UCF file, specifying templates, or producing optional design data.

   Refer to the “Selecting Options” section for more information on the Options dialog box.

4. After you have selected all of your options, you are ready to initiate the Flow Engine to implement the design.

   - In a Schematic Flow project click **`OK`** on the Options dialog box to close it and return to the Implement Design dialog box. On the Implement Design dialog box, click **`Run`**.

- In an HDL Flow project, click **OK** on the Options dialog box to close it and return to the Synthesis/Implementation dialog box. Click **Run** on the Synthesis/Implementation dialog box to start the Flow Engine. (Refer to the “HDL Flow Projects” - “Creating Revisions” section for additional ways the Flow Engine is accessed when implementing HDL Flow projects.)

Refer to the“Flow Engine” section for more information.

Setting Control Files

You can designate a user constraints file, guide files, or Floorplan files to control the current implementation. You can set the control files from the Project Manager’s Implementation pulldown menu or via the Control Files **Set** button on the Synthesis/Implementation dialog box.

User Constraints File

User constraints files (*design_name*.ucf) contain logic placement and timing requirements to control the implementation of your design. Refer to the “Foundation Constraints” appendix for detailed information on creating .ucf files and on constraint syntax.

If you want to control the implementation of your design with a user constraints file, you can specify this file in the Set Constraints File dialog box. The software implements your design to meet the specified timing requirements and other constraints specified in this file.

1. In the Project Manager, select **Implementation** → **Set Constraints File(s)** to open the dialog box shown in the following figure.

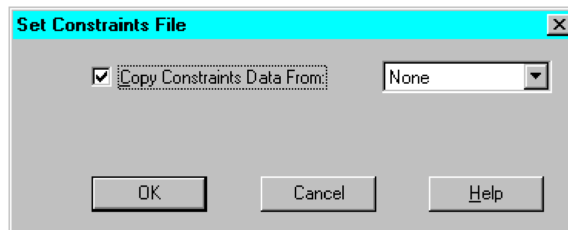


Figure 11-2 Set Constraints File Dialog Box

2. Make sure **Copy Constraints Data From** is selected.

3. In the drop-down list box, choose one of the following.
   - A revision that contains the user constraints file (UCF) you want to use for this implementation
   - **None** if you do not want to copy constraints data
   - **Custom** to guide from a specific file

     If you select **Custom**, the following dialog box appears. Type the name of a specific file in the Constraints File field, or click **Browse** to open a file selection dialog box in which you can choose an existing UCF file.

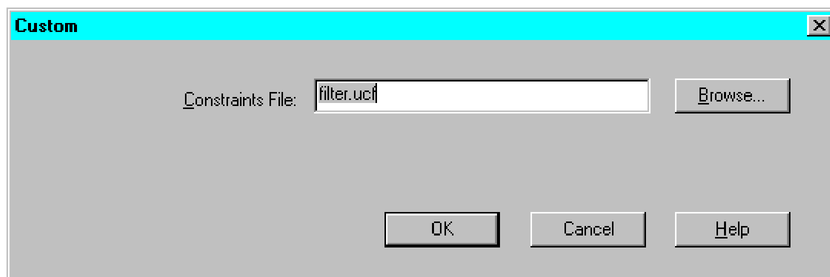


Figure 11-3 Set Constraints File Custom Dialog Box

4. In the Set Constraints File dialog box, click **OK**.

   When you implement the design, the Flow Engine uses the copied data to constrain the implementation.

Guide Files

You can select a previously routed or fitted implementation revision or a guide file to use as a guide for the current implementation. The procedure for guiding your implementation is the same for FPGAs and CPLDs. However, the way the design is implemented differs between the two.

Guiding FPGA Designs

When guiding an FPGA design, the software attempts to use the guide for placing logic and routing signals for the current implementation revision of the design. This ensures consistent implementations

between place and route iterations. Guiding a design for an FPGA works as follows.

- If a component in the new design has the same name as that of the guide design or file, it is placed as in the guide.
- If an unnamed component in the new design is the same type as a component within the guide, it is placed as in the guide.
- If the signals attached to a component in the new design match the signals attached to the component of the guide, the pins are swapped to match the guide, where possible.
- If the signal names in the input design match the guide, and have the same sources and loads, the routing information from the guide design is copied to the new design.

After these components and signals are placed and routed, the remainder of the logic is placed and routed. If you have made only minor changes to your design and want the remaining logic placed and routed exactly as in your guide design, select the Match Guide Design Exactly option. This option locks the placement and routing of the matching logic so that it cannot change to accommodate additional logic.

Note: Setting the Match Guide Design Exactly option is not recommended for synthesis based designs.

Guiding CPLD Designs

For CPLDs, each time you implement your design, a guide file is created (*design_name*.gyd) which contains your pinout information. You can reuse this file in subsequent iterations of your design if you want to keep the same pinouts. If you select a valid implementation revision or guide file name, the pinouts from that file will be used when the design is processed.

Note: You can override guide file locations by assigning locations in your design file or constraints file.

Setting Guide Files

1. In the Project Manager, select **`Implementation`** → **`Set Guide File(s)`** to open the dialog box shown in the following figure.

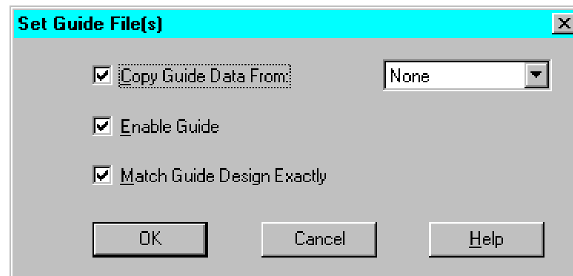


Figure 11-4 Set Guide File(s) Dialog Box

2. Make sure **`Copy Guide Data From`** is selected.
3. In the drop-down list box, choose one of the following.
   - A revision that contains the guide file you want to use for this implementation
   - **`None`** if you do not want to copy a guide file
   - **`Custom`** to guide from any mapped or routed file for FPGAs or fitted file for CPLDs, including designs not generated from within the Design Manager

     If you select **`Custom`**, the following dialog box appears. Type the name of a mapped, routed, or fitted file in the Guide File field, or click **`Browse`** to open a file selection dialog box in which you can choose an existing file. Choose an NCD file for FPGAs or a GYD file for CPLDs. You can also specify a mapping guide file for FPGAs.

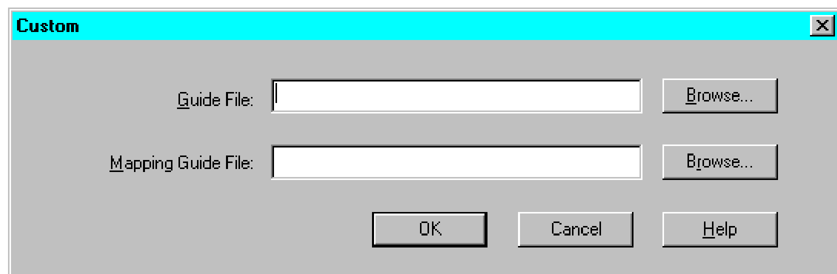


Figure 11-5 Set Guide File(s) Custom Dialog Box

Note: The implementation revision or revision data is based on a placed and routed design. Guide from a placed and routed file rather than a mapped file to reduce runtime. To guide from a mapped file,

you must use the Custom option. If you use this option, you cannot guide mapping using the Set Floorplan File(s) command. Guided mapping is not supported for Virtex devices.

4. In the Set Guide File(s) dialog box, make sure **`Enable Guide`** is selected.

   By default, this option is enabled and instructs the software to use the specified guide file. If you do not want to guide your design but want to keep your guide file intact, disable this option.

5. For FPGA devices, select **`Match Guide Design Exactly`** if you want to lock the placement and routing of matching logic.

   If you do *not* select this option, the guide files are used as a starting point only. This allows the mapper, placer, and router greater flexibility in accommodating design modifications, often resulting in greater overall success.

Note: For synthesis-based designs, use the Match Guide Design Exactly option only if the guide file is from the same design version.

6. Click **`OK`**.

   When you implement the design, the Flow Engine uses the copied data to guide the implementation.

Floorplan Files

When you use the Floorplanner, an MFP file is generated that contains mapping information. You can instruct the Design Manager to use this file as a guide for mapping an implementation revision using the Set Floorplan File(s) command. To use this command, you must select an implementation revision that has been mapped and modified using the Floorplanner. For information on using the Floorplanner, see the *Floorplanner Guide*.

Note: If you use the Set Floorplan File(s) command you cannot guide mapping using the Set Guide File(s) command Custom option. The Set Floorplan File(s) command is available for the XC4000, Virtex, and Spartan device families only.

1. From the Project Manager, select **`Implementation`** → **`Set Floorplan File(s)`** to open the dialog box shown in the following figure.

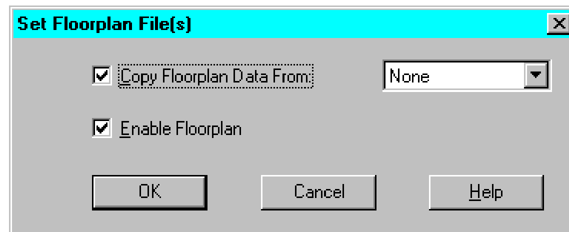


Figure 11-6 Set Floorplan File(s) Dialog Box

2. Make sure **`Copy Floorplan Data From`** is selected.
3. In the drop-down list box, choose one of the following.
   - A revision that contains the floorplan files you want to use for this implementation
   - **`None`** if you do not want to copy floorplan data
   - **`Custom`** to guide from any mapped file in your file system, including designs not generated from within the Design Manager

     If you select **`Custom`**, the following dialog box appears. Type the name of a specific file in the Floorplanning File field, or click **`Browse`** to open a file selection dialog box in which you can choose an existing file. Specify an FNF file for the Floorplanning File field and an MFP file for the Floorplanned Guide File field.

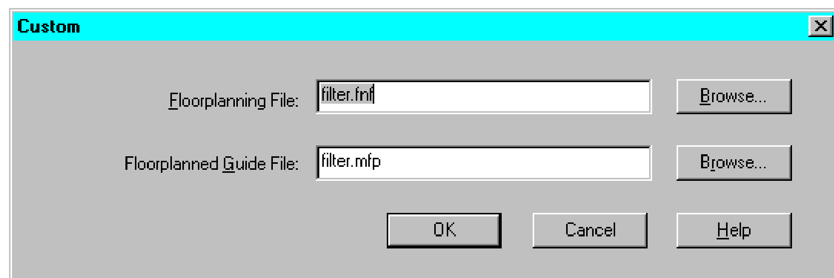


Figure 11-7 Set Floorplan File(s) Custom Dialog Box

4. In the Set Floorplan File(s) dialog box, make sure **`Enable Floorplan`** is selected.

Note: By default, this option is enabled and instructs the software to use the specified Floorplanner file. If you do not want to guide your design but want to keep your Floorplanner file intact, disable this option.

5. Click **OK**.

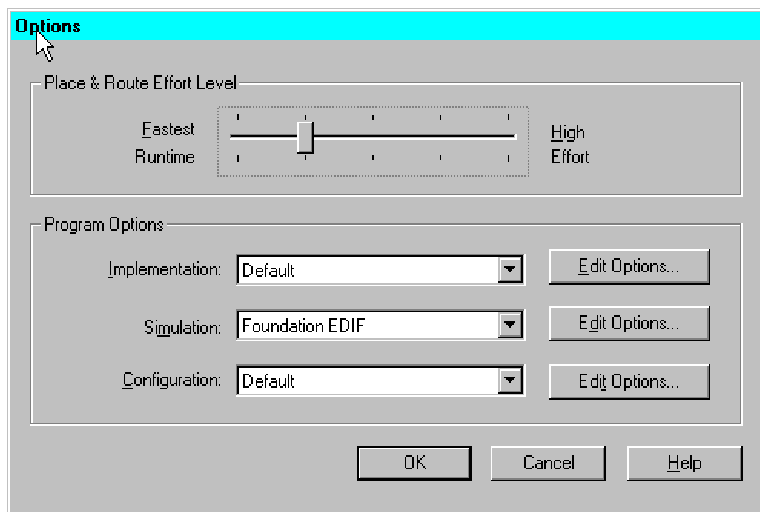
   The Flow Engine uses the copied data to guide the implementation.

Selecting Options

For FPGAs, options specify how a design is optimized, mapped, placed, routed, and configured. For CPLDs, they control how a design is translated and fit. Implementation options are specified in the Options dialog box.

In a Schematic Flow project, select **Options** on the Implement Design dialog box to access the Options dialog box shown in the following figure.

In an HDL Flow project, select **Options** on the Synthesis/Implementation dialog box to access the Options dialog box.



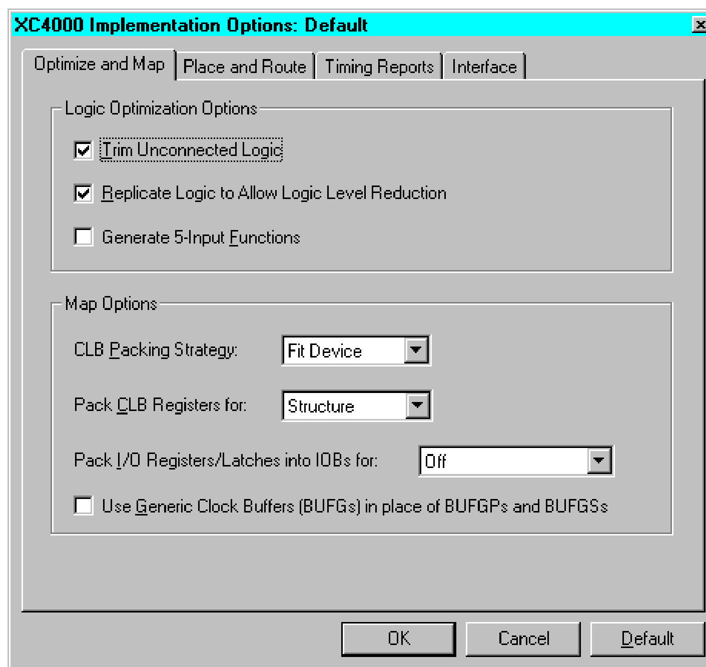
Place & Route Effort Level

The Place & Route Effort Level setting controls how much effort the placer and router should use to best place and route a design at the expense of longer runtimes.

Program Options

The Program Options are grouped into implementation, simulation, and configuration options. These can be used to create customized templates for various implementation styles you may want to try. For example, one implementation style could be Quick Evaluation, while another could be Timing Constraint Driven.

You can have multiple templates in a project. By choosing a template, you are choosing an implementation, simulation, or configuration style. In the Program Option portion of the Options Dialog, select **`Edit Options`** for Implementation, Simulation, or Configuration to access the associated template. An example of the Implementation Options dialog box is shown in the following figure. The options shown in each template depends on the target device family. For detailed information on the templates for each device family, refer to the "Implementation Flow Options" chapter of the *Design Manager/ Flow Engine Guide*.



Implementation Templates

Implementation templates control how the software maps, places, routes, and optimizes an FPGA design and how the software fits a CPLD design.

Simulation Templates

Simulation templates control the creation of netlists in terms of the Xilinx primitive set, which allow you to simulate and back-annotate your design. In back-annotation, physical design data is distributed back to the logic design to perform back-end simulation. You can perform front and back-end simulation on both pre- and post-routed designs. Select a simulation template to use from the Simulation drop-down list.

Configuration Templates (FPGAs)

Configuration templates control the configuration parameters of a device, the startup sequence, and readback capabilities. Select a

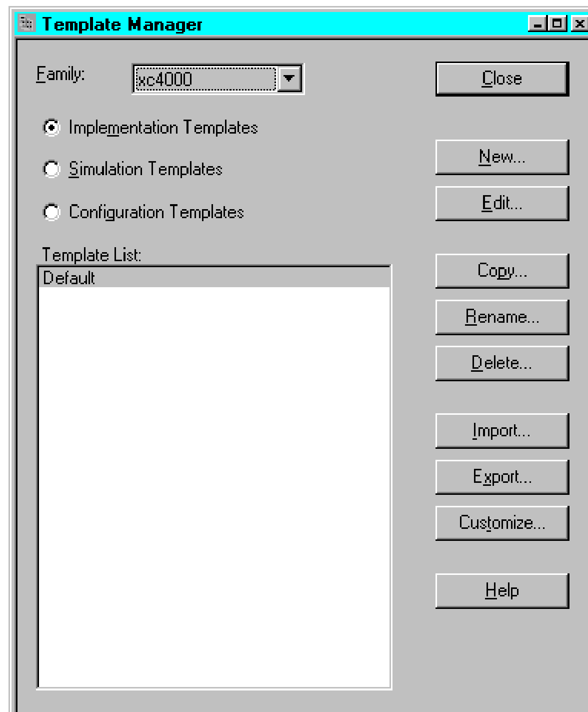
configuration template to use in this implementation from the Configuration drop-down list.

Note: Configuration options are supported for the FPGA device families only. There are no configuration options for the CPLD families.

Template Manager

To create new templates or as an alternate way to access the templates use the Template Manager.

1. From the Project Manager menu, select **`Tools`** → **`Utilities`** → **`Implementation Template Manager.`** This opens the Template Manager dialog box.



2. From the Template Manager dialog box, click the button associated with the type of template on which you wish to perform an operation (Configuration, Simulation, or Implementation).

3. Click the appropriate button for the operation (New, Edit, Copy, and so forth).

4. After you have made all of your template entries, click **Close**.

Flow Engine

The Project Manager's Implementation phase button automatically invokes and controls the Flow Engine to process the design. The Flow Engine interface prominently displays the status of each implementation stage as shown in the following figures.

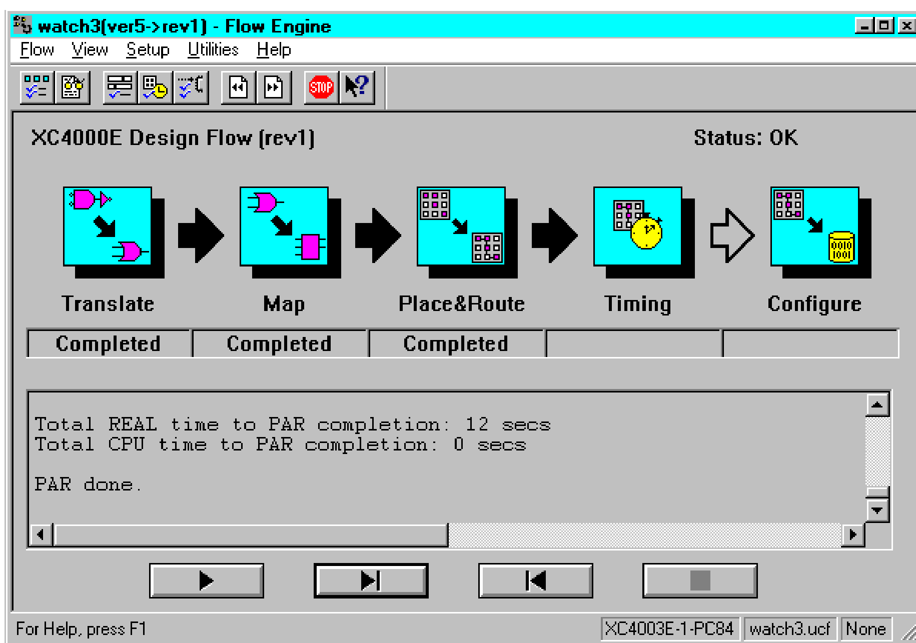


Figure 11-8 Flow Engine - FPGA Processing

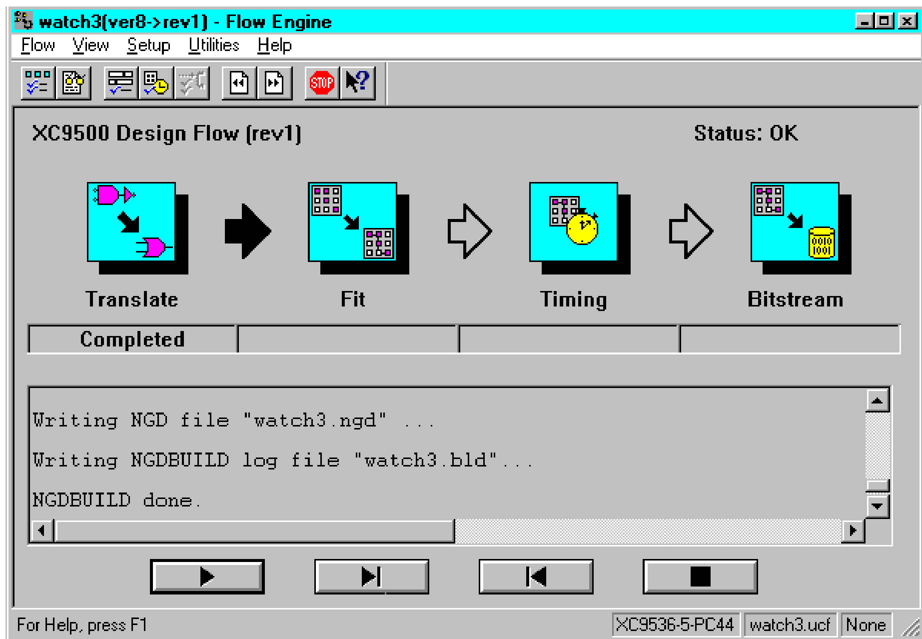


Figure 11-9 Flow Engine - CPLD Processing

When you process your design, the Flow Engine translates the design file into the Xilinx internal database format (NGD). The Flow Engine then implements your design and generates bitstream data.

