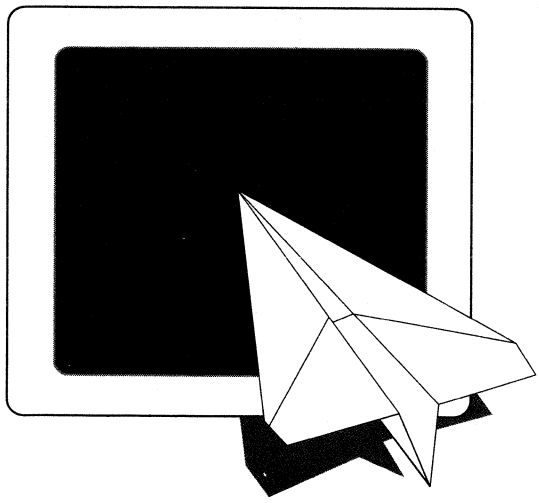


XACT User Guide

Introduction

Design Entry

Design Implementation

Design Verification

FPGA Design Implementation Flows

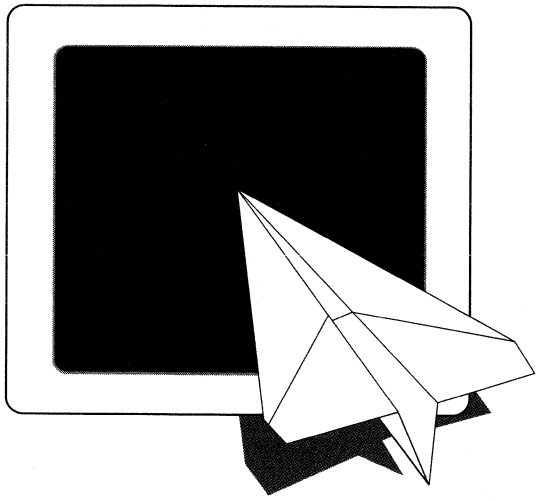
Configuration, Length Count, and Debugging

The XC4000 Readback Capability

Boundary Scan in XC4000 Devices

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.



XACT Design Editor Tutorial

Index

XACT User Guide

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Preface

About This Manual

This manual provides an overview of Xilinx FPGA logic devices, design entry, design implement, and design verification.

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

Another publications you can consult for related information is *The Programmable Logic Data Book.*

Manual Contents

This manual covers the following topics.

- Chapter 1, "Introduction," introduces Xilinx FPGA logic devices.
- Chapter 2,"Design Entry," describes the design entry process, which includes schematic entry, text-based entry, and functional simulation.
- Chapter 3, "Design Implementation," describes the design implementation process, from XNF translation to generating a bitstream.
- Chapter 4, "Design Verification," describes the design verification process, which includes simulation, static timing analysis, and in-circuit verification.
- Chapter 5, "Design Implementation Flows," describes the automatic design implementation flow using XMake for each Xilinx FPGA family.

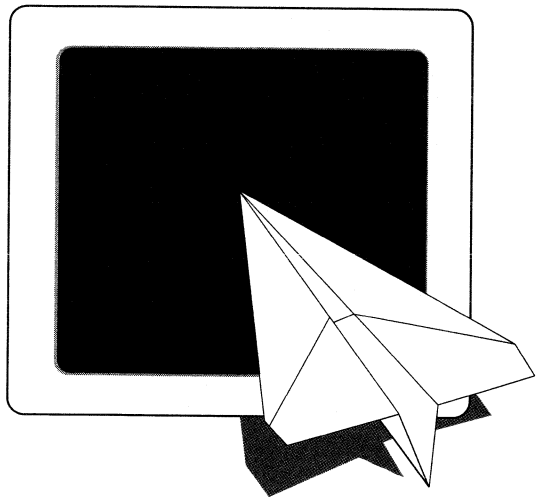
- Chapter 6, "Configuration, Length Count, and Debugging," provides tips for when you encounter problems during configuration and downloading.
- Chapter 7, "The XC4000 Readback Capability," describes performing a readback to verify data configuration.
- Chapter 8, "Boundary Scan in XC4000 Devices," describes the process for testing the integrity of your board.
- Chapter 9, "XACT Design Editor Tutorial," teaches you the basic skills you need to use XDE.

Conventions

The following conventions are used in this manual's syntactical statements:

`Courier font regular`	System messages or program files appear in regular Courier font.
`Courier font bold`	Literal commands that you must enter in syntax statements are in bold Courier font.
italic font	Variables that you replace in syntax statements are in italic font.
`[ ]`	Square brackets denote optional items or parameters. However, in bus specifications, such as bus [7:0], they are required.
`{ }`	Braces enclose a list of items from which you must choose one or more.
. . .	A vertical ellipsis indicates material that has been omitted.
. . .	A horizontal ellipsis indicates that the preceding can be repeated one or more times.
|	A vertical bar separates items in a list of choices.
↵	This symbol denotes a carriage return.

Contents



Introduction

XACT User Guide

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Chapter 1

Introduction

This manual describes the XACT Development System, which enables you to enter, implement, and verify your designs in a matter of hours. Before actually starting with the design process, you should read the following introduction to the Xilinx Field Programmable Gate Array (FPGA) devices.

For information on EPLDs, refer to the *XEPLD Design Guide* or *XEPLD Reference Guide.*

This chapter introduces Xilinx FPGA logic devices, provides an overview of the FPGA design process, and introduces the XACT Development System documentation set.

Xilinx FPGA Logic Devices

To integrate logic into smaller spaces, Xilinx has developed a type of FPGA called a Logic Cell Array (LCA). Several families can accommodate as many as 9,000 gates on a single device. The XC4000 family allows more than 20,000 gates on a single device. For more information about a specific Xilinx device, refer to *The Programmable Logic Data Book.*

Advantages of Xilinx FPGAs

The most significant advantage of using the Xilinx line of FPGA products is the ability to produce a prototype logic design on your desktop. You can create a logic design, implement it, and verify it in hours, while conventional gate array products can take months to develop and produce working silicon.

In addition, Xilinx logic products are in-circuit programmable, so even while an FPGA device is soldered to a board, you can reprogram the Xilinx part with a different FPGA design.

Xilinx FPGA Families

Xilinx markets and supports many product lines — XC2000, XC2000L, XC3000, XC3000A, XC3000L, XC3100, XC3100A, XC4000, XC4000A, and XC4000H. The primary difference between these products lies in the number of gates and the architectural features of the individual devices, as shown in the following table.

Table 1-1 Maximum Logic Capacities

Product	Logic Capacity (Gate Equivalent)	Max. No. of CLBs	Max. No. of IOBs
XC2000/L	1,200 - 1,800	100	74
XC3000/A/L	1,300 - 9,000	484	176
XC3100/A	1,300 - 9,000	484	176
XC4000	2,000 - 20,000	900	240
XC4000A	2,000 - 5,000	196	112
XC4000H	2,000 - 5,000	196	192

FPGA Architecture

The FPGA architecture consists of three types of configurable elements — a perimeter of input/output blocks (IOBs), a core array of configurable logic blocks (CLBs), and resources for interconnection. The IOBs provide a programmable interface between the internal logic array and the device package pins, CLBs perform user-specified logic functions, and the interconnect resources carry signals among the blocks.

A configuration program stored in internal static memory cells determines the logic functions and the interconnect. The configuration data is loaded into the device during power-up or when you reprogram.

FPGA devices are customized by loading configuration data into internal memory cells (latches). The FPGA device can either actively read its configuration data out of an external serial or byte-wide parallel PROM (master modes), or the configuration data can be written into the FPGA device (slave and peripheral modes).

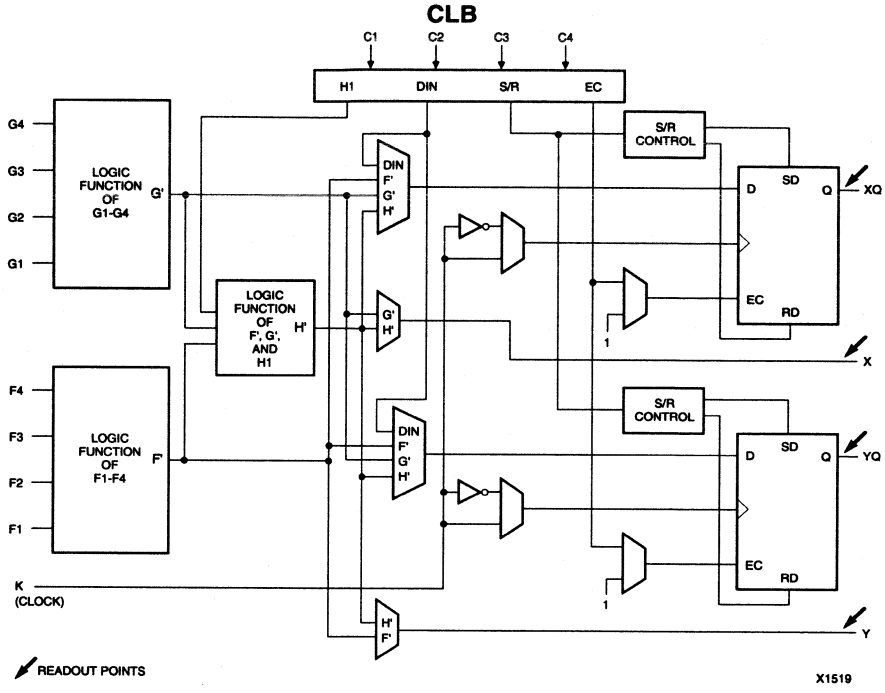


Figure 1-1 XC4000 CLB Structure

XC3000/XC3000A/XC3000L/XC3100/XC3100A devices — similar to XC2000 devices, CLBs in XC3000 devices have two function generators per CLB, a combinatorial logic section, and an internal control section. However, they have an additional storage element and more inputs.

Each XC3000 CLB includes five logic inputs, a common clock input, an asynchronous direct reset input, a clock enable, and two outputs. A data-in input is also provided for direct input to the flip-flops within the CLB.

Figure 1-2 illustrates an XC3000 CLB structure.

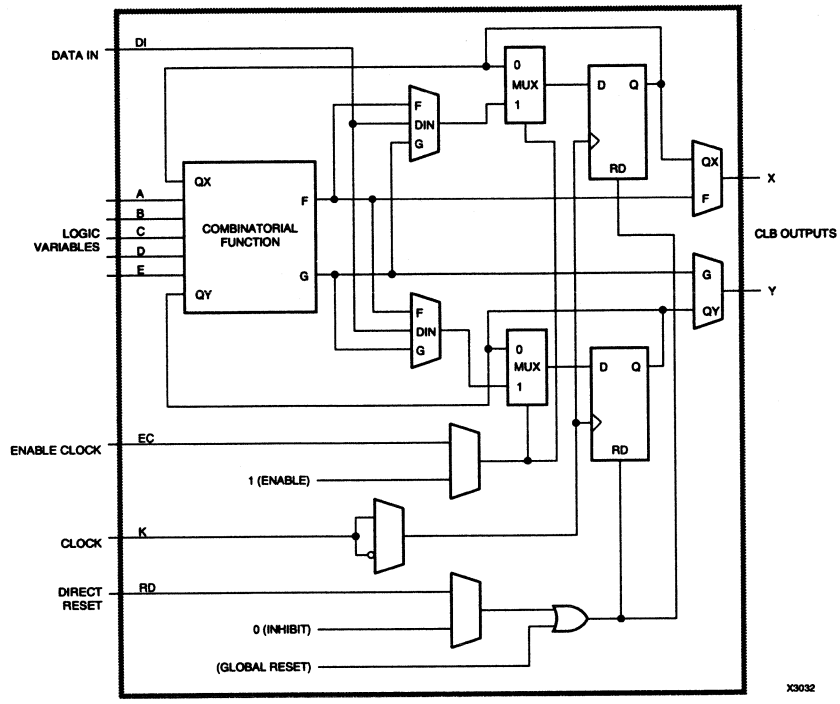


Figure 1-2 XC3000 CLB Structure

XC2000/XC2000L devices — each CLB in XC2000 devices has two function generators, a combinatorial logic section, a storage element, and an internal routing and control section. In addition, each CLB has four general-purpose inputs, a clock input, and two outputs. Figure 1-3 illustrates an XC2000 CLB structure.

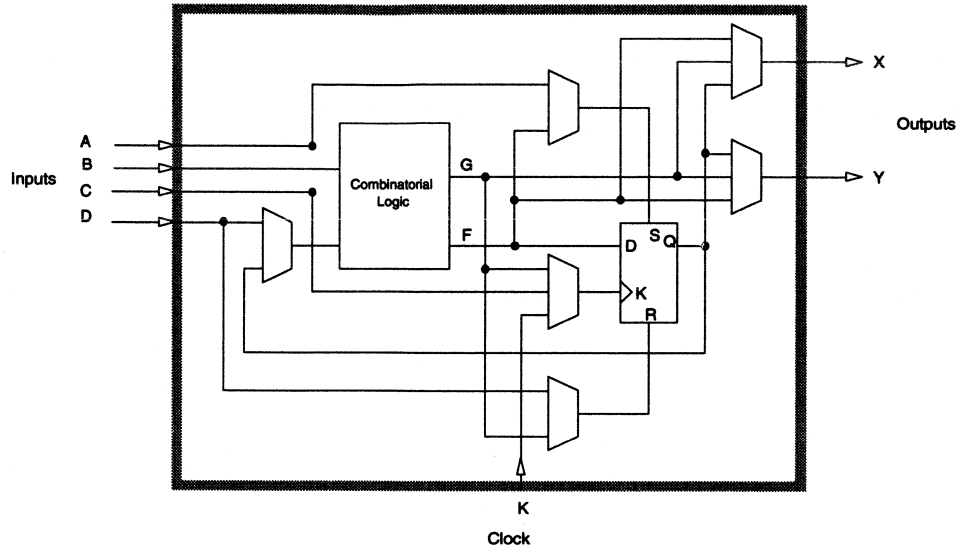


Figure 1-3 XC2000 CLB Structure

Input/Output Blocks

Input/output blocks (IOBs) on the FPGA device provide an interface between an external package pin and internal logic blocks (CLBs). FPGA routing resources then connect the IOBs to the CLB inputs and outputs.

XC4000/XC4000A devices — the XC4000 IOB input includes both registered and direct input paths, and each output provides a 3-state output buffer that can be driven by a registered or direct output signal. Configuration options on the IOB output include an inversion, a controlled slew-rate output, a 3-state control inversion, a clock inversion, and programmable flip-flop initialization states. Configuration options on the inputs include clock inversion and a programmable delay to eliminate input hold time.

A pull-up or pull-down resistor can be activated for either inputs or outputs. Input registers can be flip-flops or latches with programmable initialization states. An XC4000 IOB structure is shown in Figure 1-4.

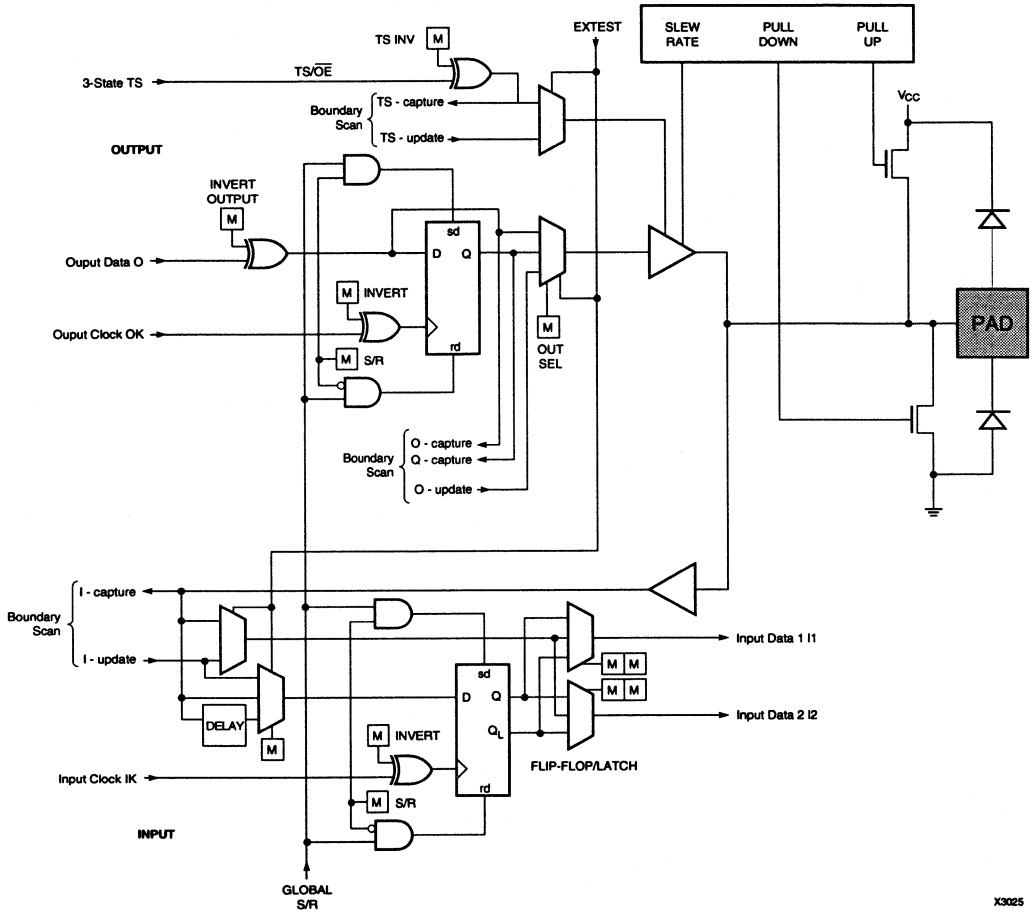


Figure 1-4 XC4000/XC4000A IOB Structure

XC4000H devices — except for the I/O structure, the XC4000H family is identical to the XC4000 family. The XC4000H family almost doubles the number of input/output pins. XC4000H devices, however, contain no input or output flips, as illustrated in the following figure.

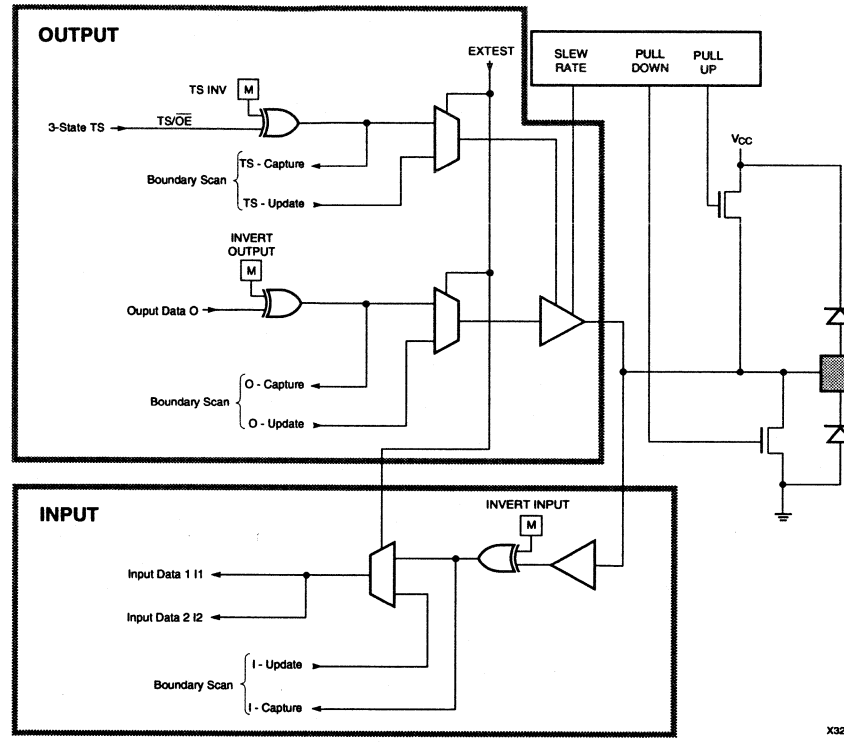


Figure 1-5 XC4000H IOB Structure

XC3000/XC3000A/XC3000L/XC3100/XC3100A devices — the XC3000 IOB input includes both registered and direct input paths, and each output provides a register, optional inversion, a controlled slew-rate output, and optional 3-state control inversion.

IOB clocks for an entire edge can be inverted. Configuration options on the inputs include a pull-up resistor. An XC3000 IOB structure is shown in Figure 1-6.

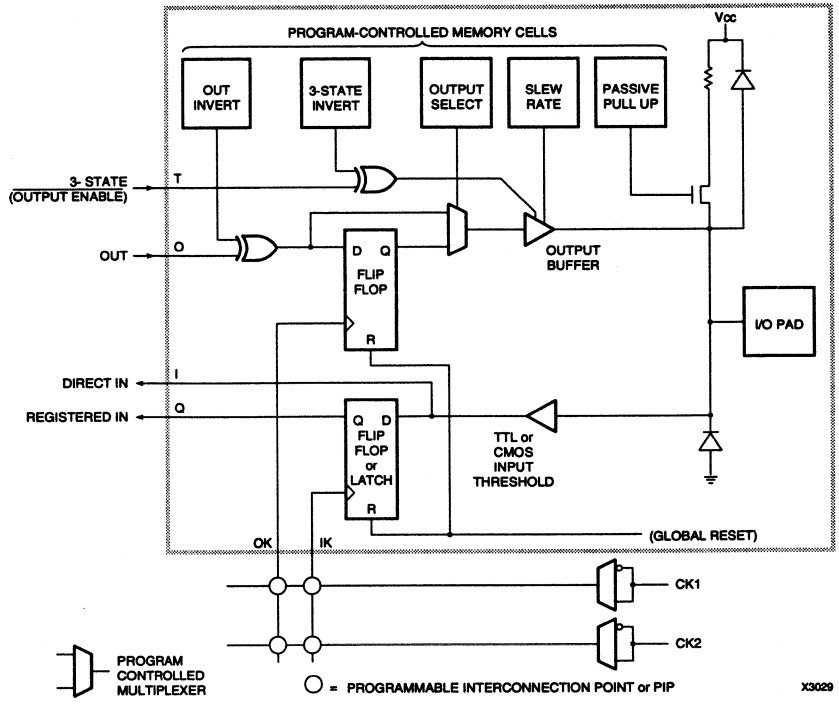


Figure 1-6 XC3000 IOB Structure

XC2000/XC2000L devices — the XC2000 IOBs include a direct or registered input and a 3-state output. Figure 1-7 illustrates an XC2000 IOB structure.

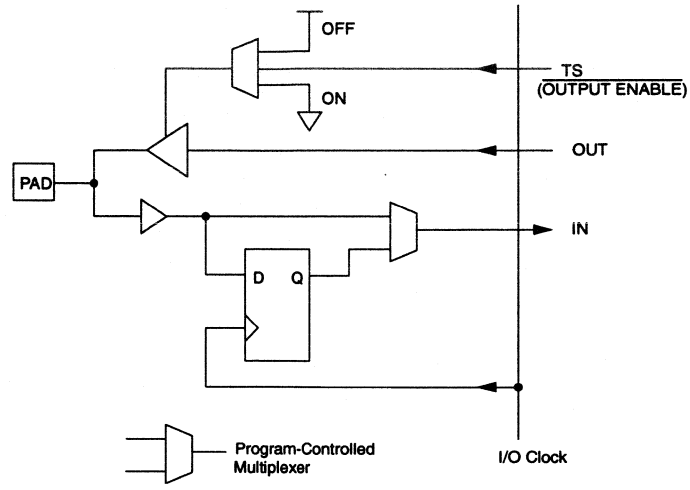


Figure 1-7 XC2000 IOB Structure

Global Resources

Each device includes global resources, which distribute clock signals throughout the device with very low skew.

- GCLK and ACLK primitives

  The GCLK and ACLK primitives correspond to the global clock buffer and the alternate clock buffer in XC2000/L, XC3000/A/L, and XC3100/A devices. You can only use one GCLK and one ACLK in a single FPGA design.

- BUFGP and BUFGS primitives

  BUFGP and BUFGS correspond to the primary and secondary global buffers, which distribute high-fanout clock or control signals throughout XC4000/A/H devices. You can use up to four BUFGP and BUFGS primitives in an XC4000/A/H design.

Routing Resources

The CLBs and IOBs, as mentioned in the preceding sections, are interconnected using the routing resources provided on the device. The Xilinx mapping, placement, and routing software chooses the

best resource to use for a particular signal type. The different types of FPGA routing resources are described in this section. For more detailed information about routing resources for each Xilinx FPGA, refer to *The Programmable Logic Data Book.*

Horizontal and vertical longlines — in all devices other than XC2000/L, you can use horizontal longlines driven by 3-state buffers for bidirectional data busses. Other longlines (vertical and non-TBUF-driven horizontal) are useful for high-fanout nets. Figure 1-8 illustrates horizontal and vertical longlines on an XC3000 device.

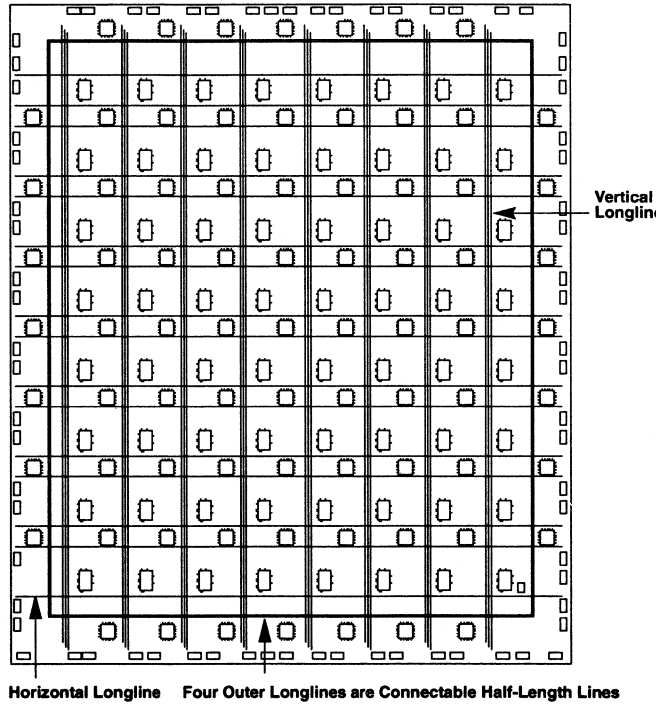


Figure 1-8 Horizontal and Vertical Longlines (XC3000)

Direct interconnect (XC2000 and XC3000 only) — direct interconnect are direct paths used to connect adjacent CLBs or to connect adjacent CLBs and IOBs. Each CLB output can be connected to the input of a CLB or IOB adjacent to it through a direct interconnect segment.

These resources are best used for high-speed signals between adjacent blocks as illustrated by Figure 1-9.

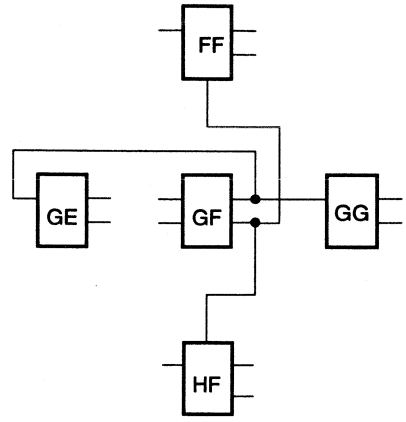


Figure 1-9 Example of CLBs with Direct Interconnect (XC3000)

General purpose interconnect — general purpose interconnect consists of an array of short adjacent metal segments oriented vertically and horizontally between the rows and columns of CLBs, as illustrated in Figure 1-10. Switch matrices connect the metal segments.

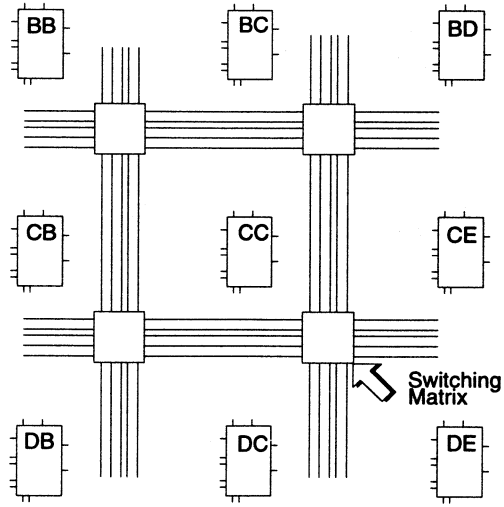


Figure 1-10 General Purpose Interconnects (GPIs)

Programmable interconnect points (PIPs) — PIPs are individual switches that enable the connection between intersecting routing segments, or from routing segments to CLB or IOB pins, as illustrated in Figure 1-11.

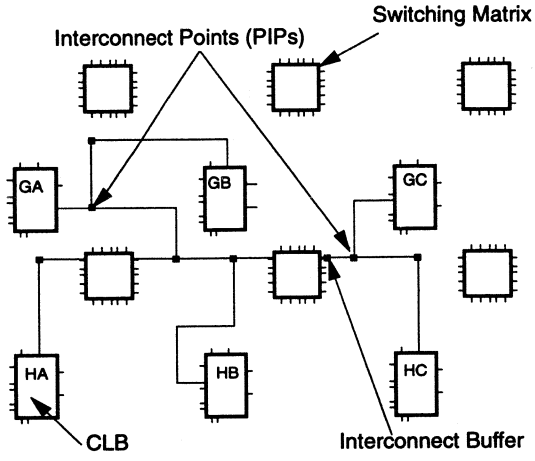


Figure 1-11 Programmable Interconnect Points (PIPs)

Switch matrices — a matrix of switches is located at the intersections of the horizontal and vertical groups of general-purpose interconnect segments. These matrices are also referred to as magic boxes. Figure 1-12 shows the possible pin-to-pin connections for an XC3000 switching matrix.

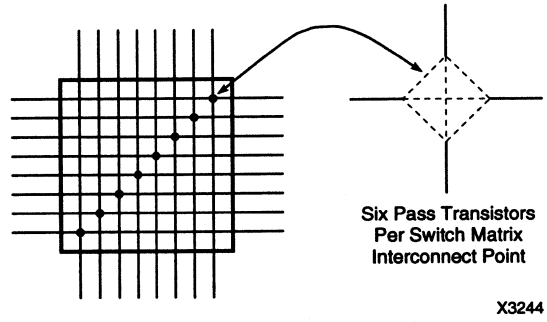


Figure 1-12 Switching Matrix

Design Flow Overview

The FPGA design flow is a 3-step process that consists of the following stages.

- Design Entry — In this stage of the design flow you create your design using a Xilinx-supported schematic editor or hardware description language (HDL).
- Design Implementation — By partitioning, placing, and routing your design, you convert the design file created in the design entry stage into an LCA file format. Then you create a bitstream file from the LCA file and optionally program a PROM or EPROM for subsequent programming of your Xilinx device.
- Design Verification — Using a simulator, the Xilinx XChecker™ cable, or the Xilinx Download cable, you ensure that your design meets your timing requirements and functions properly.