Process indicators in the Flow Engine main window show you which of these stages is currently processing. The arrows between each step turn black after the previous step is completed. Underneath each process indicator, a progress bar shows the status of each processing step, whether running, completed, aborted, or failed.

By default, all implementation processing stages are performed. If you want, you can control processing of your design by using the STOP button in the Flow Engine Tool bar to stop processing after a designated stage. Refer to the “Flow Engine Controls” section under the “Additional Implementation Tools” section for more information on additional features of the Flow Engine.

For an overview of the processing and file manipulation performed for FPGAs and CPLDs, refer to the“File Processing Overview” appendix.

Translate

The Flow Engine's first step, Translate, merges all of the input netlists. This is accomplished by running NGDBuild. For a complete description of NGDBuild, refer to the "NGDBuild" chapter of the *Development System Reference Guide.*

MAP (FPGAs)

The MAP program maps a logical design to a Xilinx FPGA. The input to a mapping program is an NGD file, which contains a logical description of the design in terms of both the hierarchical components used to develop the design and the lower level Xilinx primitives, and any number of NMC (macro library) files, each of which contains the definition of a physical macro. MAP first performs a logical DRC (Design Rule Check) on the design in the NGD file. MAP then maps the logic to the components (logic cells, I/O cells, and other components) in the target Xilinx FPGA. The output design is an NCD (Native Circuit Description) file physically representing the design mapped to the components in the Xilinx FPGA. The NCD file can then be placed and routed.

You can run the Mapper from a GUI (Flow Engine) or command line. For a description of the GUI, see the *Design Manager/Flow Engine Guide,* an online book. For a description of the MAP command and its options, see the *Development System Reference Guide,* an online book.

Place and Route (FPGAs)

After an FPGA design has undergone the necessary translation to bring it into the NCD (Native Circuit Description) format, it is ready to place and route. This phase is done by PAR (Xilinx's Place and Route program). PAR takes an NCD file, places and routes the design, and produces an NCD file, which is used by the bitstream generator (BitGen). The output NCD file can also act as a guide file when you place and route the design again after you make minor changes to it.

In the Xilinx Development System, PAR places and routes a design using a combination of two methods.

- Cost-based — This means that placement and routing are performed using various cost tables which assign weighted values to relevant factors such as constraints, length of connection and available routing resources.

- Timing-Driven — PAR places and routes a design based upon your timing constraints.

For a complete description of PAR, see the “PAR—Place and Route” chapter in the *Development System Reference Guide*.

CPLD Fitter

The CPLD Fitter implements designs for the XC9500/XL devices. The Fitter outputs the files listed below.

- The Fitting report (*design_name*.rpt) lists a summary and detailed information about the logic and I/O pin resources used by the design, including the pinout, error and warning messages, and Boolean equations representing the implemented logic.
- The Static timing report (*design_name*.tim) shows a summary report of worst-case timing for all paths in the design; it optionally includes a complete listing of all delays on each individual path in the design.
- The Guide file (*design_name*.gyd) contains all resulting pinout information required to reproduce the current pinout if you run the Lock Pins command before the next time the fitter is run for the same design. (The Guide file is written only upon successful completion of the fitter.) Multi-Pass Place and Route and Guide Files are not accessible via the Foundation Project Manager. Access these functions through the standalone Design Manager (**Start** → **Programs** → **Accessories** → **Design Manager**.
- The Programming file (*design_name*.jed for XC9000) is a JEDEC-formatted (9k) programming file to be downloaded into the CPLD device.
- Timing simulation database (design_name.nga) is a binary database representing the implemented logic of the design, including all delays, consisting of Xilinx simulation model primitives (simprims).

For detailed information about implementing CPLD designs, refer to the *CPLD Design Techniques* and *CPLD Flow Tutorial* in the Foundation on-line help.

Configure (FPGAs)

After the design has been completely routed, you must configure the device so that it can execute the desired function. 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. The BIT file contains all of the configuration information from the NCD file defining the internal logic and interconnections of the FPGA, plus device-specific information from other files associated with the target device. The binary data in the BIT file can then be downloaded into the FPGA's memory cells, or it can be used to create a PROM file.

For a complete description of BitGen, see the "BitGen" chapter in the *Development System Reference Guide*. This chapter also explains how to use the command line to run BitGen.

Within the Flow Engine, BitGen runs as part of the Configure process. For details consult the various configuration template options in the "Working with Templates" section in the "Using the Design Manager" chapter of the *Design Manager/Flow Engine Guide*.

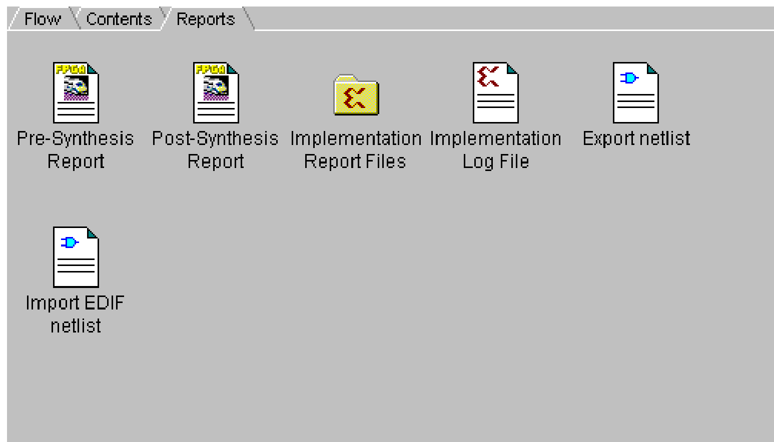
Bitstream (CPLDs)

At the end of a successful CPLD implementation, a .jed programming file is created. The JTAG Programmer uses this file to configure XC9500/XL/XV CPLD devices.

Implementation Reports

The implementation reports provide information on logic trimming, logic optimization, timing constraint performance, and I/O pin assignment. To access the reports, select the Reports tab from Project Flow area of the Project Manager. Double click the Implementation Report Files icon to access the implementation reports.

The Implementation Log on the Reports tab is a record of all the implementation processing.



Double click the Implementation Report Files icon to access the Report Browser shown in the following figures. To open a particular report, double click its icon.

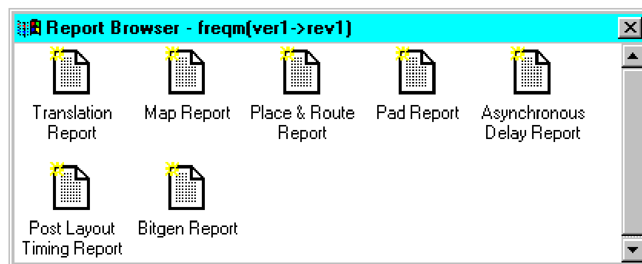


Figure 11-10 Report Browser - FPGAs

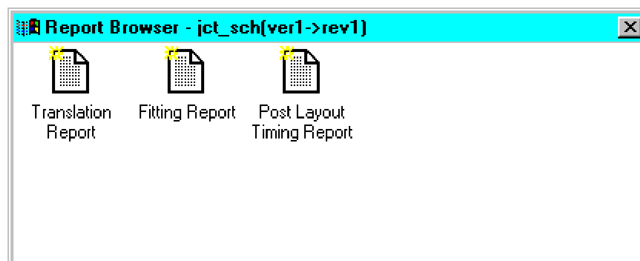


Figure 11-11 Report Browser - CPLDs

Translation Report

The translation report (.bld) contains warning and error messages from the three translation processes: conversion of the EDIF or XNF style netlist to the Xilinx NGD netlist format, timing specification checks, and logical design rule checks. The report lists the following:

- Missing or untranslatable hierarchical blocks
- Invalid or incomplete timing constraints
- Output contention, loadless outputs, and sourceless inputs

Map Report (FPGAs)

The Map Report (.mrp) contains warning and error messages detailing logic optimization and problems in mapping logic to physical resources. The report lists the following information:

- Erroneously removed logic. Sourceless and loadless signals can cause a whole chain of logic to be removed. Each deleted element is listed with progressive indentation, so the origins of removed logic sections are easily identifiable; their deletion statements are not indented.
- Logic that has been added or expanded to optimize speed.
- The Design Summary section lists the number and percentage of used CLBs, IOBs, flip-flops, and latches. It also lists occurrences of architecturally-specific resources like global buffers and boundary scan logic.

Note: The Map Report can be very large. To find information, use key word searches. To quickly locate major sections, search for the string '---' , because each section heading is underlined with dashes.

Place and Route Report (FPGAs)

The Place and Route Report (.par) contains the following information.

- The overall placer score which measures the "goodness" of the placement. Lower is better. The score is strongly dependent on the nature of the design and the physical part that is being targeted, so meaningful score comparisons can only be made between iterations of the same design targeted for the same part.

- The Number of Signals Not Completely Routed should be zero for a completely implemented design. If non-zero, you may be able to improve results by using re-entrant routing or the multi-pass place and route flow.
- The timing summary at the end of the report details the design's asynchronous delays.

Pad Report (FPGAs)

The Pad Report lists the design's pinout in three ways.

- Signals are referenced according to pad numbers.
- Pad numbers are referenced according to signal names.
- PCF file constraints are listed. This section of the Pad Report can be cut and pasted into the .pcf file after the SCHEMATIC END; statement to preserve the pinout for future design iterations.

Fitting Report (CPLDs)

The Fitting Report (*design_name*.rpt) lists summary and detailed information about the logic and I/O pin resources used by the design, including the pinout, error and warning messages, and Boolean equations representing the implemented logic.

Post Layout Timing Report

A timing summary report shows the calculated worst-case timing for the logic paths in your design.

Additional Implementation Tools

From the Project Manager's Tools menu, you can select **`Tools`** → **`Implementation`** to access the additional implementation tools described below.

Constraints Editor

You can invoke the Xilinx implementation Constraints Editor by selecting **`Tools`** → **`Implementation`** → **`Constraints Editor`**.

The Xilinx Constraints Editor is a Graphical User Interface (GUI) that provides you with a convenient way to create user constraints files without having to learn constraints syntax.

The Constraints Editor interface consists of a main window, three tab windows for creating global, port, and advanced constraints, and a number of dialog boxes.

Information on the Xilinx Constraints Editor can be found in the *Constraints Editor Guide*, an online book.

Flow Engine Controls

You can invoke and run the Flow Engine manually by selecting **Tools** → **Implementation** → **Flow Engine**. Be aware that when invoked from the Tools menu, Flow Engine processing is not under Project Management control.

Controlling Flow Engine Steps

If you want to implement your design in separate steps instead of automatically with the Implementation phase button, use the following procedure.

1. Create a new revision by selecting **Project** → **Create Revision**. In the New Revision dialog box, you can accept the defaults or change the target device, speed, and revision name. Click **OK** to create the revision.
2. In the Project Manager Versions tab, select the revision.
3. Select **Tools** → **Implementation** → **Flow Engine** from the Project Manager's menu bar.
4. If you want to modify the implementation option settings, select **Setup** → **Options** from the menu in the Flow Engine to access the Options dialog box.
5. Set the appropriate options in the Options dialog box.

   Refer to the "Selecting Options" section for information on the Options dialog box.
6. Click **OK** to return to the Flow Engine.
7. To start the Flow Engine, do one of the following.
   - In the Flow Engine window, select **Flow** → **Run**.

- Select **Flow** → **Step** to single step through the implementation process.

Optionally, you can select **Setup** → **Stop After** and select where to stop processing.

Running Re-Entrant Routing on FPGAs

You can use re-entrant routing to further route an already routed design. The design maintains its current routing and additional routing is added. You can reroute connections by running cost-based cleanup, delay-based cleanup, and additional re-entrant route passes. Cleanup passes attempt to minimize the delays on all nets and decrease the number of routing resources used. Cost-based cleanup routing is faster while delay-based cleanup is more intensive.

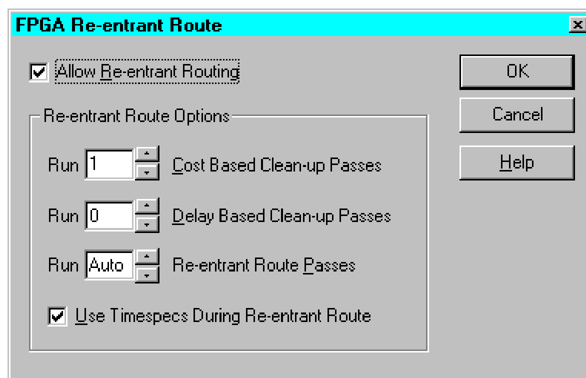
Re-entrant routing offers the following advantages.

- Cleanup passes significantly reduce delays, especially on non-timing driven runs.
- For timing-driven runs, cleanup passes can improve timing on elements not covered by timing constraints.
- For designs which do not meet timing goals by a narrow margin, delay-based cleanup passes can reorganize routing so that additional re-entrant route passes enable the design to meet timing goals.

Note: Re-entrant Routing is supported for the FPGA device families only.

Use the following procedure to perform Re-Entrant Routing.

1. In the Project Manager Versions tab, select an implemented revision.
2. Select **Tools** → **Implementation** → **Flow Engine** from the Project Manager's menu bar.
3. Select **Setup** → **FPGA Re-entrant Route** from the Flow Engine to access the FPGA Re-entrant Route dialog box.



4. Select **`Allow Re-entrant Routing`** to route the previously routed design again.

5. Select a number between 1 and 5 for the **`Run _ Cost-Based Cleanup Passes`** field.

   These cleanup passes reroute nets if the new routing uses less costly resources than the original configuration. Cost is based on pre-determined cost tables. Cost-based cleanup usually has a faster runtime than the delay-based cleanup, but does not reduce delays as significantly.

Note: If you run both cost-based and delay-based cleanup passes, the cost-based passes run first.

6. Select a number between 1 and 5 for the **`Run _ Delay-Based Cleanup Passes`** field.

   These cleanup passes reroute nets if new routing will minimize the delay for a given connection. Delay-based cleanup usually produces faster in-circuit performance.

7. Select a number between 1 to 2000 for the **`Run _ Re-entrant Route Passes`** field to run additional re-entrant routing passes.

   These passes are either timing driven or non-timing driven depending on whether you specified timing constraints.

8. Select **`Use Timespecs During Re-entrant Route`** if you want to reroute the design within the specified timing constraints in your design file.

9. Click **OK**. This causes the Place and Route icon in the Flow Engine to show a loop back arrow and the Re-Entrant route label.

10. If you are specifying timing or location constraints, you have the option to relax them to give PAR more flexibility. If you modify the UCF file, you must step backwards with the Flow Engine and re-run Translation in order to incorporate the changes.

    Since your design is already implemented, step back to the beginning of Place and Route using the Step Backward button at the bottom of the Flow Engine, and then click the button to start again.

Configuring the Flow

You can configure the implementation flow and control certain aspects of the Flow Engine interface. To configure the flow, use the following procedure.

1. In the Project Manager Versions tab, select an implemented revision (or create a new revision).

2. Select **Tools** → **Implementation** → **Flow Engine** from the Project Manager's menu bar.

3. From the Flow Engine menu, select **Setup** → **Advanced** to access the Advanced dialog box.

4. Select a state from the Implementation State list box to update the Flow Engine as to which implementation state was last completed.

Note: The advanced setting is not used in normal Flow Engine use. It is used if some processing on the design was performed outside of the Project Manager or Flow Engine framework, such as in the FPGA Editor. It can also be used if you ran the Flow Engine Step Back

button by mistake and want to reset the implementation state to its original state.

5. Select **`Use Flashing to Indicate Heartbeat`** to enable flashing icons to indicate that a process step is being processed. A trade-off of this feature is that flashing icons slow down the implementation process.
6. Click **`OK`**.

Floorplanner

The Floorplanner is a graphical placement tool that gives you control over placing a design into a target FPGA. You can access the Floorplanner through **`Tools`** → **`Implementation`** → **`Floorplanner`** on the Project Manager's menu bar.

Floorplanning is an optional methodology to help you improve performance and density of a fully, automatically placed and routed design. Floorplanning is particularly useful on structured designs and data path logic. With the Floorplanner, you see where to place logic in the floorplan for optimal results, placing data paths exactly at the desired location on the die.

With the Floorplanner, you can floorplan your design prior to or after running PAR. In an iterative design flow, you floorplan and place and route, interactively. You can modify the logic placement in the Floorplan window as often as necessary to achieve your design goals. You can save the iterations of your floorplanned design to use later as a constraints file for PAR.

The Floorplanner displays a hierarchical representation of the design in the Design Hierarchy window using hierarchy structure lines and colors to distinguish the different hierarchical levels. The Floorplan window displays the floorplan of the target device into which you place logic from the hierarchy. The following figure shows the windows on the PC version.

Logic symbols represent each level of hierarchy in the Design Hierarchy window. You can modify that hierarchy in the Floorplanner without changing the original design.

You use the mouse to select the logic from the Design Hierarchy window and place it in the FPGA represented in the Floorplan window.

Alternatively, you can invoke the Floorplanner after running the automatic place and route tools to view and possibly improve the results of the automatic implementation.

FPGA Editor

The FPGA Editor is a graphical application for displaying and configuring FPGAs. You can use the FPGA Editor to place and route critical components before running the automatic place and route tools on your designs. You can also use the FPGA Editor to manually finish placement and routing if the routing program does not completely route your design. In addition, the FPGA Editor reads from and writes to the Physical Constraints File (PCF).

For a description of the FPGA Editor, see the *FPGA Editor Guide,* an online book.

You can access the FPGA Editor through **Tools** → **Implementation** → **FPGA Editor** on the Project Manager's menu bar.

CPLD ChipViewer

The ChipViewer provides a graphical view of the CPLD fitting report. With this tool you can examine inputs and outputs, macrocell details, equations, and pin assignments. You can examine both pre-fitting and post-fitting results.

More information on using the CPLD ChipViewer is available in that tool's online help (**Tools** → **Implementation** → **CPLD ChipViewer** → **Help**) or from the Umbrella Help menu accessed by **Help** → **Foundation Help Contents** → **Advanced Tools** → **ChipViewer**.

Locking Device Pins

You can automatically generate pin locking constraints in your UCF file for use with other Xilinx implementation tools. Pinout information is taken from a placed NCD file for FPGAs or a fitted GYD file for CPLDs.

To lock device pins, do the following.

1. From the Versions tab in the Project Manager window, select an implementation revision.

2. Select **`Tools`** → **`Implementation`** → **`Lock Device Pins`** from the Project Manager menu bar.
3. When the Lock Pins Status confirmation dialog box appears, click **`OK`** or click **`View Lock Pins Report`** to view the report.

   Pin locking constraints that created with this command are added to your UCF file in the PINLOCK section.

If you want to view the report after you have dismissed the Lock Pins Status dialog box, use **`Tools`** → **`Implementation`** → **`Lock Pins Report`** from the Project Manager.

Chapter 12

Verification and Programming

This chapter contains the following sections.

- "Overview"
- "Timing Simulation"
- "Timing Analyzer"
- "In-Circuit Verification"
- "Downloading a Design"

Overview

Design verification is the process of testing the functionality and performance of your design. Design verification should occur throughout your design process. Foundation supports three complementary methods for design verification. These are described below.

- Simulation

  You can perform simulations to determine if the timing requirements and functionality of your design have been met.

  - Functional Simulation can be performed in Schematic Flow projects immediately after design entry and in HDL Flow projects after synthesis. Refer to the"Functional Simulation" chapter for information on Functional Simulation.
  - Timing Simulation is performed during the Implementation phase. The"Timing Simulation" section of this chapter discusses design verification using Timing Simulation.

- Static timing analysis

  Static timing analysis is best for quick timing checks of your design.

- For Foundation Express users, the Express Time Tracker provides post-synthesis, pre-implementation timing analysis for HDL Flow projects. Refer to "Express Time Tracker" section of the "Design Methodologies - HDL Flow" chapter for information.
- For Schematic Flow projects and HDL Flow projects, static timing analysis can be done at two different stages of the Implementation phase for FPGA devices: after Map or after Place and Route. It can be done after Fit for CPLDs. Refer to the "Timing Analyzer" section in this chapter for information on static timing analysis within the Implementation phase.

- In-circuit verification

  As a final test, you can verify how your design performs in the target application. In-circuit verification tests the circuit under typical operating conditions. To perform in-circuit verification, you download your design bitstream into a device with the Xilinx XChecker cable. Refer to "In-Circuit Verification" in the Device Programming section of this chapter for information.

When the design meets your requirements, the last step in its processing is downloading the design and programming the target device.

Timing Simulation

Timing simulation verifies that your design runs at the desired speed for your device under worst-case conditions. It can verify timing relationships and determine the critical paths for the design under worst-case conditions. It can also determine whether the design contains set-up or hold violations.

The procedures for functional and timing simulation are nearly identical. Functional simulation is performed *before* the design is placed and routed and simulates only the functionality of the logic in the design. Timing simulation is performed *after* the design is placed and routed and uses timing information based on the delays in the placed and routed design. Timing simulation describes the circuit behavior far more accurately than Functional simulation.

Like functional simulation, you must use input stimulus to run the simulation. To create stimulus, refer to the"Functional Simulation" chapter.

Note: Naming the nets during your design entry is very important for both functional and timing simulation. This allows you to find the nets in the simulations more easily than looking for a machine-generated name

Generating a Timing-annotated Netlist

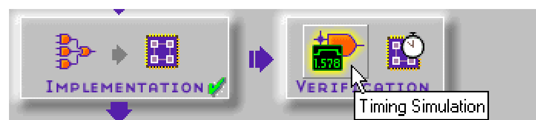
Before performing timing simulation on your design, you must generate a timing-annotated netlist by implementing the design as follows.

1. Within the Project Manager, click the Implementation icon.
   a) For Schematic Flow projects, this opens the Implement Design dialog box.
   b) For HDL Flow projects, this opens the Synthesis/Implementation dialog box.
2. Click the `Options` button. This opens the Options dialog box.
3. Verify that the Simulation Template is **`Foundation EDIF`**. (Change it to **`Foundation EDIF`**, if necessary.)
4. Implement the design.
   a) For Schematic Flow projects, click **`Run`** in the Implement Design dialog box.
   b) For HDL Flow projects, click **`OK`** in the Synthesis/Implementation dialog box.

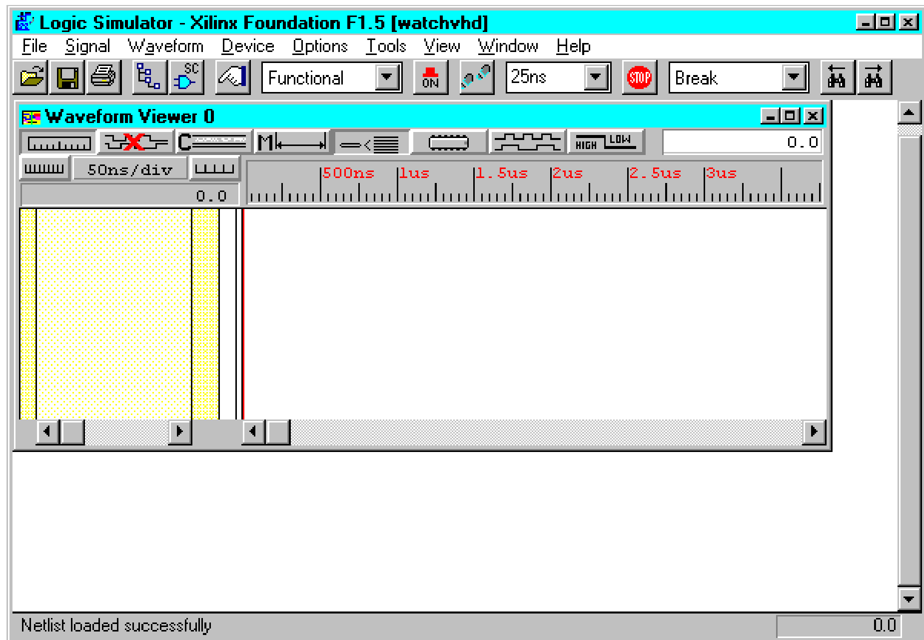
Basic Timing Simulation Process

After the design has been implemented and timing simulation data produced as described in“Generating a Timing-annotated Netlist” section, you can perform a timing simulation. This section describes the basic steps to perform timing simulation.

1. Open the Timing Simulator by clicking the Timing Simulation icon on the Verification phase button.



2. The implementation timing netlist is loaded into the simulator. The Waveform View window displays on top of the Logic Simulator window as shown in the following figure.



3. Simulate the design as described in the“Functional Simulation” chapter. Although the procedure is the same for functional and timing simulation, you are now simulating based on a design with worst-case delays in the timing simulator.

4. Use the controls from the Simulator window to verify your design.

Timing Analyzer

The Timing Analyzer performs static timing analysis of an FPGA or CPLD design. A static timing analysis is a point-to-point analysis of a design network. It does not include insertion of stimulus vectors.The FPGA design must be mapped and can be partially or completely placed, routed, or both. The CPLD design must be completely placed and routed (fitted).

The Timing Analyzer verifies that the delay along a given path or paths meets your specified timing requirements. It organizes and displays data that allows you to analyze the critical paths in a circuit, the cycle time of the circuit, the delay along any specified path, and the paths with the greatest delay. It also provides a quick analysis of the effect of different speed grades on the same design.

The Timing Analyzer works with synchronous systems composed of flip-flops and combinatorial logic. In synchronous designs, the Timing Analyzer takes into account all path delays, including clock-to-Q and setup requirements while calculating the worst-case timing of the design. However, the Timing Analyzer does not perform setup and hold checks. You must use a simulation tool for these checks.

The Timing Analyzer creates timing analysis reports, which you customize by applying filters with the Path Filters menu commands.

For a complete description of the Timing Analyzer, see the *Timing Analyzer Guide*, an online manual.

Post Implementation Static Timing Analysis

Post-implementation timing reports incorporate all delays to provide a comprehensive timing summary. If an implemented design has met all of your timing constraints, then you can proceed by creating configuration data and downloading a device. On the other hand, if you identify problems in the timing reports, you can try fixing the problems by increasing the placer effort level or using re-entrant routing. You can also redesign the logic paths to use fewer levels of logic, tag the paths for specialized routing resources, move to a faster device, or allocate more time for the paths.

Edit the Implementation template (from the Project Manager, select **Implementation → Options**) to modify the Place & Route effort level. For information on re-entrant routing, see the “Running Re-Entrant Routing on FPGAs” section in the “An Introduction to Design Implementation chapter.

Summary Timing Reports

Summary reports show timing constraint performance and clock performance. Implementing a design in the Flow Engine can automatically generate summary timing reports. To create summary timing reports, perform the following steps.

Summary reports show timing constraint performance and clock performance. Implementing a design in the Flow Engine can automatically generate summary timing reports. To create summary timing reports, perform the following steps:

1. Open the Options dialog box (**Implementation** → **Options**) from the Project Manager) and select **Edit Options** for the Implementation template.
2. Select the Timing Reports tab.
3. For a post-map report, select **Produce Logic Level Timing Report**. For a post-PAR report select **Produce Post Layout Timing Report**.
4. To modify the reports to highlight path delays or paths that have failed timing constraints, select a report format.
5. After MAP or PAR has completed, the respective timing reports appear in the Report Browser.

Detailed Timing Analysis

To perform detailed timing analysis, select **Tools** → **Simulation/Verification** → **Interactive Timing Analyzer** from the Project Manager menu. You can specify specific paths for analysis, discover paths not affected by timing constraints, and analyze the timing performance of the implementation based on another speed grade. For path analysis, perform the following:

1. Choose sources. From the Timing Analyzer menu, select **Path Filters** → **Custom Filters** → **Select Sources**.
2. Choose destinations. From the Timing Analyzer menu, select **Path Filters** → **Custom Filters** → **Select Destinations**.
3. To create a report, select one of the options under the Analyze menu.

To switch speed grades, select **Options** → **Speed Grade**. After a new speed grade is selected, all new Timing Analyzer reports will be based on the design running with new speed grade delays. The design does not have to be re-implemented, because the new delays are read from a separate data file.

In-Circuit Verification

As a final test, you can verify how your design performs in the target application. In-circuit verification tests the circuit under typical operating conditions. Because you can program your Xilinx devices repeatedly, you can easily load different iterations of your design into your device and test it in-circuit.

To verify your FPGA designs in-circuit, download your design bitstream into a device with the Xilinx XChecker cable.

Refer to the following section for more information on programming your target device.

Downloading a Design

To download your design, you must successfully run implementation to create a configuration bitstream. Xilinx provides the MultiLINX cable, Parallel Cable III, or the XChecker cable, depending on which development system you are using, to download the bitstream to a device.

You can use the XChecker cable or MultiLINX cable to read back and verify configuration data. Detailed cable connection and daisy-chain information is provided in the *Hardware Debugger Guide*.

Note: The Xilinx Parallel Cable III can be used for FPGA and CPLD design download and readback, but it does not have a design verification function.

With the XChecker cable, you can use the Hardware Debugger Pick function to take snapshots of the circuit at specific clock cycles. You can obtain these snapshots by performing serial readback of the nodes during in-circuit operation. With the Hardware Debugger software, you can speed up your analysis by limiting the readback bitstream to only those nodes and clock cycles in which you have interest.

You can also use the XChecker cable to probe your design after you download it. Probing internal nodes allows you to pinpoint the location of any design problems.

Use the XChecker cable when you do not want to specify additional IOBs and routing resources on your Xilinx FPGA for probing. This

allows you to decide how you want to probe after you have downloaded your design.

The MultiLINX cable is compatible in supporting Readback & Verify for all the FPGAs supported by the XChecker cable. Plus, the MultiLINX cable supports the XC4000E/XL, Spartan/XL, and Virtex devices whose bit file size is more than 256K bits.

Note: Debug is not currently available with the MultiLINX cable.

JTAG Programmer

You can use JTAG programmer to download, read back, and verify design configuration data and to perform functional tests on any FPGA or CPLD device. You can also use it to probe internal logic states of a CPLD design.

JTAG Programmer uses sequences of JTAG instructions to perform programming and verification operations.

You need to provide JEDEC files for each XC95000 device, BIT files for each Xilinx FPGA device in the JTAG programming chain, and BSDL files for the remaining devices.

JTAG Programmer supports the following Xilinx device families: XC4000E/L/EX/XL/XV/XLA, XC5200, XC95000/XL/XV, Spartan/XL, and Virtex.

There are two download cables available for use with the JTAG Programmer. The first is an RS232 serial cable known as the XChecker Cable. The second is the Parallel Download Cable which can be connected to a PC's parallel printer port.

There are a few advantages to be considered in selecting a cable:

- The XChecker Cable connects to the serial port of PCs.
- The Parallel Cable has better drive capability. The Parallel Cable can drive up to 10 XC9500 devices in a boundary-scan chain, and the XChecker Cable can drive up to 4 XC9500 devices.
- The Parallel Cable is at least 5 times faster.

Refer to the *JTAG Programmer Guide* in the online book collection for complete information on the JTAG Programmer.

Hardware Debugger (FPGAs only)

The Hardware Debugger is a graphical interface that allows you to download a design to a device, verify the downloaded configuration, and display the internal states of the programmed device. Use the program to perform the following tasks.

- Download a BIT file to an FPGA or a PROM file to a daisy chain of FPGAs. Downloading refers to the process of programming or configuring a device.
- Verify the configuration data of a single device using an XChecker cable. Verification consists of reading the configuration data that was sent to the device and comparing it to the original bitstream to ensure that the design was correctly received by the device.
- Debug the internal logic states of a configured device using an XChecker cable. Debugging consists of reading internal device states to verify that the design is functioning correctly.

You can use the Hardware Debugger with the following Xilinx devices: XC3000A/L, XC3100A/L, XC4000E/EX/L/XL/XV, XC5200, Virtex, Spartan/XL, and Virtex.

Your target board can be either a Xilinx FPGA demonstration board or your own board. The demonstration boards can be used to test most designs.

Refer to the *Hardware Debugger Guide* in the online book collection for complete information on the Hardware Debugger.

PROM File Formatter

The PROM File Formatter provides a graphical user interface that allows you to format BIT files into a PROM file compatible with Xilinx and third-party PROM programmers. It is also used to concatenate multiple bitstreams into a single PROM file for daisy chain applications. This program also enables you to take advantage of the Xilinx FPGA reconfiguration capability, as you can store several applications in the same PROM file.

PROM files are also compatible with the Xilinx Hardware Debugger software. You can use the Hardware Debugger to download a PROM file to a single FPGA or to a daisy chain of FPGA devices.

A Xilinx PROM file consists of one or more data streams. In this context, a data stream represents all the configuration data required to implement a given application. Each data stream contains one or more BIT files and once saved, will have a separate preamble and length count.

The PROM file can be formatted in one of three industry standard formats: Intel MCS-86®, Tektronix TEKHEX, and Motorola EXORmacs.

Note: You can also format BIT files into a HEX format file. This file type is not considered a PROM file since you cannot use it to program PROM devices. A HEX format file is ordinarily used as input to user-defined programs for microprocessor downloads.

You can store PROM files in PROM devices or on your computer. In turn, you can use the files to program your FPGA devices either from a PROM device on your board or from your computer using a serial or parallel cable. Refer to the *Hardware Debugger Reference/User Guide* for more information.

Refer to the *PROM File Formatter Guide* in the online book collection for complete information on the PROM File Formatter.

Appendix A

Glossary

This appendix contains definitions and explanations for terms used in the *Foundation Series 2.1i User Guide*.

ABEL

ABEL is a high-level language (HDL) and compilation system produced by Data I/O Corporation.

actions

In state machines, actions are HDL statements that are used to make assignments to output ports or internal signals. Actions can be executed at several points in a state diagram. The most commonly used actions are state actions and transition actions. State actions are executed when the machine is in the associated state. Transition actions are executed when the machine goes through the associated transition.

Aldec

An Electronic Design Automation (EDA) vendor. Aldec provides the Foundation Project Manager, Schematic Editor, Logic Simulator, and HDL Editor.

aliases

Aliases, or signal groups, are useful for probing specific groups of nodes.

analyze

A process performed to check the syntax of an HDL file.

architecture

Architecture is the common logic structure of a family of programmable integrated circuits. The same architecture can be realized in different manufacturing processes. Examples of Xilinx architectures are the XC4000, Spartan, and XC9500 devices.

attribute

Attributes are instructions placed on symbols or nets in a schematic to indicate their placement, implementation, naming, direction, or other properties.

binary encoding

Using the minimum number of registers to encode a state machine is called binary, or maximal, encoding, because the registers are used to their maximum capacity. Each register represents one bit of a binary number.

BitGen

The BitGen program produces a bitstream for Xilinx FPGA device configuration. The BitGen program displays as the Configure step within the Flow Engine.

Black Box Instantiation

Instantiation where the synthesizer is not given the architecture or modules.

block

A group consisting of one or more logic functions. Also called CLB.

breakpoint

A breakpoint is a condition for which a simulator must stop to perform simulation commands.

buffer

A buffer is an element used to increase the current or drive of a weak signal and, consequently, increase the fanout of the signal. A storage element.

bus

A bus is a group of nets carrying common information. In LogiBLOX, bus sizes are declared so that they can be expanded accordingly during design implementation.

CLB

The Configurable Logic Block (CLB). Constitutes the basic FPGA cell. It includes two 16-bit function generators (F or G), one 8-bit function generator (H), two registers (flip-flops or latches), and reprogrammable routing controls (multiplexers).

component

A component is an instantiation or symbol reference from a library of logic elements that can be placed on a schematic.

condition

If there is more than one transition leaving a state in a state machine, you must associate a condition with each transition. A condition is a Boolean expression.

constraint

Constraints are specifications for the implementation process. There are several categories of constraints: routing, timing, area, mapping, and placement constraints.

Using attributes, you can force the placement of logic (macros) in CLBs, the location of CLBs on the chip, and the maximum delay between flip-flops. CLBs are arranged in columns and rows on the FPGA device. The goal is to place logic in columns on the device to attain the best possible placement from the standpoint of both performance and space.

constraints editor

A GUI tool that you can use to enter design constraints. In Foundation2.1i, there are two constraint editors. The Express Constraints Editor is integrated with the synthesis tools for pre-implementation optimization. It available only in the Foundation Express product configuration. The Xilinx Constraints Editor is integrated with the Design Implementation tools and available in all product configurations.

constraints file

A constraints file specifies constraints (location and path delay) information in a textual form. An alternate method is to place constraints on a schematic.

CORE Generator

A software tool for generating and delivering parameterizable cores optimized for FPGAs. Like LogiBLOX modules, cores are high-level modules. The library includes cores as complex as DSP filters and multipliers, and as simple as delay elements. You can use these cores as building blocks in order to complete your designs more quickly.

CPLD

Complex Programmable Logic Device (CPLD) is an erasable programmable logic device that can be programmed with a schematic or a behavioral design. CPLDs constitute a type of complex PLD based on EPROM or EEPROM technology. They are characterized by an architecture offering high speed, predictable timing, and simple software.

The basic CPLD cell is called a macrocell, which is the CPLD implementation of a CLB. It is composed of AND gate arrays and is surrounded by the interconnect area.

CPLDs consume more power than FPGA devices, are based on a different architecture, and are primarily used to support behavioral designs and to implement complex counters, complex state machines, arithmetic operations, wide inputs, and PAL crunchers.

CPLD fitter

The CPLD Fitter implements designs for the XC9500 devices.

design entry tools

The Foundation design entry tools consist of the Schematic Editor, HDL Editor, and State Editor. The tools can be accessed via the Design Entry button in the Project Manager's Flow tab. The optional Base Express and Foundation Express packages contain VHDL and Verilog design entry tools.

design implementation tools

A set of tools that comprise the mainstream programs used for Xilinx design implementation. Many of these tools are invoked automatically by the Flow Engine.Those tools include NGDBuild, MAP, PAR, NGDAnno, TRCE, all the NGD2 translator tools, BitGen, and PROMGen. The GUI-based tools are Design Manager/Flow Engine, Constraint Editor, FPGA Editor, Floorplanner, PROM File Formatter, JTAG Programmer, and Hardware Debugger.

Design Manager

Xilinx Alliance graphical user interface for managing and implementing designs. In Foundation, a standalone version of the Alliance Design Manager can be accessed from **`Start`** → **`Programs`** → **`Xilinx Foundation Series 2.1i`** → **`Accessories`** → **`Design Manager`**.

effort level

Effort level refers to how hard the Xilinx Design System (XDS) tries to place and route a design. The effort level settings are.