Overviews of design entry, design implementation, and design verification are discussed in the "Design Entry," "Design Implementation," and "Design Verification" chapters, respectively.

The full design flow is an iterative process of entering, implementing, and verifying your design until it is correct and complete. The XACT Development System allows quick design iterations through the design flow cycle. Since FPGA devices permit unlimited reprogramming, you do not need to discard devices when debugging your design in-circuit. See the "Design Implementation Flow" chapter in this user guide for design flows for specific Xilinx devices.

Figure 1-13 illustrates the Xilinx design flow.

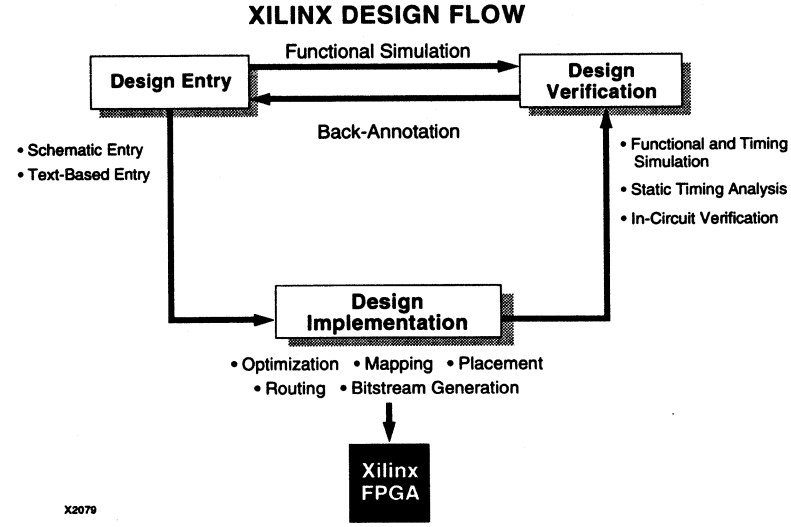


Figure 1-13 Xilinx Design Flow

The following table, Table 1-2, defines the terms used in Figure 1-13.

Table 1-2 Design Flow Terms

Term	Description
Schematic entry	Entry using graphic symbols
Text-based entry	Entry using a design language
Optimization	Converting device-independent or behavioral logic descriptions to a form that can be efficiently implemented in a Xilinx FPGA
Mapping	Dividing a design's logic into the resources of the Xilinx FPGA
Placement	Assigning design blocks created during mapping to specific locations in the FPGA
Routing	Assigning the interconnect paths
Bitstream generation	Converting a design into a bitstream that can be loaded into a Xilinx FPGA
Back-annotation	Matching schematic and routed nets so that the same input stimuli can be used for functional and timing simulation
Simulation	Software testing of the design logic and timing using input stimuli
Static timing analysis	Analyzing the signal paths to ensure that the delays are acceptable
In-circuit verification	Testing of the design in a system after it is loaded into a Xilinx FPGA

XACT Development System Documentation

The XACT Development System documentation set consists of a series of books that help you use the Xilinx Development System software in conjunction with your schematic entry tools. The documentation set includes the following manuals: *XACT User Guide, XACT Libraries Guide, X-BLOX User Guide, Xilinx ABEL User Guide, XACT Reference Guide, Volumes 1-3, XACT Hardware and Peripherals Guide,* and your CAE-specific interface user guide.

The following sections give a brief description of what information each manual contains.

XACT User Guide

This guide contains information you need during the FPGA development process. It contains an overview of design entry, design implementation, and design verification. This guide also covers how to perform a readback and boundary scan for XC4000 devices, as well as configuration and debugging hints.

XACT Libraries Guide

The *XACT Libraries Guide* manual presents information about the various Xilinx-supplied primitives and macros in your schematic editor. A cross-reference assists you in determining which primitives and macros are designed for use with a particular Xilinx device family.

X-BLOX User Guide

The *X-BLOX User Guide* describes the X-BLOX (blocks of logic optimized for Xilinx™) synthesis tool, which consists of a library of modules you can use to describe a system by means of high-level functions instead of gate-level primitives.

Xilinx ABEL User Guide

The *Xilinx ABEL User Guide* describes the Xilinx ABEL program, which consists of a Xilinx-specific version of the ABEL design entry software and a series of translation programs. It allows you to create modules for Xilinx FPGA designs using state machines, Boolean equations, and truth tables. It also allows you to create full EPLD designs or modules for these designs.

XACT Reference Guide, Volume 1

The *XACT Reference Guide, Volume 1* provides detailed information on the various design entry programs in the XACT Development System. The entry for each program includes the program's syntax, options, files, and error and warning messages. Information about program function and examples is also included.

XACT Reference Guide, Volume 2

The *XACT Reference Guide, Volume 2* guide provides detailed information on the various design implementation programs in the XACT Development System. The entry for each program includes the program's syntax, options, files, and error and warning messages. Information about program function and examples is also included.

XACT Reference Guide, Volume 3

The *XACT Reference Guide, Volume 3* provides detailed information on the various design verification programs in the XACT Development System. The entry for each program includes the program's syntax, options, files, and error and warning messages. Information about program function and examples is also included.

XACT Hardware and Peripherals Guide

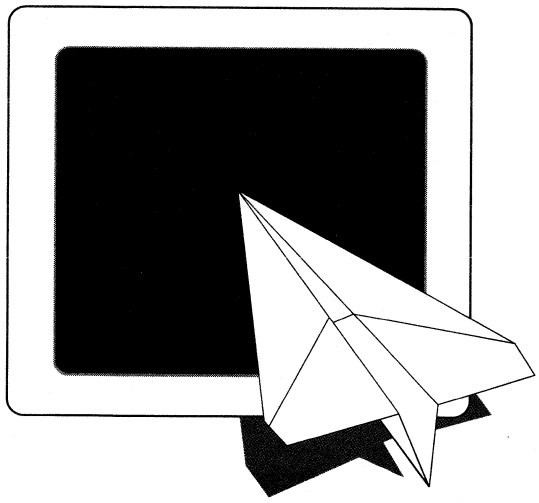
The *XACT Hardware and Peripherals Guide* provides detailed information on the various hardware and peripherals in the XACT Development System. This guide describes the XC3000, XC4000, and FPGA demonstration boards, which allow you to verify your design; the XPP Serial PROM Programmer; and the XChecker download cable and software.

CAE Interface User Guides

The Xilinx interface user guides include the following information:

- Design entry interface — This section gives detailed information on entering an LCA design using a Xilinx-supported schematic editor and then translating it into a Xilinx Netlist Format (XNF) file. Documentation for this section comes with a Xilinx-supported simulation package.
- Design verification interface — This section gives detailed information on simulating a design using a Xilinx-supported simulator. Both timing and functional simulation are described. Documentation for this section comes with a Xilinx-supported simulation package.

- Tutorials — These sections are platform-specific tutorials that walk you step by step through the major functions that your system performs, such as schematic capture and functional and timing simulation. The tutorials might also include information on design issues specific to your platform.
- Additional documentation — Xilinx offers other documents including applications information and specifications that might be helpful. Check with your local sales office or field representative.



Design Entry

XACT User Guide

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Chapter 2

Design Entry

This chapter introduces design entry using both schematic and text-based entry tools, highlights hierarchical design, and discusses ways to control design implementation during design entry.

Design entry takes the design from a concept to a netlist. You can enter a design with a schematic editor, a text-based tool, or both. These entry methods require Xilinx-supported third-party tools, which produce a design file in their own netlist formats.

The following sections describe each design entry method in detail.

Schematic Entry

Schematic tools provide a graphic interface for design entry. You can use these tools to connect symbols representing the logic components in your design. You can build your design with individual gates, or you can combine gates to create functional blocks. This section focuses on ways to enter functional blocks using library elements, the MemGen program, and the logic-design/synthesis tool X-BLOX.

Library Elements

The following sections discuss primitives, macros, and architectural resources.

Primitives and Macros

Xilinx FPGA libraries provide primitives as well as common high-level macro functions. Primitives are basic circuit elements, such as AND and OR gates, with unique library names, symbols, and descriptions. Macros contain multiple library elements, which can include primitives and other macros.

There are two types of macros you can use with Xilinx FPGAs. Soft macros, available for all FPGAs, have pre-defined functionality, but have flexible mapping, placement, and routing. Relationally placed macros (RPMs), available for XC4000/A/H devices only, have fixed mapping and relative placement.

Architectural Resources

Your choice of schematic symbol for a particular function is critical. For example, the standard AND4 symbol is implemented using the function generator of a CLB. If you need a D flip-flop with Reset direct, you must specify a FDRD primitive in your schematic.

The specific choice of schematic symbol also determines how registers are implemented. Registers are available in both CLBs and IOBs. If you want to ensure that a register is implemented using an IOB, you must use the corresponding I/O library primitive. See the *XACT Libraries Guide* for more information.

MemGen for XC4000 Devices

The Xilinx memory generator, MemGen, is a convenient tool for creating RAMs and ROMs for XC4000/A/H FPGAs. MemGen creates an XNF file, a log file, and a schematic symbol, which can be used as part of the design.

You can create memories up to 32 bits wide and 256 words deep. Two 16x1 memories or a single 32x1 memory can fit in one XC4000 CLB. For more information on MemGen commands and capabilities, refer to "The MemGen Program" in the *XACT Reference Guide, Volume 1.*

X-BLOX

The X-BLOX tool provides a library of variable-size MSI- and LSI-level design building blocks such as adders, counters, decoders, and shift-registers. This library complements the XC3000A/L, XC3100A, and XC4000/A/H macro libraries, which contain simpler, fixed-size logic and gate functions. The X-BLOX tool also includes the X-BLOX software, which integrates X-BLOX library elements into your design. For further information on this software, see the *X-BLOX User Guide.*

Text-Based Entry

Text-based entry is well suited for many designs, including state machines and decoders. You can use several hardware description languages (HDLs) to design Xilinx FPGAs, including ABEL-HDL, VHSIC HDL (VHDL), Verilog HDL, PALASM, MINC, and CUPL. This section highlights Xilinx ABEL, VHDL, and Verilog HDL.

Xilinx ABEL

You can enter Boolean equations, state machine descriptions, and truth tables in ABEL-HDL using Xilinx ABEL. Xilinx ABEL is best used for creating designs that are functional blocks within designs entered with a schematic editor. XMake merges functional blocks containing logic described in ABEL-HDL with the rest of your design during design entry. For a more detailed description of Xilinx ABEL, refer to the *Xilinx ABEL User Guide*.

XSI (Xilinx Synopsys Interface)

The XSI design tool kit enables you to implement FPGA designs using the Synopsys High-Level Design Automation (HLDA) synthesis software. Synopsys HLDA synthesis software creates and optimizes circuit designs from hardware description languages such as VHDL and Verilog HDL.

Both the Synopsys Design Compiler and the FPGA Compiler support the XC3000/A/L and XC4000/A/H libraries.

Hierarchical Design

Schematics usually contain hierarchy, which is important because it:

- Helps you conceptualize your design.
- Adds structure to your design.
- Makes it easier to debug your design.
- Makes it easier to combine different design entry methods (schematic and text) for different parts of your design.

- Makes it easier to design incrementally. Incremental design consists of designing, implementing, and verifying individual sub-blocks that build a design in stages.
- Facilitates concurrent design. Concurrent design is the process of dividing a design among a number of people who develop different parts of the design in parallel.

Hierarchical Names

A specific hierarchical name indicates each library element, unique block, and instance you create. For example, the last three terms in the name

```
/Acc/alu_1/mult_4/8count_3/4bit_0/mux_1/or2
```

refer to the 2-input OR gate in the first instance of a multiplexer in a 4-bit counter.

Note: Xilinx strongly recommends that you name the components and nets in your design. In schematic editors, component names and net names are preserved and used by the XACT Design Editor. The component names and net names are used for back-annotation and appear in the debug and analysis tools. If you do not name your components and nets, the schematic editor automatically generates the names. For example, the software might name the previous example the following:

```
/$1a123/$1b942/$1c23/$1d235/$1e121/$1g123/$1h57
```

Consequently, it can be very difficult to analyze circuits with automatically generated names, since they only have any significance to the Xilinx software.

Controlling Implementation

If your application requires that you constrain your design, you can specify mapping and block placement during design entry.

Mapping

You can specify how a particular block of logic is mapped into CLBs using a CLBMAP for XC2000, XC3000, XC3000A/L, and XC3100/A FPGAs, and an FMAP or HMAP for XC4000/A/H FPGAs. These

mapping symbols can be used in your schematic. However, if you overuse these specifications, it might be harder to route your design.

Block Placement

Block placement can be constrained to a specific location, an area on the device, or along a longline. All three can be specified in a constraints (CST) file. Location and area constraints can be specified directly from the schematic. Poor block placement can adversely affect both the placement and the routing. Typically, block placement defines IOB placement.

Note: The design implementation software allows specific nets to be flagged as critical or given weighted priorities. However, too many net weights and critical flags can easily degrade the placement and routing of a design, resulting in lower performance. For this reason, Xilinx strongly recommends that critical flags or net weights be assigned only where absolutely necessary.

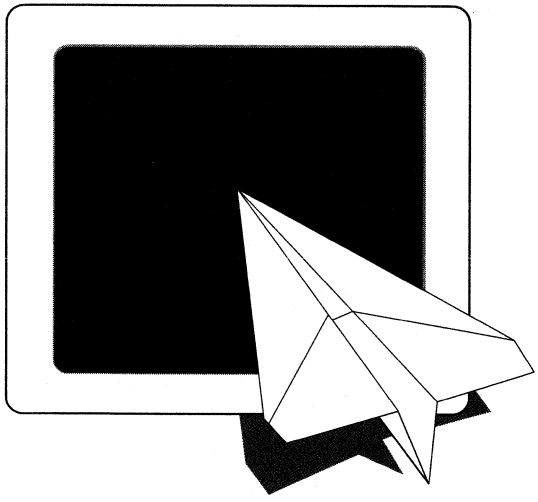
Timing Specifications

You can specify timing requirements for paths in your design directly from your schematic for the following devices: XC3000A/L, XC3100A, and XC4000/A/H. PPR uses these timing specifications to achieve optimum performance when placing and routing your design. See the "XACT-Performance Utility" chapter in the *XACT Reference Guide, Volume 1* for detailed instructions on using this feature.

Performing Functional Simulation

After you have entered your design, you can either simulate or implement your design. Functional simulation tests the logic in your design to determine if it works properly. You can save a lot of time during subsequent design steps if you perform functional simulation early in the design flow. Details on functional simulation can be found in the CAE-specific interface user guide provided with your Xilinx interface.

The design implementation process implements your design into the specific Xilinx FPGA. For more information on design implementation, refer to the "Design Implementation" chapter in this user guide.



Design
Implementation

XACT User Guide

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Chapter 3

Design Implementation

Design implementation, performed by the XMake program, generates a bitstream for a Xilinx FPGA from the design netlist. Design implementation includes the following steps: translation from the design netlist to the Xilinx Netlist Format (XNF), design optimization, merging, mapping, placement, routing, and bitstream generation. The XMake program automatically carries out all these steps.

Design flows for each family are discussed in the "Design Implementation Flows" chapter in this user guide.

Table 3-1 outlines the design implementation programs. Although XMake automatically runs many of these programs, the following table enables you to examine the role each program plays in design implementation.

Table 3-1 Programs Used in Design Implementation

Program	Function	Input	Output	Families
XMake	Automatic design implementation	Schematic output or XNF file	BIT file and LCA file	All
XNFMerge	Merges multiple XNF files	XNF files	Single flattened XFF file	All
XNFPrep	Performs a design rule check (DRC) and removes unused and redundant logic	XNF file	XTF file or XTG file (X-BLOX)	XC2000 XC2000L XC3000 XC3000A XC3000L XC3100 XC3100A XC4000
XNFMAP	Divides the design among device resources	XTF file	MAP file and a CRF file	XC2000 XC2000L XC3000 XC3000A XC3000L XC3100 XC3100A
MAP2LCA	Converts a MAP file to an LCA file	MAP file	LCA file SCP file	XC2000 XC2000L XC3000 XC3100
APR	Automatically places and routes	LCA, SCP, and CST files	Routed LCA file	XC2000 XC2000L XC3000 XC3100
PPR	Maps, places, and routes	Merged XNF file and constraint file	Placed and routed LCA file	XC3000A XC3000L XC3100A XC4000 XC4000A XC4000H

Program	Function	Input	Output	Families
MakeBits	Generates FPGA bitstream	Routed LCA file	BIT file	All
XDE	Manual design editing	LCA file	LCA file	All

Each of the programs in Table 3-1 produces report files, which you should examine to determine the progress of your design. For a description of report file contents, refer to the appropriate volume of the *XACT Reference Guide* as indicated in Table 3-2.

Table 3-2 Design Implementation Documentation

Program	Reference Guide Volume
XMake	*XACT Reference Guide, Volume 1*
XNFMerge	*XACT Reference Guide, Volume 2*
XNFPrep	*XACT Reference Guide, Volume 2*
XNFMAP	*XACT Reference Guide, Volume 2*
MAP2LCA	*XACT Reference Guide, Volume 2*
APR	*XACT Reference Guide, Volume 2*
PPR	*XACT Reference Guide, Volume 2*
MakeBits	*XACT Reference Guide, Volume 2*
XDE	*XACT Reference Guide, Volume 3*

XNF Translation

The XNF file provides a common interface between design entry tools and the Xilinx development software. It is an ASCII text netlist format containing the logic and connectivity information for the design. It also supports timing information for the design.

Each entry tool has a specific type of output. A translator must be used to convert the entry tool output into an XNF file. You can also translate an XNF file that contains the timing information into a format that can be used by your simulator. The software documentation for each entry tool supported by Xilinx details how to translate to and from an XNF file.

Optimization

Optimization converts device independent or behavioral logic descriptions into a form that can be best implemented in a Xilinx FPGA. This procedure is performed by programs like Xilinx ABEL and X-BLOX. An example of design optimization is the encoding of symbolic state machines using Xilinx ABEL. The following sections discuss the three types of encoding.

Binary Encoding

Binary encoding uses the minimum number of registers to encode a state machine. It is also called maximal encoding, since the registers are used to their maximum capability. Each register represents one bit of a binary number. Although binary encoding keeps the number of registers to a minimum, more combinational logic is required to decode each state.

Binary encoding can be used with Xilinx FPGAs or EPLDs; it is well-suited to EPLD devices, which have wide gates and few registers.

One-Hot Encoding

One-hot encoding takes an approach that is opposite to that of binary 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.

This type of encoding can significantly reduce the amount of combinational logic used to implement a state machine. Highly encoded designs tend to require many high fan-in logic functions to interpret the inputs. One-hot encoding uses a simpler interpretation process, since each state has its own register, or flip-flop. As a result, the state machine is already "decoded," and the state of the machine is determined simply by finding out which flip-flop is active. This process reduces the width of the combinational logic, so the state machine requires fewer levels of logic between registers, thereby reducing its complexity and increasing its speed.

Standard Encoding

Standard encoding incorporates features of both binary encoding and one-hot encoding. It forms clusters of states and uses binary encoding for each cluster. One-hot encoding is a special case of standard encoding in which each cluster contains exactly one state. Binary encoding is a special case in which all states belong to a single cluster.

Standard encoding can be used with FPGAs only.

Merging

Design hierarchy and multiple design entry sources result in multiple files that XMake automatically merges into a single XNF file before proceeding with the subsequent implementation steps. XMake uses the XNFMerge program to perform this task.

Mapping

Mapping, also known as partitioning, is the process of breaking a design into small parts that correspond to CLBs, IOBs, or other Xilinx FPGA resources. This process is represented in Figure 3-1. Because the available resources differ among different Xilinx FPGA families, the mapping program chooses different options, depending upon which device is used.

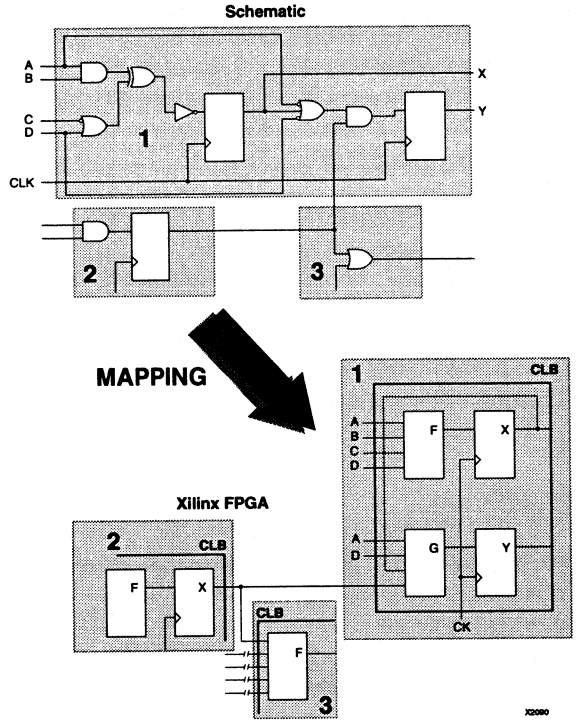


Figure 3-1 Logic Being Mapped into CLBs

Placement

XMake uses the automated placement software programs APR or PPR to place the mapped CLBs and IOBs in optimal positions inside the device, so a minimum amount of routing is necessary to connect them.

The XACT Development System does not require constraints in order to achieve optimum placement. In fact, constraints such as manually assigned pins often limit efficient logic placement. Constraints can create significant restrictions on the placement algorithm and should be avoided.

Routing

XMake automatically routes the design using either APR or PPR. The routing algorithms are designed to determine the interconnect paths that lead to the minimum delays.

Generating a Bitstream

As a final step, XMake uses the MakeBits program to automatically generate a bitstream for the Xilinx FPGA. MakeBits generates specific FPGA configuration data in a BIT file.

Before MakeBits generates the final bitstream for production, you should run MakeBits with the -d option to create a Design Rule Checker (DRC) report. Then you should run MakeBits using the Tie operation to define valid logic levels for Xilinx FPGA resources not specifically used in the design. For more information on MakeBits, MakeBits options, and configuration modes, consult "The MakeBits Program" in the *XACT Reference Guide, Volume 2.*

XACT Design Editor (XDE)

Since XMake can perform all steps for design implementation automatically, manual implementation using XDE is usually unnecessary. However, you can use XDE to get a closer look at how XMake implemented your design. The "Design Verification" chapter in this guide contains several examples of ways to look at timing information. You can find additional information on XDE in the *XACT Reference Guide, Volume 3.*

Design Size and Performance

Information about design size and performance can help you to optimize your design. When you place and route your complete design, the resulting report files list the number of CLBs, IOBs and other device resources used.

If you want to determine the design size and performance without running automatic implementation software, you can quickly obtain an estimate from a rough calculation based on the Xilinx FPGA architecture. See *The Programmable Logic Data Book* for more information on all Xilinx FPGA architectures.

Estimating Design Size

You can obtain a reasonably accurate size estimate of your design simply from the number of CLBs and IOBs your design uses. To estimate the number of CLBs, count the number of registers. You can improve your estimate by comparing blocks in your design with macro library components. You can obtain the number of IOBs by determining the amount of I/O your design uses and adding extra if you plan to bring out test probes.

Synchronous Design

The Xilinx FPGA architecture is best suited for synchronous design. Strict synchronous design ensures that all registers are driven from the same time base with no clock skew. This section outlines several tips for producing high-performance synchronous designs.

Global Clock Distribution

Xilinx clock networks guarantee extremely small clock skew values. Table 3-3 lists the global clock resources for XC2000, XC3000, and XC4000 families.

Table 3-3 Global Clock Resources

FPGA Family	Resource	Number	Destination Pins
XC2000 XC2000L	ACLK GCLK	1 1	Clock or Control Clock or Control
XC3000 XC3000A XC3000L XC3100 XC3100A	ACLK GCLK	1 1	Clock Clock
XC4000 XC4000A XC4000H	BUFGP BUFGS	4 4	Clock Clock or Control

Note: XC4000 BUFGP and BUFGS also connect to control pin and logic inputs; however, limited routing resources can add extra delay on these loads.

Other Synchronous Design Considerations

Other considerations for achieving a synchronous design include the following:

- Use clock enables instead of gated clocks to control the latching of data into registers. See Figure 3-2 and Figure 3-3.
- If your design has more clocks than the number of global clock distribution networks, try to redesign to minimize the number of clocks. Otherwise, put the clocks that have the lowest fan-out onto normally routed nets and give them a high net weighting. Remember that a clock net routed through a normal net has skew.

Data Feedback and Clock Enable

The circuit in Figure 3-2 shows a gated clock. The timing diagram indicates that this implementation can lead to clock glitches that can cause the flip-flop to clock at the wrong time.

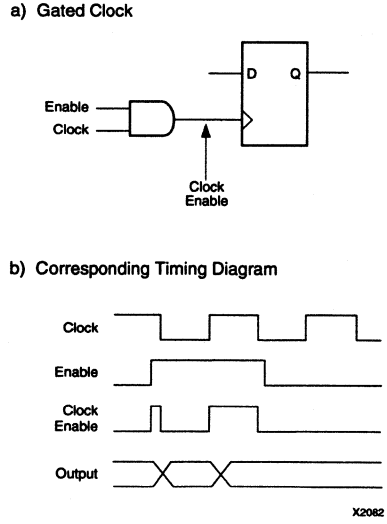


Figure 3-2 Gated Clock

Figure 3-3 shows a synchronous alternative to the gated clock using a data path. The flip-flop is clocked at every clock cycle and the data path is controlled by an enable. When the enable is Low, the

multiplexer feeds the output of the register back on itself. When the enable is High, new data is fed to the flip-flop and the register changes its state. This circuit guarantees a minimum clock pulse width and it does not add skew to the clock. As illustrated in Figure 3-3, the XC3000 or XC4000 flip-flop has a built-in clock-enable (CE).

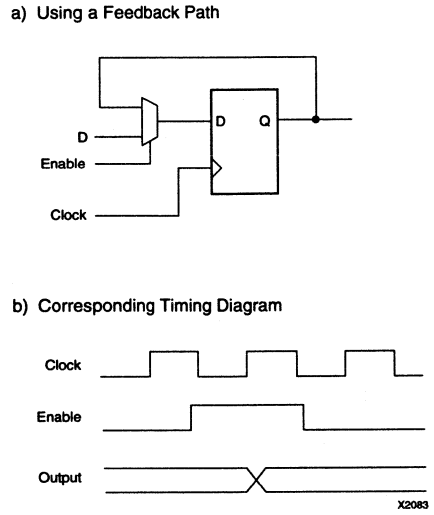


Figure 3-3 Synchronous Design Using Data Feedback

Counters

Cascading several small counters to create a larger counter is similar to a gated clock. For example, if two 8-bit counters are connected, the TC of the first counter is a large AND function gating the second clock input. Using the CE input, you can create a synchronous design as shown in Figure 3-4. In this case, the TC (terminal counter) of the first stage is connected directly to the CE of the second stage.

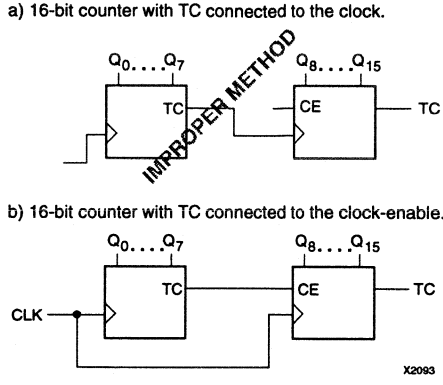
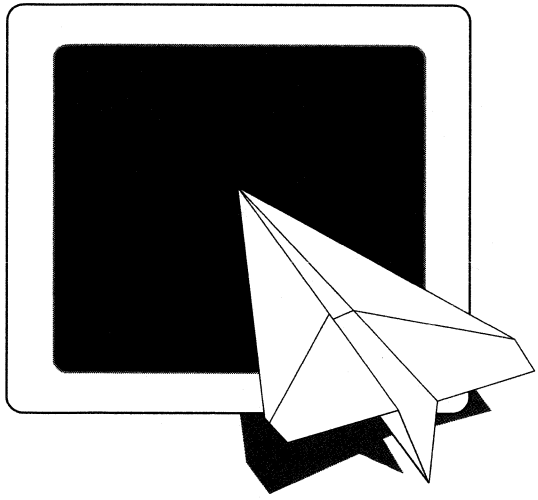


Figure 3-4 Two 8-Bit Counters Connected to Create a 16-Bit Counter



Design Verification

XACT User Guide

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Chapter 4

Design Verification

This chapter introduces design verification, which is the process of testing the functionality and performance of your design. The XACT Development System provides three complementary tools for design verification: simulation, static timing analysis, and in-circuit verification.

Verification Design Flow

Design verification occurs throughout the design process, as illustrated in Figure 4-1. You can verify Xilinx FPGA designs in three different ways:

- Simulation
- Static timing analysis
- In-circuit verification.

This section focuses on these three design verification methods.

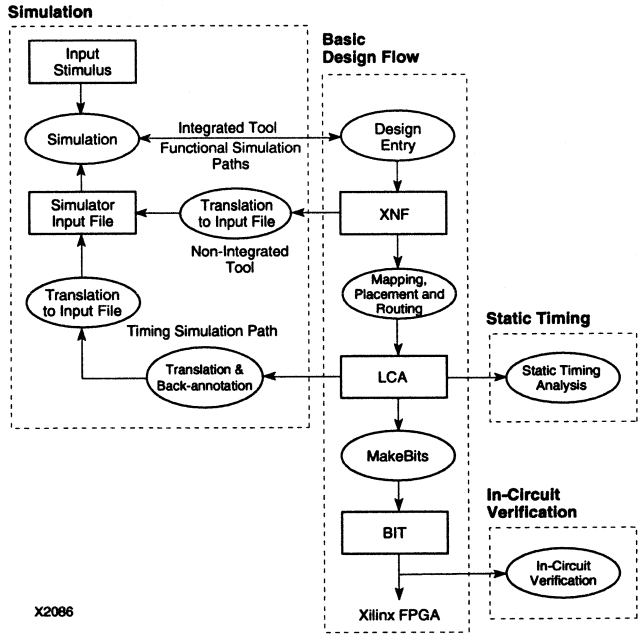
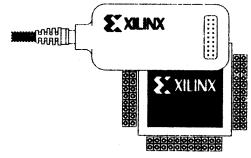


Figure 4-1 The Three Verification Methods in the Design Flow

Figure 4-2 summarizes the tools used for simulation, static timing analysis, and in-circuit verification.

Verification Tools

Type of Verifcation	Tools
Simulation 	• Third Party Simulators (Integrated and Non-Integrated)
Static Timing 	• QueryNet • XDelay
In-Circuit Verification 	• Design Rule Checker • Download or XChecker Cable

X2087

Figure 4-2 Verification Tools

Simulation

Design simulation involves testing your design using software models. It is most effective when testing the functionality of your design and its performance under worst-case conditions. You can easily probe internal nodes with test vectors or other input stimuli and use these results to make changes in your schematic.