- High, which provides the highest quality placement but requires the longest execution time. Use high effort on designs that do not route or do not meet your performance requirements.
- Medium, which is the default effort level. It provides the best trade-off between execution time and high quality placement for most designs.
- Low, which provides the fastest execution time and adequate placement results for prototyping of simple, easy-to-route designs. Low effort is useful if you are exploring a large design space and only need estimates of final performance.

elaborate

The HDL process that combines the individual parts of a into a single design and then synthesizes the design.

Express Compiler

Engine used to compile VHDL and Verilog code for the Base Express and Foundation Express products.

Express Constraints Editor

GUI available in the synthesis phase of Foundation Express containing spreadsheets used to define specific optimization requirements. See also Express Time Tracker. The Express Time Tracker is available at the end of the synthesis phase of Foundation Express. It contains spreadsheets detailing optimization results.

Express Time Tracker

GUI available at the end of the synthesis phase of Foundation Express. It contains spreadsheets detailing optimization results.

Finite State Machine Editor

Design Entry tool to create and edit state machine descriptions.

fitter

The fitter is the software that maps a PLD logic description into the target CPLD.

floorplanning

Floorplanning is the process of choosing the best grouping and connectivity of logic in a design.

It is also the process of manually placing blocks of logic in an FPGA where the goal is to increase density, routability, or performance.

FPGA

Field Programmable Gate Array (FPGA), is a class of integrated circuits pioneered by Xilinx in which the logic function is defined by the customer using Xilinx development system software after the IC has been manufactured and delivered to the end user. Gate arrays are another type of IC whose logic is defined during the manufacturing process. Xilinx supplies RAM-based FPGA devices.

FPGA applications include fast counters, fast pipelined designs, register intensive designs, and battery powered multi-level logic.

FPGA Editor

The FPGA Editor is a graphical application for displaying and configuring FPGAs. You can use the FPGA Editor to place and route critical components before running the automatic place and route tools on your designs.

FSM

Finite State Machine.

functional simulation

A process to test the logic in a design before implementation to determine if it works properly. Uses unit delays because timing information is not available before implementation.

guided design

Guided design is the use of a previously implemented version of a file for design mapping, placement, and routing. Guided design allows logic to be modified or added to a design while preserving the layout and performance that have been previously achieved.

guided mapping

An existing NCD file is used to “guide” the current MAP run. The guide file may be used at any stage of implementation: unplaced or placed, unrouted or routed. In 2.1i, guided mapping is supported through the Project Manager.

HDL

Hardware Description Language. A language that describes circuits in textual code. The two most widely accepted HDLs are VHDL and Verilog.

HDL Editor

Design entry tool to produce/edit HDL files. The HDL Editor also provides a syntax checker, language templates, and access to the synthesis tools.

HDL Flow

An HDL Flow project can contain VHDL, Verilog, or schematic top-level designs. It can contain underlying schematic, HDL (VHDL or Verilog), or State Machine designs. The entire design is always exported in HDL terms and synthesized. Top level schematic designs in an HDL Flow are exported as schematic netlists, optimized by the synthesis tool, and then exported for Implementation. On the Project Manager Flow tab, a Synthesis button is included between the Design Entry and Implementation buttons for this project type.

hierarchical designs

A hierarchical design is a design composed of multiple sheets at different levels of your schematic or of multiple HDL files with a top-level modules calling other modules.

Hierarchy Browser

The left-hand portion of the Foundation Project Manager that displays the current design project. The browser also displays two tabs, Files and Versions.

implementation

For FPGAs, implementation is the mapping, placement and routing of a design. For CPLDs, implementation is the fitting of a design.

Implementation Constraints Editor

See Xilinx Constraints Editor.

instantiation

Incorporating a macro or module into a top-level design. The instantiated module can be a LogiBLOX module, VHDL module, Verilog module, schematic module, state machine, or netlist.

Language Assistant

The Language Assistant in the HDL Editor provides templates to aid you in common VHDL and Verilog constructs, common logic functions, and architecture-specific features.

Library Manager

The Library Manager is the tool used to perform a variety of operations on the design entry tools libraries and their contents. These libraries contain the primitives and macros that you use to build your design.

locking

Lock placement applies a constraint to all placed components in your design. This option specifies that placed components cannot be unplaced, moved, or deleted.

LogiBLOX

A Xilinx design tool for creating high-level modules such as counters, shift registers, RAM, and multiplexers.The modules are customizable and pre-optimized for Xilinx FPGA and CPLD architectural features. All Xilinx devices with the exception of Virtex support LogiBLOX.

logic

Logic is one of the three major classes of ICs in most digital electronic systems — microprocessors, memory, and logic. Logic is used for data manipulation and control functions that require higher speed than a microprocessor can provide.

Logic Simulator

The Logic Simulator, a real-time interactive design tool, can be used for both functional and timing simulation of designs. The Logic Simulator creates an electronic breadboard of your design directly from your design's netlist. The Logic Simulator can be accessed by clicking the Functional Simulation icon on the Simulation button or the Timing Simulation icon on the Verification button in the Project Manager.

macro

A macro is a component made of nets and primitives (flip-flops or latches) that implements high-level functions, such as adders, subtractors, and dividers. Soft macros and RPMs are types of macros.

A macro can be unplaced, partially placed, or fully placed, and it can also be unrouted, partially routed, or fully routed. See also "physical macro."

MAP

The MAP program maps a logical design to a Xilinx FPGA. The input to a mapping program is an NGD file. The MAP program is initiated within the Flow Engine during Implementation.

mapping

Mapping is the process of assigning a design's logic elements to the specific physical elements that actually implement logic functions in a device.

MRP file

An MRP (mapping report) file is an output of the MAP run. It is an ASCII file containing information about the MAP run. The information in this file contains DRC warnings and messages, mapper warnings and messages, design information, schematic attributes, removed logic, expanded logic, signal cross references, symbol cross references, physical design errors and warnings, and a design summary.

NCD file

An NCD (netlist circuit description) file is the output design file from the MAP program, LCA2NCD, PAR, or EPIC. It is a flat physical design database correlated to the physical side of the NGD in order to provide coupling back to the user's original design. The NCD file is an input file to MAP, PAR, TRCE, BitGen, and NGDAnno.

net

A net is a logical connection between two or more symbol instance pins. After routing, the abstract concept of a net is transformed to a physical connection called a wire.

A net is an electrical connection between components or nets. It can also be a connection from a single component. It is the same as a wire or a signal.

netlist

A netlist is a text description of the circuit connectivity. It is basically a list of connectors, a list of instances, and, for each instance, a list of the signals connected to the instance terminals. In addition, the netlist contains attribute information.

NGA file

An NGA (native generic annotated) file is an output from the NGDAnno run. An NGA file is subsequently input to the appropriate NGD2 translation program.

NGDAnno

The NGDAnno program distributes delays, setup and hold time, and pulse widths found in the physical NCD design file back to the logical NGD file. NGDAnno merges mapping information from the NGM file, and timing information from the NCD file and puts all this data in the NGA file.

NGDBuild

The NGDBuild program performs all the steps necessary to read a netlist file in XNF or EDIF format and create an NGD file describing the logical design. The NGDBuild program executes as the Translate step within the Flow Engine.

NGD file

An NGD (native generic database) file is an output from the NGDBuild run. An NGD file contains a logical description of the design expressed both in terms of the hierarchy used when the design was first created and in terms of lower-level Xilinx primitives to which the hierarchy resolves.

NGM file

An NGM (native generic mapping) file is an output from the MAP run and contains mapping information for the design. The NGM file is an input file to the NGDAnno program.

one-hot encoding

For state machines, in one-hot encoding, an individual state register is dedicated to one state. Only one flip-flop is active, or hot, at any one time.

optimization

Optimization is the process that decreases the area or increases the speed of a design. Foundation allows you to control optimization of a design on a module-by-module basis. This means that you have the ability to, for instance, optimize certain modules of your design for speed, some for area, and some for a balance of both.

optimize

The third step in the FPGA Express synthesis flow. In this stage, the implemented design is re-synthesized with constraints the user specifies. This is the final step before writing out the XNF file from FPGA Express.

PAR (Place and Route)

PAR is a program that takes an NCD file, places and routes the design, and outputs an NCD file. The NCD file produced by PAR can be used as a guide file for reiterative placement and routing. The NCD file can also be used by the bitstream generator, BitGen.

path delay

A path delay is the time it takes for a signal to propagate through a path.

PCF file

The PCF file is an output file of the MAP program. It is an ASCII file containing physical constraints created by the MAP program as well as physical constraints entered by you. You can edit the PCF file from within the FPGA Editor. (FPGA only)

PDF file

Project Description File. The PDF file contains library and other project-specific information. Not to be confused with an Adobe Acrobat document with the same extension.

physical Design Rule Check (DRC)

Physical Design Rule Check (DRC) is a series of tests to discover logical and physical errors in the design. Physical DRC is applied from the FPGA Editor, BitGen program, PAR program, and Hardware Debugger. By default, results of the DRC are written into the current working directory.

physical macro

A physical macro is a logical function that has been created from components of a specific device family. Physical macros are stored in files with the extension .nmc. A physical macro is created when the FPGA Editor is in macro mode. See also “macro.”

pin

A pin can be a symbol pin or a package pin. A package pin is a physical connector on an integrated circuit package that carries signals into and out of an integrated circuit. A symbol pin, also referred to as an instance pin, is the connection point of an instance to a net.

pinwires

Pinwires are wires which are directly tied to the pin of a site (CLB, IOB, etc.)

project

Foundation organizes related files into a distinct logical unit called a project, which contains a variety of file types. A project is created as either a Schematic Flow or an HDL Flow project.

Project Flowchart

The right-hand portion of the Foundation Project Manager that provides access to the synthesis and implementation tools, and the current design project. The project flowchart can display up to four tabs: Flow, Contents, Reports, and Synthesis (Schematic Flow only).

Project Manager

The Project Manager, the overall Foundation project management tool, contains the Foundation Series tools used in the design process.

PROM File Formatter

The PROM File Formatter is the program used to format one or more bitstreams into an MC86, TEKHEX, EXORmacs or HEX PROM file format.

route

The process of assigning logical nets to physical wire segments in the FPGA that interconnect logic cells.

route-through

A route that can pass through an occupied or an unoccupied CLB site is called a route-through. You can manually do a route-through in the FPGA Editor. Route-throughs provide you with routing resources that would otherwise be unavailable.

Schematic Editor

The schematic design tool accessed by selecting the Schematic Editor icon on the Design Entry button in the Project Manager.

Schematic Flow

A project that uses the Schematic Flow can have top-level schematic, ABEL, or state machine files. It can contain underlying schematic, HDL (VHDL, Verilog, or ABEL), state machine designs, or netlists.

state diagram

A state diagram is a pictorial description of the outputs and required inputs for each state transition as well as the sequencing between states. Each circle in a state diagram contains the name of a state. Arrows to and from the circles show the transitions between states and the input conditions that cause state transitions. These conditions are written next to each arrow.

state machine

A state machine is a set of combinatorial and sequential logic elements arranged to operate in a predefined sequence in response to specified inputs. The hardware implementation of a state machine design is a set of storage registers (flip-flops) and combinatorial logic, or gates. The storage registers store the current state, and the logic network performs the operations to determine the next state.

state machine designs

State machine designs typically start with the translation of a concept into a “paper design,” usually in the form of a state diagram or a bubble diagram. The paper design is converted to a state table and, finally, into the source code itself.

states

The values stored in the memory elements of a device (flip-flops, RAMs, CLB outputs, and IOBs) that represent the state of that device for a particular readback (time). To each state, there corresponds a specific set of logical values.

static timing analysis

A static timing analysis is a point-to-point delay analysis of a design network.

static timing analyzer

A static timing analyzer is a tool that analyzes the timing of the design on the basis of its paths.

status bar

The status bar is an area located at the bottom of a tool window that provides information about the commands that you are about to select or that are being processed.

stimulus information

Stimulus information is the information defined at the schematic level and representing a list of nodes and vectors to be simulated in functional and timing simulation.

Symbol Editor

With the Symbol Editor, you can edit features of component symbols such as pin locations, pin names, pin numbers, pin shape, and pin descriptions for component symbols.

Synopsys

Synopsys supports HDL, a behavioral language for entering equations. HDL also enables you to include LogiBLOX schematic components in a design.

synthesis

The HDL design process in which each design module is elaborated and the design hierarchy is created and linked to form a unique design implementation. Synthesis starts from a high level of logic abstraction (typically Verilog or VHDL) and automatically creates a lower level of logic abstraction using a library containing primitives

Time Tracker

See Express Time Tracker.

transitions

Transitions define the movement from one state to another in a state machine. They are drawn as arrows between state bubbles.

TRCE

TRCE (Timing Reporter and Circuit Evaluator) “trace” is a program that will automatically perform a static timing analysis on a design using the specified (either timing constraints. The input to TRCE is an NCD file and, optionally, a PCF file. The output from TRCE is an ASCII timing report which indicates how well the timing constraints for your design have been met.

TWR file

A TWR (Timing Wizard Report) file is an output from the TRCE program. A TWR file contains a logical description of the design expressed both in terms of the hierarchy used when the design was first created and in terms of lower-level Xilinx primitives to which the hierarchy resolves.

UCF file

A UCF (user constraints file) contains user-specified logical constraints.

verification

Verification is the process of reading back the configuration data of a device and comparing it to the original design to ensure that all of the design was correctly received by the device.

Verilog

Verilog is a commonly used Hardware Description Language (HDL) that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level. It is IEEE standard 1364-1995. Foundation Express and Base Express products include design entry tools to create Verilog designs. Recognizable as a file with a .v extension.

VHDL

VHDL is an acronym for VHSIC Hardware Description Language (VHSIC is an acronym for Very High-Speed Integrated Circuits). An

industry-standard (IEEE 1076.1) HDL. Recognizable as a file with a .vhd or .vhdl extension.

VHDL can be used to model a digital system at many levels of abstraction ranging form the algorithmic level to the gate level. It is IEEE standard 1076-1987. Foundation Express and Base Express products include design entry tools to create VHDL designs.

Wire

A wire is either a net or a signal.

Xilinx Constraints Editor

A GUI tool that you can use to enter design constraints. The Xilinx Constraints Editor is integrated with the Design Implementation tools and available in all product configurations.

Appendix B

Foundation Constraints

This appendix discusses some of the more common constraints you can apply to your design to control the timing and layout of a Xilinx FPGA or CPLD; it describes how to use constraints at each stage of design processing.

This appendix contains the following sections.

- "Constraint Entry Mechanisms"
- "Translating and Merging Logical Designs"
- "The Xilinx Constraints Editor"
- "Constraints File Overview"
- "Timing Constraints"
- "Layout Constraints"
- "Efficient Use of Timespecs and Layout Constraints"
- "Standard Block Delay Symbols"
- "Table of Supported Constraints"
- "Basic UCF Syntax Examples"
- "User Constraint File Example"
- "Constraining LogiBLOX RAM/ROM with Synopsys"

For a complete listing of all supported constraints, refer to the *Libraries Guide* (Chapter 12, "Attributes, Constraints, and Carry Logic"). For a more complete discussion of how timing constraints work in Foundation, refer to the *Development System Reference Guide* ("Using Timing Constraints"). For information on all attributes, including timing constraints, used in CPLD designs, refer to the Foundation online help.

Constraint Entry Mechanisms

With the Foundation version of the Xilinx design implementation tools, you control the implementation of a design by defining constraints that affect the mapping and layout of the physical circuit. Additionally, you can specify the “path” timing requirements of the circuit to obtain the best results and allow the implementation tools to choose the layout which best satisfies these requirements.

The various design constraints available within Foundation can be entered at the time you create the design (i.e., the logical domain) or after the design is mapped (that is, the physical domain).

Constraints entered in the logical domain are created in the following ways

- Entered into the schematic
- Applied to a synthesis process and then forward-annotated through a netlist constraints file (NCF)
- Created with the Constraints Editor (see “The Xilinx Constraints Editor” section.)

Constraints entered in the physical domain are entered directly into the Physical Constraints File (PCF). These constraints are conceptually the same as those entered during design creation; however, they are directly related to objects within the physical design database and are therefore applied using the PCF syntax.

The following figure illustrates the constraints entry approach for the Foundation version of the Xilinx tools.

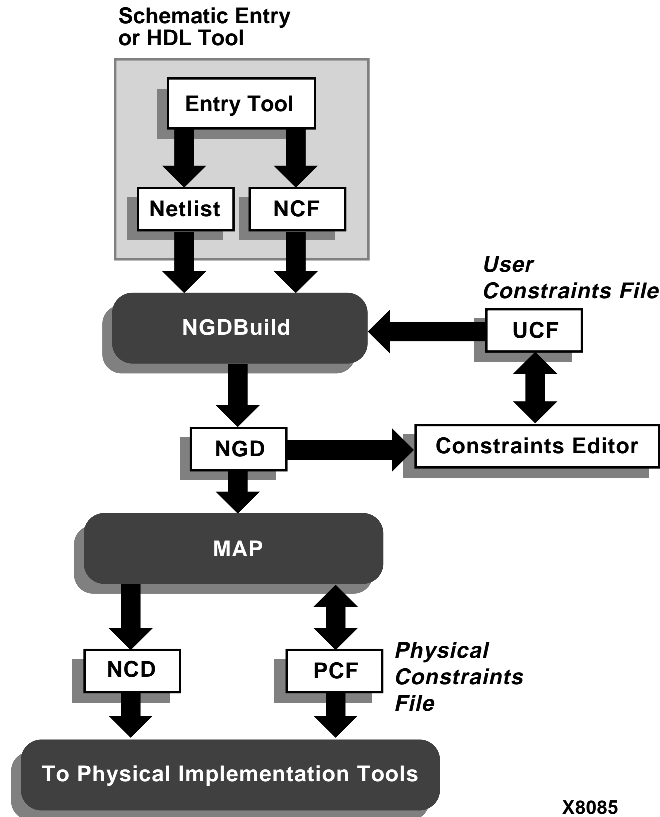


Figure B-1 Constraint Entry Flow

Translating and Merging Logical Designs

The process of implementing a design within the Foundation tools starts with a logical design file (NGD) that represents the design created by the NGDBuild application (as shown in the “Constraint Entry Flow” figure).

The NGD file contains all of the design’s logic structures (gates) and constraints. The NGD file is produced through the NGDBuild process which controls the translation and merging of all of the related logic design files.

All design files are translated from industry standard netlists into intermediate NGO files by one of two netlist translation programs XNF2NGD or EDIF2NGD. The exception to this rule is logic, which is created through the use of LogiBLOX components. LogiBLOX components may be compiled directly in memory, and are, therefore never written to disk as a separate intermediate NGO file.

The Xilinx Constraints Editor

The Xilinx Constraints Editor is a Graphical User Interface (GUI) that provides a convenient way for you to create certain new constraints. Constraints created with the Constraints Editor are written to the UCF (User Constraints File). See the “User Constraints File (UCF)” section.

For more information on the Constraints Editor, see the *Constraints Editor Guide,* an online book.

Constraints File Overview

The following subsections describe the Netlist Constraints File, User Constraints File, and the Physical Constraints File.

Netlist Constraints File (NCF)

The Netlist Constraints File (NCF) is an ASCII file generated by the synthesis program. It contains the logical contraints entered in the design.

User Constraints File (UCF)

The User Constraint File was developed to provide a convenient mechanism for constraining a logical design without returning to the design entry tools. UCF constraints intentionally overwrite constraints that are present in the netlist.

UCF constraints override any constraints contained within the netlist created by the schematic or synthesis tools. A constraint that is being applied via the UCF file must specify the complete hierarchical path name for the instance or net being constrained.

In the Foundation, UCF constraints are considered more significant because they appear later in the design flow and provide a mecha-

nism for establishing or modifying logical design constraints without requiring you to re-enter a schematic or synthesis tool.

The process of building the complete logical design representation (NGD files) is the job of NGDBuild. In developing this complete design database, NGDBuild annotates design constraints with those it finds in a UCF file. If a UCF file exists with the same name as the top-level netlist then it will automatically be read. Otherwise, you must indicate a specific file for User Constraints in the Options dialog box. The syntax for the UCF constraints file is explained (on a per-constraint basis) in the “Timing Constraints” section.