You perform simulation using third-party tools that are linked to the XACT Development System. Use the various CAE-specific interface user guides, which cover the specific commands and features of the simulators supported by Xilinx, as your primary reference.

The software models provided by simulation tools are designed to perform detailed characterization of your design.You can perform functional or timing simulation.

The following sections discuss functional and timing simulation.

Functional Simulation

Functional simulation determines if the logic in your design is correct before you implement it in a device.

As shown in Figure 4-1, functional simulation can take place at the earliest stages of the design flow. Since timing information for the implemented design is not available, the simulator tests the logic in the design using unit delays. It is usually faster and easier to correct design errors if you perform functional simulation early in the design flow.

Figure 4-1 shows the design flows for integrated and non-integrated simulation tools. Integrated tools combine the simulator with a schematic editor, so that no translation is needed between them. You can move directly from entry to simulation using an integrated tool. Non-integrated tools do not combine the simulator and schematic editor. Translation programs use your XNF file to create a simulator input file, so you can simulate with a non-integrated tool.

Timing Simulation

Timing simulation verifies that your design runs at the desired speed in your device under worst-case conditions. It is performed after your design is mapped, placed, and routed, when all design delays are known.

Figure 4-1 outlines how to perform timing simulation once delay information is available after placement and routing. Timing simulation is valuable because it can verify timing relationships and determine the critical paths for the design under worst-case conditions. It can also determine whether or not the design contains set-up or hold violations.

To input timing information into your simulator, you must convert the routed LCA file for your design into an XNF file using LCA2XNF (for XC4000 designs you must run MakeBits with the -w option on your LCA file before you run LCA2XNF). The back-annotation tool, XNFBA, combines the information in this XNF file with the net names contained in the original design XNF file. The resulting XNF file can then be translated into an input format suitable for the simulator.

Note: Naming the nets in your design is very important for both functional and timing simulation.

Static Timing Analysis

By using XDelay, you can quickly check for timing problems in your design. XDelay provides access to signal path delay information for all paths in your design.

You can use XDelay to determine path delays in your design. Static timing analysis is best for quick timing checks of a design after placement and routing is complete.

As illustrated in Figure 4-3, you can perform static timing analysis using the XDelay program and the QueryNet program in XDE. The remainder of this section discusses these tools and the timing information they provide.

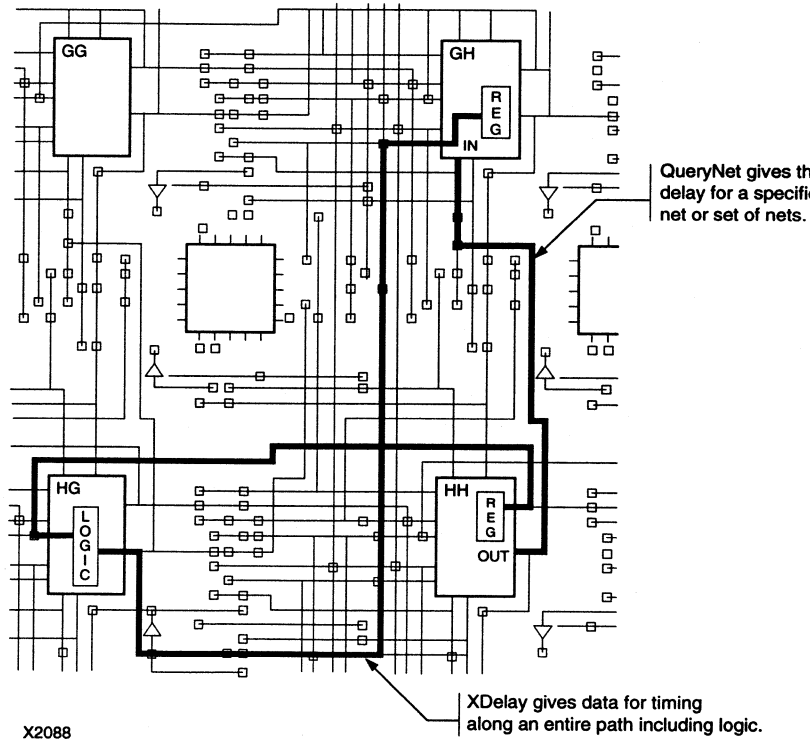


Figure 4-3 XDelay and QueryNet

XDelay

XDelay is a static timing analysis tool, which is also available from the XDE Timing menu and the XACT Design Manager interface (XDM). XDelay can give an overall analysis of design performance. For example, XDelay can calculate the maximum clock frequencies for each clock in the design. A sample from an XDelay output file is shown in Figure 4-4.

```
XDelay: sample.lca (3020PC68-70), xact 4.29b, Mon Aug 31
17:51:04 1992

Xdelay path report options:

From all.
To all.
Output will be sorted by decreasing path delays.

Logical Path                        Delay            Cumulative
------------                        -----            ----------
From: Blk CQL              PAD to P14.I    :     6.0ns ( 6.0ns)
Thru: Net GOSCSET               to BA.C    :     2.6ns ( 8.6ns)
Thru: Blk GOSC                  to BA.X    :     8.0ns (16.6ns)
Thru: Net CLKIN                 to BB.K    :    11.6ns (28.2ns)
  To: Clock Input, Blk J01                 :     0.0ns (28.2ns)
Target FFY drives output net "J1"

From: Blk CQ               PAD to P12.I    :     6.0ns ( 6.0ns)
Thru: Net GOSCRESET             to BA.B    :     2.5ns ( 8.5ns)
Thru: Blk GOSC                  to BA.X    :     8.0ns (16.5ns)
Thru: Net CLKIN                 to CB.K    :    11.6ns (28.1ns)
  To: Clock Input, Blk J23                 :     0.0ns (28.1ns)
Target FFY drives output net "J3"

From: Blk CQ               PAD to P12.I    :     6.0ns ( 6.0ns)
Thru: Net GOSCRESET             to BA.B    :     2.5ns ( 8.5ns)
Thru: Blk GOSC                  to BA.X    :     8.0ns (16.5ns)
Thru: Net CLKIN                 to BB.K    :    11.6ns (28.1ns)
  To: Clock Input, Blk J01                 :     0.0ns (28.1ns)
Target FFY drives output net "J1"
```

Figure 4-4 Sample Output from XDelay

XDelay is best suited for examining delay paths. There are four types of signal paths: pad-to-pad, pad-to-setup, clock-to-setup (register transfer), and clock-to-pad. Each signal path contains block and routing delays. You can further specify paths that you want to verify using options like "from" and "to." Figure 4-3 illustrates a register transfer path from a register output in the HH CLB, through logic in the HG CLB, to a register input in the GH CLB.

XDelay can import the timing information from the XACT-Performance tool and use it to analyze your design.

QueryNet

Figure 4-3 shows a net that has been examined using QueryNet. You can use QueryNet to obtain the delay for a single net or specify a number of nets that you wish to probe. QueryNet enables you to quickly determine if there are nets with large delays or skew. A sample from a QueryNet output file is shown in Figure 4-5.

```
Querynet: demo3020.lca (3020PC68-70), xact 4.29b, Mon Aug 31
17:50:10 1992

---- CLKIN. . . . .     BA.X (GOSC). . .   6.3 BA.A (GOSC)
                                           11.6 BB.K (J01)
                                           11.6 CB.K (J23)
                                           11.6 DB.K (J4)
                                            6.6 P12.O (CQ)
                                            7.6 P12.T (CQ)
---- GOSCQL . . . .     BA.Y (GOSC). . .   4.6 P14.O (CQL)
                                            4.6 P14.T (CQL)
---- GOSCRESET. . .     P12.I (CQ) . . .   2.5 BA.B (GOSC)
---- GOSCSET. . . .     P14.I (CQL). . .   2.6 BA.C (GOSC)
---- J1 . . . . . .     BB.Y (J01) . . .   1.0 CB.A (J23)
---- J3 . . . . . .     CB.Y (J23) . . .   1.0 DB.A (J4)
---- J4 . . . . . .     DB.X (J4). . . .   9.7 BB.A (J01)
                                            9.6 HB.A (FLASH)
---- OUT7 . . . . .     HB.X (FLASH) . .   7.0 P28.O (O7A)
                                            6.3 P29.O (O7B)
```

Figure 4-5 Sample Output from QueryNet

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 FPGAs repeatedly, you can easily load your design into your device and test it in-circuit. To verify your design in-circuit, download your design bitstream into a device with the Xilinx download cable or the XChecker cable. You can also use the XChecker cable to probe your design after you download it.

You can also probe the internal nodes of your downloaded design using the XChecker cable. Probing makes it easier for you to pinpoint the location of any design problems. Refer to the "XDE" chapter in the *XACT Reference Guide, Volume 3* for more information.

Design Rule Checker

Before generating the final bitstream, it is important to use the DRC option in MakeBits to evaluate the LCA file for problems that prevent the design from functioning properly. You can directly invoke DRC under MakeBits in the XACT Design Manager (XDM) or inside XDE.

Xilinx Download and XChecker Cables

Xilinx provides either the Download or XChecker cable depending on which development system you are using. To download your design, you must create a configuration bitstream using MakeBits.

To read back and verify configuration data, you must use the XChecker cable. Refer to the "XChecker" chapter in the *XACT Hardware and Peripherals Guide* if you need more information.

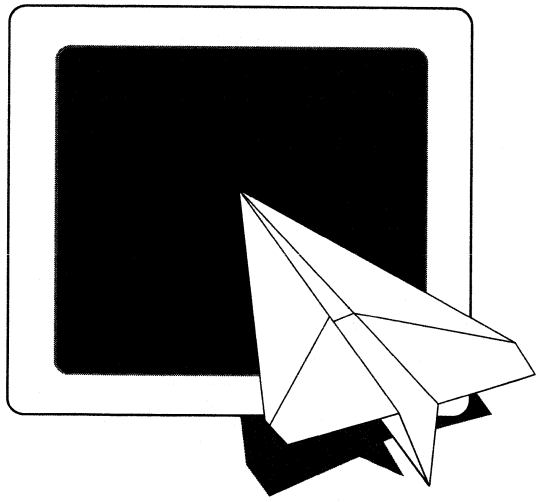
With the XChecker cable, you can use the XChecker Pick function to take snapshots of the circuit at specific clock cycles. You can obtain these snapshots by performing serial readback of the nodes during in-circuit operation. With XChecker, you can increase the speed of your analysis by limiting the readback bitstream to only those nodes and clock cycles in which you have interest.

Use the XChecker cable when you do not want to allocate IOBs and routing resources on your Xilinx FPGA for probing. As a result, you

can decide how you want to probe after you have downloaded your design.

Probe

Using the Probe command in XDE, you can connect internal nodes in your design to IOBs for analysis. With this capability, you can use an oscilloscope or a logic analyzer to perform real-time analysis of nodes that would not normally be accessible. To use Probe, you must carefully allocate sufficient IOB and routing resources in your Xilinx FPGA. Probe is a very flexible tool, since you can change the probe location based on the resources you allocate. Refer to the XDE chapter in the *XACT Reference Guide, Volume 3* for more information.



XACT User Guide

FPGA Design Implementation Flows

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Chapter 5

FPGA Design Implementation Flows

This section provides an overview of the Xilinx Field Programmable Gate Array (FPGA) design process and the XACT™ Development System software. For a more extensive coverage of specific features and development system commands, refer to the appropriate volume of the *XACT Reference Guide.*

The XACT Development System software consists of several packages that provide integrated design entry, implementation, and verification capability.

Using the XACT Design Manager (XDM) program, a graphical menu interface, you can select and run any Xilinx program from one menu. The XMake program in XDM automatically translates designs. For more information about design entry, design implementation, and design verification, refer to the appropriate chapters in this user guide.

This chapters covers automatic design implementation for the following families:

- XC2000, XC2000L, XC3000, and XC3100
- XC3000A, XC3000L, and XC3100A
- XC4000, XC4000A, and XC4000H

XC2000, XC2000L, XC3000, and XC3100 Families

This section introduces some of the programs XMake runs when performing design implementation of XC2000, XC2000L, XC3000, and XC3100 devices. The following figure illustrates the design implementation flow.

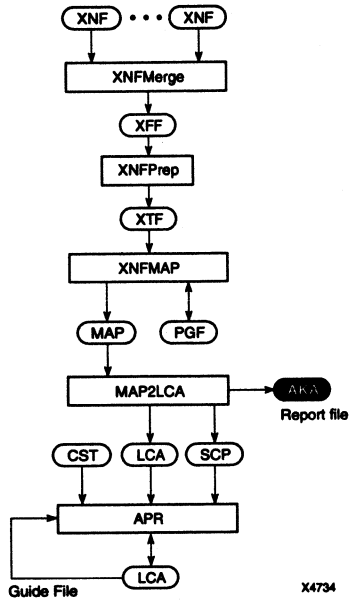


Figure 5-1 XC2000/XC2000L/XC3000/XC3100 Flow

Logic Reduction and Partitioning

As part of the automatic design-translation process, XMake activates the logic reduction and partitioning programs, XNFMAP and MAP2LCA, and the place and route program, APR. XMake uses a utility called XNFMerge to combine various XNF files into a single XFF file. For detailed information about each program, refer to the *XACT Reference Guide, Volume 2.*

The XNFMerge program merges lower-level XNF files into a top-level XNF file and writes the result to an output file. After XNFMerge produces a single, flattened XFF file for the design, the design is processed through XNFPrep. XNFPrep analyzes the design and reports all errors and warnings. If no errors are found, XNFPrep performs; logic reduction, which consists of removing sourceless and loadless nets, and optimizing the logic for the specific architecture. The XNFPrep report file has a PRP extension and contains explanations of the errors and warnings, as well as a detailed description of why logic was removed.

XNFMAP partitions the logic in the XNF file into the CLBs and IOBs that comprise the FPGA architecture then writes the results to a MAP file. The MAP2LCA program converts this MAP file into an LCA file.

Automatic Place and Route (APR)

The Automatic Placement and Routing program uses an LCA design file (*name*.lca) to automatically arrange the CLBs and IOBs; optimize the block-pin net assignments; and determine interconnection patterns to route the design and write the resulting design file to disk. The APR input file can result from XACT-EditLCA or a MAP2LCA conversion. The program uses the input design file (*name*.lca), a file of constraints produced from parameters assigned during schematic capture (*name*.scp), and an optional text file of constraints you produced (*name*.cst).

The primary results are a new design file (*output*.lca) and a report file of routing results, routing order, flag tables, delay table, and placement results. Refer to the *XACT Reference Guide, Volume 2* for complete descriptions of APR commands and options.

APRLoop

The APRLoop command runs iterations of the APR program and saves the results of each run. You can run APRLoop overnight and evaluate several resulting placements in terms of areas of congestion, routing delays, number of unrouted pins. See the APR chapter in the *XACT Reference Guide, Volume 2* for more information.

You can edit the best APRLoop result with XDE by moving blocks, using longlines, and routing critical nets. Try running APR on the best result with blocks locked (-l) and higher net weights on unrouted nets.

XC3000A, XC3000L, and XC3100A Families

For XC3000A, XC3000L, and XC3100A parts, the first part of the design flow is similar to that discussed in the previous section — designs are processed using XNFMerge and XNFPrep. However, the design flows are different from this point on, as illustrated by the following figure.

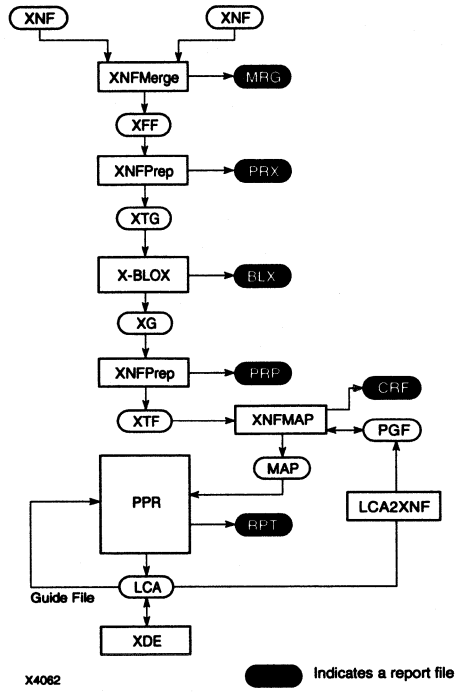


Figure 5-2 XC3000A/XC3000L/XC3100A Design Flow

If XMake detects the use of X-BLOX components in the schematic, it runs the X-BLOX program to synthesize the logic for these functions. XMake runs XNFPrep again to generate the XTF file.

The Partition, Place, and Route program, PPR, uses the MAP file as input and reads the constraints from the CST file, if it exists in the current directory. PPR creates two output files — the design file with an LCA extension and a summary report file with an RPT extension.

XC4000, XC4000A, and XC4000H Families

The design flow for the XC4000, XC4000A, and XC4000H parts starts with XNFMerge, which combines all XNF files associated with the design into one XFF file. The following figure illustrates the design implementation flow.

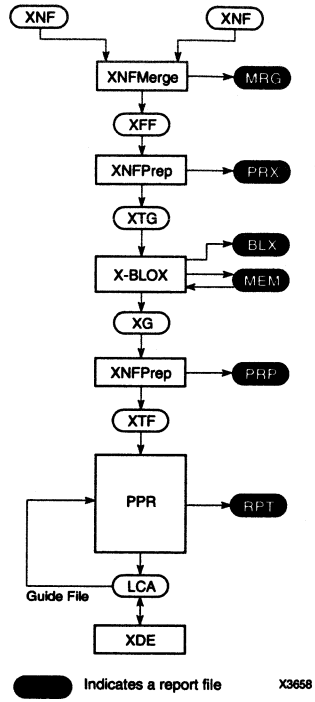
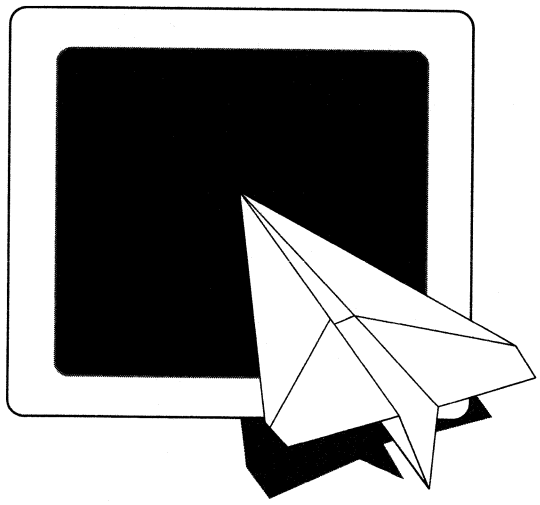


Figure 5-3 XC4000/XC4000A/XC4000H Design Flow

Next, XMake runs XNFPrep, which analyzes the design and reports all errors and warnings. If no errors are found, XNFPrep performs logic reduction, which consists of removing sourceless and loadless nets, and optimizing the logic for the specific architecture. XNFPrep creates a report file with a PRP extension or a PRX extension, if X-BLOX is used. The report file contains explanations of error and warning messages, as well as a detailed description of why logic was removed. XNFPrep outputs an XTF file or an XTG file, if X-BLOX is used.

If XMake detects the use of X-BLOX modules, it runs the X-BLOX program. X-BLOX outputs a design file with an XTG extension and a report file with an RPT extension.



XACT User Guide

Configuration, Length Count, and Debugging

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Chapter 6

Configuration, Length Count, and Debugging

This chapter provides solutions to problems you might encounter during the downloading process, such as configuring the device and verifying the length count.

The Xilinx Development System enables you to enter, place, route, and verify Field Programmable Gate Array (FPGA) designs. Then you can use the MakeBits and MakePROM programs to create configuration files. The resulting configuration data is loaded into the FPGA configuration memory cells, which control Configurable Logic Block (CLB) and I/O Block (IOB) look-up functions, multiplexers, and interconnections.

Configuration

Configuration is the process in which you download a bitstream to your device or daisy chain of devices. This section covers the following topics:

- Data generation
- Data format
- Modes
- Loading and framing configuration data
- States for XC2000/XC3000 devices
- States for XC4000 devices

Data Generation

Once an LCA file has been created for a design, you can use MakeBits and MakePROM from within the XACT Design Editor (XDE) to

generate configuration data files. These files can then be used to configure the device.

MakeBits

The MakeBits program, run from within XDE, uses the LCA file to generate the specific FPGA configuration data. In MakeBits, you can use the Configure command of the Configure menu to determine the device options. These options can also be set directly with the mouse in the Program Options table at the bottom left corner of the screen. The MakeBits profile provides a default set of device options. You can revise these defaults by performing a Saveprofile with the desired options selected. For future use, the saved makebits.pro file can be read into MakeBits.

Executing the MakeBits command in the Configure menu with the desired device options selected generates a bitstream file (BIT) for the currently loaded design. MakeBits first prompts with the options: Tie, Norestore, Verbose, UseCriticalNetsLast, IgnoreCriticalNetFlags, and MakeLL. If none of these options are selected, selecting the Done button generates an untied bitstream in memory that can be downloaded directly to the device using the Download option from the Download menu. You can use an "untied" version for a quick logic check of a room-temperature breadboard where the increased supply current is not critical.

The following sections discuss two steps in the MakeBits process: tying the design and creating the bitfiles.

Tying the design — production designs require the use of Tie to define valid logic levels for FPGA resources that are not used in the design. The unused interconnects produce floating CMOS gate input levels that can result in simultaneous p- and n-channel conduction, causing additional supply current, heat, and electrical noise. The use of the Tie option defines all outputs of unused CLBs and IOBs as logic Lows and uses them to drive accessible unused routing resources.

The remaining unused resources are then added to nets that were not flagged as critical with the Flagnet command in the Net menu of EditLCA. This might result in timing changes relative to the original LCA design file. The additional selection of the Norestore option retains a temporary version of the tied design in memory, allowing the use of the Querynet command with the Tiechange option to

evaluate the effects of the Tie operation on timing. To protect the original design file, the tied version of the design is not saved on your disk unless you specifically request it.

In some cases, the design might not tie if there are too many critical nets. At this time, you can use the UseCriticalNetsLast option. This option gives MakeBits the option to use the critical nets in the design, but only as a last resort. If MakeBits is still unable to tie the design, you can use the IgnoreCriticalNetFlags option. This option tells MakeBits to ignore the critical flags found in the design.

If you used either the UseCriticalNetsLast or IgnoreCriticalNetFlags options, Xilinx strongly recommends that you run the XDelay program on the design to ensure the timing of the critical nets in the design.

Unused and unbonded I/O blocks must also have defined pad voltages. In the XC3000 and XC4000 families, this is automatically accomplished with the internal pull-up resistor as the default for unconfigured IOBs. For the XC2000 family, the pad level can be defined with an external signal, a pull-up resistor, or the output buffer can be enabled and driven by an internal signal.

Creating the bitfiles — Writebits saves the bitstream to a file. This binary format (BIT) file can be used by the Download cable or the XChecker cable to configure an in-circuit FPGA. A user-readable ASCII file of the configuration data can also be generated by using the Rawbits command found in the Config menu. This file is useful for verifying the actual bit order loaded into an FPGA.

MakePROM

The MakePROM program translates BIT files into PROM programmer compatible data files. You can then use these files to program PROMs that provide the configuration data used to define an FPGA operation.

MakePROM also appends BIT files and produces a composite PROM file that contains the total length count and configuration data for a chain of devices. This chain is called a daisy chain. In a daisy chain, multiple devices are configured in a serial fashion from the same PROM file. The lead device is configured first, with each succeeding device programmed in the order that it is arranged on the board.

The MakePROM default options are determined by the MakePROM profile. If the default Promsize setting is not correct, use the Set command on the Misc menu. The Format command of the Prom menu selects the desired PROM-programmer format. The formats written by MakePROM are Mcs86 (Intel), Exormax (Motorola), Tekhex (Tektronix). Most PROM programmers support one or more of these formats.

The following two sections describe the most common function.

Loading — once you set the format and size options, you can load the BIT files into the system memory in the order desired by using the Load command found on the Prom menu. The Load command prompts for a hexadecimal PROM starting address and direction of address sequence (up or down). The Load command then displays the BIT files of the current directory. To configure multiple FPGAs, select multiple BIT files in order — lead device first. When configuring in parallel multiple devices having the same design, you only have to select the BIT file once.

You can also generate PROM files containing alternative configurations with individual starting addresses, so that you can control the high-order PROM addresses and select different configurations from different segments within the same PROM.

Saving — use the Save command to produce an ASCII file that can be transferred to a conventional PROM programmer. This hex-character file contains header, address, PROM data, and checksum information. The header information identifies record type, record size, and the address of the first data byte of that record. The PROM data includes the FPGA configuration data.

Data Format

Two common types of bitstreams are single device streams and daisy chain streams. In both cases, the data bitstream consists of header and program data.

The header contains the start sequence bits and the length count that specifies the number of configuration data bits needed to configure the device or devices. The program data contains the information needed to configure each frame of the device. The program data also includes a tail-bit sequence that ensures the last data frame is loaded. The bitstream is padded so it ends on a byte boundary.

Creating Bitstreams

You must create a bitstream for every FPGA design. The MakeBits program generates a properly formatted file. If devices are daisy chained, MakePROM creates a PROM hex file for the chain. Specify the bitstream (the BIT file) for the first device in the chain, then the bitstream for the second device, and so on.

Single Device Streams

The 40-bit header of a single device bitstream begins with a minimum of eight ones (dummy bits) followed by a 0010 preamble. A 24-bit length count follows the preamble. For more information see the "Length Count" section at the back of this chapter. The header ends with a 4-bit separator field of ones.

The program data of a single device bitstream contains configuration data frames, tail bits, and pad bits. The number of frames and bits that make up the frames varies from part to part; however, the basic breakdown is very similar.

The configuration data frames are broken up into a start bit, configuration bits, and stop bits. Each frame begins with a zero start bit followed by the actual configuration data bits for the device. XC2000 and XC3000 parts have 3 ones as their stop bits. XC4000 parts have a 4-bit field that is either a default 0110 or a calculated CRC checksum, if selected in the MakeBits options. The CRC check sum is based on the accumulation of all configuration bits in the current device.

For XC2000 and XC3000 devices, the tail bits are 1111. For XC4000 devices, the tail bits are 01111. The difference in the number of tail bits combined with the varying number of frame bits for each device means that the number of byte boundary pad bits vary as well. The complete sequence of end bits is the combination of the tail bits and the pad bits, so its length varies from part to part.

Daisy Chain Streams

Daisy chain bitstreams also contain a single composite 40- bit header. The header is much like the single device header because it begins with a minimum of eight dummy bits followed by a preamble. The header contains a 24-bit length count. The length count, however, is

common to all the devices in the daisy chain. Daisy chain headers also end with a 4-bit separator field.

The program data of a daisy chain bitstream contains each device's program data followed by its tail bits and ending with the pad bits for the entire daisy chain. The configuration data frames for each device appear in the order in which the actual devices are arranged on the board. The lead device is at the beginning of the program data and the last device in the daisy chain is at the end of the program data. Each device's set of data includes the device's tail bits (1111 for XC2000 and XC3000 devices and 01111 XC4000 devices). After the last device's program data and tail bits, the entire stream's pad bits are placed. The stream's pad bits are calculated based on the total number of devices in a daisy chain plus control of the start-up sequence relative to a byte boundary. See the "Length Count" section at the end of the chapter.

PROMs

Typically, after verifying a design through simulation and implementing the breadboard, you can make a final production board. Many boards use a PROM to configure the FPGAs; the serial PROM is the most common.

Once the bitstreams have been created, you can use MakePROM to create the necessary PROM file. There are three formats supported by MakePROM: MCS-86, EXORMAX, and TEKHEX. During the programming of the XC17XXD devices, the polarity of the PROMS RESET/OE must be set. You can accomplish this task by setting the polarity bits in the device to either 0000 HEX for $\overline{\text{RESET}}$/OE or FFFF HEX for RESET/$\overline{\text{OE}}$.

Modes

FPGAs accept configuration data in one of several modes. Logic levels applied to the mode selection pins, M0, M1, and M2, at the start of configuration determine the method to be used. Some of these modes use parallel data while others use serial data. Some modes are active, that is, the device provides its own clock. Others are passive, that is, the device accepts externally generated timing.

Master versus Non-Master

There are two different configuration methods for FPGAs: Master mode and non-Master mode. In Master mode, the FPGA loads itself from an external memory such as a parallel or serial PROM. The FPGA provides clocks and/or addresses to the PROM and receives data that it loads into its internal configuration memory. There are three different Master configuration modes: Master Parallel Up, Master Parallel Down, and Master Serial.

In non-Master mode, the FPGA is loaded by another device such as a microprocessor or another FPGA, which provides the data and the clocks necessary to load the non-Master device's internal configuration memory. There are two non-Master configuration modes for the XC2000 and XC3000 devices: Peripheral and Slave Serial. XC4000 devices support three: Asynchronous Peripheral, Synchronous Peripheral, and Slave Serial.

Note: Wire AND the $\overline{\text{INIT}}$ pin of the slave devices and tie the signal to the $\overline{\text{INIT}}$ pin of the lead device to delay configuration until all devices have cleared their configuration memories. $\overline{\text{INIT}}$ is an input/output on the XC4000. The XC4000 requires an additional clock to complete startup after loading configuration data; therefore, it should be used as a master in the daisy chains of mixed device families.

XC2000/XC3000 Modes

This section describes XC2000/XC3000 modes: Master Parallel Up/Down, Master Serial, and Peripheral Mode. Table 6-1 lists how to adjust the mode pins to set the device configuration mode.

Table 6-1 XC2000/XC3000 Configuration Modes

XC2000 Family					
M0	**M1**	**M2**	**CCLK**	**Mode**	**Data**
0	0	0	output	Master	Bit serial
0	0	1	output	Master	Byte wide addr. = 0000 up
0	1	0	—	reserved	—
0	1	1	output	Master	Byte wide addr. = FFFF down
1	0	0	—	reserved	—
1	0	1	output	Peripheral	Bit serial
1	1	0	—	reserved	—
1	1	1	input	Slave	Bit serial
XC3000 Family					
M0	**M1**	**M2**	**CCLK**	**Mode**	**Data**
0	0	0	output	Master	Bit serial
0	0	1	output	Master	Byte wide addr. = 0000 up
0	1	0	—	reserved	—
0	1	1	output	Master	Byte wide addr. = FFFF down
1	0	0	—	reserved	—
1	0	1	output	Peripheral	Byte wide
1	1	0	—	reserved	—
1	1	1	input	Slave	Bit serial

Master Parallel Up/Down — in Master Parallel mode, the device directly addresses an industry-standard byte-wide EPROM and accepts eight data bits right before the 16-bit address counter is incremented (decremented for Master Parallel High mode).

Figure 6-1 illustrates the Master Parallel Mode for XC2000 devices with daisy-chained Slave Mode devices.