Note: Versions prior to M1.2 required the -uc switch to identify a User Constraint File that needed to be annotated to the design. Versions M1.2 and later allow UCF file annotation to be performed by default—if the UCF file has the same base name as the input.

Physical Constraints File (PCF)

(FPGA only) The layout tools work on the physical design, so the PCF file is written in terms that these tools can readily interpret. Layout and timing constraints are written in terms of the physical design’s components (COMPs), fractions of COMPs (BELs), and collections of COMPs (macros).

Because of this different design viewpoint, the PCF syntax is not necessarily the same as that used in the logical design constraint files (UCF/NCF). Furthermore, because the PCF file is written for the physical design implementation tools, its syntax may not be as intuitive as the UCF file. Regardless of the syntactical challenges associated with using a PCF file, many designers will choose to work at the physical level of design abstraction for the following reasons.

- It is readily modified and immediately applicable to the present task —implementing an FPGA (that is, there is no need to re-run NGDBuild or MAP in order to run layout or analysis tools).
- The implications of logical design structures on the physical design’s implementation only become obvious once the design is evaluated using the physical tools. Altering the PCF file for “what-if” analysis can be desirable.
- Certain constraints are only available within the PCF file.

Note: If you modify the PCF file, you should be certain that you enter your constraints after the line “SCHEMATIC END ;”. Otherwise, your constraints will be overwritten every time MAP is re-executed.

Case Sensitivity

Since EDIF is a case-sensitive format, the Foundation constraints are case sensitive as well. Always specify the net names and instance names exactly as they are in your schematic or code. Be consistent when using TNMs and other user-defined names in your constraints file; always use the same case throughout. For site names (such as “CLB_R2C8” or “P2”), you should use only upper case letters, since site names within Xilinx devices are all upper case.

Timing Constraints

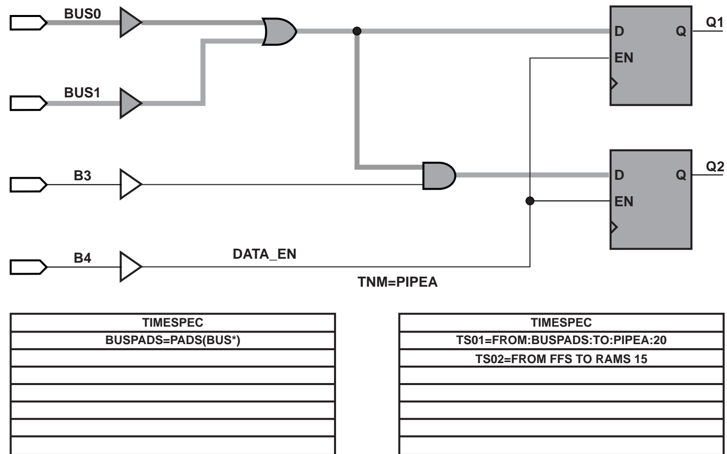
The following subsections discuss timing constraints. Many timing constraints can be created using the Constraints Editor. If a constraint can be created with the Constraints Editor, it will be noted in the sections that follow.

The “From:To” Style Timespec

When using the From:To style of constraint, the path(s) that are constrained are specified by declaring the start point and end point, which must be a pad, flip-flop, latch, RAM, or user-specified sync point (see TPSYNC). To group a set of endpoints together, you may attach a TNM attribute to the object (or to a net that is an input to the object). With a macro, the TNM traverses the hierarchy to tag all relevant objects. A TIMEGRP is a method for combining two or more sets of TNMs or other TIMEGRPs together, or alternatively, to create a new group by pattern matching (grouping a set of objects that all have output nets that begin with a given string)

You can create a From:To timespec with the Constraints Editor.

You use TNMs to identify a group of design objects which are to be referenced within a Timespec. If a TNM is placed on a net, the Foundation tools determine TNM membership by tracing forward from the specified net to all the valid endpoints of the net. Refer to the *Development System Reference Guide* (“Using Timing Constraints”) for more information on this subject. The following schematic shows an example of TNM, TIMESPEC, and TIMEGRP statements.



The following file corresponds to the preceding figure.

```
# This is a comment line
# UCF FROM:TO style Timespecs

  NET DATA_EN TNM = PIPEA ;
  TIMEGRP BUSPADS = PADS(BUS*) ;
  TIMESPEC TS01 = FROM:BUSPADS:TO:PIPEA:20 ;

# Spaces or colons (:) may be used as field separators

  TIMESPEC TS02 = FROM FFS TO RAMS 15 ;
```

The first line of the above example illustrates the application of the TNM (Timing Name) PIPEA to the net named DATA_EN. The second line illustrates the TIMEGRP design object formed using a pattern matching mechanism in conjunction with the predefined TIMEGRP “PADS”. In this example, the TIMEGRP named BUSPADS will include only those PADs with names that start with *BUS*.

Each of the user-defined Timegroups is then used to define the object space constrained by the timing specification (Timespec) named TS01. This timing specification states that all paths from each member of the BUSPADS group to each member of the PIPEA group need to have a path delay that does not exceed 20 nanoseconds (ns are the default units for time). The TIMESPEC TS02 constraint illustrates a similar type of timing constraint using the predefined groups FFS and RAMS.

Note: All From:To Timespecs must be relative to a Timegroup. The above example illustrates that you can define Timegroups either explicitly (TIMEGRPs) or implicitly (TNMs), or they may be predefined groups (PADS, LATCHES, FFS, RAMS).

There is an additional keyword that you can add to the From:To specification that allows the user to narrow the set of paths that are covered—THRU. By using the From:Thru:To form of a Timespec, you are able to constrain only those paths that go through a certain set of nets, defined by the TPTHRU keyword, as shown in the following example.

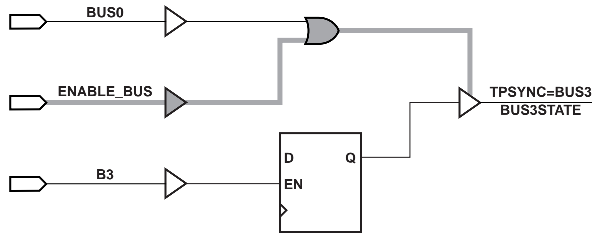
```
# UCF example of FROM:TO Timespec using THRU

  NET $1I6/thisnet TPTHRU=these ;
  NET $1I6/thatnet TPTHRU=these ;

  TIMEGRP sflops=FFS(DATA*) ;
  TIMEGRP dflops=FFS(OUTREG*) ;

  TIMESPEC TS23=FROM:sflops:THRU:these:TO:dflops:20 ;
```

Here, only those paths that go from the Q pin of the *sflops* through the nets `$1I6/thisnet` and `$1I6/thatnet` and on to the D pin of *dflops* will be controlled by TS23.

Using TPSYNC

(FPGA only.) In the Foundation design implementation tools, you can define any node as a source or destination for a Timespec with the TPSYNC keyword. The use of TPSYNC is similar to TPTHRU—it is a label that is attached to a set of nets, pins, or instances in the design.

For example, suppose a design has a PAD ENABLE_BUS that must arrive at the enable pin of several different 3-state buffers in less than a specified time. With the Foundation tools, you can now define that 3-state buffer as an endpoint for a timing spec. The following figure illustrates TPSYNC.



TIMESPEC
TSNewSpa3=FROM:PAD(ENABLE_BUS):TO:bus3:20ns

X8569

The following UCF file corresponds to the above example.

```
# TPSYNC example; pad to a 3-state buffer enable pin
# Note TPSYNC attached to 3-state buffer's output NET

NET BUS3STATE TPSYNC=bus3;
TIMESPEC TSNewSpc3=FROM:PAD(ENABLE_BUS):TO:bus3:20ns;
```

In the NET statement shown above, the TPSYNC is attached to the output net of a 3-state buffer called *BUS3STATE.* If a TPSYNC is attached to a net, then the source of the net is considered to be the endpoint (in this case, the 3-state buffer itself). The subsequent TIMESPEC statement can use the TPSYNC name just as it uses a TNM name.

The next TPSYNC UCF file example shows you how to use the keyword PIN instead of NET if you want to attach an attribute to a pin.

```
# Note TPSYNC attached to 3-state buffer's enable PIN

PIN $1I6/BUSMACRO1/TRIBUF34.T TPSYNC=bus1;
TIMESPEC TSNewSpc1=FROM:PAD(ENABLE_BUS):TO:bus1:20ns;
```

In this example, the instance name of the 3-state buffer is stated followed by the pin name of the enable (.T). If a TPSYNC is attached to a primitive input pin, then the primitive's input is considered the startpoint or endpoint for a timing specification. If it is attached to a output pin, then the output of the primitive is used.

The last TPSYNC example shows you how to use the keyword INST if you want to attach an attribute to a instance:

```
# Note TPSYNC attached to 3-state buffer INSTANCE (UCF
# file)

INST $1I6/BUSMACRO2/BUFFER_2 TPSYNC=bus2;
TIMESPEC TSNewSpc2=FROM:PAD(ENABLE_BUS):TO:bus2:20ns;
```

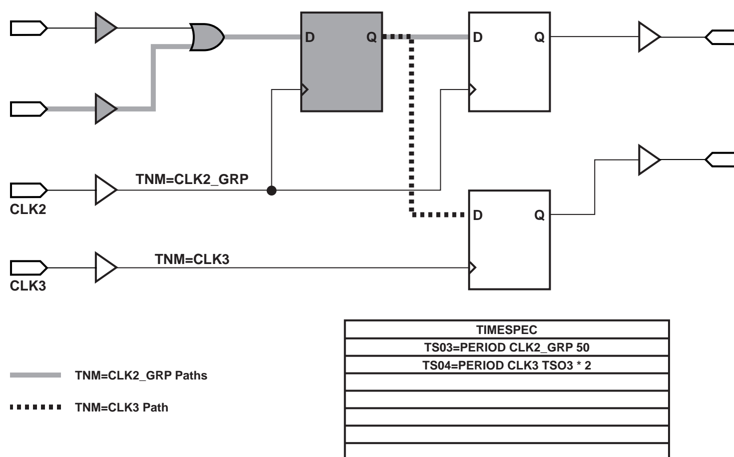
If a TPSYNC is attached to an instance, then the output of the instance is considered the startpoint or endpoint for a timing specification.

The Period Style Timespec

The TIMESPEC form of the PERIOD constraint allows flexibility in group definitions and allows you to define clock timing relative to another TIMESPEC.

You can create a Period constraint with the Constraints Editor.

The following schematic example illustrates the use of the PERIOD Timespec referenced to timegroups *CLK2_GRP* and *CLK3*.



The following syntax is the corresponding UCF file.

```
# UCF PERIOD style Timespecs

  NET CLK2 TNM = CLK2_GRP ;
  NET CLK3 TNM = CLK3 ;
```

```
TIMESPEC TS03 = PERIOD CLK2_GRP 50 ;
TIMESPEC TS04 = PERIOD CLK3 TS03 * 2 ;
```

Furthermore, the example shows how constraints and nets may be given the same name because they occupy separate name-spaces. Also, it shows the constraint syntax whereby one Timespec is defined relative to another (the value of TS04 is declared to be two times that of TS03).

The PERIOD constraint covers all timing paths which start or end at a register, latch, or synchronous RAM that is clocked by the referenced net. The only exception to this rule are paths to output pads, which are not covered by the PERIOD constraint. (Input pads, which are the source of a "pad-to-setup" timing path for one of the specified synchronous elements, *are* covered by the PERIOD constraint.)

The flexibility of the TIMESPEC form of the PERIOD constraint arises from being able to modify the contents of the TIMEGRP once the design has been mapped. By adding or removing objects from the TIMEGRP, which are listed in the PCF file, you can alter the paths that are covered by the PERIOD constraint.

If you do not need the flexibility offered by the TIMESPEC form, you can use the NET form of the PERIOD constraint may be used. The syntax for the NET form of the PERIOD constraint is simpler than the TIMESPEC form, while continuing to provide the same path coverage. The following example illustrates the syntax of the NET form of the PERIOD constraint.

```
# NET form of the PERIOD timing constraint
# (no TSidentifier)

  NET CLK PERIOD = 40 ;
```

This is the recommendation of using PERIOD on a single clock design in which data does not pass between the clock domains.

With the Foundation 1.5 release, PERIOD will now include clock skew in the path analysis.

The Offset Constraint

Use offsets to define the timing relationship between an external clock and its associated data-in or data-out-pin. Using this option, you can do the following.

- Calculate whether a setup time is being violated at a flip-flop whose data and clock inputs are derived from external nets.
- Specify the delay of an external output net derived from the Q output of an internal flip-flop being clocked from an external device pin.

You can create a Pad to Setup or Clock to Pad offset constraint with the Constraints Editor.

There are basically three types of offset specifications.

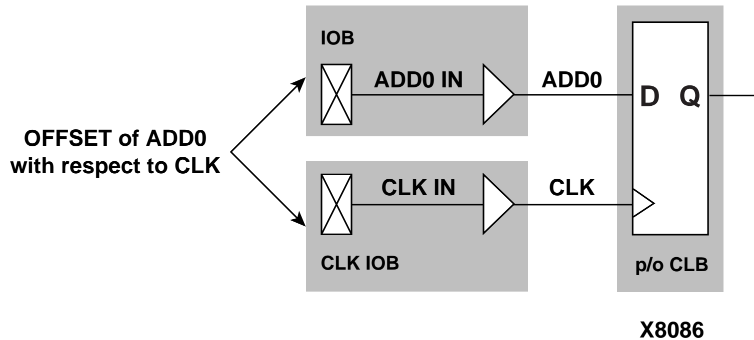
- Global
- Specific
- Group

Since the global and group OFFSET constraints are not associated with a single data net or component, these two types can also be entered on a TIMESPEC symbol in the design netlist with TS*id.* See the "Using Timing Constraints" in the *Development System Reference Guide* for details.

In the following example, the OFFSET constraint is applied to a net that connects with a PAD (as shown in the figure later in this section). It defines the delay of a signal relative to a clock and is only valid for registered data paths. The OFFSET constraint specifies the signal delay external to the chip, allowing the implementation tools to automatically adjust relevant internal delays (CLK buffer and distribution delays) to accommodate the external delay specified with the following.

```
# Net form of the OFFSET timing constraint

  NET ADD0_IN OFFSET = IN 14 AFTER CLK ;
```

In analyzing OFFSET paths, the Xilinx timing tools adjust the PERIOD associated with the constrained synchronous element based on both the timing specified in the OFFSET constraint and the delay of the referenced clock signal. In the following figure, assume a delay of 8 ns for the signal CLK to arrive at the CLB, a 5 ns setup time for ADD0, and a 14 ns OFFSET delay for the signal ADD0. Assume a period of 40 ns is specified. The Foundation tools allocate 29 ns for the signal ADD0 to arrive at the CLB input pin (40 ns - 14 ns + 8 ns - 5 ns = 29 ns).



This same timing constraint could be applied using the FROM:PADS:TO:FFS timing constraint. However, using a From:To methodology would require you to know the intrinsic CLK net delay, and you would have to adjust the value assigned to the From:To Timespec. The internal CLK net delay is implicit in the OFFSET/ PERIOD constraint. Furthermore, migrating the design to another speed grade or device would require modification of the From:To Timespec to accommodate the new intrinsic CLK net delay. An alternative solution is to use the flip-flop in the IOB of certain FPGA architectures (XC4000E/EX, for instance), as the clock-to-setup time *is* specified in the *Programmable Logic Data Book*.

Note: Relative Timespecs can only be applied to similar Timespecs. For example, a PERIOD Timespec may be defined in terms of another PERIOD Timespec, but not a FROM:TO Timespec.

Ignoring Paths

(FPGA only.) When you declare a a Timespec that includes paths where the timing is not important, the tools may create a less optimal route since there is more competition for routing resources. This problem can be alleviated by using a TIG (timing ignore) attribute on the non-critical nets. TIG causes all paths that fan out from the net or pin where it is applied to be "ignored" during timing simulation.

You can create a Timing Ignore constraint with the Constraints Editor.

The following syntax indicates that `$1I456/slow_net` should not have the Timespec TS01 or TS04 applied to it.

```
#Timespec-specific TIG example (UCF file)
```

```
NET $1I456/slow_net TIG=TS01, TS04 ;
```

On the other hand, the following syntax indicates that the layout tools should ignore paths through the `$1I456/slow_net` net for *all* known Timespecs.

```
#Global TIG example (UCF file)

  NET $1I456/slow_net TIG
```

Controlling Skew

(FPGA only.) Skew is the difference between the minimum and maximum of the maximum load delays on a net. You can control the maximum allowable skew on a net by attaching the MAXSKEW attribute directly to the net.

```
#MAXSKEW example (UCF file)

  NET $1I345/net_a MAXSKEW=3 ;
```

The above example indicates that 3 ns is the maximum skew allowed on `$1I345/net_a`. For a detailed example of how MAXSKEW works, see the “Additional Timing Constraints” section in the *Development System Reference Guide*.

Constraint Precedence

A design may assign a precedence to Timespecs only within a certain class of constraints. For example, you may specify a priority for a particular From:To specification to be greater than another, but you may not specify a From:To constraint to have priority over a TIG constraint. The following example illustrates the explicit assignment of priorities between two same-class timing constraints, the lowest number having the highest priority.

```
# Priority UCF example

  TIMESPEC TS01 = FROM GROUPA TO GROUPB 40 PRIORITY 4;
  TIMESPEC TS02 = FROM GROUP1 TO GROUP2 35 PRIORITY 2;
```

The following sections illustrate the order of precedence for the various types (and various sources) of timing constraints.

Across Constraint Sources

Across constraint sources, the following priorities apply

- Physical Constraint File (PCF)—the highest priority
- User Constraint File (UCF)
- Input Netlist / Netlist Constraint File (NCF)—the lowest priority

Within Constraint Sources

Within constraint sources, the following priorities apply.

- TIG (Timing Ignore)—the highest priority
- FROM:*source*:THRU:*point*: TO:*destination* specification

  The priority of each type of FROM:THRU:TO specification is as follows (highest priority is listed first).

  FROM:USER1:THRU:USER_T:TO:USER2 specification
  (USER1 and USER2 are user-defined groups)

  FROM:USER1:THRU:USER_T:TO:FFS specification
  or
  FROM:FFS:THRU:USER_T:TO:USER2 specification
  (FFS is any pre-defined group)

  FROM:FFS:THRU:USER_T:TO:FFS specification
- FROM:*source*:TO:*destination* specification

  The priority of each type of FROM:TO specification is as follows (highest priority is listed first).

  FROM:USER1:TO:USER2 specification

  FROM:USER1:TO:FFS specification
  or
  FROM:FFS:TO:USER2 specification

  FROM:FFS:TO:FFS specification
- PERIOD specification

- "Allpaths" type constraints—the lowest priority

Layout constraints also have an inherent precedence which is based on the type of constraint and the site description provided to the tools. If two constraints have the same priority and cover the same path, then the last constraint in the constraint file will override any other constraints that overlap.

Layout Constraints

(FPGA only.) The mapping constraints in the example below illustrate some of the capabilities to control the implementation process for a design. The OPTIMIZE attribute is attached to the block of logic associated with the instance "GLUE."All of the combinatorial logic within the block GLUE will be optimized for speed (minimizing levels of logic) while other aspects of the design will be processed by the default mapping algorithms (assuming the design-based optimization switches are not issued).

```
# Mapping constraint

INST GLUE OPTIMIZE = SPEED ;

# Layout constraint

NET IOBLOCK/DATA0_IN LOC = P12 ;
```

The layout constraint in the example above illustrates the use of a full hierarchical path name for the net named DATA0_IN in the application of the I/O location constraint. In this example, IOBLOCK is a hierarchical boundary that contains the net DATA0_IN. Location constraints applied to "pad nets" are used to constrain the location of the PAD itself, in this case to site P12.

Note: If the design contains a PAD, the constraint could have been just as easily applied to it directly (some design flows do not provide explicit I/O pads in the design netlist).