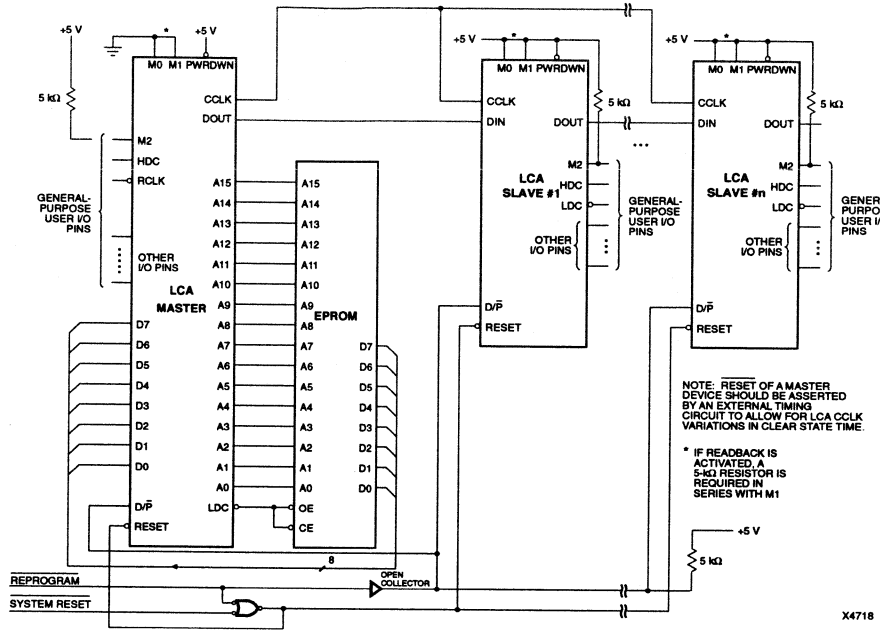


Figure 6-1 Master Parallel Mode for XC2000 Devices

Figure 6-2 illustrates the Master Parallel Mode for XC3000/XC3100 devices.

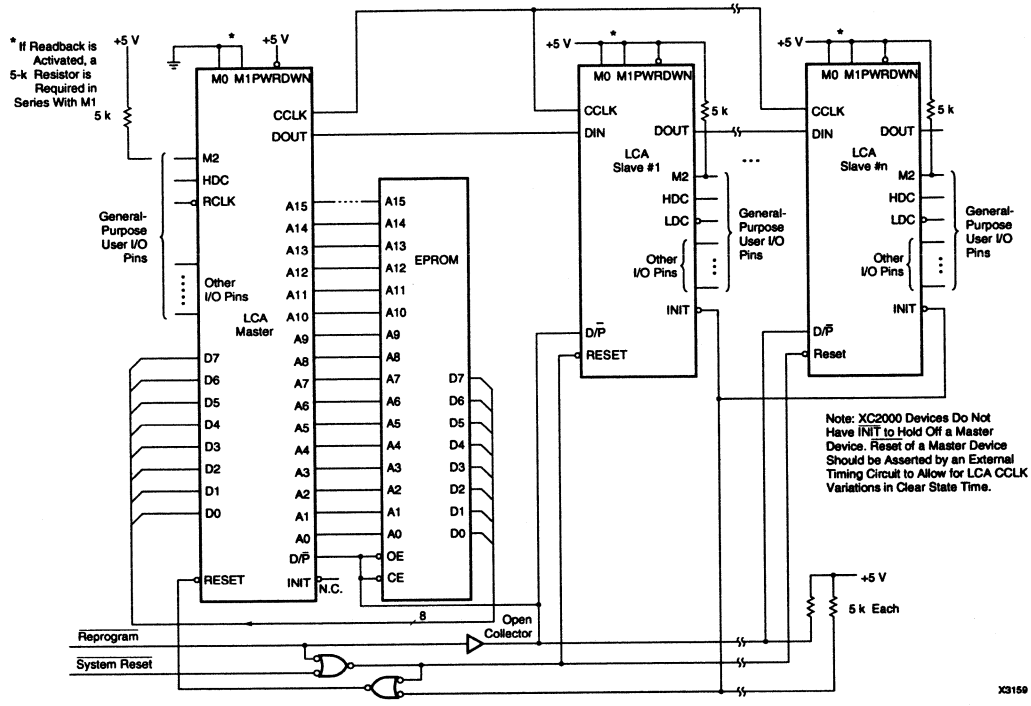


Figure 6-2 Master Parallel Mode for XC3000/XC3100 Devices

The eight data bits are serialized in the device and generate CCLKs that are used internally by the Master device and by any Slave devices connected in parallel or in a daisy chain. The device presents the serialized 40-bit header on the DOUT pin and then applies a High while accepting its configuration data frames.

There is an internal delay of 1.5 CCLK periods, after the rising CCLK edge that accepts a byte of data and changes the EPROM address until the falling CCLK edge that makes the LSB (D0) of this byte appear on DOUT. This means that DOUT changes on the falling CCLK edge, and the next device in a daisy chain accepts the data on the subsequent rising CCLK edge.

Master Serial — in Master Serial mode, the CCLK output drives a serial PROM that feeds the FPGA DIN input. Each rising edge of the CCLK output increments the serial PROM's internal address counter. This positive transition on CCLK puts the next data bit on the PROM's data output connected to the device's DIN pin. The device accepts this data on the subsequent rising CCLK edge.

Figure 6-3 and Figure 6-4 illustrate Master Serial Mode for XC2000/ XC3000 devices.

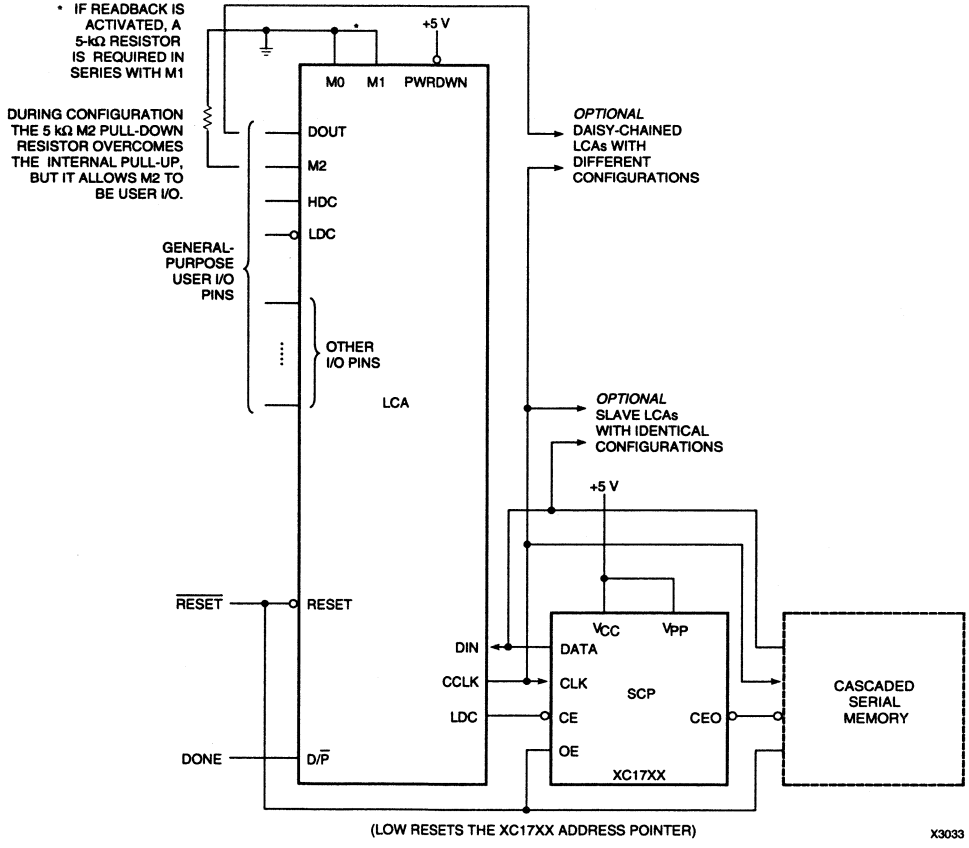


Figure 6-3 Master Serial Mode for XC2000 Devices

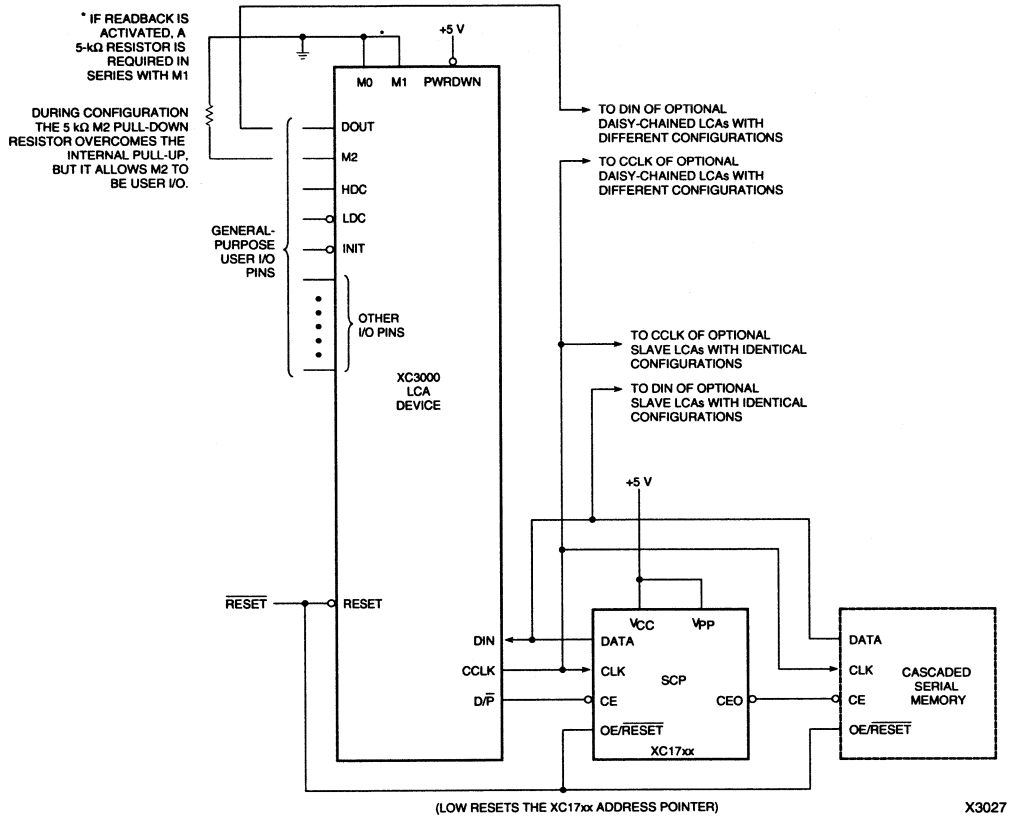


Figure 6-4 Master Serial Mode for XC3000/XC3100 Devices

The Master Serial device presents the 40-bit header (and all data that overflows its configuration frames) on its DOUT pin. As in the Master Parallel modes, there is a 1.5 CCLK internal delay from DIN to DOUT.

The serial PROM CE input can be driven from either $\overline{LDC}$ or DONE. Using $\overline{LDC}$ avoids potential contention on the DIN pin if this PIN is configured as user I/O, but $\overline{LDC}$ is then restricted to be permanently High user output. Using DONE also avoids contention on DIN, provided the early DONE option is selected.

Peripheral Mode — Peripheral mode provides an interface for loading the FPGA as a processor peripheral. Figure 6-5 shows the serial data Peripheral mode connections of the XC2000 devices.

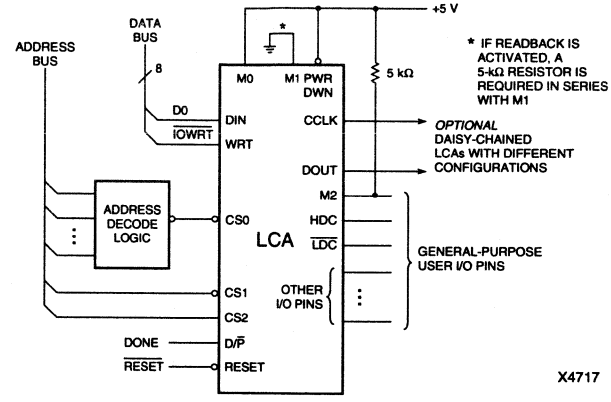


Figure 6-5 Peripheral Mode for XC2000 Devices

Figure 6-6 shows the parallel data Peripheral mode connections of the XC3000 devices.

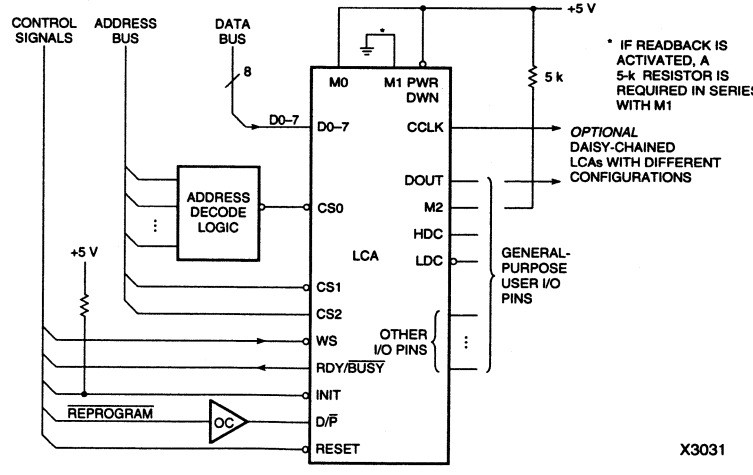


Figure 6-6 Peripheral Mode for XC3000/XC3100 Devices

Processor Write cycles are decoded from the common assertion of the active Low Write Strobe ($\overline{WS}$), and two active Low and one active High Chip Selects ($\overline{CS0}$, $\overline{CS1}$, and CS2). If all these signals are not applicable, the unused inputs must be driven to their active levels.

XC2000 devices accept one serial bit of configuration data on the DIN pin for each selected Write cycle. XC3000 devices accept one parallel byte of configuration data on the D0-D7 inputs for each selected Write cycle.

For the switching characteristics, see *The Programmable Logic Data Book.*

Each byte of XC3000 configuration data is loaded into a register. XC3000 devices generate a configuration clock from the internal timing generator and serialize the parallel input data for internal framing and daisy chained Slave devices on DOUT.

When INIT goes High, the RDY/$\overline{BUSY}$ signal indicates the state of the device. Prior to the INIT pin going High, the RDY/$\overline{BUSY}$ signal is tristated through a pull-up resistor, which gives a false indication that the device is ready to accept data. A High on the XC3000 RDY/$\overline{BUSY}$ output pin indicates loading completion of the previous byte when the input register is ready. You can use a Peripheral-mode device as a lead device for a daisy-chain of Slave devices. It can have configuration inputs connected in parallel with another device for identical configurations.

Note: Additional clocks beyond the last data are required to complete start-up (I/O activation and DONE generation) after length-count compare. In Peripheral mode, the use of the chip-select pins as user outputs can result in signal contention during the last CCLK cycles of configuration as the user I/Os become active.

Slave Mode — this mode provides a simple interface for loading configuration data. This mode uses serial data and an externally provided synchronizing configuration clock (CCLK). Bit-serial configuration data is read at the rising edge of CCLK. Data on DOUT is provided on the falling edge of CCLK. CCLK must not remain Low for more than 5 µs (1 µs on Military B devices). Multiple Slave-mode devices can be connected in parallel if they use identical configurations.

Most Slave-mode applications are daisy-chain configurations in which the data inputs are supplied by the previous device's data

outputs, while the common clock is supplied by a lead device in Master or Peripheral mode. A processor or other special circuit can also supply data and a clock. XC2000 devices require some Slave-mode data hold time, while XC3000 devices do not.

Figure 6-7 and Figure 6-8 illustrate slave mode for XC2000 and XC3000/XC3100 devices.

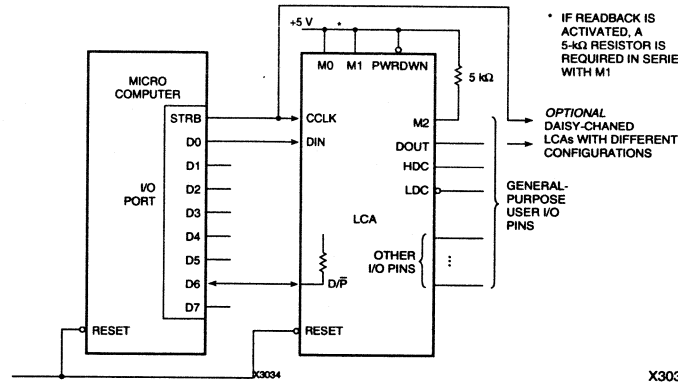


Figure 6-7 Slave Serial Mode for XC2000 Devices

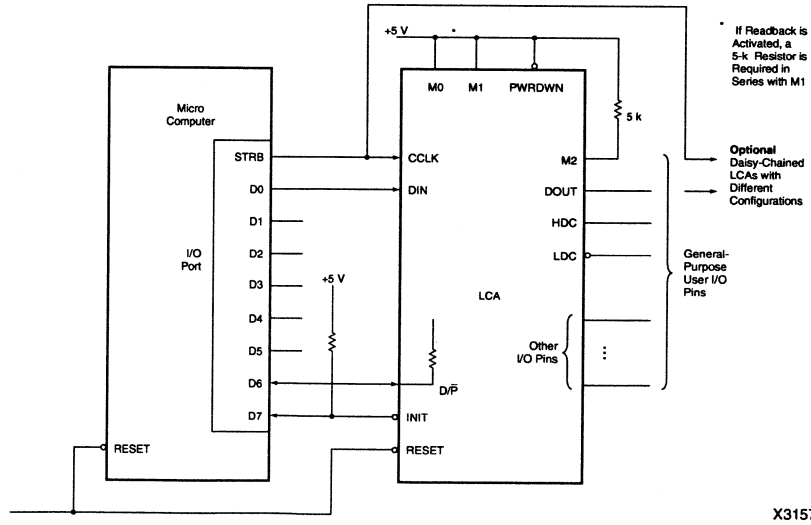


Figure 6-8 Slave Serial Mode for XC3000/XC3100 Devices

XC4000 Modes

This section describes XC4000 modes: Master Parallel Up/Down, Master Serial, Synchronous Peripheral, Asynchronous Peripheral, and Slave Serial. The following table lists how to adjust the mode pins to set the device's configuration mode.

Table 6-2 XC4000 Configuration Modes

XC4000 Family					
Mode	**M2**	**M1**	**M0**	**CCLK**	**Data**
Master Serial	0	0	0	output	Bit serial
Slave Serial	1	1	1	input	Bit serial
Master Parallel Up	1	0	0	output	Byte wide, 0000 up
Master Parallel Down	1	1	0	output	Byte wide, 3FFFF down
Peripheral Synchronous	0	1	1	input	Byte wide
Peripheral Asynchronous	1	0	1	output	Byte wide
Reserved	0	1	0	—	—
Reserved	0	0	1	—	—

Master Parallel Up/Down — Master Parallel modes Up and Down use byte-wide data from a PROM that is supplied to the D0-D7 pins in response to the 18-bit address generated by the FPGA configuration logic. Figure 6-9 illustrates Master Parallel Mode for XC4000 devices.

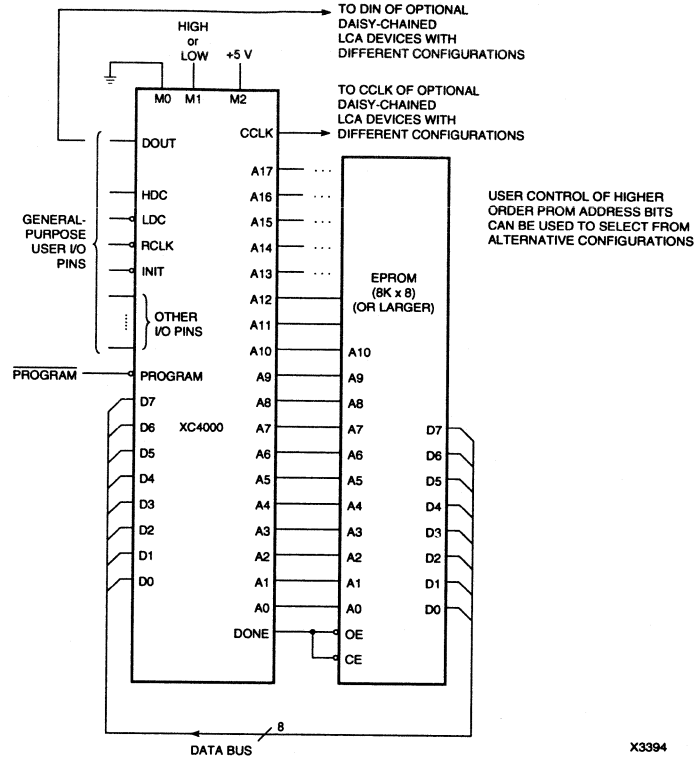


Figure 6-9 Master Parallel Mode for XC4000 Devices

The HEX starting address is 00000 and increments for Master Parallel Up mode, and is 3FFFF and decrements for Master Parallel Down mode. These two modes provide opposite-address compatibility with microprocessors that begin executing from either end of memory. For Master Up or Down, 8-bit data bytes are read by each read clock (RCLK) and internally serialized (least significant bit first) by the configuration clock (CCLK).

For multiple devices in a daisy chain, the Master device interfaces with the parallel PROM and passes configuration data to the other devices serially. The slave devices use the CCLK generated by the lead device to shift in the data through DIN. If multiple Slave Serial devices have identical configurations, their DIN pins can be

connected in parallel. Master devices cannot be connected in parallel due to lack of CCLK synchronization.

Master Serial — this mode uses serial configuration data supplied to the DIN pin from a synchronous serial source such as a serial PROM, as illustrated in Figure 6-10.

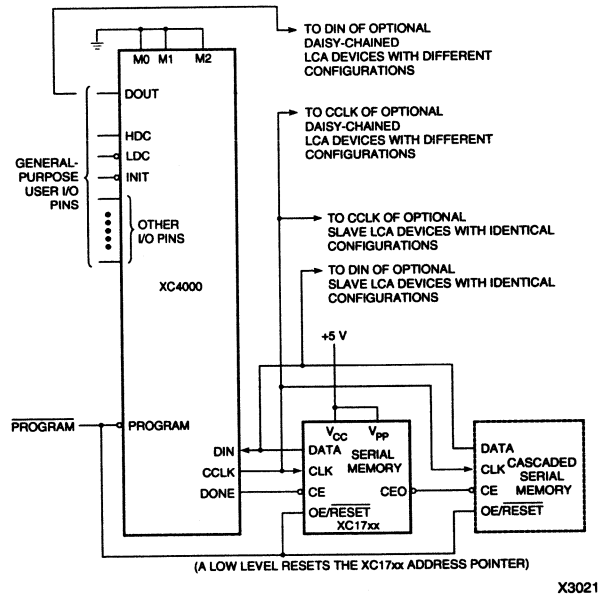


Figure 6-10 Master Serial Mode for XC4000 Devices

The device automatically loads itself from the serial PROM upon power-up or re-configuration. An internal oscillator generates the CCLK that is used to clock in data and also to drive Slave Serial devices. For Master Serial Mode switching characteristics, refer to *The Programmable Logic Data Book.*

Synchronous Peripheral — functionally, the Peripheral Synchronous mode looks like a Parallel Slave mode. The configuration clock is generated externally and drives CCLK. Figure 6-11 illustrates Synchronous Peripheral Mode for XC4000 devices.

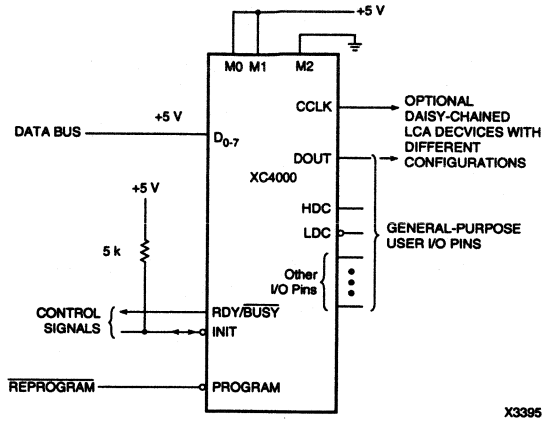


Figure 6-11 Synchronous Peripheral Mode for XC4000 Devices

The data is accepted byte-wide from a processor bus or DMA controller. The second user-generated CCLK after $\overline{\text{INIT}}$ goes High clocks in the first byte of data on D0-D7. The next seven CCLKs internally serialize the data into the configuration frames. The RDY/$\overline{\text{BUSY}}$ signal goes High for one clock cycle after each byte of data is loaded, signaling that the device is ready to accept a new byte. The next byte of data can be placed on the D0-D7 data bus any time between the first and the seventh CCLK after the RDY/$\overline{\text{BUSY}}$ pulse. For Synchronous Peripheral switching characteristics, refer to *The Programmable Logic Data Book.*

Asynchronous Peripheral — in Asynchronous Peripheral mode, the device looks like a peripheral on a processor bus. Data is loaded through byte-wide data input pins, D0-D7. The chip must be selected, CS1=1 and $\overline{\text{CS0}}$=0, before a write to the device or a read of the status flag can occur. For Asynchronous Peripheral switching characteristics, refer to *The Programmable Logic Data Book.*

For the Write cycle, the combination of $\overline{\text{WS}}$=0, $\overline{\text{CS0}}$=0, CS1=1, and $\overline{\text{RS}}$=1 loads the data byte into a data latch for a parallel to serial shift register to load into the internal configuration frames. When the Write condition becomes false, it latches the last data at the D0-D7 pins to the data latch. A High on the RDY/$\overline{\text{BUSY}}$ output pin indicates that the data latch is empty and that another Write cycle can be

issued. The next byte of data can be presented to D0-D7 after each Write cycle after some hold time.

For the Read cycle, $\overline{RS}$=0, $\overline{CS0}$=0, CS1=1, and $\overline{WS}$=1, the RDY/$\overline{BUSY}$ status flag is available on D7 so that you can read from the same bus you write to.

Note: In Peripheral mode, use of the chip select pins as user outputs can result in signal contention in the last cycle of configuration as the user I/O becomes active.

Slave Serial — Slave Serial devices accept serial data with an externally supplied configuration clock. Multiple Slave Serial devices can be connected in parallel if they use identical configurations. Most Slave Serial applications are daisy chain configurations in which the data is supplied by the previous devices DOUT, while the common clock is supplied by a Master or Peripheral mode device at the head of the chain. A processor or other logic can also supply data.

The XC2000 devices require some Slave Serial mode data hold time, while the XC3000 and XC4000 devices do not. Figure 6-12 illustrates Slave Serial Mode for XC4000 devices.

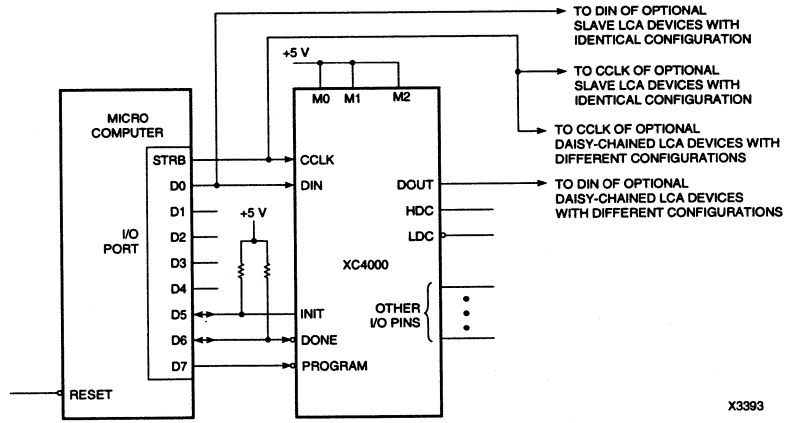


Figure 6-12 Slave Serial Mode for XC4000 Devices

Note: When using the Download or XChecker cables, the devices must be in Slave Serial mode.

Loading and Framing Configuration Data

At power-up or when you reprogram, the device clears its memory as indicated by $\overline{INIT}$. After the device clears its memory, configuration data is loaded into an FPGA from an external storage source such as an EPROM, RAM, microprocessor, or file. For serial configuration modes, each byte of PROM data is loaded into the FPGA least significant bit first. For parallel configuration modes, the data is serialized internally by the device to produce the same configuration result.

Figure 6-13 shows clock and data path selections in the device. These determine the configuration clock source and provide data serialization for DOUT and for the frame register of the internal configuration memory. The serial data of the preamble/length count and configuration data for additional devices is clocked out by the negative edge of CCLK and supplied to DOUT for daisy chained Slave mode devices.

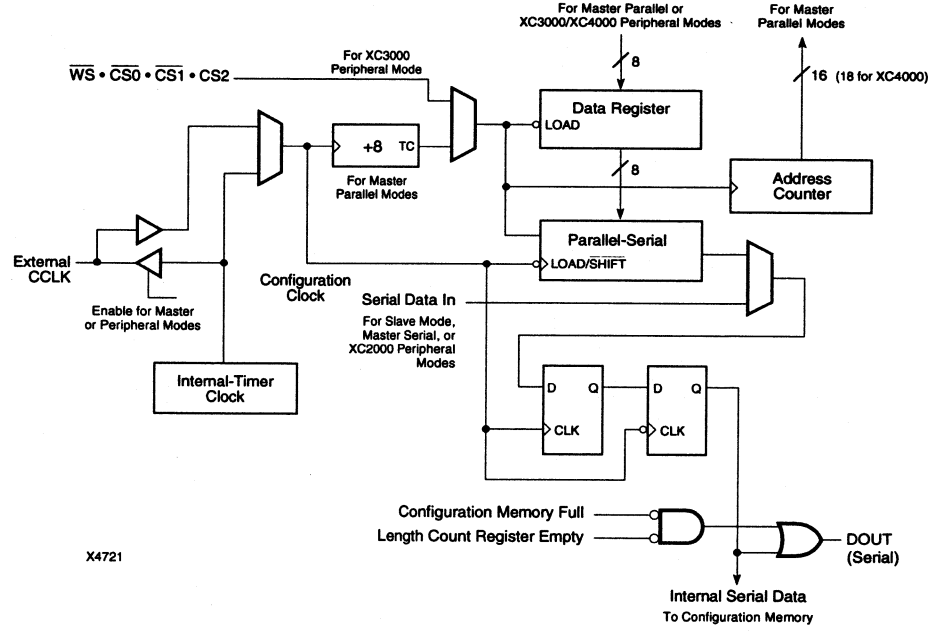


Figure 6-13 Mode-Line Logic Level Inputs to the FPGA Control Configuration Data Path Multiplexers

The length count control of operation allows a system of multiple FPGAs of assorted sizes to begin operation in a synchronized fashion, as illustrated by Figure 6-14. The frame register accumulates frames of memory data.