Converting a Logical Design to a Physical Design

The process of mapping translates a design from the logical design domain to the physical design domain. The MAP process creates both the physical design components (CLBs, IOBs, and so forth) and the physical design constraints (layout and timing). The physical design components are written into a Native Circuit Description (NCD) file.

The physical design constraints are written into a Physical Constraints File (PCF).

As the design flow of the “Constraint Entry Flow” figure shows, MAP not only writes a PCF file, but also reads a specified pre-existing PCF file. MAP reads an existing PCF file in order to facilitate the overriding of constraints that are contained within another logic design using the “last one wins” resolution mechanism provided by the PCF file. The following subsection briefly describes this approach.

“Last One Wins” Resolution

MAP creates new physical design constraints each time it converts a logical design into a physical design. The constraints that are created during this process are written into the “Schematic” section of the PCF file. This section is recreated each time MAP is run based on the constraints that are contained within the NGD file. The schematic section is always written at the top of the PCF file, and constraints that are in the PCF file but outside of the Schematic section (after the line “SCHEMATIC END”) are considered to be in the “User” section of the PCF file. The user section is read, syntactically checked, and rewritten each time MAP is run. Since these constraints always follow those written into the schematic section, they will always take precedence (following the “last-one-wins” rule).

Note: If the design contains a PAD, the constraint could have been just as easily applied to it directly (some design flows do not provide explicit I/O pads in the design netlist).

XC5200XL Constraints

There are some special considerations for constraints for the XC5200XL family.

- The XC5200XL family requires that some constraints (such as LOC and RLOC) specify a logic cell name within the CLB. For instance, the following constraint will LOC the instance *ISYM52* to the lowest logic cell of the CLB in row 5 and column 2.

  ```
  INST ISYM52 LOC = CLB_R5C2.LC0 ;
  ```

  If this was an XC4000 design, an extension would be optional (the XC4000 family does not use the LCx notation; it uses FFX and FFY to specify which flip-flop and F and G to specify which function generator).

- The XC5200XL family does not have flip-flops in the IOB, so two new constraints have been provided: INREG and OUTREG. PAR will attempt to place a register with a INREG attribute near the IOB that drives its Din pin, so it can use fast routes. OUTREG will cause PAR to attempt to place a register near the IOB that Qout sources, as shown in the following example.

```
INST near_input_flop INREG ;
INST near_output_flop OUTREG ;
```

Efficient Use of Timespecs and Layout Constraints

The previous section described the mechanisms available for constraining a design's timing within the Foundation tools. The sections that follow summarize each of the constraints that are available.

The robust nature of the language enables you to define your design requirements at the highest level of abstraction first, and then fine tune the timing requirements by using more specific Timespecs, if needed. This is the methodology that will best describe your requirements to the tools.

The following observations help to illustrate the reasons why this methodology should be followed (from a tool runtime perspective).

- Using explicit Timegroups causes slower runtimes than using implicit timegroups arising from the use of constraints such as PERIOD.
- Processing larger Timegroups takes longer than processing smaller Timegroups.
- Using many specific Timespecs results in slower runtimes than using a smaller set of more general Timespecs.

In conclusion, overall design runtime is improved when a "qualified global" timing methodology is employed instead of a "thorough-detailed" timing methodology.

The "Starter Set" of Timing Constraints

The following examples clearly identify the "preferred" mechanism for controlling the timing of your design. The preferred method assumes a goal of getting the required results in the fastest run time possible. If the design has a single clock and required I/O timing that

equals the clock period, all that you need are the three constraints shown in the following example.

```
# Global UCF example

  NET CLK1 PERIOD = 40 ;
  NET OUT* OFFSET = OUT 13 AFTER CLK ;

  TIMESPEC TS01 = FROM PADS TO PADS 40 ;
```

Note: When you use net name wild cards in OFFSETS, make sure that the name is unique to valid nets; otherwise processing errors will occur.

If you need to account for extra delay external to the FPGA, then you could add the following.

```
  NET INPUT* OFFSET = IN 8 BEFORE CLK ;
```

The PERIOD constraint covers all pad-to-setup and clock-to-setup timing paths. The OFFSET constraint covers the clock-to-pad timing for each of the output nets beginning with *OUT*. Both the OFFSET and PERIOD constraints account for the delay of the Clock Buffer/ Net in the I/O timing calculations.

The following PCF fragment illustrates the differences in syntax between the UCF and PCF languages. In addition to the syntactical changes, remember that net and instance names may change. As an example, one of the net matches resulting from the UCF "NET *OUT**" constraint is now applied to "COMP *OUT1_PAD*". The name OUT1_PAD is the name assigned to the pad instance. In addition to name changes, another difference is the verbosity of the PCF. In the PCF there is additional syntax for "MAXDELAY," "TIMEGRP," and "PRIORITY." These are all optional qualifications of the Timespec within the UCF, but written explicitly to the PCF file illustrating the full flexibility of the language.

```
# Global PCF example

SCHEMATIC START;

. . .

NET PERIOD "CLK_IN" = 40 nS HIGH 50.00% ;
COMP "OUT1_PAD" OFFSET = OUT 40 ns AFTER COMP "CLK";
COMP "OUT2_PAD" OFFSET = OUT 40 ns AFTER COMP
"CLK";COMP "INPUT1_PAD" OFFSET = IN 28 ns BEFORE COMP
"CLK";
```

```
TS01 = MAXDELAY FROM TIMEGRP “PADS” TO TIMEGRP “PADS”
        40000 pS PRIORITY 0;

SCHEMATIC END;
```

The next UCF example illustrates the use of both global constraints (PERIOD, OFFSET) to generally constrain the design and detailed Timespecs (FROM:THRU:TO) to provide fast and slow exceptions to the general timing requirements. Because the amount of constraints placed on a design directly impact runtime, Xilinx recommends that you first apply global constraints, then apply individual constraints only to those elements of the design that require additional constraints (or an exception to a constraint). The more global the constraints, the better the runtime performance of the tools.

```
# Sample UCF file
# Specify target device and package

CONFIG PART = XC4010e-PQ208-3 ;

# Global constraints

NET CLK1 PERIOD = 40 ;
NET DATA_OUT* OFFSET = OUT 15 AFTER DCLK ;
TIMESPEC TS01 = FROM PADS TO PADS  40 ;

# Layout constraints

NET SCLINF LOC = P125 ;

# Detailed constraints
# Exception to cover X_DAT and Y_DAT buses

# Ignore timing on reset net

NET RESET_N TIG ;

# Slow exception for data leaving INA FFs

TIMESPEC TS02 = FROM FFS(INA*) TO FFS  80 ;

# Faster timing required for data leaving RAM

TIMESPEC TS03 = FROM RAMS TO FFS 20 ;

# Form special timegroups related to RAMs

INST $1I64 TNM = SPDRAM ;
NET RAMBUS0 TPTHRU = RAMVIA ;
NET RAMBUS1 TPTHRU = RAMVIA ;

# Specify timing for this special timing path
```

```
TIMESPEC TS04 = FROM SPDRAM THRU RAMVIA TO FFS 45 ;
```

Standard Block Delay Symbols

The “Timing Symbols and Their Default Values” table lists the block delay symbols, each with their corresponding description. There is a one-to-many correspondence between these symbol names and the *Programmable Logic Data Book* symbol names. For those symbols listed with a disabled default, no timing analysis is performed on paths that have a segment composed of symbol path. For example, paths which have a set/reset to output path will not be analyzed. Any of the block delays (Symbol) listed in the table may be explicitly enabled or disabled using the PCF file.

The following example shows the PCF syntax that enables the path tracing for all paths that contain RAM data to out paths. This PCF directive is placed in the user section of the PCF.

```
SCHEMATIC END;

// This is a PCF comment line
// Enable RAM data to out path tracing

  ENABLE = ram_d_o;
```

Table B-1 Timing Symbols and Their Default Values

Symbol	Default	Description
reg_sr_q	Disabled	Set/reset to output propagation delay
lat_d_q	Disabled	Data to output transparent latch delay
ram_d_o	Disabled	RAM data to output propagation delay
ram_we_o	Enabled	RAM write enable to output propagation delay
tbuf_t_o	Enabled	TBUF tristate to output propagation delay
tbuf_i_o	Enabled	TBUF input to output propagation delay
io_pad_I	Enabled	IO pad to input propagation delay
io_t_pad	Enabled	IO tristate to pad propagation delay
req_sr_clk	Disabled	Set/Reset to clock setup and hold checks

Table B-1 Timing Symbols and Their Default Values

Symbol	Default	Description
io_o_I	Enabled	IO output to input propagation delay (Disabled for tristated IOBs.)
io_o_pad	Enabled	IO output to pad propagation delay

Table of Supported Constraints

The following table summarizes all supported constraints; it also shows whether the constraint must be entered at the schematic level or whether it can be specified in one or more of the valid constraint file types (NCF, UCF, or PCF). For further explanation and examples of each of the constraints, see the online *Libraries Guide* (Chapter 12, “Attributes, Constraints, and Carry Logic”).

Certain constraints can only be defined at the design level, whereas other constraints can be defined in the various configuration files. The following table lists the constraints and their applicability to the design, and the NCF, UCF, and PCF files.

The CE column indicates which constraints can be entered using the Xilinx Constraints Editor, a GUI tool in the Xilinx Development System. The Constraints Editor passes these constraints to the implementation tools through a UCF file.

A check mark (√) indicates that the constraint applies to the item for that column.

Table B-2 Constraint Applicability Table

Attribute/Constraint	Design	NCF	UCF	CE	PCF
BASE	√				
BLKNM	√	√	√		
BUFG	√	√	√		
CLKDV_DIVIDE	√	√	√		
COLLAPSE	√	√	√		
COMPGRP					√
CONFIG**	√				
DECODE	√	√	√		

Table B-2 Constraint Applicability Table

Attribute/Constraint	Design	NCF	UCF	CE	PCF
DIVIDE1_BY	√	√			
DIVIDE2_BY	√	√			
DOUBLE	√				
DRIVE	√	√	√	√	
DROP_SPEC		√	√		√*
DUTY_CYCLE_CORRECTION	√	√	√		
EQUATE_F	√				
EQUATE_G	√				
FAST	√	√	√	√	
FILE	√				
FREQUENCY					√
HBLKNM	√	√	√		
HU_SET	√	√	√		
INIT	√	√	√***		
INIT_0x	√	√	√		
INREG	√	√	√		√
IOB	√	√	√		
IOSTANDARD	√	√	√	√	
KEEP	√	√	√		
KEEPER	√	√	√	√	
LOC	√	√	√	√	√*
LOCATE					√
LOCK					√
MAP	√	√	√		
MAXDELAY	√	√	√		√*
MAXSKEW	√	√	√		√*
MEDDELAY	√	√	√		
NODELAY	√	√	√		
NOREDUCE	√	√	√		

Table B-2 Constraint Applicability Table

Attribute/Constraint	Design	NCF	UCF	CE	PCF
OFFSET		√	√	√	√*
ONESHOT	√				
OPT_EFFORT	√	√	√		
OPTIMIZE	√	√	√		
OUTREG	√	√	√		√
PATH					√
PART	√	√	√		
PENALIZE TILDE					√
PERIOD	√	√	√	√	√*
PIN					√
PRIORITIZE					√
PROHIBIT	√	√	√	√	√*
PULLDOWN	√	√	√	√	
PULLUP	√	√	√	√	
PWR_MODE	√	√	√		
REG	√	√	√		
RLOC	√	√	√		
RLOC_ORIGIN	√	√	√		√
RLOC_RANGE	√	√	√		√
S(ave) - Net Flag attribute	√	√	√		
SITEGRP					√
SLOW	√	√	√	√	
STARTUP_WAIT	√	√	√		
TEMPERATURE	√	√	√	√	√
TIG	√	√	√	√	√*
Time group attributes	√	√	√	√	√
TNM	√	√	√	√	
TNM_NET	√	√	√	√	
TPSYNC	√	√	√		

Table B-2 Constraint Applicability Table

Attribute/Constraint	Design	NCF	UCF	CE	PCF
TPTHRU	√	√	√	√	
TSidentifier	√	√	√	√	√*
U_SET	√	√	√		
USE_RLOC	√	√	√		
VOLTAGE	√	√	√	√	√
WIREAND	√	√	√		
XBLKNM	√	√	√		

*Use cautiously — although the constraint is available, there are differences between the UCF/NCF and PCF syntax.

**The CONFIG attribute configures internal options of an XC3000 CLB or IOB. Do not confuse this attribute with the CONFIG primitive, which is a table containing PROHIBIT and PART attributes.

***INIT is allowed in the UCF for CPLDs only.

Basic UCF Syntax Examples

The following sections summarize the functions of timespecs.

PERIOD Timespec

The PERIOD spec covers all timing paths that start or end at a register, latch, or synchronous RAM which are clocked by the reference net (excluding pad destinations). Also covered is the setup requirement of the synchronous element relative to other elements (for example, flip flops, pads, and so forth).

Note: The default unit for time is nanoseconds.

```
NET clk20MHz PERIOD = 50 ;
NET clk50mhz TNM = clk50mhz ;
TIMESPEC TS01 = PERIOD : clk50mhz : 20 ;
```

FROM:TO Timespecs

FROM:TO style timespecs can be used to constrain paths between time groups.

Note: Keywords: RAMS, FFS, PADS, and LATCHES are predefined time groups used to specify all elements of each type in a design.

TIMESPEC TS02 = FROM : PADS : TO : FFS : 36 ;
TIMESPEC TS03 = FROM : FFS : TO : PADS : 36 ns ;
TIMESPEC TS04 = FROM : PADS : TO : PADS : 66 ;
TIMESPEC TS05 = FROM : PADS : TO : RAMS : 36 ;
TIMESPEC TS06 = FROM : RAMS : TO : PADS : 35.5 ;

OFFSET Timespec

To automatically include clock buffer/routing delay in your "PADS:TO: *synchronous element* or *synchronous element* :TO:PADS timing specifications, use OFFSET constraints instead of FROM:TO constraints.

- For an input where the maximum clock-to-out (Tco) of the driving device is 10 ns.

```
NET in_net_name OFFSET=IN:10:AFTER:clk_net ;
```

- For an output where the minimum setup time (Tsu) of the device being driven is 5 ns.

```
NET out_net_name OFFSET=OUT:5:BEFORE:clk_net ;
```

Timing Ignore

If you can ignore timing of paths, use Timing Ignore (TIG).

Note: The "*" character is a wild-card which can be used for bus names. A "?" character can be used to wild-card one character.

- Ignore timing of net *reset_n*:

NET : reset_n : TIG ;

- Ignore *data_reg(7:0)* net in instance *mux_mem*:

NET : mux_mem/data_reg* : TIG ;

- Ignore *data_reg(7:0)* net in instance *mux_mem* as related to a TIMESPEC named TS01 only:

NET : mux_mem/data_reg* : TIG = TS01 ;

- Ignore *data1_sig* and *data2_sig* nets:

NET : data?_sig : TIG ;

Path Exceptions

If your design has outputs that can be slower than others, you can create specific timespecs similar to this example for output nets named *out_data(7:0)* and *irq_n*.

TIMEGRP slow_outs = PADS(out_data* : irq_n) ;
TIMEGRP fast_outs = PADS : EXCEPT : slow_outs ;
TIMESPEC TS08 = FROM : FFS : TO : fast_outs : 22 ;
TIMESPEC TS09 = FROM : FFS : TO : slow_outs : 75 ;

If you have multi-cycle FF to FF paths, you can create a time group using either the TIMEGRP or TNM statements.

Warning: Many VHDL/verilog synthesizers do not predictably name flip flop Q output nets. Most synthesizers do assign predictable instance names to flip flops, however.

- TIMEGRP example.

TIMEGRP slowffs = FFS(inst_path/ff_q_output_net1* : inst_path/ff_q_output_net2*);

- TNM attached to instance example.

INST inst_path/ff_instance_name1_reg* TNM = slowffs ;
INST inst_path/ff_instance_name2_reg* TNM = slowffs ;

- If a FF clock-enable is used on all flip flops of a multi-cycle path, you can attach TNM to the clock enable net.

Note: TNM attached to a net “forward traces” to any FF, LATCH, RAM, or PAD attached to the net.

NET ff_clock_enable_net TNM = slowffs ;

- Example of using “slowffs” timegroup, in a FROM:TO timespec, with either of the three timegroup methods previously shown.

TIMESPEC TS10 = FROM : slowffs : TO : FFS : 100 ;

Miscellaneous Examples

- Assign an IO pin number or place a basic element (BEL) in a specific CLB. BEL = FF, LUT, RAM, etc...

INST io_buf_instance_name LOC = P110 ;
NET io_net_name LOC = P111 ;
INST instance_path/BEL_inst_name LOC = CLB_R17C36 ;

- Prohibit IO pin C26 or CLB_R5C3 from being used.

CONFIG PROHIBIT = C26 ;
CONFIG PROHIBIT = CLB_R5C3 ;

- Assign an OBUF to be FAST or SLOW.

INST obuf_instance_name FAST ;
INST obuf_instance_name SLOW ;

- Constrain the skew or delay associate with a net.

NET any_net_name MAXSKEW = 7 ;
NET any_net_name MAXDELAY = 20 ns;

- Declare an IOB input FF delay (default = MAXDELAY).

Note: MEDDELAY/NODELAY can be attached to a CLB FF that is pushed into an IOB by the "map -pr i" option.

INST input_ff_instance_name MEDDELAY ;
INST input_ff_instance_name NODELAY ;

- Also, constraint priority in your .ucf file is as follows.

Highest

1. Timing Ignore (TIG)
2. FROM : THRU : TO specs
3. FROM : TO specs lowest
4. OFFSET
5. PERIOD specs

See the on-line documentation set for additional timespec features or additional information.

User Constraint File Example

The user constraint file (.ucf) is a user-created ASCII file that holds timing and location constraints. It is read by NGDBuild during the translate process and is combined with an EDIF or XNF netlist into an NGD file. If a UCF file exists with the same name as the top-level netlist, then it will automatically be read. Otherwise, specify a file for User Constraints in the Implement Control Files Settings dialog box.

For Foundation 2.1i, if you already have an existing UCF file associated with a Revision, this UCF file is automatically copied and used as your UCF file within a new revision.

The following example shows how to lock I/Os to pin locations and how to write Timespec and Timegroup constraints.

Note: You can also lock pin locations within the Project Manager by selecting **`Tools`** → **`Implementation`** → **`Lock Device Pins`**.

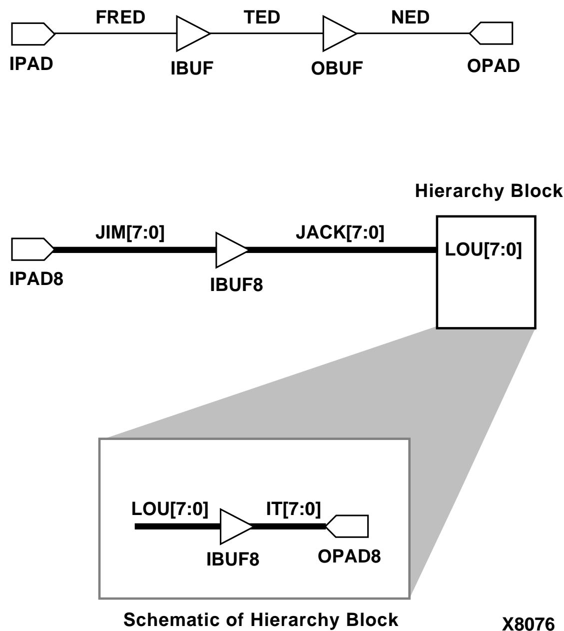
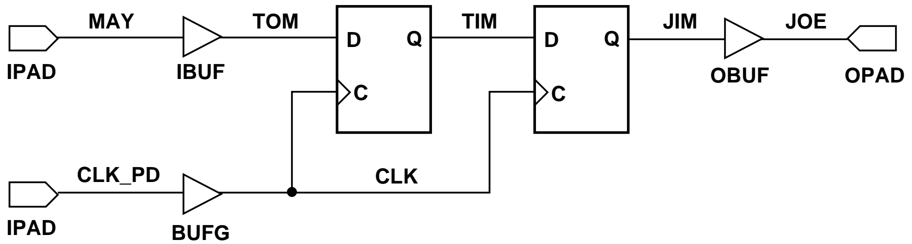


Figure B-2 Locking I/Os to Pin Locations

```
# This is a UCF comment

# The constraints below lock the I/O signals to pads.

# The net name that connects to the pad is used to
# constrain the I/O.

# The pin grid array packages use pin names like B3 or
# T1, instead of P<Pin Number>.


# Lock the input pins

NET FRED LOC = P18;
NET JIM<0> LOC = P20;
NET JIM<1> LOC = P23;
NET JIM<2> LOC = P24;
NET JIM<3> LOC = P25;
NET JIM<4> LOC = P26;
NET JIM<5> LOC = P27;
NET JIM<6> LOC = P28;
NET JIM<7> LOC = P38;

# Lock the output pins

NET NED LOC = P19;
NET HIERARCHY_BLOCK/<IT0> LOC = P44
NET HIERARCHY_BLOCK/<IT1> LOC = P45
NET HIERARCHY_BLOCK/<IT2> LOC = P46
NET HIERARCHY_BLOCK/<IT3> LOC = P47
NET HIERARCHY_BLOCK/<IT4> LOC = P48
NET HIERARCHY_BLOCK/<IT5> LOC = P49
NET HIERARCHY_BLOCK/<IT6> LOC = P50
NET HIERARCHY_BLOCK/<IT7> LOC = P462
```

For more information on constraint precedence, refer to the “Using Timing Constraints” chapter in the *Development System Reference Guide*.