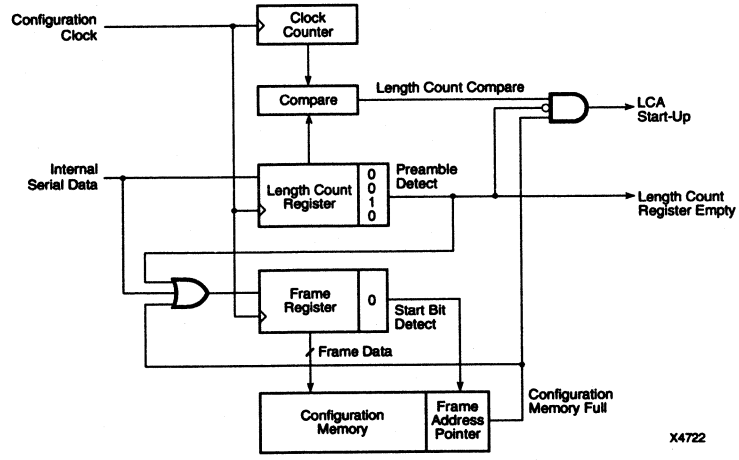


Figure 6-14 Frame Register

The configuration program generated by the MakeBits program begins with a preamble of 1111 1111 0010 followed by a 24-bit length count representing the total number of configuration clocks needed to complete loading of the configuration programs. After the preamble and length count have been loaded, serial data is shifted into the internal frame register.

The lead device always passes the preamble, length count, and 4-bit separator field to the attached devices though the DOUT pin. If there is only one device in the system, the DOUT pin still reflects this 40-bit header. Following the length count, the device presents a High DOUT signal until it intercepts the number of frames required to fill its configuration memory. It is at this time that the device begins to pass the data directly through to its DOUT pin.

This scheme allows daisy chained devices to receive the common length count and inhibits data-frame start bits for downstream devices until the leading device memory is filled.

When the internal clock counter matches the length count value loaded at the beginning of the configuration, the device begins start-up, if its program memory is full. If the values do not match, then the device continues to shift data from its DIN pin to its DOUT pin until they do.

Serial Loading of Daisy Chains

If several devices are used in a design, they can be configured in a daisy chain. The first device in the chain can be loaded in any mode and is known as the lead device; subsequent devices are loaded in Slave mode.

For daisy chain configurations, the XACT Development System creates a composite configuration bitstream for the selected FPGA designs. The bitstream includes a preamble and a length count for the total bitstream, followed by one or more linked data groups separated by the device tail bits. With a 24-bit length-count register, FPGAs can accommodate over 16 million configuration bits or about 700 XC3030s in a single daisy chain.

After loading and passing on the preamble and length count to a possible daisy chain, a lead device loads its configuration data frames while providing a High DOUT to possible downstream devices.

When the lead device has received its configuration data and if the current clock count has not reached the full length count value, the additional data is shifted out and appears in serial form on the DOUT pin. The lead device also generates the configuration clock to synchronize the serial data in and data out of following devices.

Slave devices use the positive edge of CCLK to read data in and the negative edge of CCLK to shift data out. A Master-Parallel mode device uses its internal timing generator to produce a continuous CCLK of eight times its EPROM address rate. XC3000 and XC4000 Peripheral-mode devices produce a burst of eight CCLKs for each chip-select and write-strobe cycle; an XC2000 device produces a single CCLK per Write.

The internal timing generator continues to operate for general timing and synchronization of inputs in all modes.

Concurrent Loading of Multiple Devices

It is possible to configure several FPGAs concurrently, which is especially useful when you want to load several devices with the same design. There are two methods of concurrent loading that differ only in the source of the configuration clock as follows.

Method 1: Lead device provides CCLK — follow these steps to perform Method 1:

1. Configure one device in the Master Serial mode and the other devices in the Slave Serial mode.
2. Connect the CCLK of the lead device to the CLK pin of the serial PROM and to the CCLK pins of all the Slaves.
3. Connect the DATA pin of the serial PROM to the DIN pins of all the devices.
4. Enable the serial PROM with the wired-AND of all the device DONE/$\overline{\text{PROG}}$ (DONE on XC4000 devices) pins.

The Master device provides clocks to the Serial PROM, which sends out data to the Master device and all the devices in Slave mode. The Master device also clocks the data into the devices in Slave mode. When all the DONE/$\overline{\text{PROG}}$ and DONE pins go High, the PROM is disabled.

Method 2: External source for CCLK — follow these steps to perform Method 2:

1. Configure all the devices in Slave Serial mode.
2. Connect all the CCLK pins together and drive them from the device providing the configuration clock.
3. Connect the DIN pins together and connect them to the device sending out the data. The DONE/$\overline{\text{PROG}}$ and DONE pins can be wire-ANDed together to create a signal to monitor the time at which all devices have been configured.
4. When loading the devices, use a wire AND of $\overline{\text{INIT}}$ pins to make sure that you have allowed enough time for them to finish the Clear state before sending data.
5. Be careful not to violate any of the CCLK timing specifications.

Note: Due to dynamic circuitry, there is a 5-µs maximum limit on CCLK Low time for XC2000 and XC3000 devices. Since there is no dynamic configuration circuitry in the XC4000 parts, there is no limit on the configuration CCLK Low time.

Loading Alternate Configurations

You can accomplish selection from alternate configurations with external control of high-order PROM address inputs or by selecting M1 for incrementing or decrementing addresses. For Master High or Low, 8-bit data bytes are read and internally serialized (least significant bit first) by the configuration clock. One Master-mode device can be used to interface to the PROM and pass additional linked configuration data to additional devices in a serial daisy chain fashion.

When all the data of a parallel configuration PROM has been read, the data output pin is disabled by a High from LDC or an early DONE. The configuration clock output is generated for the Slave Serial devices, which use the serialized data output supplied from previous devices DOUT pin. If multiple Slave Serial mode devices have identical configurations, their DIN pins can be connected in parallel.

With a Master Serial configuration, additional identical devices can be configured simultaneously, each in Serial Slave mode. Multiple Master mode devices cannot be connected in parallel because their clocks cannot be synchronized.

States for XC2000/XC3000 Devices

During configuration, the FPGA traverses through several states. During these states the FPGA performs such things as power-on time delay, configuration memory clear, and configuration programming. When the device has loaded its last frame of configuration bits, it goes through startup and becomes operational. The device can also be sent back into re-configuration if the proper signals are applied.

The state diagram of the configuration process for the XC2000 and XC3000 families is described in the following subsections and illustrated by Figure 6-15.

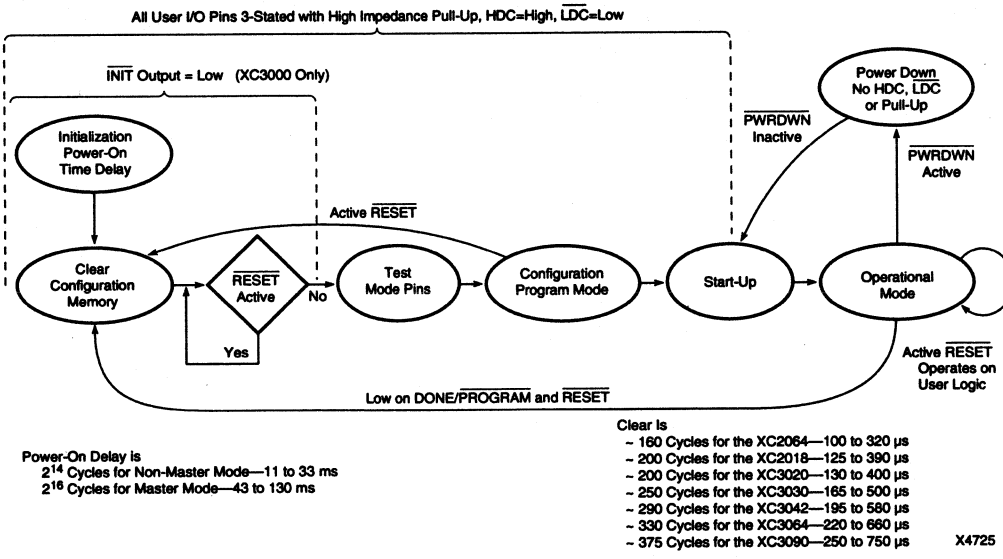


Figure 6-15 State Diagram for XC2000/XC3000 Devices

Power-up and Initialization

An internal power-on-reset circuit is triggered when power is applied as V_{CC} reaches the voltage at which portions of the device begin to operate (approximately 2.5 to 3.0 V).

Note: At power-up, V_{CC} must rise from 2.0 V to V_{CC} minimum in less than 25 ms. If this is not possible, configuration can be delayed by holding $\overline{RESET}$ Low until V_{CC} has reached 4.0 V (2.5 V for the XC2000L and XC3000L parts). A very long V_{CC} rise time of greater than 100 ms, or a non-monotonically rising V_{CC} might require a High level on $\overline{RESET}$ for greater than 6 µs, followed by a minimum pulse of 6 µs on both $\overline{RESET}$ and DONE/$\overline{PROG}$ after V_{CC} has reached 4.0 V (2.5 V for the XC2000L and XC3000L parts).

Once stable power has been established, the user-programmable I/O output buffers not used in configuration are disabled and high-impedance pull-up resistors are provided for the I/O package pins. Configuration handshake output signals, such as HDC, LDC, DONE, $\overline{INIT}$ (XC3000 only), and so on, assume their configuration-state logic levels. An internal time-out delay is initiated so that the power

supply voltage can stabilize. The Initialization state time-out of non-Master mode device (11 to 33 ms) timing is determined by a 14-bit counter.

Even a slow daisy-chained Slave Serial device must be allowed to complete initialization before a fast Master device. Master mode devices extend their initialization state to four times the non-Master delay (43 to 130 ms).

The time-out counter is clocked by a 1-MHz (nominal) internal timer, which is subject to variation from 0.5 to 1.5 MHz with process temperature and supply voltage. This internal timer is also used to synchronize inputs and provides CCLK in Master modes.

The external $\overline{\text{RESET}}$ pin must be held Low if V_{CC} power does not rise from 2.0 V to V_{CC} minimum in less than 25 ms. Until Initialization and Clear are completed, the power-down input ($\overline{\text{PWRDWN}}$) is inhibited in the XC3000 devices. You can assert $\overline{\text{PWRDWN}}$ after Initialization and Clear are completed on XC3000A and XC3000L devices. However, you should not assert $\overline{\text{PWRDWN}}$ for an XC2000, XC3000, and XC3100 devices until after configuration is completed.

Clear

After initialization, the FPGA enters the Clear state, where it clears the configuration memory. This process takes one internal timer cycle per configuration data frame. After the Clear state, the device tests for the absence of an external active Low $\overline{\text{RESET}}$ before it makes a final sample of the mode lines and enters the Configuration state.

If $\overline{\text{RESET}}$ is active, the device waits before entering the Configuration state. For Slave- or Peripheral-mode XC2000 devices, an external time delay must assure that the Initialization and Clear states are completed before configuration begins.

For the XC3000 devices, a High on the active-Low, open-drain, initialization signal $\overline{\text{INIT}}$ indicates that the Initialization and Clear states are complete. An external wired-AND of multiple XC3000 $\overline{\text{INIT}}$ pins can be used to indicate that Clear for all devices is complete. This wired-AND signal can be used to control the start of configuration and re-configuration by asserting the active Low $\overline{\text{RESET}}$ of a lead Master-mode device or by signaling a processor that all connected XC3000 devices are initialized.

Configuration

Before entering the configuration cycle, the complete configuration memory is cleared. During configuration program loading, combinatorial inputs begin to propagate as the device becomes configured, but device flip-flops and latches are held reset and output buffers are held high-impedance.

High During Configuration (HDC) and Low During Configuration ($\overline{\text{LDC}}$) are two user I/O pins that define active output levels when the device is in its Initialization, Clear, or Configuration state. HDC, $\overline{\text{LDC}}$ and DONE/$\overline{\text{PROG}}$ provide signals for control of external logic functions such as reset, bus enable or PROM enable during configuration. This allows other configuration pins to be shared with user logic functions.

Other user I/O pins that, depending on the configuration mode, have temporary functions during configuration include the address pins, data pins, chip-select/write-strobe, data in, data out, and the XC3000 $\overline{\text{INIT}}$. Output pins that are shared with configuration functions become active a clock cycle before DONE in any XC2000 device and in XC3000 devices programmed for late DONE. Contention between active device outputs and configuration inputs must be avoided. This contention can be avoided with isolation resistors, isolation buffers, an external signal used to temporarily disable user outputs, or internal FPGA logic used to match initial outputs to external levels.

User inputs can be programmed globally to have either TTL- or CMOS-compatible thresholds. The inputs have TTL thresholds at power-up and during configuration. The thresholds change to CMOS levels at the completion of configuration if the user has selected that MakeBits option. TTL-input thresholds are required for interfacing because of their 2.4 V minimum High levels. CMOS thresholds generally provide better system noise immunity.

Note: The TTL-threshold option only affects the inputs, except for $\overline{\text{PWRDWN}}$, which is fixed at a CMOS level. All XC2000 and XC3000 outputs are always CMOS compatible, that is, they switch rail-to-rail.

If the crystal oscillator is used, it begins operation early in the configuration to provide time for stabilization before it is connected by DONE to the internal circuitry.

An assertion of $\overline{\text{RESET}}$ during configuration is recognized after two or three internal-clock cycles (2 to 6 µs), and the device initiates an abort, returning to the Clear state to clear the partially loaded configuration-memory words. The device then tests for an inactive $\overline{\text{RESET}}$ and re-samples the mode lines before re-entering the Configuration state.

Note: The Clear time-out for a Master mode re-program or abort does not have the four-times delay of the Initialization state. If a daisy chain is used, an external $\overline{\text{RESET}}$ is required long enough to guarantee clearing of all non-Master mode devices. In the XC3000, you can achieve the necessary time-out delay from a wired-AND of the slave device $\overline{\text{INIT}}$ signals.

Start-up

When the initially loaded length count and internal clock count compare and the configuration memory is full, the devices execute a synchronous start-up sequence and becomes operational. For both XC2000 and XC3000 devices, this includes the following:

- The release of the DONE pin
- The activation of the user I/O
- De-activation of the internal global Reset signal

Figure 6-16 illustrates start-up timing for XC2000 and XC3000 devices.

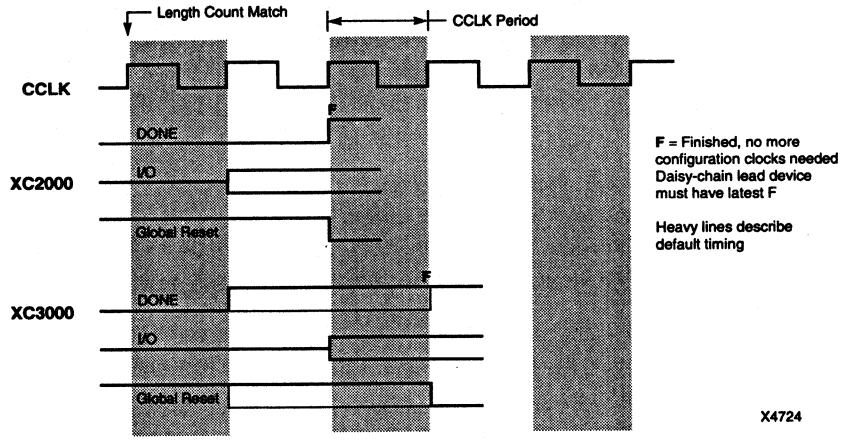


Figure 6-16 Start-up Timing for XC2000/XC3000 Devices

One CCLK after the length count and CCLK compare, the user I/O becomes active. Then, one CCLK period after the user I/O is released, the DONE pin is released and the internal global Reset is de-activated. At the same time the DONE signal is released and the global Reset signal is deactivated, the CCLK becomes an input with a pull-up resistor.

The XC3000 family offers some flexibility with the ability to program both the DONE timing and the de-activation of the global Reset relative to the release of the user I/O — either one CCLK before or one CCLK after. The user I/O is released two CCLKs after the length count compares. The timing selections of the DONE signal and the de-activation of the global Reset are programmed using the MakeBits options when the BIT file is created.

In both the XC2000 and XC3000 devices, you can program the DONE/$\overline{\text{PROG}}$ signal to be an open-drain output or to include an internal pull-up resistor to accommodate external wire-ANDing. Use of internal pull-ups must be limited to less than 16 devices on any wired-AND to allow any single device to drive the combined load Low. The devices begin operation even if the DONE/$\overline{\text{PROG}}$ pin is externally held Low.

Reprogramming

The configuration memory can be rewritten while the device is operating in your system. The device returns to the Clear state where the configuration memory is cleared, I/O pins are disabled, and the mode lines re-sampled, as described above for an aborted configuration.

Re-program control is often implemented using an external open-collector driver that pulls DONE/$\overline{\text{PROG}}$ Low. Once it recognizes a stable request, the device itself holds DONE/$\overline{\text{PROG}}$ Low until the new configuration has been completed. Even if the DONE/$\overline{\text{PROG}}$ pin is externally held Low beyond the configuration period, the device begins operation upon completion of configuration.

To reduce sensitivity to noise, these re-program signals are filtered for two-to-three cycles of the internal timing generator (2 to 6 µs). Note that the Clear time-out for a Master mode re-program or abort does not have the four-times delay of the Initialization state. If a daisy chain is used, an external $\overline{\text{RESET}}$ must be long enough to guarantee clearing all non-Master mode devices.

For the XC2000 family, indicate an external time delay of 400 µs. With XC3000 devices, this can be achieved with an 800-µs external time delay or with a wired-AND of $\overline{\text{INIT}}$ pins.

In some applications, the system power supply might have momentary failures, which can leave the device control logic in an invalid state. There are two methods to recover from this state:

- The first method is to cycle the V_{CC} supply to less than 0.1 V and re-apply valid V_{CC}.
- The second method is to provide simultaneous Low levels for 6 µs on $\overline{\text{RESET}}$ and DONE/$\overline{\text{PROG}}$ pins *after the $\overline{\text{RESET}}$ pin has been High for 6 µs following a return to valid V_{CC}.*

This guarantees the FPGA returns to the Clear state. Either of these methods is needed in the event of an incomplete supply-voltage interruption. They are not needed for a normal application of power from an off condition.

States for XC4000 Devices

The XC4000 family of parts undergoes a configuration process similar to the XC2000 and XC3000 families. The configuration process is broken into four sections:

- Power-up and Memory Clear
- Initialization
- Configure
- Startup-up
- Reprogramming

The state diagram of the configuration process for a XC4000 device is shown below.

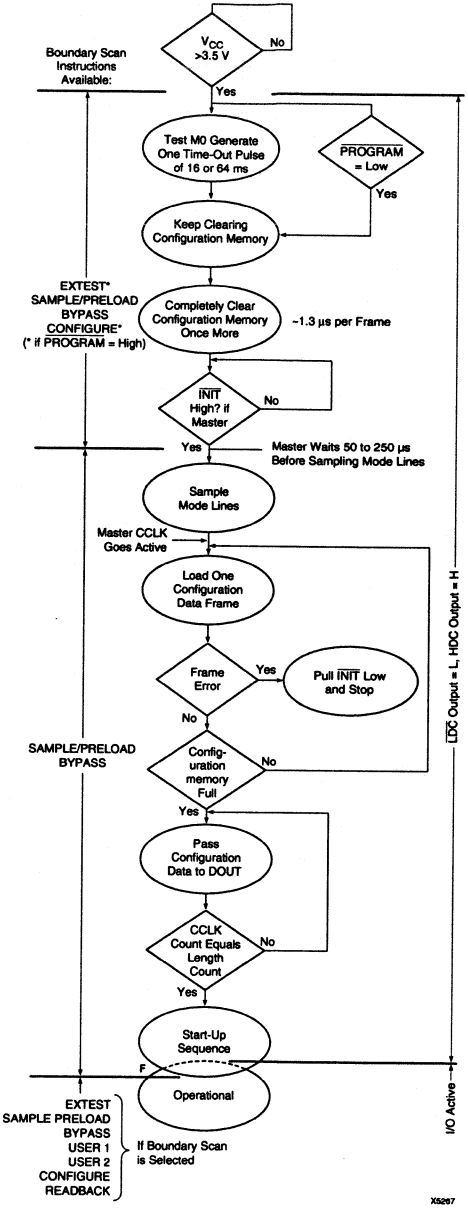


Figure 6-17 State Diagram for XC4000 Configuration Process

Power-up and Memory Clear

When power is first applied or re-applied to an XC4000 FPGA, an internal circuit forces initialization of the configuration logic. When V_{CC} reaches an operational level (approximately 2.5 to 3.0 V), and the circuit passes a write and read test of a sample pair of configuration bits, a nominal 85 ms time delay is started. During this time delay or as long as the $\overline{\text{PROGRAM}}$ input is asserted, the configuration logic is held in a Configuration Memory Clear state. The user-programmable I/O output buffers not used in the configuration are disabled and High impedance pull-up resistors are provided for the I/O package pins. Configuration handshake output signals assume their configuration state logic levels — HDC=High, $\overline{\text{LDC}}$=Low and $\overline{\text{INIT}}$=Low. The CCLK pin becomes high impedance. The configuration memory frames are consecutively initialized, using the internal oscillator. The configuration clear takes approximately 1.3 µs per frame. At the end of each complete pass through the frame addressing the power-on time-out delay circuitry and the level of the $\overline{\text{PROGRAM}}$ pin are tested. If neither is asserted, the logic initiates an additional complete clearing of the configuration frames and then releases the $\overline{\text{INIT}}$ pin.

Initialization

The open drain $\overline{\text{INIT}}$ pin is released after the final pass of memory clearing. In the XC4000, there is a deliberate delay of 40 to 160 µs before a Peripheral or Master mode device recognizes an inactive $\overline{\text{INIT}}$. To hold off configuration for an extended period of time, the $\overline{\text{INIT}}$ signal can be driven with an external open collector Low. Two internal clocks after the $\overline{\text{INIT}}$ pin is recognized as High, and the three mode lines are sampled to determine the configuration mode. The appropriate configuration interface pins then become active.

Configuration

The preamble and length count configuration data is passed to all devices in a daisy chain. The 24-bit length count is loaded into a register to be compared with an internal counter that counts the number of configuration clocks (CCLKs). After the preamble and length count is passed to all devices in the daisy chain, DOUT is held High to prevent frame start bits from reaching any other devices.

A specific configuration bit early in the first frame of a master device controls the configuration clock rate. This configuration bit can be set in the MakeBits options for XC4000 devices. The default is Slow, approximately 0.75 MHz. This rate is compatible with XC2000 and XC3000 devices and should be used in mixed daisy chains.

The Fast rate, approximately 6 MHz, can be used to speed up configuration for XC4000 devices only. The XC4000 devices end each frame with four bits of a 16-bit CRC checksum. During the loading of each frame, a 16-bit CRC checksum is generated. The final four bits of the data frame are checked against the 16-bit checksum. If there is an error found within a particular frame, the device stops loading and signals an error by pulling the open-drain $\overline{\text{INIT}}$ Low. If a frame error occurs, the device must be reprogrammed by pulling the $\overline{\text{PROGRAM}}$ pin Low.

The internal counter is checked against the length count. If these values match and the device has filled its configuration memory, the device moves on to the start-up phase. Otherwise, the device keeps sending data from the DIN pin to the DOUT pin.

Start-up

The start-up state has several options to control the sequence of events when the device becomes operational. The following figure illustrates start-up timing for XC4000 devices:

Note: Due to the large number of clocks from length count to "Finished" in the XC4000, do not use an XC2000/XC3000 device as a master with XC4000 devices as slaves.

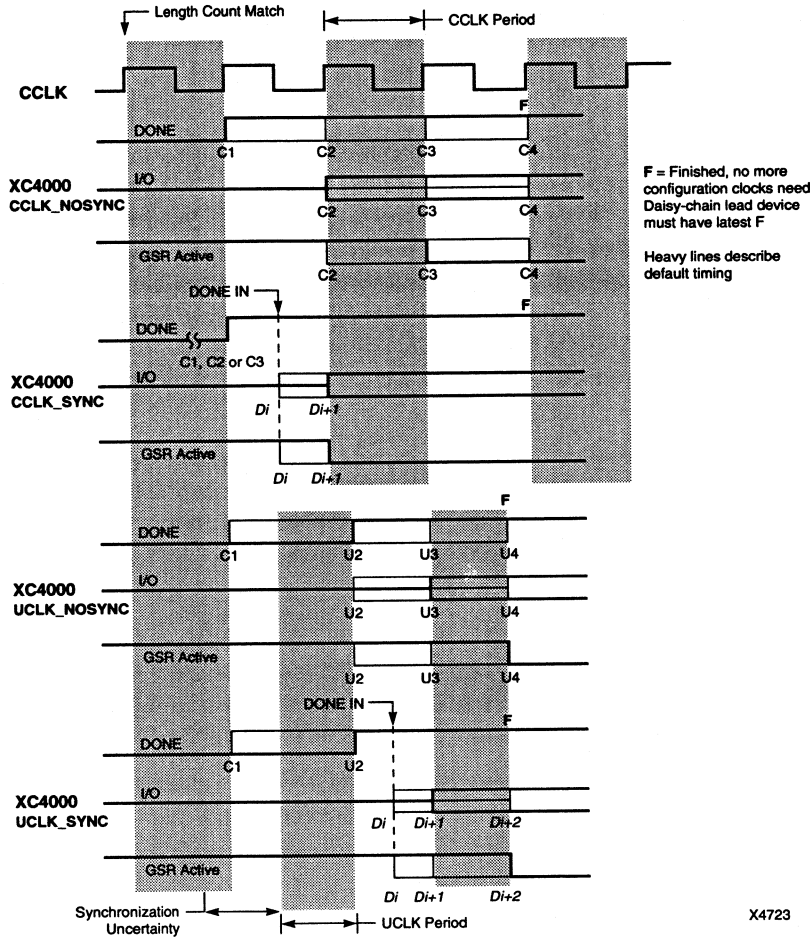


Figure 6-18 Start-up Timing for XC4000 Devices

Three events in the start-up sequence can be programmed to occur on different configuration clock edges. This flexibility allows the XC4000 to "wake up" according to user requirements. The timing for DoneActive, GSRInactive, and OutputsActive are set to different defaults according to how the StartupClk and SynchToDone option are set.

- DoneActive determines when the open-drain DONE output signal is released.
- GSRInactive determines when the global SET/RESET signal for all CLB and IOB flip-flops is released. At the end of configuration, flip-flops configured with set direct (FDSD) are set while all others are reset.

Note: The XC4000 does not have a power-down mode. The global 3-state net is provided to hold all outputs in the high-impedance state to minimize power consumption.

- OutputsActive determines the time the configuration-related pins change to the user-defined functions.

Note: The input pins of the device are active throughout the configuration process, which means that signals applied to inputs other than the configuration pins propagate through the device.

When selecting a particular start-up scheme for a given design, first decide which clock to use. Then decide whether the system on the board needs to be synchronized by using the DONE signal.

The XC4000 family allows you to use the internal CCLK to drive the start-up sequence or an external User clock. The MakeBits default indicates use of the internal CCLK. However, if the FPGA needs to be started with the rest of the system, you can use an external User clock. To use the User clock, you must connect the clock to the CLK pin of the Startup symbol in the design and set the appropriate MakeBits option, as follows:

- CCLK_NOSYNC — internal CCLK driven start-up, not synchronized to DONE
- CCLK_SYNC — internal CCLK driven start-up, not synchronized to DONE
- UCLK_NOSYNC — external User clock driven start-up, not synchronized to DONE
- UCLK_SYNC — external User clock driven start-up, synchronized to DONE

When configuring multiple FPGAs in parallel or in a daisy chain, the DONE signal can be wired together so that all devices go High simultaneously. The resulting DONE signal can be used to synchronize the OutputsActive and GSRInactive for all the devices.

You can choose between internal and external clocks, and whether or not to select synch to DONE. In the event of a data error, you can use Sync-To-DONE to prevent correctly configured upstream devices from going active.

Table 6-3 summarizes the four options you can set in MakeBits. The default values for the various settings are indicated in *italic*. The table footnotes describe each setting.

Note: When you use a user-supplied start-up clock, the first user clock after C1 clocks an additional register to resolve potential metastability problems caused by the asynchronous relationship between CCLK and the user clock. For multiple XC4000 devices in a chain, it is recommended to synchronize the start-up sequence to DONE if the user clock option is desired.

Table 6-3 MakeBits Options

MakeBits Options	DoneActive	OutputsActive	GSRInactive
CCLK_NOSYNC	*C1*[a], C2, C3, C4	*C2*[b], C3, C4	C2, *C3*[c], *C4*[d]
CCLK_SYNC	*C1*, C2, C3	C2, C3, DI[e], *DI_PLUS_1*[f]	C2, C3, DI, *DI_PLUS_1*
UCLK_NOSYNC	C1, *U2*[g], U3, U4	U2, *U3*[h], U4	U2, U3, *U4*[i]
UCLK_SYNC	C1, *U2*	U2, DI, *DI_PLUS_1*[j], DI_PLUS_2	U2, DI, DI_PLUS_1, *DI_PLUS_2*[k]

a. First rising edge of CCLK after the length count compares
b. Second rising edge of CCLK after length count compares
c. Third rising edge of CCLK after length count compares
d. Fourth rising edge of CCLK after length count compares
e. When the DONE input goes High
f. First rising edge of CCLK after DONE goes High
g. Second valid rising edge of user clock after C1
h. Third valid rising edge of user clock after C1
i. Fourth valid rising edge of user clock after C1
j. First rising edge of user clock after DI
k. Second rising edge of user clock after DI

Note: When using the NOSYNC modes, the device becomes operational even if the DONE pin is externally held Low.

Reprogramming

The configuration memory can be rewritten while the device is operating in your system. Reprogramming might be required to load a different configuration or to reload the configuration after a power

supply interruption. To reprogram, pull the $\overline{\text{PROGRAM}}$ pin Low. The device returns to the Clear state where the configuration memory is cleared, I/O pins are disabled, and the mode lines are re-sampled, as described previously.

To reduce sensitivity to noise, the $\overline{\text{PROGRAM}}$ signal is filtered for 2-3 cycles of the devices internal timing generator (2-6 µs). Once the device recognizes a stable request, it holds DONE Low until the new configuration has been completed. Even if DONE is externally held Low beyond the configuration period, the device begins operation upon completion of configuration unless you used Sync-To-DONE.

On a re-program, Xilinx devices do not provide a time-out delay; therefore, you must provide one to ensure that downstream devices can clear their own configuration memories. See *The Programmable Logic Data Book* for the appropriate time-out delays. The Clear time-out for a Master or Peripheral mode re-program does not have the 40-160 µs delay as in powering up condition. If a daisy chain is used, wiring the $\overline{\text{INIT}}$ pins together guarantees that the chain does not start configuring until all devices are finished clearing.

Length Count

When an FPGA begins its configuration sequence, it loads in the 24-bit length count (LC) found in the 40-bit bitstream header. It then proceeds to clock in data and increment an internal counter that keeps tracks of the number of configuration clocks that have occurred. The counter begins its enumeration of configuration clocks after $\overline{\text{INIT}}$ has gone inactive. When this internal counter equals the loaded 24-bit length count value, the device begins its individual start-up sequence if the memory is full.

Note: This section provides an overview of how the Xilinx software calculates LC. Use this information for verification only.

Two methods of calculating length count have been devised. The first method is called the DONE Alignment and the second is the Length Count Alignment. Also, when discussing the position of length count, the nth bit in the bit stream is being referred to where n is the value of length count.

The following table lists the frames and bits per frame that are needed to calculate the length count using either of the described methods.

Table 6-4 Bits per Frame and Frames for Xilinx Devices

Device	Bits per Frame	Frames
XC2064	74	160
XC2018	91	196
XC3020 XC3020A XC3020L XC3120 XC3120A	75	197
XC3030 XC3030A XC3030L XC3130 XC3130A	92	241
XC3042 XC3042A XC3042L XC3142 XC3142A	108	285
XC3064 XC3064A XC3064L XC3164 XC3164A	140	329
XC3090 XC3090A XC3090L XC3190 XC3190A	172	373
XC3195 XC3195A	188	505
XC4002A	102	310
XC4003A	122	374

Device	Bits per Frame	Frames
XC4003 XC4003H	126	428
XC4004A	142	435
XC4005A	162	502
XC4005 XC4005H	166	572
XC4006	186	644
XC4008	206	716
XC4010	226	788
XC4013	266	932
XC4025	346	1,220

DONE Alignment

The DONE Alignment Method is used to calculate LC in the XACT tools prior to XACT 5.0. The DONE Alignment Method, however, does not readily accommodate peripheral mode devices. Because the last bit of the byte of data is not clocked in until the next byte, DONE might never occur if the device does not receive enough configuration clocks, which is referred to as the Pig and the Python Syndrome. Also, if the number of XC4000 devices is greater than three, LC is greater than the number of bits in the stream.

When calculating LC using the DONE Alignment method, LC occurs somewhere in the last byte. This positioning of LC might seem arbitrary, but in fact is calculated based on the number of configuration clocks needed to produce a DONE signal at the end of a data byte.

The following description shows how the DONE Alignment Method calculates the value of LC for an arbitrarily sized daisy chain.

General

To manually calculate the length count using the DONE alignment method, follow these steps:

1. Initialize LC and then *add* 40 to represent the 40-bit header.

2. For each of the devices in the chain, repeat the following:
   a) *Add* (bits per frame) * (number of frames) to LC using the values given for the particular device.
   b) *Add* number of tailbits to LC, where the number of tail bits is 4 for XC2000 and XC3000 parts, and 5 for XC4000 parts.
3. If there is more than 1 device in the bit stream, *add* the number of devices to LC.
4. *Add* the value K to LC using the following criteria:
   - K = 3 if the chain consists only of XC2000 devices.
   - K = 2 if the chain consists only of XC3000 devices all with DONE scheduled before the release of the user I/O.
   - K = 4 if the chain contains an XC3000 device with its DONE scheduled after the release of the user I/O.
   - K = 3 with any mix of devices with no late XC3000 DONEs (this includes single XC4000 bit files and XC4000 chains).
5. Add the value P to LC where P is the number of pad bits.
   - P = 0 if LC is evenly divisible by 8.
   - P = n where n is the number added to LC to make it evenly divisible by 8 if it is not already evenly divisible by 8.
6. Take the rounded value of LC and *subtract* the K value you added in step 4.

Note: Although it appears in the following examples as if the K value does nothing to LC, it needs to be added so that the proper value of P is calculated.

7. If the chain does not contain any XC4000 devices the LC calculation is complete; otherwise, *add* the number of XC4000 devices in the chain to LC.

Examples

The following table illustrates the formula that calculates the length count for the specified device chain.

Table 6-5 Length Count Formulas for DONE Alignment Method

Part	Formula	Length Count
XC2018	[(40) + [((91)*(196)) + (4)] + (0) + (K=3) + (P=5) - (K=3)]	17885
XC3020[a]	[(40) + [((75)*(197)) + (4)] + (0) + (K=2) + (P=3) - (K=2)]	14822
XC3020[b]	[(40) + [((75)*(197)) + (4)] + (0) + (K=4) + (P=1) - (K=4)]	14820
XC4010	[(40) + [((226)*(788)) + (5)] + (0) + (K=3) + (P=0) - (K=3) + (1)]	178134
XC2018[c]	[(40) + [((91)*(196)) + (4) + ((91)*(196)) + (4)] + (2) + (K=3) + (P=3) - (K=3)]	35725
XC3020[d]	[(40) + [((75)*(197)) + (4) + ((75)*(197)) + (4)] + (2) + (K=4) + (P=4) - (K=4)]	29604
XC4010[e]	[(40) + [((226)*(788)) + (5) + ((226)*(788)) + (5)] + (2) + (K=3) + (P=1) - (K=3) + (2)]	356321
XC2018; XC3020[f]	[(40) + [((91)*(196)) + (4) + ((75)*(197)) + (4)] + (2) + (K=3) + (P=0) - (K=3)]	32661
XC2018; XC3020[g]	[(40) + [((91)*(196)) + (4) + ((75)*(197)) + (4)] + (2) + (K=4) + (P=7) - (K=4)]	32668
XC2018; XC4010[h]	[(40) + [((91)*(196)) + (4) + ((226)*(788)) + (5)] + (2) + (K=3) + (P=6) - (K=3) + (1)]	195982
XC3020; XC4000[i]	[(40) + [((75)*(197)) + (4) + ((226)*(788)) + (5)] + (2) + (K=3) + (P=3) - (K=3) + (1)]	192918

a. Early DONE
b. Late DONE
c. Daisy chain with two devices
d. Daisy chain with early and late DONE
e. Daisy chain with two devices
f. Daisy chain with early DONE
g. Daisy chain with late DONE
h. Daisy chain
i. Daisy chain with early DONE

The above examples do not cover the entire set of possible daisy chains, but do represent a large subset.

Note: To make the position of LC more consistent and to provide you with better handling of the peripheral mode "pig and python" syndrome, the Length Count Alignment Method is recommended.

Length Count Alignment

The Length Count Alignment Method forces the LC position to be the first bit of the last byte in the bit stream. This technique ensures that there is a sufficient number of configuration clocks provided to the devices in the chain, regardless of their type or start-up sequence.

General

To manually calculate the length count using the Length Count alignment method, follow these steps:

1. Initialize LC and then ADD 40 to represent the 40-bit header.
2. For each of the devices in the chain repeat the following:
   a) Add (bits per frame) * (number of frames) to LC using the values given for the particular device.
   b) Add number of tailbits to LC, where the number of tail bits is four for XC2000 and XC3000 parts, and five for XC4000 parts.
3. If there is more than one device in the bit stream, add the number of devices to LC.
4. Add the value P to LC where P is the number of pad bits.
   - P = 0 if LC is evenly divisible by 8.
   - P = n where n is the number added to LC to make it evenly divisible by eight if it is not already evenly divisible by eight.
5. Add one to LC and a byte (FF) to the actual bit stream. This forces LC to fall in the first bit of the last byte.

Note: There is no K value used in the Length Count Alignment Method.

Examples

The following table illustrates the formula that calculates the length count for the specified device chain.

Table 6-6 LC Formulas for Length Count Alignment Method

Part	Formula	Length Count
XC2018	[(40) + [((91)*(196)) + (4)] + (P=0) + (1)]	17881
XC3020[a]	[(40) + [((75)*(197)) + (4)] + (0) + (K=2) + (P=3) - (K=2)]	14822
XC3020[b]	[(40) + [((75)*(197)) + (4)] + (P=5) + (1)]	14825
XC4010	[(40) + [((226)*(788)) + (5)] + (P=3) + (1)]	178137
XC2018; XC3020[c]	[(40) + [((91)*(196)) + (4) + ((75)*(197)) + (4)] + (2) + (P=3) + (1)]	32665
XC4010[d]	[(40) + [((226)*(788)) + (5) + ((226)*(788)) + (5)] + (2) + (P=4) + (1)]	356233
XC2018; XC3020; XC4010[e]	[(40) + [((91)*(196)) + (4) + ((75)*(197)) + (4) + ((226)*(788)) + (5)] + (3) + (P=5) + (1)]	210761

a. Early or late DONE
b. Late DONE
c. Daisy chain
d. Daisy chain with two devices
e. Daisy chain with three devices

Debugging Hints

There are two categories of configuration problems.

- The FPGA fails to configure, that is, the DONE/$\overline{\text{PROG}}$ (XC2000 and XC3000) or DONE (XC4000) pin does not go High.
- The FPGA does not configure correctly, that is, the configuration program is wrong.

The following hints help with debugging both categories of configuration problems. First, read the general debugging hints. Then turn to the page that has hints for the type of configuration mode you are using and follow those suggestions.

General Debugging Hints

During design verification, if the DONE output does not go High, there are several things to check.

All Families

The following debugging hints apply to all families:

- Even without a Slave mode device, CCLK and DOUT are very informative signals. CCLK synchronizes input data and shifts it through DOUT. The preamble/length count begins with the onset of CCLK negative edges and the bit order must match.

  `1111 1111 0010 (`*length count*`) 1111 . . .`

  Verify the length count is produced on each DOUT pin.

Note: In XC2000 and XC3000 devices low pulses on $\overline{\text{RESET}}$ abort configuration. This input/output pattern can be repeated at the Clear-state time interval rather than waiting for complete configuration attempts.

- If extraneous configuration clocks are applied after Clear but before the preamble data, the clock count equals the length count before the configuration data is completely loaded. In this case, the DONE output does not become active until the clock counter again equals length count. This requires 2^{24} extra clocks, or about 16 s at 1 MHz.

Note: If configuration takes 15 to 25 seconds, there is a mismatch between length count and the number of CCLK pulses generated.

- Check supply and configuration-related pins with an oscilloscope or logic analyzer to provide invaluable information, such as wiring errors, bad socket pins, noisy ground, missing V_{CC} on a serial configuration PROM V_{PP}, and so on.
- Ringing on a clock line can cause extraneous clocking and loss of frame synchronization in an FPGA.
- XChecker and the XACT Download Cable provide alternate methods of configuration to verify configuration data and isolate wiring errors, such as interchanged or inverted configuration data or control signals.
- Try a different device; the devices are 100% tested at the factory, but they could be damaged after leaving the factory.
- Configuration functions can be disrupted by signal contention between configuration inputs and the device user outputs, which become active at the end of configuration. This disruption can be

indicated by I/O pins being active and HDC/$\overline{\text{LDC}}$ no longer being active at their configuration levels. Avoid these contentions by rearranging pin-outs, maintaining additional 3-state control of user I/O outputs, or matching start-up output levels to the configuration input levels on inputs other than chip-select. It is also possible to use a series resistor (1-10 kΩ) to provide isolation between conflicting signal sources that could occur after configuration is complete.

- During reprogramming, user logic must generate a time-out that ensures all devices have completed the Clear cycle before any configuration data is sent to them.
- When externally powered signals continue to drive input pins, removing V_{CC} from the device may leave V_{CC} at a 0.5-to-2.0 V level, which can leave a device in an invalid state. The input-protection circuits of the device include diodes to clamp input-voltage excursions to ground and V_{CC}. When V_{CC} falls more than half a volt below the input signal, the input signal might begin to supply degenerate V_{CC} levels. If input signals are not current-limited, the input-protection circuit can be damaged by the excessive current required.
- If the preamble and length count appear correctly on DOUT but DONE is missing, some dummy bits can be added to the end of the normal configuration data. Some PROM programmers can be used to edit the PROM length count and append this dummy Slave data. An alternative is to add an additional device in MakePROM. This step is rarely necessary.
- If the device chain is failing to go though the start-up sequence, remake your bit stream using the alternate length count generation method.

XC2000 and XC3000

The following debugging hints apply to XC2000 and XC3000 families only:

- A slowly rising or noisy $\overline{\text{RESET}}$ can cause multiple devices to get out of synchronization. Always debounce reset switches.
- The internal state of the device can be disrupted if the $\overline{\text{PWRDWN}}$ is not pulled up or if the $\overline{\text{RESET}}$ pin is experiencing noise.

- An undefined (floating) or active Low $\overline{\text{PWRDWN}}$ during configuration can disturb the operation.
- The configuration-clock input signal drives some quasi-static circuitry that requires a maximum Low-time input specification. *Never hold the configuration clock input Low for more than 5 µs.*
- Check for missing pull-up resistors on DONE/$\overline{\text{PROG}}$ (or $\overline{\text{INIT}}$ in the XC3000). Refer to *The Programmable Logic Data Book* for appropriate pin assignments.
- Make sure V_{CC} rises in 25 ms or less. If this cannot be guaranteed, hold $\overline{\text{RESET}}$, active on the devices and any serial PROMs until power is up on your system.
- If $\overline{\text{RESET}}$ is used to delay configuration, make sure it has a rise time of less than 200 ns and that it is error-free.

XC4000

The following debugging hints apply to XC4000 families only:

- Check for missing pull-up resistors on DONE, and so on. Refer to *The Programmable Logic Data Book* for appropriate pin assignments.
- Make sure V_{CC} rises in 25 ms or less. If this cannot be guaranteed, hold $\overline{\text{PROGRAM}}$ or $\overline{\text{INIT}}$ active on the devices and reset any serial PROMs until power is up on your system.
- An excessive number of loads on the configuration clock can cause noise incorrect information to be loaded into the device.
- The boundary scan input pins are active during configuration even if boundary scan is *not* used in the design. During configuration, it is possible to toggle the boundary scan inputs and send the device into boundary scan mode. This process halts configuration.

Master Parallel Up and Down Modes

The following sections provide debugging information for failed and incorrect configuration during Master Parallel Up and Down Modes.

Debugging Hints for Failed Configuration

For a failed configuration, try the following:

- Verify that the FPGA is sending addresses to the PROM. If it is not, make sure the FPGA mode pins (M0, M1, M2) are at their proper values, shown below.
  - M0 = 0, M1 = 0, M2 = 1 for Master Parallel Up
  - M0 = 0, M1 = 1, M2 = 1 for Master Parallel Down
  - Make sure V_{CC} is at 5 V and ground is at 0V.
- Check to see that the PROM is sending out data. If it is not, try the following:
  - Make sure that the addresses from the FPGA are arriving at the PROM address pins.
  - Make sure that power and ground are connected to the PROM.
  - Make sure that the PROM is enabled.
  - Verify that the PROM is programmed with the correct data.
- Check the PROM data pins to be sure they are connected to the FPGA data pins D0-D7. Be sure the PROM address pins are connected to the FPGA address pins A0-A15. Verify that all connections are in the appropriate order. Monitor the FPGA pins, not the socket pins. Check the socket connections.
- Check for contention between the FPGA address pins or the PROM data pins and other signals on the board.

Debugging Hints for Incorrect Configuration

For an incorrect configuration, try the following:

- Make sure that the device is addressing the correct part of the PROM. The device starts at address 0000 hex and counts up in Master Parallel Up mode and at address FFFF hex, and counts

down in Master Parallel Down mode. If the PROM addressing is different, it must be taken care of by external hardware.

- Check for contention between the device address pins or the PROM data pins and other signals on the board.
- Check for ringing and noise on address and data lines.
- Make sure the data and address pins on the PROM are wired to the correct data and address pins on the device.
- Make sure the data in the PROM is correct. You can check it against the Rawbits file.

Master Serial Mode

The following sections provide debugging information for failed and incorrect configuration during Master Serial Mode.

Debugging Hints for Failed Configuration

For a failed configuration, try the following:

- Monitor the DOUT pin of the Master device. In the beginning of configuration, you should see the following.

  `1111 1111 0010` (*length count*) `1111 . . .`

  After this sequence, DOUT should be all ones — a continuous High level — until the DONE/$\overline{\text{PROG}}$ or DONE pin goes High. If this is a Master device for a daisy chain, this pattern remains until the Master device has received all its data. After receiving the data, the Master device becomes transparent and passes data through the DOUT pin to the Slave devices. If you do not see this pattern, you have an error. Try some of the other hints.

- Verify that the FPGA is sending a clock signal on its CCLK pin and that this signal is reaching the Serial-Configuration PROM CLK pin. If it is not, make sure the device mode pins (M0, M1, M2) are at their proper values, shown below.

  M0 = 0, M1 = 0, M2 = 0 for Master Serial mode

- Verify that the Serial-Configuration PROM is sending data.
  - Make sure that power and ground are applied to the PROM.

  - Make sure V_{PP} is connected to V_{CC}. A floating V_{PP} pin results in temperature-dependent unreliable operation.
  - Make sure the PROM is enabled with $\overline{OE}$ and $\overline{CE}$ Low.
- Verify that the PROM is programmed with the correct data.
- Make sure that the DATA pin of the PROM is connected to the DIN pin of the device.
- Check for contention between the device pins or the PROM data pin and other signals on the board.
- A symptom of slow V_{CC} rise time is that the device sends out clocks continuously, the $\overline{CEO}$ pin on the PROM goes Low, but the DONE/$\overline{PROG}$ or DONE pin never goes High.
- If you abort the FPGA configuration by asserting device $\overline{RESET}$, you must also reset the serial PROM by asserting the OE/RESET pin.

Debugging Hints for Incorrect Configuration

For incorrect configuration, try the following:

- Make sure that the V_{PP} pin on the Serial Configuration PROM is tied to V_{CC}. A floating V_{PP} pin results in temperature-dependent unreliable operation.
- Check for contention between device pins or the PROM data pins and other signals on the board.
- Check for ringing and noise on the clock and data lines.
- Make sure the PROM data is correct.

Peripheral Mode

The following sections provide debugging information for failed and incorrect configuration during Peripheral Mode.

Debugging Hints for Failed Configuration

For a failed configuration, try the following:

- Monitor the DOUT pin of the Master device. In the beginning of configuration, you should see the following.

1111 1111 0010 (*length count*) 1111 . . .

After this sequence, DOUT should be all ones, a continuous High level, until the DONE/$\overline{\text{PROG}}$ or DONE pin goes High. If this is a Master device for a daisy chain, this pattern remains until the Master device has received all its data. When it has, the Master device becomes transparent and passes data through the DOUT pin to the Slave devices. If you do not see this pattern, you have a gross error somewhere. Check the following hints.

- Verify that the device is receiving data at its input pins and that it is receiving valid Write-Strobe and Chip-Select signals. If it is not receiving the proper data and valid signals, check the device loading the device. Make sure that these signals meet the device timing requirements listed in *The Programmable Logic Data Book.*
- Observe minimum times listed in *The Programmable Logic Data Book.*
- Make sure the device mode pins (M0, M1, M2) are at their proper values.

  M0 = 1, M1 = 0, M2 = 1 for Peripheral mode
- Make sure that the device is ready to receive data.

  XC3000 and XC4000 devices — on power up, make sure that the $\overline{\text{INIT}}$ pin has gone High, or wait at least 34 ms (22 ms for XC4000) before you begin sending data to the device. Make sure that the RDY/$\overline{\text{BUSY}}$ signal is High before sending each data byte.

  XC2000 Family — on power up, make sure that the device has had time to "wake up," at least 34 ms, before sending it data.
- Check for contention between the Chip Selects ($\overline{\text{CS0}}$, $\overline{\text{CS1}}$, and CS2) or the Write Strobe ($\overline{\text{WS}}$) and signals on these pins after configuration. It is best to use the Chip Select pins as inputs after configuration. Avoid contention if they are used as outputs. With XC2000 Family devices, the I/Os become active before the device receives its final data bits and clocks, and also before the DONE/$\overline{\text{PROG}}$ pin goes High. If the I/Os for any of the Chip Selects or the Write Strobe become outputs after configuration, they could contend and, in effect, de-select the device so that it never receives its final data bits.

Warning: Beware of contention!

- Check for contention between the FPGA pins and other signals on the board.

Debugging Hints for Incorrect Configuration

For an incorrect configuration, try the following:

- Make sure that you are sending the data to the device in the correct order. Monitor DOUT and verify that the pattern below appears at the beginning of configuration.

  `1111 1111 0010 (`*length count*`) 1111 . . .`

  - XC3000 and XC4000 devices — data is sent as a byte. Make sure bit 0 is connected to the FPGA D0 pin, bit 1 to D1 pin, and so on.
  - XC2000 devices — data is sent serially. If a PROM file is used as a data source, make sure you serialize the data by sending the LSB of each byte first. For example, if the data byte is 14 (0001 0100), it must be sent to the device as 0010 1000.
- Check for contention between device pins and other signals on the board.
- Check for ringing and noise on the Chip-Select and Write-Strobe lines.

Slave Mode

The following sections provide debugging information for failed and incorrect configuration during Slave Mode.

Debugging Hints for Failed Configuration

For failed configuration, try the following:

- Monitor the DOUT pin of the Master device. In the beginning of configuration, you should see the following pattern.

  `1111 1111 0010 (`*length count*`) 1111 . . .`

  After this sequence, the DOUT pin should output all ones (be continuously High) until the DONE/$\overline{\text{PROG}}$ or DONE pin goes High. If this is the first device for a daisy chain, this pattern remains until the Master device has received all its data. Then, the

Master device becomes transparent and passes data through the DOUT pin to the Slave devices. If you do not see this pattern, you have a gross error somewhere. Check the following hints.

- Verify that the device is receiving data at its input pin (DIN) and that it is receiving a valid clock signal on its CCLK pin.
  - Check the device sending the data.
  - Check the device sending the clock signal and make sure the clock meets the device timing requirements. Refer to the Slave Mode Programming section of *The Programmable Logic Data Book*. If the clock is being generated by the configuration clock of a Master device, it always meets the proper timing requirements.
- Make sure the device mode pins (M0, M1, M2) are at their proper values.

  M0 =1, M1 = 1, M2 = 1 for Slave mode
- Make sure the device is ready to receive data.
  - XC3000 and XC4000 devices — on power up, make sure the $\overline{\text{INIT}}$ pin is High or wait at least 34 ms (22 ms for XC4000 devices) before you begin sending data to the device.
  - XC2000 devices — on power up, make sure that the device has had time to "wake up," at least 34 ms, before sending it data.
- Check for contention between the device pins and other signals on the board.

Debugging Hints for Incorrect Configuration

For incorrect configuration, try the following:

- Make sure that you are sending the data to the device in the correct order. Monitor DOUT and verify that the following pattern appears at the beginning of configuration.

  `1111 1111 0010 (`*length count*`) 1111 . . .`
- Data is sent serially. Make sure you send the LSB of each byte first if a PROM file is used as a data source. For example, if the data byte is 14 (0001 0100) it would be sent to the device as 0010 1000.

- Check for contention between device pins and other signals on the board.
- Check for ringing and noise on the data and clock signals.

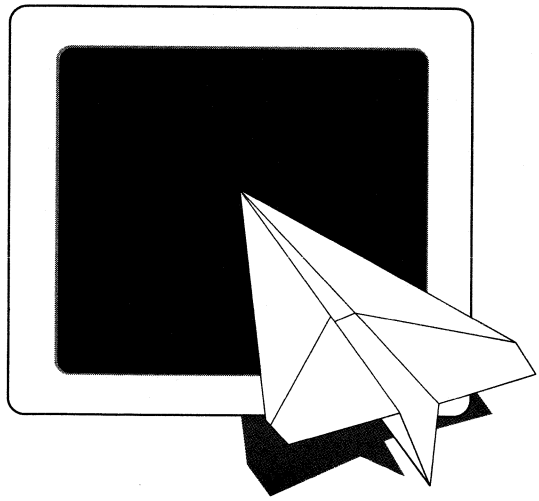
Daisy Chain Debugging Hints

The key to debugging daisy-chain configurations is to isolate the problem and attempt to configure a single device. Remove all but the first device from the board and configure it. Then insert the second device and configure both. Repeat as you add one device at a time until they all configure.

- The first device in the chain can be in any of the configuration modes. Debug it first, using the hints provided for the appropriate mode.
- All devices after the first one are in a Slave mode, so refer to Slave Mode Debugging Hints section to solve any problems with Slave device.
- Monitor the DOUT pin of each device in the chain and verify the pattern below appears at the beginning of configuration.

  `1111 1111 0010 (`*length count*`) 1111 . . .`

- Make sure that the length count is long enough to represent a chain of all the devices.
- Add a dummy bitstream or just zeros to the end of the bitstream. Change the LC to reflect the number of bits you added to the end. This verifies that the last device in the daisy chain goes transparent.



XACT User Guide

The XC4000 Readback Capability

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Chapter 7

The XC4000 Readback Capability

This chapter describes the XC4000 readback capability and covers the following topics:

- When is readback necessary?
- Readback features
- Performing a readback
- Readback initialization
- Configuration and readback bitstreams
- Software support for readback
- Readback timing
- Cyclic redundancy check

Every FPGA device shipped by Xilinx is tested using the device readback capability. All CLBs and IOBs are configured and read back using extensive test patterns to guarantee 100 percent functionality of the FPGA device.

You can perform a readback on a device at any time after configuration.The readback data consists of the configuration data and, optionally, the current state of the CLBs and IOBs.

When is Readback Necessary?

Xilinx devices are 100 percent pretested. You can use cyclic redundancy checking (CRC) for XC4000 series devices on the configuration bitstream to check the integrity of the bitstream loaded into the configuration memory.

The configuration bitstream includes a 4-bit partial check of a 16-bit CRC of the configuration data for each data frame transmitted into the FPGA device. Using this technique, the device detects invalid data bits and aborts the configuration process. The INIT status pin is pulled Low, signaling that an error occurred during loading of the configuration memory.

Consequently, readback is useful only in few cases as follows.

- To verify the configuration in an unstable environment
- To read back the internal state of the RAM, CLBs, and IOBs during the FPGA development phase
- To test high-reliability applications that require in-system functional analysis and verification

Readback Features

The readback operation does not interfere with the FPGA operation. After a valid readback request, the current state of the internal nodes can be captured into a special shift register. Then the data can be transferred out of the device using a user-defined clock signal.

The following internal configuration data and circuit nodes are available for readback as follows:

- Configuration memory bits that define the logic configuration of CLBs, IOBs, and the interconnects
- X and Y output pins of CLB function generators
- XQ and YQ output pins of CLB flip-flops
- Output pins of flip-flops in IOB output paths
- I1 and I2 input pins of IOBs

Note: The XC4000 CLB includes special connections between the C1-C4 input pins and the XQ and YQ output pins that can be used to route signals from one side of the CLB to the other. Although these signals leave the CLB through the XQ or YQ pin, they are not captured in the readback bitstream; only the actual flip-flop outputs are captured.

Figure 7-1 and Figure 7-2 illustrate the logic configurations of IOBs and CLBs.

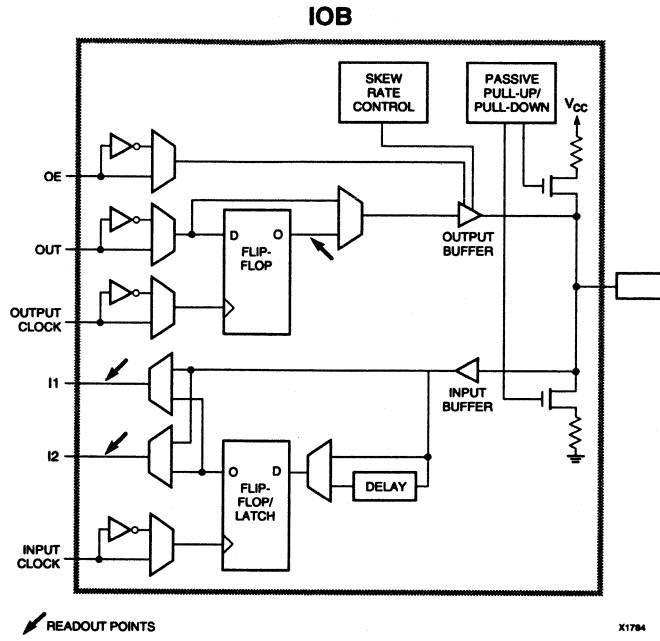


Figure 7-1 IOB Logic Configuration

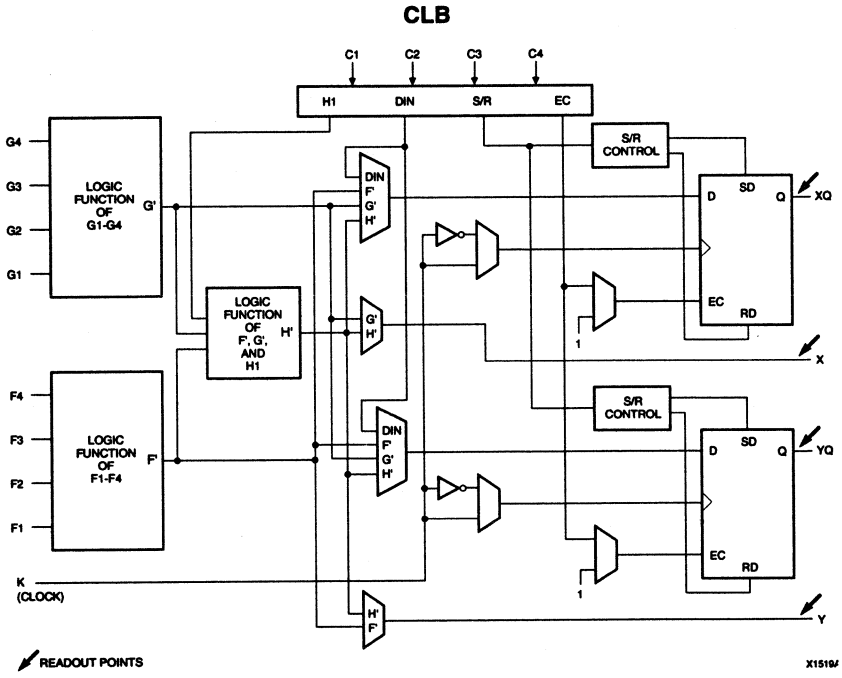


Figure 7-2 CLB Logic Configuration

A mask file (*design*.LL), generated by the MakeBits program, contains information about the location of the user data bits in the readback bitstream and the names of the signals connected.

You can implement comparison logic in CLBs to compare the readback bitstream with data stored in the configuration PROM. This technique does not work if any CLB is used as RAM, since changing the RAM contents alters the data in the configuration memory. In this case, an additional mask PROM is needed to disable the comparison of readback bitstream locations that represent the RAM data.

The XC4000 family features a boundary-scan instruction that initiates a readback sequence. Refer to the "Boundary Scan in XC4000 Devices" chapter in this user guide for more information.

Daisy chaining FPGA devices for readback is not possible. Each device must be read back individually.

The XChecker download cable and logic probe handles configuration and readback of XC2000, XC3000, and XC4000 FPGA families. In addition, it displays selected internal nodes on the screen.

Performing a Readback

The following section describes performing a readback and covers these topics:

- Readback state diagram
- Readback primitive

Readback State Diagram

An FPGA-internal state machine controls the readback process. See Figure 7-3 for the readback state flow diagram.