This example shows how to specify timing constraints.



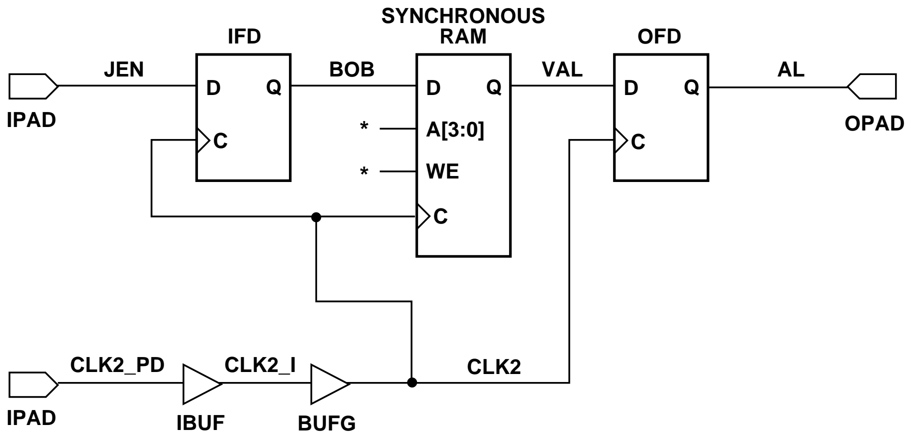


Figure B-3 Specifying Timing Constraints

```
---User Constraint File (UCF):

# This is a comment

# Period specifies minimum PERIOD of CLK net. Offset specifies that
# data on MAY can arrive up to 6 ns before the clock edge arrives on CLK.

# NOTE: Period constraints do not apply to elements in input or output
pads.

NET CLK PERIOD = 20 ns ;
NET MAY OFFSET = IN 6ns before CLK_PD ;

# Groups all clocked loads of CLK2 into CLK2_LOADS timegroup
# Groups all clocked loads of VAL into VAL_LOADS
# timegroup TNM # => Timegroup NaMe

NET CLK2 TNM=CLK2_LOADS ;
NET VAL TNM=VAL_LOAD ;

# Specifies worst case speed of path from IPAD to CLK2 # loads. Includes
# pad, buffer, and net delays. TS0l is a Timespec identifier; it can
# have names of the form TS<string>. PADS (CLK2_PD) is a Timegroup name
# specified inside of a Timespec.

TIMESPEC TS01=FROM PADS (CLK2_PD) TO CLK2_LOADS=15ns ;

# Specifies the maximum frequency for all loads clocked by CLK2.

TIMESPEC TS02=FROM CLK2_LOADS TO CLK2_LOADS=30Mhz;

# Specifies the minimum delay on the path from Synchronous RAM to OFD.
# Includes clock-to-out“User Constraint File Example” delay, net delay,
and setup time.

TIMESPEC TS03=FROM CLK2_LOADS TO VAL_LOAD=15000ps ;
```

Constraining LogiBLOX RAM/ROM with Synopsys

In the M1 XSI (Xilinx Synopsys Interface) HDL methodology, whenever large blocks of RAM/ROM are needed, LogiBLOX RAM/ROM modules are instantiated in the HDL code. With LogiBLOX RAM/ROM modules instantiated in the HDL code, timing and/or placement constraints on these RAM/ROM modules, and the RAM/ROM primitives that comprise these modules, can be specified in a UCF file. To create timing and/or placement constraints for RAM/ROM LogiBLOX modules, knowledge of how many primitives will be used and how the primitives, and/or how the RAM/ROM LogiBLOX modules are named is needed.

Estimating the Number of Primitives Used

When a RAM/ROM is specified with LogiBLOX, the RAM/ROM depth and width are specified. If the RAM/ROM depth is divisible by 32, then 32x1 primitives are used. If the RAM/ROM depth is not divisible by 32, then 16x1 primitives are used instead. In the case of dual-port RAMs, 16x1 primitives are always used. Based on whether 32x1 or 16x1 primitives are used, the number of RAM/ROM can be calculated.

For example, if a RAM48x4 was required for a design, RAM16x1 primitives would be used. Based on the width, there would be four banks of RAM16x1s. Based on the depth, each bank would have three RAM16x1s.

How the RAM Primitives are Named

Using the example of a RAM48x4, the RAM primitives inside the LogiBLOX are named as follows.

MEM0_0	MEM1_0	MEM2_0	MEM3_0
MEM0_1	MEM1_1	MEM2_1	MEM3_1
MEM0_2	MEM1_2	MEM2_2	MEM3_2

Each primitive in a LogiBLOX RAM/ROM module has an instance name of MEMx_y, where y represents the primitive position in the bank of memory and where x represents the bit position of the RAM/ROM output.

For the next two items, refer to the Verilog/VHDL examples included at the end of this section. The Verilog/VHDL example instantiates a RAM32x2S, which is in the bottom of the hierarchy. The RAM32x2S was implemented with LogiBLOX. The next two items are written within the context of the Verilog examples but also apply to the VHDL examples as well.

Referencing a LogiBLOX Module/Component in the HDL Flow

LogiBLOX RAM/ROM modules in the HDL Flow are constrained via a UCF file. LogiBLOX RAM/ROM modules instantiated in the HDL code can be referenced by the full-hierarchical instance name. If a LogiBLOX RAM/ROM module is at the top-level of the HDL code, then the instance name of the LogiBLOX RAM/ROM module is just the instantiated instance name.

In the case of a LogiBLOX RAM/ROM, which is instantiated within the hierarchy of the design, the instance name of the LogiBLOX RAM/ROM module is the concatenation of all instances which contain the LogiBLOX RAM/ROM. The concatenated instance names are separated by a "_¨. In the example, the RAM32X1S is named `memory`. The module `memory` is instantiated in Verilog module `inside` with an instance name U0. The module `inside` is instantiated in the top-level module test. Therefore, the RAM32X1S can be referenced in a .ucf file as U0/U0. For example, to attach a TNM to this block of RAM, the following line could be used in the UCF file.

```
INST U0_U0 TNM=block1 ;
```

Since U0/U0 is composed of two primitives, a Timegroup called block1 would be created; block1 TNM could be used throughout the .ucf file as a Timespec end/start point, and/or U0/U0 could have a LOC area constraint applied to it. If the RAM32X1S has been instantiated in the top-level file, and the instance name used in the instantiation was U0, then this block of RAM could just be referenced by U0.

Referencing the Primitives of a LogiBLOX Module in the HDL Flow

Sometimes it is necessary to apply constraints to the primitives that compose the LogiBLOX RAM/ROM module. For example, if you choose a floorplanning strategy to implement your design, it may be necessary to apply LOC constraints to one or more primitives inside a LogiBLOX RAM/ROM module.

Returning to the RAM32x2S example above, suppose that the each of the RAM primitives had to be constrained to a particular CLB location. Based on the rules for determining the MEMx_y instance names and using the example from above, each of the RAM primitives could be referenced by concatenating the full-hierarchical name to each of the MEMx_y names. The RAM32x2S created by LogiBLOX would have primitives named MEM0_0 and MEM1_0. So, for an HDL Flow project, CLB constraints in a UCF file for each of these two items would be.

```
INST U0_U0/MEM0_0 LOC=CLB_R10C10 ;
INST U0_U0/MEM0_1 LOC=CLB_R11C11 ;
```

HDL Flow Verilog Example

Following is a Verilog example.

test.v:

```
module test(DATA,DATAOUT,ADDR,C,ENB);

input [1:0] DATA;
output [1:0] DATAOUT;
input [4:0] ADDR;
input C;
input ENB;
wire [1:0] dataoutreg;
reg [1:0] datareg;
reg [1:0] DATAOUT;
reg [4:0] addrreg;

inside U0 (.MDATA(datareg),.MDATAOUT(dataoutreg),
            .MADDR(addrreg),.C(C),.WE(ENB));

always@(posedge C) datareg = DATA;
always@(posedge C) DATAOUT = dataoutreg;
always@(posedge C) addrreg = ADDR; endmodule
```

inside.v:

```
module inside(MDATA,MDATAOUT,MADDR,C,WE);

input [1:0] MDATA;
output [1:0] MDATAOUT;
input [4:0] MADDR;
input C;
input WE;

memory U0 ( .A(MADDR), .DO(MDATAOUT),
            .DI(MDATA), .WR_EN(WE), .WR_CLK(C));

endmodule
```

test.ucf

```
INST “U0_U0” TNM = usermem;
TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;
INST “U0_U0/mem0_0” LOC=CLB_R7C2;
```

HDL Flow VHDL Example

Following is a VHDL example.

test.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity test is
   port(
        DATA: in STD_LOGIC_VECTOR(1 downto 0);
        DATAOUT: out STD_LOGIC_VECTOR(1 downto 0);
        ADDR: in STD_LOGIC_VECTOR(4 downto 0);
        C, ENB: in STD_LOGIC);
   end test;

architecture details of test is
signal dataoutreg,datareg: STD_LOGIC_VECTOR(1 downto 0);
signal addrreg: STD_LOGIC_VECTOR(4 downto 0);

component inside
   port(
        MDATA: in STD_LOGIC_VECTOR(1 downto 0);
        MDATAOUT: out STD_LOGIC_VECTOR(1 downto 0);
        MADDR: in STD_LOGIC_VECTOR(4 downto 0);
```

```
        C,WE: in STD_LOGIC);
   end component;

begin
        U0: inside port
map(MDATA=>datareg.,MDATAOUT=>dataoutreg.,MADDR=>addrreg,C=>C,WE=>ENB);

        process( C )
             begin
                  if(Cevent and C=1) then
                       datareg <= DATA;
                  end if;
             end process;

        process( C )
             begin
                  if(Cevent and C=1) then
                       DATAOUT <= dataoutreg;
                  end if;
             end process;

        process( C )
             begin
                  if(Cevent and C=1) then
                       addrreg <= ADDR;
                  end if;
             end process;

end details;
```

inside.vhd

```
entity inside is
   port(
        MDATA: in STD_LOGIC_VECTOR(1 downto 0);
        MDATAOUT: out STD_LOGIC_VECTOR(1 downto 0);
        MADDR: in STD_LOGIC_VECTOR(4 downto 0);
        C,WE: in STD_LOGIC);
   end inside;

architecture details of inside is component memory
   port(
        A: in STD_LOGIC_VECTOR(4 downto 0);
        DO: out STD_LOGIC_VECTOR(1 downto 0);
        DI: in STD_LOGIC_VECTOR(1 downto 0);
        WR_EN,WR_CLK: in STD_LOGIC);
   end component;
```

```
begin
        U0: memory port map(A=>MADDR,DO=>MDATAOUT,
                DI=>MDATA,WR_EN=>WE,WR_CLK=>C);
end details;
```

test.ucf