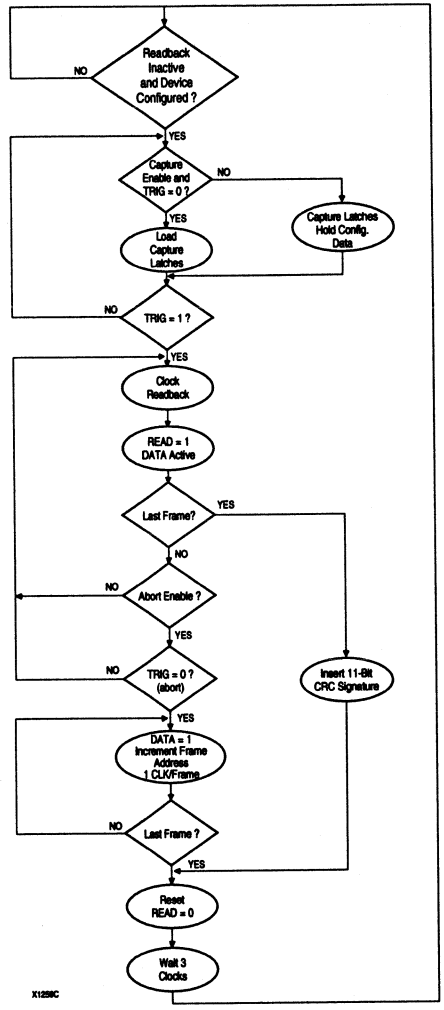


Figure 7-3 Readback State Diagram

READBACK Primitive

The XC4000 FPGA device has a dedicated primitive that handles all readback functions. The primitive has two inputs and two outputs as illustrated by Figure 7-4.

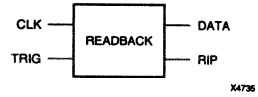


Figure 7-4 The READBACK Primitive

The READBACK primitive can connect to the user I/Os and CLBs as follows.

Table 7-1 READBACK Primitive Connections

Readback Pin	XDE Pin	Description
CLK	rdclk.I	The clock input can be connected to any device input pin, or any CLB output. If it is not connected to a user net, it connects to the device CCLK input pin, if the appropriate option is selected in the MakeBits program.
TRIG	rdbk.TRIG	A Low-to-High transition on the TRIG input starts a readback sequence. The minimum required pulse width is one rdclk.I cycle. A valid trigger causes the current value of certain nodes to be latched into an internal holding register. If ReadAbort was selected as an option in MakeBits, a High-to-Low on the TRIG input aborts the readback. In this case, additional clocks must be provided until rdbk.RIP signals the end of a readback. The rdbk.TRIG cannot be reasserted until at least three clock periods after the previous readback has been terminated correctly.
RIP	rdbk.RIP (readback in progress)	A High on this output indicates that a readback is being performed. RIP goes active one readback clock cycle after a valid readback trigger has occurred. It goes Low with the last data bit shifted out of the FPGA device. If a readback aborts, RIP remains active until the readback sequence is terminated correctly.

Readback Pin	XDE Pin	Description
DATA	rdbk.DATA	The readback data is available on the DATA output of the readback primitive. Each rising edge on rdclk.I shifts one data bit from the LCA-internal holding register to the DATA output. You can opt to disable the user data bits in the readback bitstream.

Note: In XC3000 devices, the input pin M0/RTRIG is used as a readback trigger pin and M1/RDATA as a readback data pin. In XC4000, you can use the M0 pin as an input pin and the M1 pin as a 3-state output; you could also use any other pins for these functions.

Also, XC3000 has a MakeBits option to inhibit readback. In XC4000, conventional readback is possible if you use the readback primitive in the design or perform a boundary-scan readback.

Readback Initialization

You can prepare an FPGA design for readback by any of these methods, which are described in the following sections:

- Using the readback primitive on the schematic
- Activating readback from XDE
- Performing a readback during a boundary scan

Using the Readback Primitive on the Schematic

The Xilinx Libraries include a READBACK primitive that can be used in a schematic like any other library primitive. Simply connect the inputs and outputs of the READBACK primitive to your user nets as desired. See Figure 7-5 for an example.

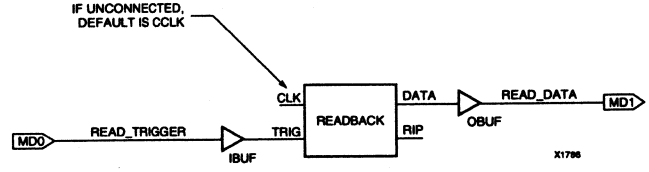


Figure 7-5 READBACK Symbol on the Schematic

Note: If the CLK input is not connected to any net, the bitstream generation software connects it to the CCLK input pin, if you selected the appropriate ClkSelect=Cclk in the MakeBits program.

Activating Readback from XDE

In XDE, the readback blocks are located in the lower left and lower right corners of the device. It is activated if these rdbk.TRIG and the rdbk.DATA signals are connected. The rdclk.I pin is connected to the CCLK pin, if not connected otherwise.Figure 7-6 illustrates the XDE readback blocks.

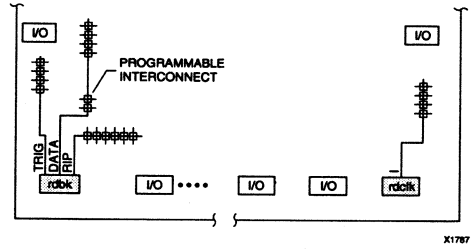


Figure 7-6 The XACT Readback Blocks

Performing a Readback during a Boundary Scan

No changes are required to prepare a design for readback through the boundary-scan port.

Configuration and Readback Bitstreams

The following sections describe the XC4000 configuration and readback bitstreams.

The XC4000 Configuration Bitstream

Figure 7-7 shows the format of the XC4000 configuration bitstream, as generated by the MakeBits program. The bitstream consists of header and program data. The header consists of four dummy bits, the preamble code, the configuration-program-length count, and an additional four dummy bits. The program data is divided into frames consisting of a start bit (0), the data field, and four error check bits (eeee). The bitstream ends with eight or more postamble bits

(01111111). The exact number of the bits in the bitstream is represented by the 24-bit program-length count.

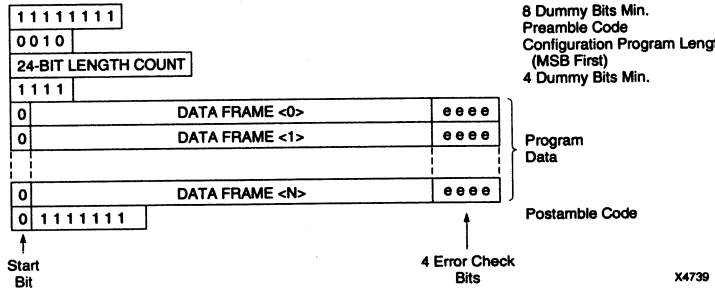


Figure 7-7 XC4000 Configuration Bitstream Format

Note: The configuration bitstreams are subject to change without notice.

For more information on length count, refer to the "Configuration, Length Count, and Debugging" chapter in this user guide. For more information on gate count, number of CLBs, number of RAMs, number of IOBs, and number of flip-flops for the various XC4000 devices, refer to *The Programmable Logic Data Book.*

The XC4000 Readback Bitstream

The readback bitstream contains configuration information as well as the state of internal user logic. The readback bitstream starts with five dummy bits. The readback data frame has the same format as the configuration data frame, which eases a bit-by-bit comparison between readback and configuration data. Each data frame consists of

a start bit (0), the data field, and four stop bits (1111). The bitstream ends with 11 CRC bits, as illustrated by Figure 7-8.

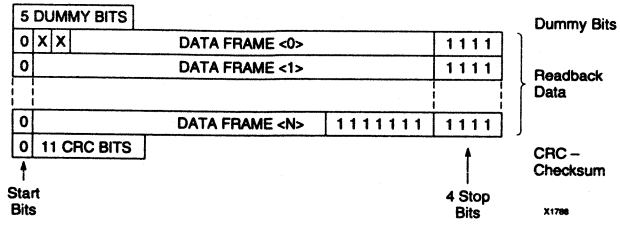


Figure 7-8 XC4000 Readback Bitstream

Both the configuration data and the internal-logic data are included in the readback bitstream. In the readback bitstream, the configuration data bits are not inverted with respect to the configuration bitstream, unlike XC3000 readback. The user-logic data bits, however, are inverted with respect to their values during readback capture.

The readback configuration data might differ from the original data downloaded into the device if CLB RAM is used in the design. The RAM data is stored in the F- and G-function tables of the CLB.

The first two bits of the first readback data frame are variable; they are non-user, non-configuration bits. Their input state is dependent on the configuration speed and the configuration error-check mode of the device. The last seven bits of the last readback data frame are always ones.

If readback capture of user data is disabled in the MakeBits program, logic Highs replace the user data. The RAM data is not part of the captured user logic data; it is contained in the readback configuration data.

The bitstream ends with eleven bits of a CRC signature appended. If ReadCapture is disabled and the design does not use any CLB RAM, this signature is constant in successive readbacks. See the "Cycle Redundancy Check" chapter in this user guide for more information about the polynomial cyclic redundancy check, CRC-16.

Software Support for Readback

The user can set readback options with the MakeBits program.The following MakeBits options are relevant for readback of XC4000 devices.

ReadCapture

Settings	Default
Enable, Disable	Disable

This option determines whether the state of internal user logic is included in the readback bitstream. If ReadCapture is disabled, the user data is replaced by ones.

ReadAbort

Settings	Default
Enable, Disable	Disable

ReadAbort enables the level-sensitive signal rdbk.TRIG to abort the readback. A High-to-Low transition stops the readback. You must supply additional clocks to terminate the readback correctly. As a minimum, the number of data frames contained in the device plus three must be sent as additional clocks. During this period, the readback data is High. The rdbk.RIP signal indicates the completion of a readback process.

ReadClk

Settings	Default
CCLK, RDBK (user supplied)	CCLK

The rdclk.I pin can be connected to any user net or to the CCLK I/O pin. With this option, you can choose between the alternatives.

LL File

MakeBits features an option used to create a logic location file (*design*.LL) that contains information on which bit in the readback bitstream corresponds to which signal in the design. This ASCII mask file, illustrated in Figure 7-9, indicates the offset from the beginning of the readback bitstream, the frame number, the offset within a frame, and names of user signals in the readback bitstream.

```
;  Offset   Column   (Frame)  Row  (Frame Offset)   Description

     21               1                100        P57 I1
     32               1                 90        U37 I1
     41               1                 79        P60 U1
      .               .                  .        .
      .               .                  .        .
      .               .                  .        .
  36640             303                 23        CD YQ
  36650             303                 13        BD YQ
  37044             307                103        LD XQ    CFG/TOGGLE
  37054             307                 93        KD XQ    CFG/RDATA_REG/Q9
  37064             307                 83        JD XQ    CFG/RDATA_REG/Q1
  37074             307                 73        ID XQ    CFG/RDATA_REG/Q2
  37084             307                 63        HD XQ    REFDATA_REG/Q5
  37095             307                 52        FD XQ
  37105             307                 42        ED XQ
      .               .                  .        .
      .               .                  .        .
      .               .                  .        .
```

Figure 7-9 Sample Logic Location File

Readback Timing

Minimum readback frequency is 10 kHz; maximum readback frequency is 1 MHz. The rdclk.I High time and Low time are each 0.5 µs minimum.

The readback speed is 10 kHz minimum, 1 MHz maximum. The following table indicates the timing parameters for readback.

Table 7-2 Timing Parameters

	Description	Symbol		Limits		
				Min	Max	Units
rdbk.TRIG	rdbk.TRIG setup	1	T_{RTRC}	200	—	ns
	rdbk.TRIG hold	2	T_{RCRT}	50	—	ns
	rdbk.TRIG Low to abort readback	3	T_{RTL}	100	—	ns
rdclk.I	rdbk.DATA delay	7	T_{RCRD}	—	250	ns
	rdbk.RIP delay	6	T_{RCRR}	—	250	ns
	High time	5	T_{RCH}	0.5	50	µs
	Low time	4	T_{RCL}	0.5	50	µs

See Figure 7-10 for additional preliminary readback switching characteristics.

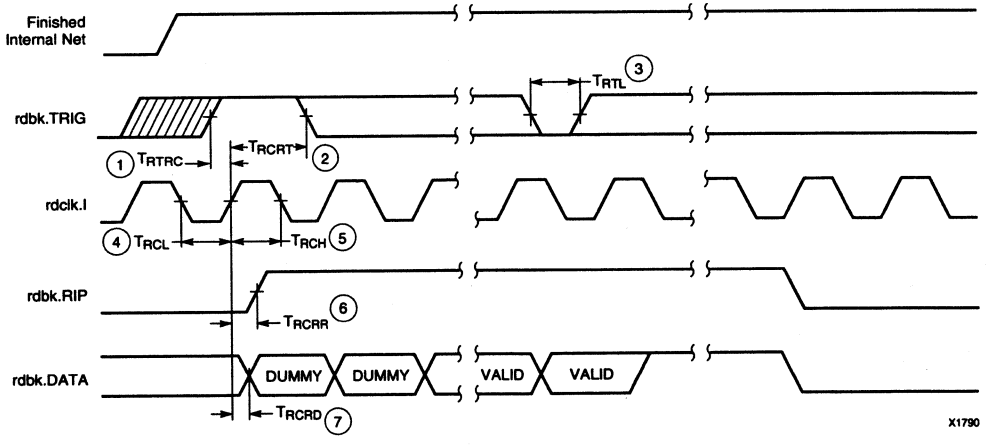


Figure 7-10 Readback Switching Characteristics

Cyclic Redundancy Check

The following section discusses the cyclic redundancy check (CRC) for FPGA configuration and readback. This section contains the following topics:

- What is CRC?
- CRC during FPGA configuration
- CRC during FPGA readback

What is CRC?

CRC is a method of error detection in data transmission applications. Generally, the transmitting system performs a calculation on the serial bitstream. The result of this calculation is tagged onto the data stream as additional check bits. The receiving system performs an identical calculation on the bitstream and compares the result with the received checksum. CRC checksum compare is often referred to as signature analysis.

CRC During FPGA Configuration

Each data frame of the FPGA configuration bitstream has four error bits at the end, as illustrated by Figure 7-7. If a frame data error is detected during the loading of the device, the configuration process with a potentially corrupted bitstream is terminated. The FPGA pulls the INIT pin Low and goes into a wait state.

CRC During Readback

During a readback, 11 bits of the 16-bit checksum are appended to the end of the readback data stream, as illustrated by Figure 7-8.

The checksum is computed using the CRC-16 polynomial, as illustrated in Figure 7-11. The checksum consists of the 11 most significant bits of the 16-bit code. A change in the checksum indicates a change in the readback bitstream. Statistically, one in 2048 errors might go undetected.

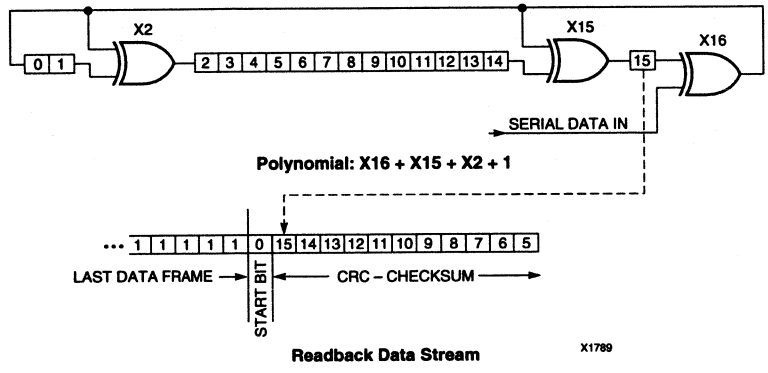
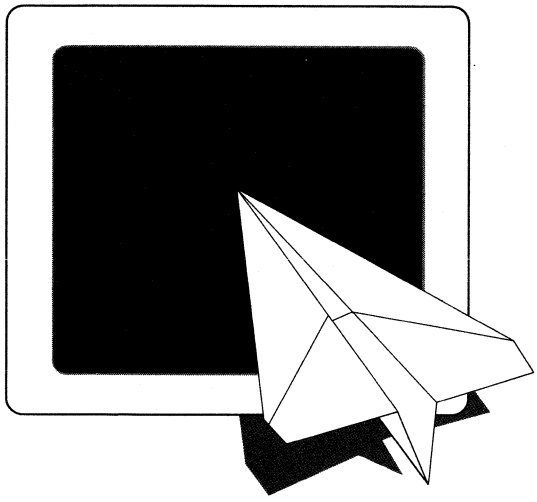


Figure 7-11 Circuit for Generating the CRC-16



XACT User Guide

Boundary Scan in XC4000 Devices

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Chapter 8

Boundary Scan in XC4000 Devices

The following information assumes that you are familiar with boundary-scan testing and the IEEE standard. Only issues specific to the XC4000 implementation are discussed in detail. For general information on boundary scan, please refer to the bibliography at the end of the chapter.

In production, boards must be tested to assure the integrity of the components and the interconnections. However, as integrated circuits have become more complex and multi-layer PC-boards have become more dense, it has become increasingly difficult to test assembled boards.

The inclusion of boundary-scan registers in ICs greatly improves the testability of boards. Boundary scan provides a mechanism for testing component I/Os and interconnections, while requiring as few as four additional pins and a minimum of additional logic in each IC. Component testing can also be supported in ICs with self-test capability.

Devices containing boundary scan have the capability of driving or observing the logic levels on I/O pins. To test the external interconnect, devices drive values onto their outputs and observe input values received from other devices. Data to be driven onto outputs is distributed through a chain of shift registers, and observed input data is returned through the same shift-register path. A central test controller compares the received data with expected results.

Data is passed serially from one device to the next, thus forming a boundary-scan path or loop that originates at the test controller and returns there. Any device can be temporarily removed from the boundary-scan path by bypassing its internal shift registers and passing the serial data directly to the next device.

XC4000 FPGA devices contain boundary-scan registers that are compatible with the IEEE Standard 1149.1, that was derived from a proposal by the Joint Test Action Group (JTAG). External (I/O and interconnect) testing is supported; there is also limited support for internal self-test.

XC4000 Boundary-Scan Features

XC4000 devices support all the mandatory boundary-scan instructions specified in the IEEE Standard 1149.1. A test access port (TAP) and registers are provided that implement the Extest, Sample/Preload and Bypass instructions. The TAP can support two User Code instructions, configure the FPGA device, and read back the configuration data.

Boundary-scan operation is independent of individual IOB configuration and package type. All IOBs are treated as independently controlled bidirectional pins, including any unbonded IOBs. Retaining the bidirectional test capability even after configuration affords tremendous flexibility for interconnect testing.

Additionally, internal signals can be captured during Extest by connecting them to unbonded IOBs or to the unused outputs in IOBs used as unidirectional input pins. This partially compensates for the lack of Intest support.

The public boundary-scan instructions are always available prior to configuration. After configuration, the public instructions and any User Code instructions are only available if specified in the design. While Sample and Bypass are available during configuration, it is recommended that boundary-scan operations not be performed during this transitory period.

Deviations from the IEEE Standard

The XC4000 boundary scan implementation deviates from the IEEE standard in that three dedicated pins — CCLK, $\overline{\text{PROGRAM}}$, and DONE — are not scanned.

Also note that the test data register contains three Xilinx test bits (BSCANT.UPD, TDO.O and TDO.T) and that bits of the register might correspond to unbonded or unused pins.

Additionally, the Extest instruction incorporates Intest-like functionality that is not specified in the standard, and system clock inputs are not disabled during Extest, as recommended in the standard.

The TAP pins (TMS, TCK, TDI and TDO) are scanned, but connections to the TAP controller are made before the boundary-scan logic. Consequently, the operation of the TAP controller cannot be affected by boundary-scan test data.

Boundary-Scan Hardware Description

The following sections describe the boundary scan hardware:

- Test Access Port
- TAP Controller
- Instruction Register
- The Boundary-Scan Data Register
- The Bypass Register
- User Registers

Test Access Port

The boundary-scan logic is accessed through the test access port (TAP), which comprises four semi-dedicated pins: test mode select (TMS), test clock (TCK), test data input (TDI) and test data output (TDO), as defined in the IEEE specification.

The TAP pins are permanently connected to the boundary-scan circuitry. However, once the device is configured, the connections can be ignored unless the use of boundary scan is specified in the design. See "Using Boundary Scan" at the middle of this chapter.

If you specify the use of boundary scan, the TAP input pins (TMS, TCK and TDI) can still be shared with other logic, subject to limitations imposed by external connections and the operation of the TAP controller. In designs that do not use boundary scan after configuration, you can use TAP pins as inputs to or outputs from the user logic in the FPGA device. TMS, TCK and TDI are available as unrestricted I/Os, while TDO only provides a 3-state output.

TAP Controller

The TAP controller is a 16-state machine that controls the operation of the boundary-scan circuitry in response to TMS. This state machine implements the state diagram specified by the IEEE standard, Figure 8-1, and is clocked by TCK.

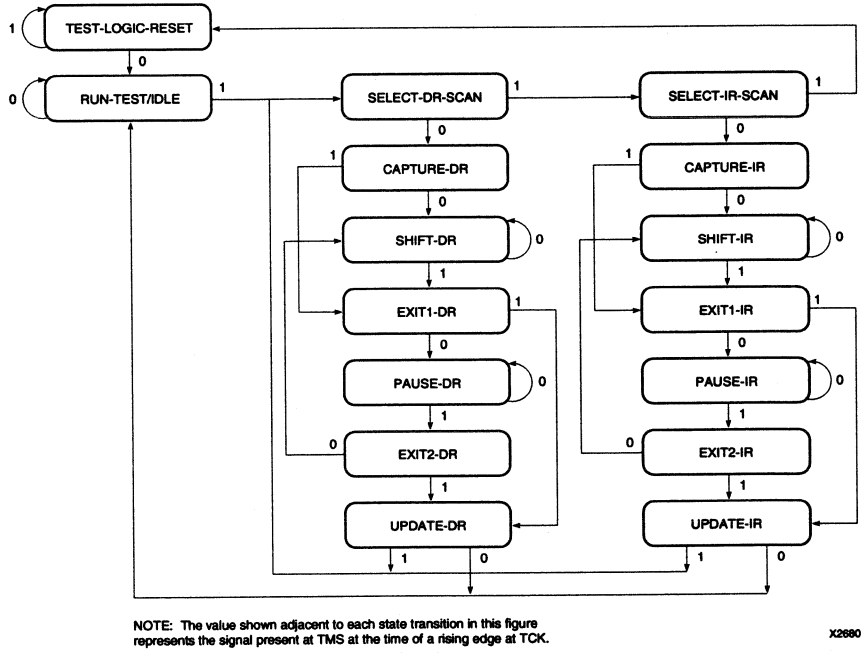


Figure 8-1 State Diagram for the TAP Controller

Upon power-up or assertion of $\overline{\text{PROGRAM}}$, the TAP controller is forced into the Test-Logic-Reset state. After configuration, the controller is disabled, unless its use is explicitly specified in your design.

Instruction Register

Loading a 3-bit instruction into the instruction register (IR) determines the subsequent operation of the boundary-scan logic, as illustrated in Table 8-1. The instruction selects the source of the TDO

pin and selects the source of device input and output data (boundary-scan register or input pin/user logic).

Note: In XC4000, whenever the TAP controller is in the Shift-DR state, all data registers are shifted, regardless of the instruction. DR data is modified even if a Bypass instruction is executed.

A 3-bit status word returned to the central test controller during an IR cycle comprises a boundary-scan availability flag bit, preceded by two mandatory bits; I0 is a one and I1 is a zero. This flag is High before and after configuration, when the full boundary-scan capability is available, and Low during configuration, when only Sample/Preload and Bypass are available.

Table 8-1 Boundary Scan Instructions

Instruction			Test Selected	TDO Source	I/O Data Source
I_2	I_1	I_0[a]			
0	0	0	EXTEST	DR	DR
0	0	1	SAMPLE/ PRELOAD	DR	Pin/Logic
0	1	0	USER 1	TDO1	Pin/Logic
0	1	1	USER 2	TDO2	Pin/Logic
1	0	0	READBACK	Readback Data	Pin/Logic
1	0	1	CONFIGURE	DOUT	Disabled
1	1	0	RESERVED	—	—
1	1	1	BYPASS	Bypass Reg	Pin/Logic

a. I_0 is closet to the TDO pin.

The Boundary-Scan Data Register

The data register (DR) is a serial shift register implemented in the IOBs of the FPGA device, as illustrated by Figure 8-2. Potentially, you can configure each IOB as an independently controlled bidirectional pin. Therefore, three data register bits are provided per IOB: for input data, output data and 3-state control. In practice, many of these bits are redundant, but they are not removed from the scan chain.

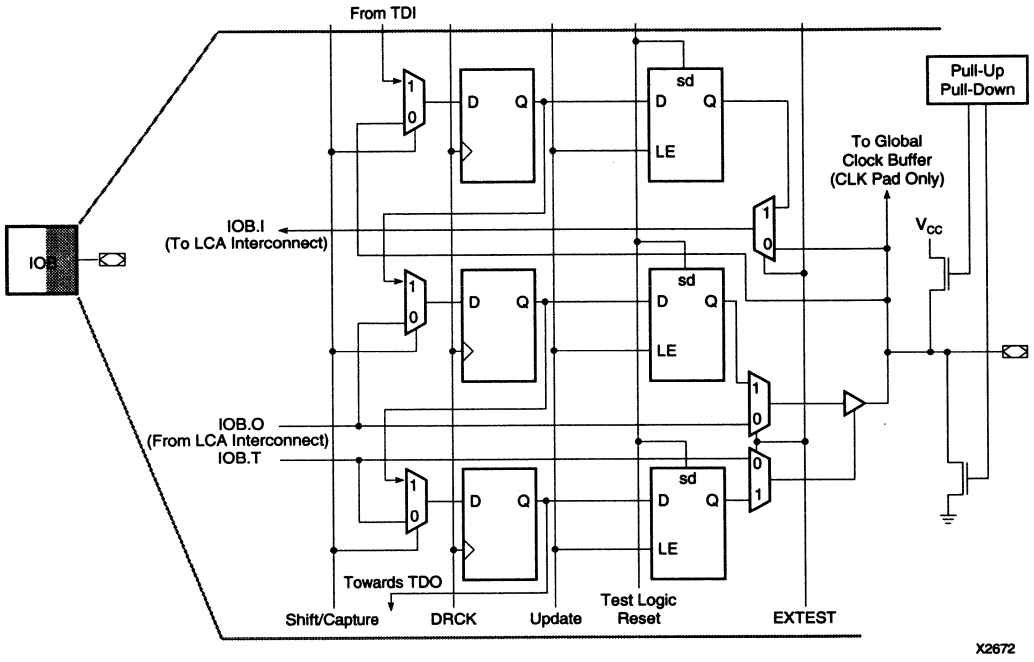


Figure 8-2 Boundary Scan Logic in a Typical IOB

An update latch accompanies each bit of the DR that is used to hold injected test data stable during shifting. The update latch is opened during the Update-DR state of the TAP controller when TCK is Low.

In a typical DR instruction, the DR captures data during the Capture-DR state (on the rising edge of TCK). This data is then shifted out and replaced with new test data. Subsequently, the update latch opens, and the new test data becomes available for injection into the logic or the interconnect. The injection of data occurs only if an Extest instruction is in progress.

Note: The update latch is opened whenever the TAP controller is in the Update-DR state, regardless of the instruction. Make sure that appropriate data is contained in the update latch prior to initiating an Extest. Any DR instruction, including Bypass, that you execute after the test data is loaded, but before the Extest commences, changes the test data.

The IEEE standard does not require the ability to inject data into the on-chip system logic and observe the results during Extest. However, this capability helps compensate for the lack of Intest. You can set logic inputs to specific levels by a Sample/Preload or Extest instruction. You can capture the resulting logic outputs during a subsequent Extest. However, all DR bits captured during an Extest might change.

Pull-up and pull-down resistors remain active during boundary scan. Before and during configuration, all pins are pulled up. After configuration, the IOB can be configured with a pull-up resistor, a pull-down resistor, or neither. Internal pull-up/pull-down resistors must be taken into account when designing test vectors to detect open circuit PC traces.

The primary and secondary global clock inputs (PGCK1-4 and SGCK1-4) are taken directly from the pins, and cannot be overwritten with boundary-scan data. However, if necessary, you can drive the clock input from boundary scan. The external clock source is 3-stated, and the clock net is driven with boundary scan data through the output driver in the clock-pad IOB. If the clock-pad IOBs are used for non-clock signals, the data can be overwritten normally.

Figure 8-3 shows the data-register cell for a TAP pin. An OR-gate permanently disables the output buffer if boundary-scan operation is selected. Consequently, it is impossible for the outputs in IOBs used by TAP inputs to conflict with TAP operation. TAP data is taken directly from the pin, and cannot be overwritten by injected boundary-scan data.

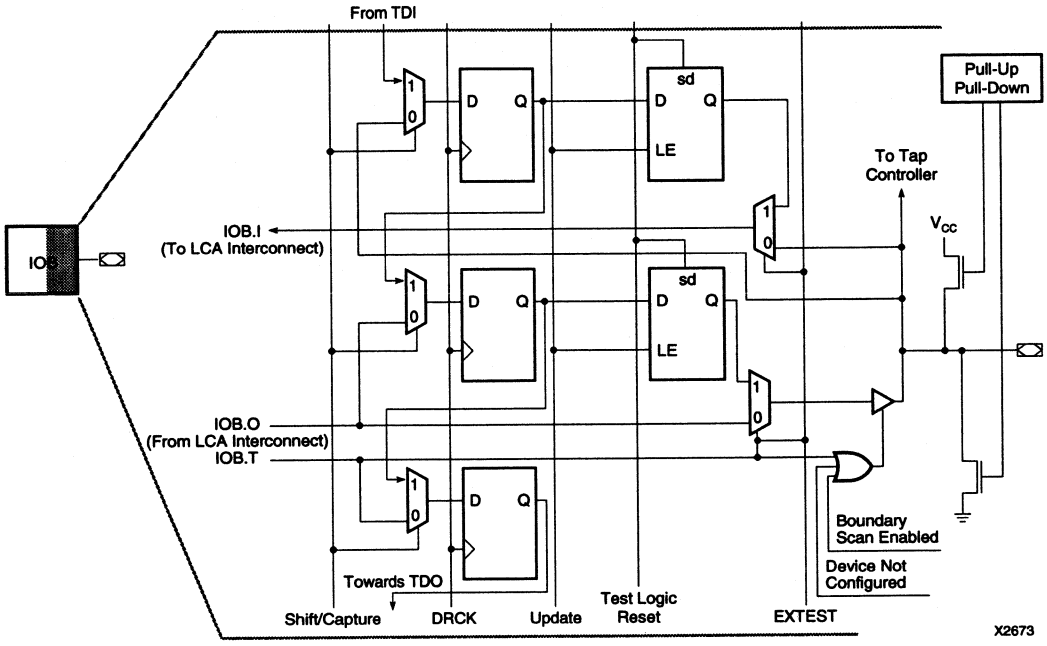


Figure 8-3 Boundary Scan Logic in a TAP Input IOB (TMS, TCK, and TDI Only)

Figure 8-4 lists, in data-stream order, the boundary-scan cells that make up the DR. The cell closest to TDO corresponds to the first bit of the data-stream, and is at the top of the table. This order is consistent with the BSDL description.