```
INST “U0_U0” TNM = usermem;
TIMESPEC TS_6= FROM : FFS :TO: usermem: 50;
INST “U0_U0/mem0_0” LOC=CLB_R7C2;
```

Appendix C

Instantiated Components

This appendix lists the Xilinx Unified Library components most frequently instantiated in synthesis designs for FPGAs. This appendix contains the following sections:

- “Library/Architecture Definitions”
- “STARTUP Component”
- “BSCAN Component”
- “READBACK Component”
- “RAM and ROM”
- “Global Buffers”
- “Fast Output Primitives (XC4000X only)”
- “IOB Components”
- “Clock Delay Components”

The function of each component is briefly described and the pin names are supplied, along with a listing of the Xilinx product families involved. Associated instantiation can be used to include the component in an HDL design. For complete lists of the Xilinx components, see the online *Libraries Guide*.

Note: To check which components can be instantiated for a design for a given device, go to c:/fndtn/synth/lib/*device_name* (if Foundation is not installed at c:/fndtn, go to where you installed it). Compare the list of components shown in the *device_name* (xc4000e, virtex, for example) directory against the Libraries Guide. Items that match can be instantiated.

Library/Architecture Definitions

The following subsections describe which Xilinx architectural families are included in each library.

XC3000 Library

Information appearing under the title of XC3000 pertains to the XC3000A and XC3100A families. This includes the XC3000L and XC3100L, which are identical in architecture and features to the XC3000A and XC3100A, respectively, but operate at a nominal supply voltage of 3.3 V.

XC4000E Library

Wherever XC4000E is mentioned, it includes the XC4000E and XC4000L families. The XC4000L is identical in architecture and features to the XC4000E but operates at a nominal supply voltage of 3.3 V.

XC4000X Library

Information under the title XC4000X pertains to the XC4000EX, XC4000XL, XC4000XV, and XC4000XLA families. The XC4000XL is identical in architecture and features to the XC4000EX but operates at a nominal supply voltage of 3.3 V. The XC4000XV has identical library symbols to the XC4000EX and XC4000XL but operates at a nominal supply voltage of 2.5 V and includes additional features (the DRIVE attribute).

XC5200 Library

The title XC5200 pertains to the XC5200 family.

XC9000 Library

The title XC9000 pertains to the XC9500, XC9500XL, and XC9500XV CPLD families.

Spartan Library

The Spartan library pertains to the Spartan family XCS* devices.

SpartanXL Library

The SpatanXL library pertains to the SpartanXL family XCS*XL devices.

Virtex Library

The Virtex Library pertains to the Virtex family XCV* devices.

STARTUP Component

The STARTUP component is typically used to access the global set/reset and global 3-state signals. STARTUP can also be used to access the startup sequence clock.

For information on the startup sequence and the associated signals, see the *Programmable Logic Data Book* and the online *Libraries Guide.*

Table C-1 Design STARTUP Components

Name	Library	Description	Outputs	Inputs
STARTUP	XC4000E XC4000X XC5200* Spartan SpartanXL	Used to connect Global Set/Reset, global 3-state control, and user configuration clock.	Q2, Q3, Q1Q4, DONEIN	GSR, GTS, CLK
STARTUP_ VIRTEX	Virtex	Used to connect Global Set/Reset, global 3-state control, and user configuration clock.		GSR, GTS, CLK

* For 5200, GSR pin is GR

STARTBUF Component

The STARTBUF component allows you to functionally simulate the STARTUP component. As with STARTUP, a STARTBUF component instantiated in your design specifies to the implementation tools to

use GSR. Using the STARTBUF component in VHDL designs is the preferred method for using GSR/GR.

Table C-2 STARTBUF Library Component

Name	Library	Description	Outputs	Inputs
STARTBUF	XC4000E XC4000X XC5200* Spartan SpartanXL	Used to connect Global Set/Reset, global tristate control, and user configuration clock.	GSROUT, GTSOUT, Q2OUT, Q3OUT, Q1Q4OUT, DONEINOUT	GSRIN, GTSIN, CLKIN

BSCAN Component

To use the boundary-scan (BSCAN) circuitry in a Xilinx FPGA, the BSCAN component must be present in the input design. The TDI, TDO, TMS, and TCK components are typically used to access the reserved boundary scan device pads for use with the BSCAN component but can be connected to user logic as well. For more information on the BSCAN component, the internal boundary scan circuitry, and the directional properties of the four reserved boundary scan pads, refer to *Programmable Logic Data Book* and the online *Libraries Guide.*

Table C-3 Boundary Scan Components

Name	Library	Description	Outputs	Inputs
BSCAN	XC4000E XC4000X XC5200 Spartan SpartanXL	Indicates that the boundary scan logic should be enabled after the FPGA has been configured.	TDO, DRCK, IDLE, SEL1, SEL2	TDI, TMS, TCK, TDO1, TDO2
BSCAN_ VIRTEX	Virtex	Used to create internal boundary scan chains in a Virtex device.	TDO1, TDO2	,TDO1, TDO2
TDI	XC4000E XC4000X XC5200 Spartan SpartanXL	Connects to the BSCAN TDI input. Loads instructions and data on each low-to-high TCK transition.	I	—

Table C-3 Boundary Scan Components

Name	Library	Description	Outputs	Inputs
TDO	XC4000E XC4000X XC5200 Spartan SpartanXL	Connects to the BSCAN TDO output. Provides the boundary scan data on each low-to-high TCK transition.	—	O
TMS	XC4000E XC4000X XC5200 Spartan SpartanXL	Connects to the BSCAN TMS input. It determines which boundary scan is performed.	I	—
TCK	XC4000E XC4000X XC5200 Spartan SpartanXL	Connects to the BSCAN TCK input. Shifts the serial data and instructions into and out of the boundary scan data registers.	I	—

* The XC5200 has three additional pins: Reset, Update, Shift

READBACK Component

To use the dedicated readback logic in a Xilinx FPGA, the READBACK component must be inserted in the input design. The MD0, MD1, and MD2 components are typically used to access the mode pins for use with the readback logic but can be connected to user logic as well. For more information on the READBACK component, the internal readback logic, and the directional properties

of the three reserved mode pins, see the *Programmable Logic Data Book* and the online *Libraries Guide*.

Table C-4 Readback Components

Name	Library	Description	Outputs	Inputs
CAPTURE_ VIRTEX	Virtex	Controls when to capture register information for readback.	—	CAP, CLK
READBACK	XC4000E XC4000X XC5200 Spartan SpartanXL	Accesses the bitstream readback function. A low-to-high transition on the TRIG input initiates the readback process.	DATA, RIP	CLK, TRIG
MD0	XC4000E XC4000X XC5200	Connects to the Mode 0 (M0) input pin, which is used to determine the configuration mode.	I	—
MD1	XC4000E XC4000X XC5200	Connects to the Mode 1 (M1) input pin, which is used to determine the configuration mode.	—	O
MD2	XC4000E XC4000X XC5200	Connects to the Mode 2 (M2) input pin, which is used to determine the configuration mode.	I	—

RAM and ROM

Some of the most frequently instantiated library components are the RAM and ROM primitives. Because most synthesis tools are unable to infer RAM or ROM components from the source HDL, the primitives must be used to build up more complex structures. The following list of RAM and ROM components is a complete list of the primitives available in the Xilinx library. For more information on the

components, see the *Programmable Logic Data Book* and the online *Libraries Guide.*

Table C-5 Memory Components

Name	Library	Description	Outputs	Inputs
RAM16X1	XC4000E XC4000X	A 16-word by 1-bit static read-write random-access memory component.	O	D, A3, A2, A1, A0, WE
RAM16X1D	XC4000E XC4000X Spartan SpartanXL Virtex	A 16-word by 1-bit dual port random access memory with synchronous write capability and asynchronous read capability.	SPO, DPO	D, A3, A2, A1, A0, DPRA3, DPRA2, DPRA1, DPRA0, WE, WCLK
RAM16X1S	XC4000E XC4000X Spartan SpartanXL Virtex	A 16-word by 1-bit static random access memory with synchronous write capability and asynchronous read capability.	O	D, A3, A2, A1, A0, WE, WCLK
RAM32X1	XC4000E XC4000X	A 32-word by 1-bit static read-write random access memory.	O	D, A0, A1, A2, A3, A4, WE
RAM32X1S	XC4000E XC4000X Spartan SpartanXL Virtex	A 32-word by 1-bit static random access memory with synchronous write capability and asynchronous read capability.	O	D, A4, A3, A2, A1, A0, WE, WCLK
RAMB4_Sn	Virtex	4096-Bit dedicated random access memory blocks with synchronous write capability	DOA DOB	WEA, ENA, RSTA, CLKA, ADDRA, DIA

Table C-5 Memory Components

Name	Library	Description	Outputs	Inputs
RAMB4_ Sn_Sn	Virtex	4096-Bit dual-ported dedicated random access memory blocks with synchronous write capability	DOA DOB	WEA, ENA, RSTA, CLKA, ADDRA, DIA, WEB, ENB, RSTB, CLKB, ADDRB, DIB
ROM16X1	XC4000E XC4000X Spartan SpartanXL	A 16-word by 1-bit read-only memory component.	O	A3, A2, A1, A0
ROM32X1	XC4000E XC4000X Spartan SpartanXL	A 32-word by 1-bit read-only memory component.	O	A4, A3, A2, A1, A0

Global Buffers

Each Xilinx PLD device has multiple styles of global buffers; the XC4000EX devices have 20 actual global buffers—eight BUFGLSs, eight BUFEs, and four BUFFCLKs. For some designs it may be necessary to use the exact buffer desired to ensure appropriate clock distribution delay.

For most designs, the BUFG, BUFGS, and BUFGP components can be inferred or instantiated, thus allowing the design implementation

tools to make an appropriate physical buffer allocation. For more information on the components, see the *Programmable Logic Data Book*.

Table C-6 Global Buffer Components

Name	Library	Description	Outputs	Inputs
BUFG	XC3000 XC4000E XC4000X XC5200 XC9000 Spartan SpartanXL Virtex	An architecture-independent global buffer, distributes high fan-out clock signals throughout a PLD device.	O	I
BUFGP	XC4000E XC5200 Spartan Virtex	A primary global buffer, distributes high fan-out clock or control signals throughout PLD devices.	O	I
BUFGS	XC4000E XC5200 Spartan	A secondary global buffer, distributes high fan-out clock or control signals throughout a PLD device.	O	I
BUFGLS	XC4000X SpartanXL	Global low-skew buffer. BUFGLS components can drive all flip-flop clock pins.	O	I
BUFGE	XC4000X	Global early buffer. XC4000EX devices have eight total, two in each corner. BUFGE components can drive all clock pins in their corner of the device.	O	I
BUFFCLK	XC4000X	Fast clocks. XC4000EX devices have 4 total, 2 each on the left and right sides. BUFFCLK components can drive all IOB clock pins on their left or right half edge.	O	I
BUFGSR	XC9000	Global Set/Reset buffer	O	I
BUFGTS	XC9000	Global Tri-State Enable buffer.	O	I

Fast Output Primitives (XC4000X only)

One of the features added to the XC4000X architecture is the fast output MUX. There is one fast output MUX located in each IOB which can be used to implement any two input logic functions. Each component can have zero, one, or two inverted inputs. Because the output MUX is located in the IOB, it must be connected to the input pin of either an OBUF or an OBUT. For more information on the output primitives, see the *Programmable Logic Data Book*.

Note: For information on how to instantiate output MUXs with inverted inputs, see the *Synopsys (XSI) Interface/ Tutorial Guide*.

Table C-7 Fast Output Primitives

Name	Library	Description	Outputs	Inputs
OAND2	XC4000X	2-input AND gate that is implemented in the output multiplexer of the XC4000EX IOB.	O	F, I0
ONAND2	XC4000X	2-input NAND gate that is implemented in the output multiplexer of the XC4000EX IOB.	O	F, I0
OOR2	XC4000X	2-input OR gate that is implemented in the output multiplexer of the XC4000EX IOB.	O	F, I0
ONOR2	XC4000X	2-input NOR gate that is implemented in the output multiplexer of the XC4000EX IOB.	O	F, I0
OXOR2	XC4000X	2-input exclusive OR gate that is implemented in the output multiplexer of the XC4000EX IOB.	O	F, I0
OXNOR2	XC4000X	2-input exclusive NOR gate that is implemented in the output multiplexer of the XC4000EX IOB.	O	F, I0
OMUX2	XC4000X	2-by-1 MUX implemented in the output multiplexer of the XC4000EX IOB.	O	D0, D1, S0

IOB Components

Depending on the synthesis vendor being used, some IOB components must be instantiated directly in the input design. Most synthesis tools support IOB D-type flip-flop inferences but may not yet support IOB D-type flip-flop inference with clock enables. Because there are many slew rates and delay types available, there are many derivatives of the primitives shown. For a complete list of the IOB primitives, see the online *Libraries Guide*.

Table C-8 Input/Output Block Components

Name	Library	Description	Outputs	Inputs
IBUF	XC3000 XC4000E XC4000X XC5200 XC9000 Spartan SpartanXL	Single input buffers. An IBUF isolates the internal circuit from the signals coming into a chip.	O	I
OBUF	XC3000 XC4000E XC4000X XC5200 XC9000 Spartan SpartanXL	Single output buffers. An OBUF isolates the internal circuit and provides drive current for signals leaving a chip.	O	I
OBUFT	XC3000 XC4000E XC4000X XC5200 XC9000 Spartan SpartanXL	Single 3-state output buffer with active-low output enable. (3-state High.)	O	I,T
OBUFE	XC9000	Single 3-state output buffer with active-high output enable. (3-state Low.)	O	I, T

Table C-8 Input/Output Block Components

Name	Library	Description	Outputs	Inputs
IFD	XC3000 XC4000E XC4000X XC5200 Spartan SpartanXL	Single input D flip-flop.	Q	D, C
OFD	XC3000 XC4000E XC4000X XC5200 Spartan SpartanXL	Single output D flip-flop.	Q	D, C
OFDT	XC3000 XC4000E XC4000X XC5200 Spartan SpartanXL	Single D flip-flop with active-high 3-state active-low output enable buffers.	O	D, C,T
IFDX	XC4000E XC4000X Spartan SpartanXL	Single input D flip-flop with clock enable.	Q	D0, D1, S0
OFDX	XC4000E XC4000X Spartan SpartanXL	Single output D flip-flop with clock enable.	Q	D, C, CE

Table C-8 Input/Output Block Components

Name	Library	Description	Outputs	Inputs
OFDTX	XC4000E XC4000X XC5200 Spartan SpartanXL	Single D flip-flop with active-high tristate and active-low output enable buffers.	O	D, C, CE, T
ILD_1	XC3000 XC4000E XC4000X XC5200 Spartan SpartanXL	Transparent input data latch with inverted gate. (Transparent High.)	Q	D, G

Clock Delay Components

These components are delay locked loops that are used to eliminate the clock delay inside the device. The delay locked loop is a digital variation of the analog phase locked loop.

Table C-9 Clock Delay Component

Name	Library	Description	Outputs	Inputs
CLKDLL	Virtex	Clock delay locked loop used to minimize clock skew.	CLK0, CLK90, CLK180, CLK270, CLS2X, CLKDV, LOCKED	CLKIN, CLKFB, RST
CLKDLLHF	Virtex	High frequency clock delay locked loop used to minimize clock skew.	CLK0, CLK180, CLKDV, LOCKED	CLKIN, CLKFB, RST

Appendix D

File Processing Overview

This section shows the files created/used by Foundation to process FPGA and CPLD designs. These figures represent 1) a high-level overview of Foundation's processing tools and 2) the manipulation of the various netlist and constraint files by these tools.

FPGAs

The following three figures illustrate the processing that Foundation performs to create FPGA designs.

Figure D-1 Manipulation of Netlist and Constraint Files for FPGAs (Part 1)

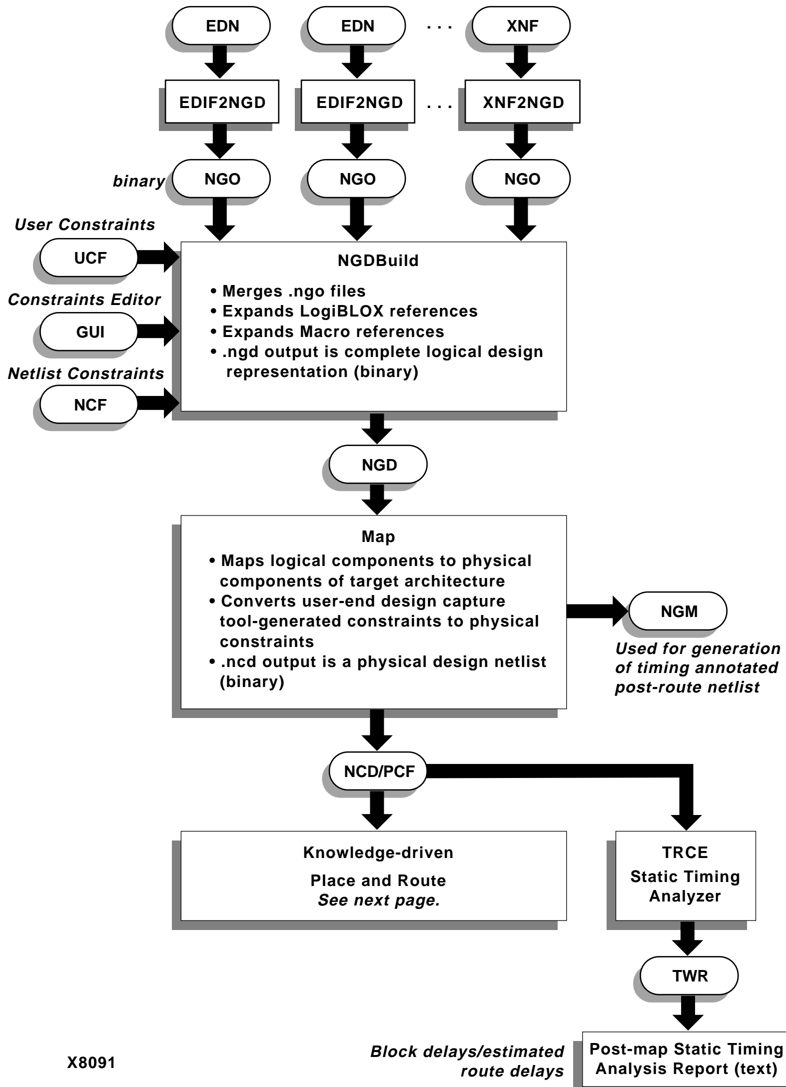


Figure D-2 Manipulation of Netlist and Constraint Files for FPGAs (Part 2)

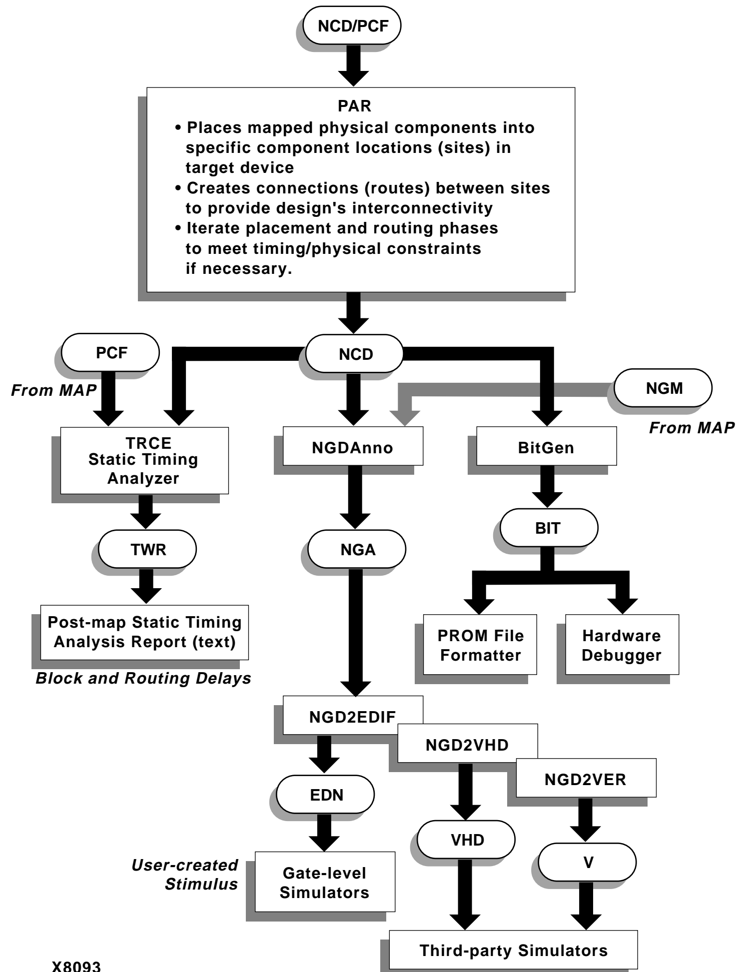


Figure D-3 Manipulation of Netlist and Constraint Files for FPGAs (Part 3)

CPLDs

The following three figures illustrate the processing that Foundation performs to create CPLD designs.

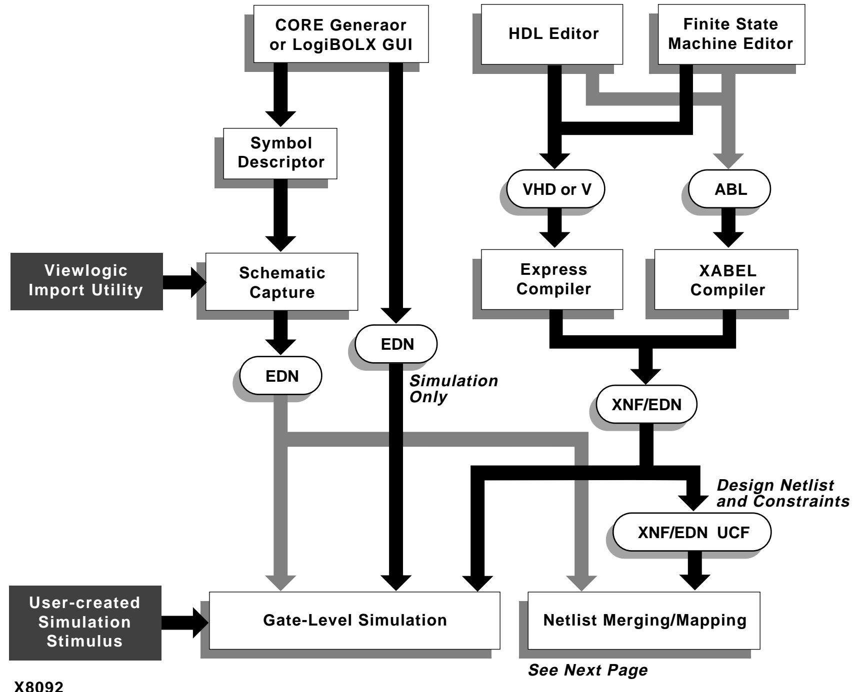


Figure D-4 Manipulation of Netlist and Constraint Files for CPLDs (Part 1)

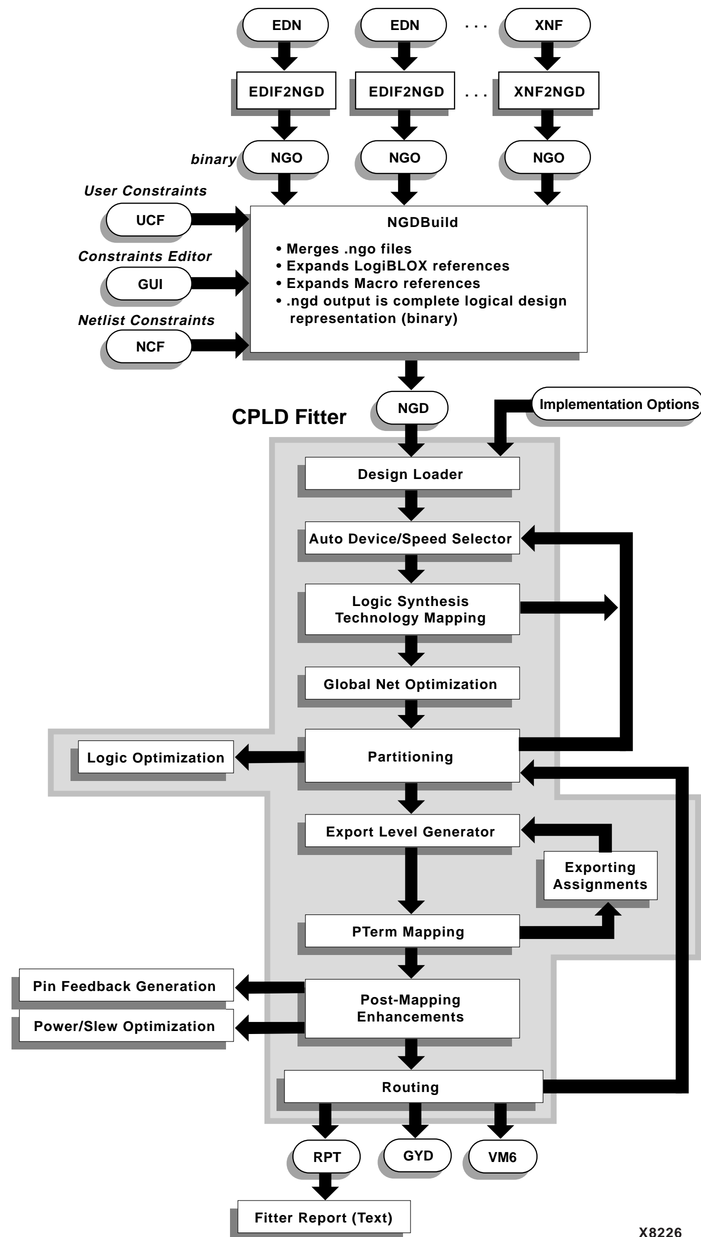


Figure D-5 Manipulation of Netlist and Constraint Files for CPLDs (Part 2)

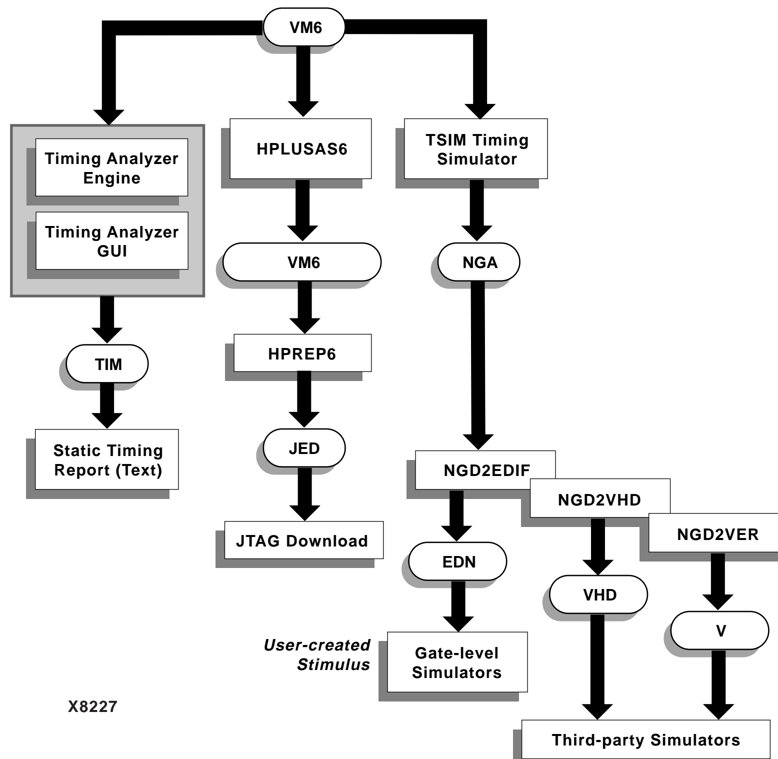


Figure D-6 Manipulation of Netlist and Constraint Files for CPLDs (Part 3)