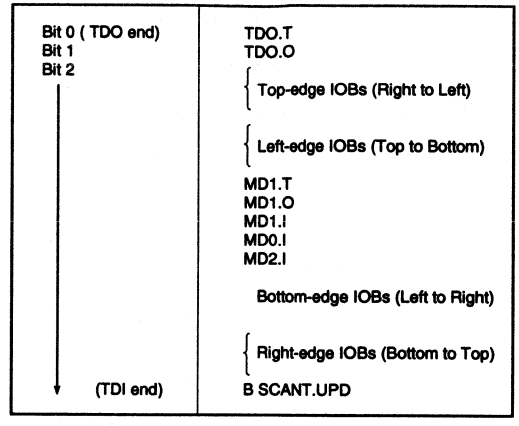


Figure 8-4 Boundary Scan Order

Each IOB corresponds to three bits in the DR. The 3-state control is first (closest to TDO), the output is next, and the input is last. Other signals correspond to individual register bits. IOB locations assume that the die is viewed from the top, as in XDE.

Note: All IOBs remain in the DR, whether they are actually used or even bonded. Three bits, BSCANT.UPD, TDO.O and TDO.T, are included for Xilinx test purposes, and can be ignored. CCLK, $\overline{\text{PROGRAM}}$ and DONE are not included in the boundary scan.

The DR also includes the following non-pin bits: TDO.T and TDO.I, which are always bits 0 and 1 of the DR, respectively, and BSCANT.UPD which is always the last bit of the DR.

The Bypass Register

This is a 1-bit shift register that passes the serial data directly to TDO when a bypass instruction is executed.

User Registers

The XC4000 boundary-scan instruction set includes two User Code instructions, User1 and User2. Connections are provided to the TAP and TAP controller that, together with direct connections to the TAP pins, permit you to include boundary-scan self-test features in the design.

The boundary scan block has six connections for user registers: SEL1, SEL2, TDO1, TDO2, DRCK and IDLE. TDI is available directly from the IOB that provides the TDI pin.

Note: The TDI signal supplied to user test logic is overwritten by boundary-scan test data during Extest. During user tests, it is not altered.

Table 8-2 Connections for User Registers

Connection	Description
SEL1, SEL2	They enable user logic. They are asserted (High) when the instruction register contains instructions User1 and User2, respectively.
TDO1, TDO2	They are inputs to the TDO output multiplexer, permitting user access to the serial boundary-scan output. They are selected when executing the instructions User1 and User2, respectively. Input to user data registers can be derived directly from the TDI pin, thus completing the boundary-scan chain. There is a one flip-flop delay between TDO1/TDO2 and the TDO output. This flip-flop is clocked on the falling edge of TCK.
DRCK	The data register clock (DRCK) is a gated and inverted version of TCK. It is provided to clock user test-data registers. TDI data should be sampled with the falling edge of DRCK (rising edge of TCK). The TDO output flip-flop accepts data on the rising edge of DRCK (falling edge of TCK). DRCK is active only during the Capture-DR and Shift-DR states of the TAP controller. While inactive, DRCK is parked High.
IDLE	It is a second gated and inverted version of TCK. It is active during the Run-Test/Idle state of the TAP controller, and may be used to clock user test logic a set number of times, determined through TMS by the central test controller.

Using Boundary Scan

This section covers the following topics:

- Boundary Scan Availability
- Post-Configuration Boundary-Scan Operation
- XC4000 Boundary-Scan Instructions

Boundary Scan Availability

Access to the built-in boundary-scan logic is always available between power-up and the completion of configuration. Optionally, the built-in logic is fully available after configuration if boundary scan is specified in the design. At this time, user test logic is also available, and can be accessed through the boundary-scan port. During configuration, a reduced boundary-scan capability remains available: the Sample/Preload and Bypass instructions only.

Figure 8-5 is a flow chart of the FPGA start-up sequence that shows when the boundary-scan instructions are available.

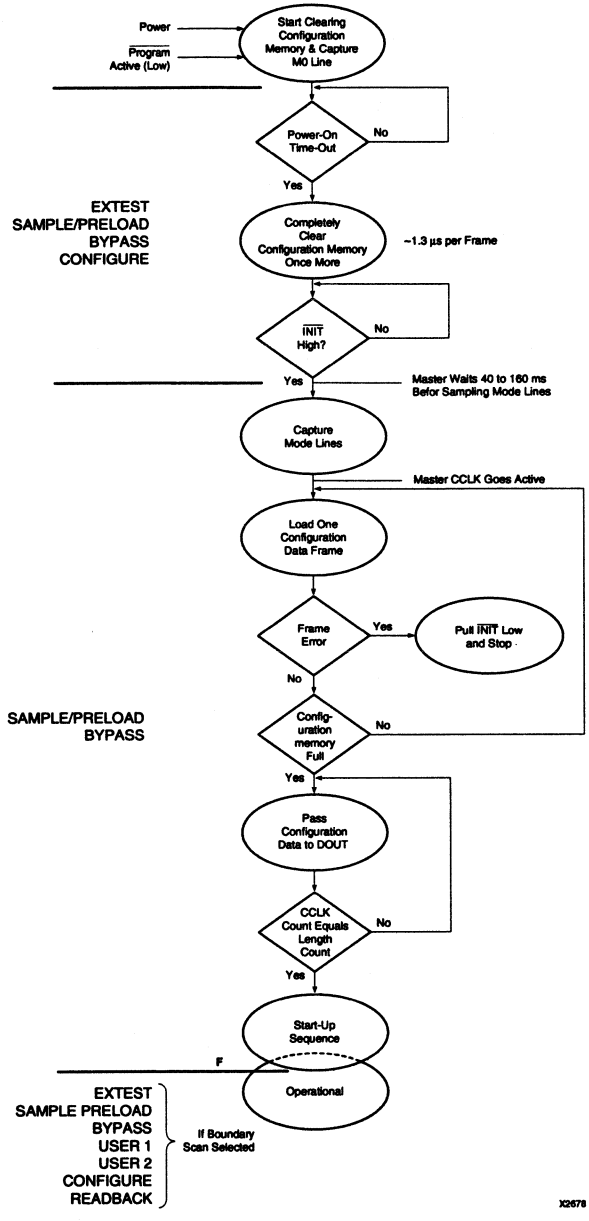


Figure 8-5 Start-up Sequence

Extest and Configure capabilities are available until $\overline{\text{INIT}}$ is High. Without external intervention, $\overline{\text{INIT}}$ automatically goes High after ~1 ms. If more time is required for boundary-scan testing, $\overline{\text{INIT}}$ can be held Low beyond this period by applying an external Low signal to the $\overline{\text{INIT}}$ pin until testing is complete. As $\overline{\text{INIT}}$ is an open-drain output with a 40K-100K pull-up, a resistor can be sufficient to hold it Low.

Warning: The $\overline{\text{INIT}}$ pin is included in the boundary-scan coverage, and can be replaced by boundary-scan test data during an Extest instruction. When performing an Extest prior to configuration, make sure that boundary-scan input data does not force $\overline{\text{INIT}}$ High; this condition terminates the Extest.

Sample/Preload and Bypass are available unless the device has been configured and Boundary Scan disabled. Since the duration of the configuration period is determined by the configuration process, which you cannot externally control, Xilinx recommends that you do not use this period for boundary-scan operations.

If you are not going to use boundary scan after configuration, delay the start of configuration until all boundary-scan operations are complete. This delay can be achieved by controlling $\overline{\text{INIT}}$ as described above. If you enable boundary scan after configuration, you can conduct unrestricted boundary-scan operations once the configuration process is complete.

The exact point at which you can resume boundary-scan operations after configuration (point F) depends upon the configuration mode. It is the point defined as Finished in the configuration timing diagram found in the Start-up section of the XC4000 Data Sheet. Refer to *The Programmable Logic Data Book* for the XC4000 Data Sheet.

The period of reduced boundary-scan availability during configuration is identified by a flag in the status word that is returned through the boundary-scan path by the Capture IR preceding a Shift IR whenever an instruction is loaded into the IR. The flag is High prior to $\overline{\text{INIT}}$ being High and after the configuration is finished, and Low when only Sample/Preload and Bypass are available. See the "Instruction Register" section at the beginning of this chapter.

Post-Configuration Boundary-Scan Operation

In a configured FPGA device, the boundary-scan logic might or might not be active depending on the configuration data loaded into the part. Activation of the boundary-scan logic, if desired, is part of the FPGA design process. After configuration, boundary scan cannot be activated or de-activated without changing the configuration.

Figure 8-6 illustrates the typical connections to the boundary scan schematic symbols.

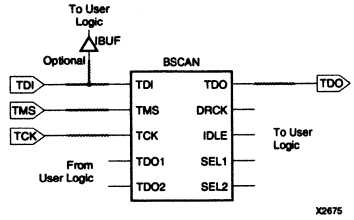


Figure 8-6 Boundary-Scan Schematic Symbols

If the BSCAN primitive is not included, boundary scan is not selected, and the IOBs used by the TAP input pins are freely available to PPR as general purpose IOBs. You can use the TDO output pin as a logic output by explicitly connecting the TDO pad primitive to an OBUF or OBUFT as required. Figure 8-7 illustrates a non boundary-scan TDO connection.

Note: Until configuration without boundary scan is complete, the boundary-scan logic is active and can respond to random inputs on TCK, TMS, and TDI.

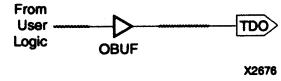


Figure 8-7 Typical Non-Boundary-Scan TDO Connection

You can also select boundary scan in XDE. You can use the EditBlk command to change the configuration of the BSCAN block, found in the top left corner of the die. In the Configuration Options box, select USED so that it is highlighted. The TAP pins are permanently connected to the BSCAN block, although the connections are not

explicitly shown. You can make connections to user test logic using XDE, if required.

XC4000 Boundary-Scan Instructions

The XC4000 boundary scan supports three IEEE-defined instructions: Extest, Sample/Preload and Bypass; two user-definable instructions: User1 and User2; and two FPGA-specific instructions: Configure and Readback. The instruction codes are shown in Table 8-1 at the beginning of this chapter.

The following sections describe each XC400 boundary-scan instruction in detail.

Extest

While the Extest instruction is present in the instruction register, the user 3-state control and data presented to the device output buffers is replaced by data loaded through the boundary-scan data register and stored in the update latch, as illustrated in Figure 8-8.

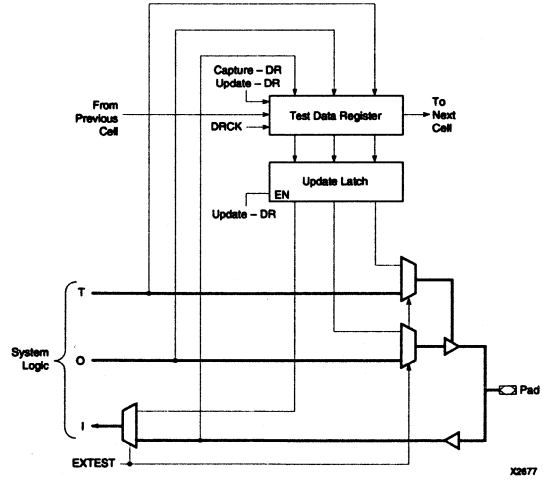


Figure 8-8 Extest Data Flow

When a shift data register operation is performed, data arriving at the device input pins is loaded into the data register. The data from the system logic that drives output buffers and their 3-state controls is

also loaded. This action occurs during the Capture-DR state of the TAP controller, as illustrated in Figure 8-1. Data is serially shifted out of the DR during the Shift-DR state as new data is shifted in. In the Update-DR state, the data shifted into the data register is transferred into the update latch for use as I/O control.

The replacement of system data with update latch data happens while the Extest instruction is in the instruction register. For this data to be valid, it must have been loaded at the start of the instruction by a previous Extest or Sample/Preload operation.

Since the data register and update latch are modified during any data register operation, including Bypass, the data in the update latch is only valid if it was loaded in the last data register operation before an Extest is performed.

The IEEE definition of Extest only requires that test data be driven onto outputs, that 3-state output controls be overridden, and that input data be captured. The capture of output data and 3-state controls and the forcing of test data into the system logic is normally performed during Intest.

The XC4000 effectively performs Extest and Intest simultaneously. This functionality permits the testing of internal logic and compensates for the absence of a separate Intest instruction. However, when performing an Extest, you must decide carefully what signals are driven into the system logic; data captured from internal system logic must be masked out of the test-data stream before performing check-sum analysis.

Sample/Preload

The Sample/Preload instruction permits visibility into system operation by capturing the values of the I/O signals. It also permits valid data to be loaded into the update register before commencing an Extest.

The DR and update latch operate exactly as in Extest, described above. However, user logic operate the IOB.

Bypass

The Bypass instruction permits data to be passed synchronously to the next device in the boundary-scan path. There is a one-bit shift register between the TDI and TDO flip-flop.

User1 and User2

These instructions permit test logic, designed by you and implemented in CLBs, to be accessed through the TAP. Test clocks and paths to TDO are provided, together with two signals that indicate that user instructions have been loaded. For details, see the "User Registers" section in the middle of this chapter.

User tests depend upon CLBs and interconnect that must be configured to operate. Consequently, they can only be performed after configuration.

A simple example of a User instruction might be a 3-state instruction. You can define your own User instruction. You could have a flip-flop, clocked by DRCK whose data input is connected to Vcc, driving the GSR net. Therefore, when the flip-flop is clocked, the GSR net 3-states all the IO.

In Figure 8-9, the clock goes from DRCK to K, and the Reset-Direct goes from SEL1 to RD if the User1 instruction is used.

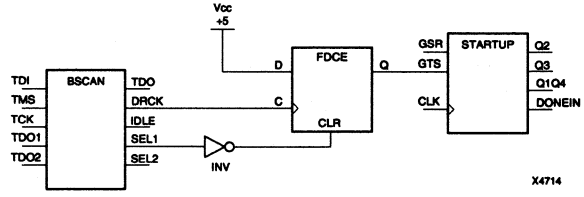


Figure 8-9 3-state Instruction

Configure

From the update of a configure instruction until the next positive edge of TCK, an internal signal is ORed with $\overline{\text{PROGRAM}}$. The device continues to clear its memory, make HDC High, and pull LDC, $\overline{\text{INIT}}$, and DONE Low.

After the TCK edge has time to clear the memory, the $\overline{\text{INIT}}$ is released and the devices accepts configuration data. The logic uses TCK and TDI in place of CCLK and Din (as in slave serial). There is no convenient boundary scan test of $\overline{\text{INIT}}$. You must monitor it externally or do a time-out.

The configure instruction must be updated when $\overline{\text{PROGRAM}}$ is High and $\overline{\text{INIT}}$ is Low prior to configuration or after a configuration is finished and boundary scan is enabled. The status of these conditions is indicated by a "1" in the third bit of an instruction capture. Configure, readback, and Extest instructions are cleared by assertion of $\overline{\text{PROGRAM}}$. All instructions are cleared by Test-Logic-Reset, which is initialized as power is applied.

The error checking is done at the end of each data frame by the device loading that frame. Daisy chained devices do not look at data they pass through to other devices. Preamble "ones" before the first "0010" only affect the length count. Any extra bits due to bypassed parts up stream require an increase in length count (or chop off one of the eight dummy preamble ones).

Readback

Readback permits the configuration data of an FPGA device to be read back through the TAP. This instruction differs from other boundary instructions in two ways.

- The readback logic is triggered (equivalent to Capture-DR) during the Update-IR state when the Readback instruction is loaded. To re-trigger the readback logic, some other boundary-scan instruction must first be loaded, and then the Readback instruction reloaded.
- DI does not connect to the input end of the Readback shift register. Consequently, data from upstream devices is lost.

For details on the readback bitstream, refer to "Verification by Readback and Signature Analysis" in this user guide.

Note: Readback data can be captured by a user net (trigger) using the READBACK symbol. This data can then be shifted out by holding the user trigger active (High), later loading the instruction register with the readback instruction, and shifting the data out by performing a shift data register.

Boundary Scan Description Language Files

Boundary Scan Description Language (BSDL) files describe boundary-scan-capable parts in a standard format used by automated test generation software. The order and function of bits in the boundary-scan data register are included in this description.

BSDL files for XC4000 devices can be obtained from your Xilinx Field Applications Engineer (FAE) or by calling the Xilinx Applications Hotline. These files can also be downloaded from the Xilinx Technical Bulletin Board (BBS), and have file names *device*.bsm.

Boundary Scan Bibliography

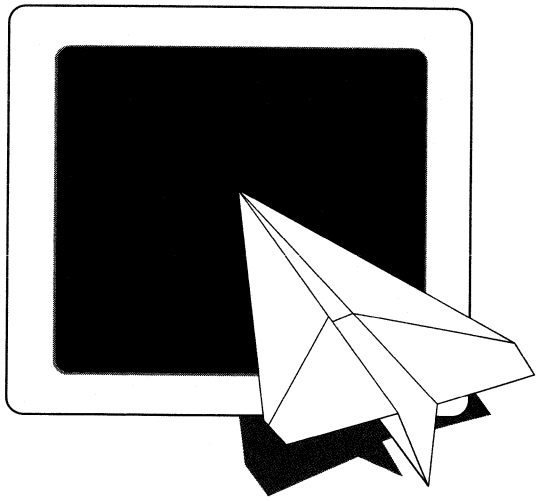
The following publications contains information about the IEEE Standard 1149.1. Consult them for general boundary-scan information beyond the scope of this chapter.

Colin M. Maunder & Rodham E. Tulloss. *The Test Access Port and Boundary Scan Architecture.* IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. BOX 3014, Los Alamitos, CA 90720-1264.

John Fluke Mfg. Co. Inc. *The ABC of Boundary Scan Test.* John Fluke Mfg. Co. Inc., P.O. BOX 9090, Everett, WA 98206.

GenRad Inc. *Meeting the Challenge of Boundary Scan.* GenRad Inc., 300 Baker Ave., Concord, MA 01742-2174.

Ken Parker. *The Boundary Scan Hanbdbook.* Kluwer Academic Publications, (617) 871-6600.



XACT Design Editor Tutorial

XACT User Guide

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Chapter 9

XACT Design Editor Tutorial

The XACT Design Editor (XDE) contains several programs that enable you to make changes to an existing design or to create a new design. This tutorial demonstrates the EditLCA program, a device-level graphical editor for your FPGA designs.

Warning: Your schematic does not reflect the changes you made using XDE; therefore, you should document any changes to placement and logic on your schematic. Indicating the changes on the schematic ensures the integrity of the design and keeps all changes documented in the design.

Generally, you will use XDE to look at timing information found in the XDelay program or to make changes to an existing FPGA design file.

This tutorial serves as an introduction to some of the more useful commands in XDE. It is a presentation of the XC3000 FPGA family; however, you can apply many of the concepts to the XC2000 and XC4000 families. The exercises presented in this tutorial familiarize you with the Logic Cell Array (LCA) architecture and show you how to take advantage of this information when using XDE. For detailed information about XDE, refer to "The XACT Design Editor" chapter of the *XACT Reference Guide, Volume 3.*

This chapter contains the following sections:

- "Getting Started" describes the notational conventions used in this tutorial. It also steps through the installation and set-up procedures necessary to get XDE running in the correct mode.
- "High-level Editing" contains examples that use some of the design editor higher-level commands.

- "Low-level Editing" describes the different types of routing resources in an FPGA and how to use them effectively.
- "Building a Four-bit Multiplier" involves entering a design using XDE.
- "Downloading a Bitstream" shows how to download a design to the demo board.

Getting Started

In XDE, you can enter commands and information in many different ways. For consistency, this manual uses specific notational conventions, which are described in the "Preface" at the beginning of this manual.

Before you can begin the XDE tutorial, you must perform the tasks described in the following sections:

- Setting the Mode
- Setting the Directory
- Choosing the Design File
- Loading the Design

Once you load the design, you can begin the tutorial by performing the following tasks:

- Viewing the Logic Cell Array
- Setting the Profile
- Using the EditBlk Display

Begin XDE by entering:

```
xact ↵
```

The XDE program loads, and a message appears stating that the xact.pro file is being read. This file contains user-preferred settings. Rather than having to choose the same options at the beginning of every session, these settings can be selected once and saved in the xact.pro file using the SAVEPROFILE command found in the PROFILE pull-down menu. The next time you start XDE, these options are the default settings.

The Executive Display appears on the screen, as shown in Figure 9-1.

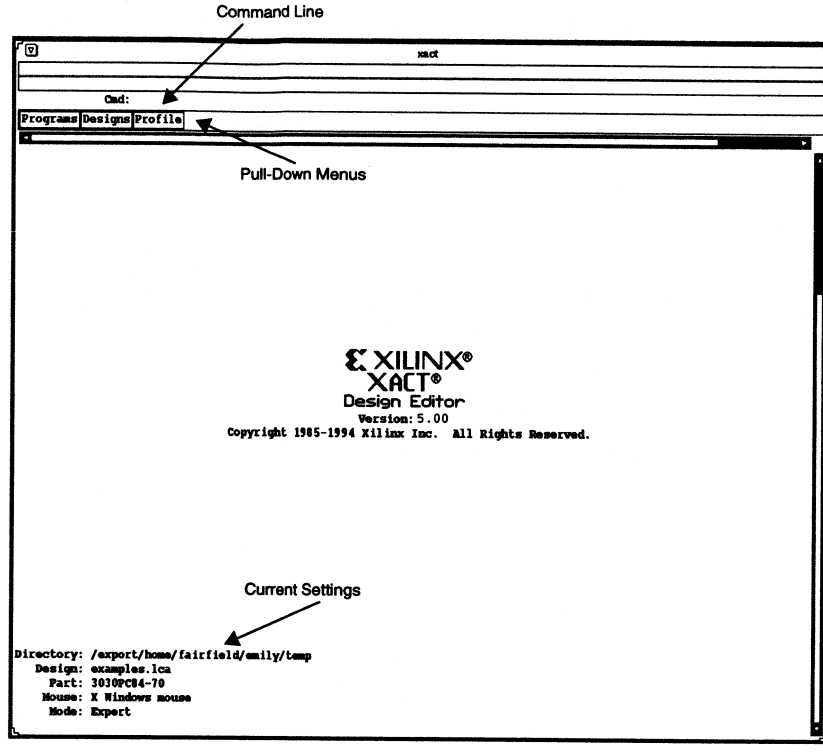


Figure 9-1 The Design Editor Executive Screen on a Sun Workstation

Setting the Mode

You must run XDE in Expert mode to perform this tutorial. If you are not already in Expert mode, do the following:

1. With the left mouse button, click on the current mode. XDE displays a dialogue box with the two mode options: Expert or Safe.
2. Click once on **Expert**, which highlights your selection.
3. Select **Done** to close the dialogue box and return you to the main screen.

Setting the Directory

The path of the current directory is shown on the directory line of the current settings section. To switch to the directory in which the tutorial file is located, refer to the appropriate subsection that follows.

PCs

To access the tutorial files, follow these steps.

1. With your left mouse button, select **Designs** → **Directory** from the menu. The system displays a dialogue box that enables you to change the current directory without exiting the program.
2. Set the path so that the current working directory reads:

   C:\XACT\TUTORIAL\CORE\EDIT_LCA

3. Then select **Done** to change directories.

All Other Platforms

To access the tutorial files, follow these steps.

1. With your left mouse button, select **Designs** → **Directory** from the menu. The system displays a dialogue box that enables you to change the current directory without exiting the program.
2. Set the path so that the current working directory reads:

   $XACT/TUTORIAL/CORE/EDIT_LCA

3. Then select **Done** to change directories.

Choosing the Design File

This section describes how to load the edit_lca.lca file, which is used in the first part of this tutorial. Perform the following steps:

1. With your left mouse button, select **Designs** → **Design** from the main XDE menu.
2. Indicate the name of the sample design file, edit_lca.lca, in one of the following ways:
   - From the Select File command line, type **edit** ↵

- From the pop-up window. Select **edit_lca.lca** and then select **Done**.

The file name appears on the Design Line of the current settings section.

Loading the Design

To load the design, select **Programs** → **Editlca** from the menu with your left mouse button. XDE loads the design called "example," reads the editlca.pro profile, and displays the LCA Editor screen.

Viewing the FPGA

The EditLCA Screen, shown in Figure 9-2, is a graphical representation of the FPGA. It consists of an array of configurable logic blocks (CLBs), configurable input/output blocks (IOBs), 3-state buffers (TBUFs), and programmable interconnect.

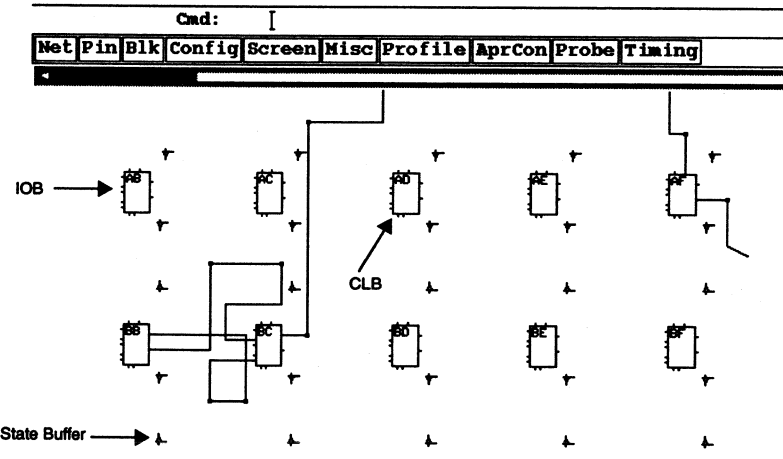


Figure 9-2 The EditLCA Screen

The following figure illustrates the input and output pins for these block types.

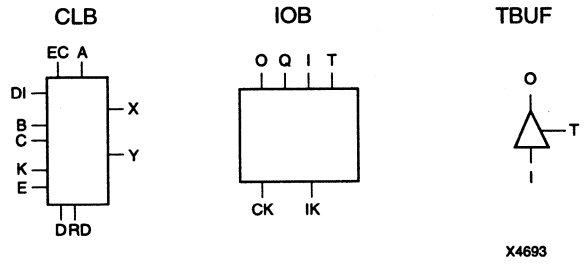


Figure 9-3 Block Types

Definitions

Xilinx uses the following definitions:

Block	A logic component of the design, for example, CLB or IOB.
Pin	Any input or output of a block.
Net	A set of pins that should be interconnected in the final design.

Each CLB has a physical location name defined by its horizontal row position followed by its vertical column position, for example, JC. Each IOB has a location name defined by its package pin number, for example, P36. When you use blocks in a design, "logical" names, which describe the block function, replace these location labels.

The EditLCA Screen

Perform the following steps to familiarize yourself with the EditLCA screen:

1. Press a mouse button and move the mouse; the world view appears in the lower right corner of the screen and shows the entire chip.

   The red rectangle encases the part of the FPGA that currently appears on the screen.

2. Continue moving the mouse with a button pressed to pan over the rest of the design.

The unused logic cell resources are shown in green and the used blocks are shown in yellow.

3. Release the mouse button. The world view window disappears.
4. Pan over the design until you find a set of configured logic blocks.
5. Place the cursor on a used input pin of the CLB, and note that the cursor status line displays information about that pin.

Setting the Profile

When you run the EditLCA program, XDE reads the editlca.pro file. This file contains default settings for the mouse buttons, function keys, screen settings, and color settings. Since there are so many available options, this section shows you how to set some of these options so that you can customize the editor to your own personal tastes.

This section explains how to change your profile for the following options:

- Setting the mouse buttons
- Defining the function keys

Setting the Mouse Buttons

To set the mouse buttons, perform the following steps:

1. With your left mouse button, select **`Profile`** → **`Mouse`** from the EditLCA menu.

   A pop-up window appears that lists the left, middle, and right mouse buttons, which are designated by B1, B2, and B3, respectively.

2. To set the left mouse button, select B1 from the pop-up window. Another pop-up window appears that lists the functions you can assign to the mouse buttons.
3. Choose **`Select`**. From now on, when XDE prompts you to select something, put the cursor on the item and push the left mouse button.

   Once you choose a function in this pop-up window, XDE returns you to the pop-up window that lists the three mouse buttons.

4. To set the middle mouse button, select B2.
5. Choose **Done** from the pop-up window. From now on, to finish a command, either select Done from the top of the screen or press the middle mouse button.
6. To set the right mouse button, select B3 from the pop-up window.
7. Select **Switch**.

   From now on, the right mouse button performs the Switch command. When using the EditBlk command, clicking this button switches between the EditLCA screen and the EditBlk screen.
8. Now that you have defined all three mouse buttons, select **Done** to exit the pop-up window.

Defining the Function Keys

Next, set the function keys. You can define function keys to perform frequently used commands. Instead of repeatedly typing commands or searching through menus, you can define the function keys to perform these commands at the push of a button as follows:

1. Select **Profile** → **Keydef** from the EditLCA menu.

   The system displays a pop-up window that lists all available function keys.

Note: The F1 key is reserved for Help, so start the definitions with the F2 key.

2. Select the specific function key in the pop-up window, for example, **F2**. The system closes the pop-up window and positions the cursor on the Enter Key Definition command line.
3. Type the command that you want to map to this function key as follows:

   editnet ↵

   This process defines F2 to perform the EditNet command.

You can also define function keys by entering the following syntax directly on the command line:

keydef f3 routepin ↵

To easily create function key mappings using the keyboard, perform the following steps:

1. Use the up arrow key to bring the last command back to the command line.
2. Use the left arrow key to move back along the line.
3. Edit the command so that it reads:

   keydef f4 unroutepin ↵

   This way you can enter similar commands without having to re-type the entire line.
4. Set the rest of the function keys according to Table 9-1. These commands are explained in the following sections. At this time you might also want to write these key definitions on a strip of paper and place them by your keyboard.
5. When you are finished, save your profile by typing:

   saveprofile ↵

From now on, any time you use XACT in this directory, these mouse button and function key settings are the default.

Table 9-1 Function Key Settings

Function Key	Definition
F5	SwapBlk
F6	SwapSig
F7	MoveBlk
F8	EditBlk
F9	Route
F10	Unroute
F11	QueryNet
F12	Dos

Using the EditBlk Display

This next section describes XDE's graphical representation of the interior of IOBs and CLBs. When you edit a block, the display switches from the EditLCA screen, which shows the exterior of all blocks, to the EditBlk screen, which displays the interior of a single block. To practice using the EditBlk display, perform the following steps:

1. Move the cursor so that the lower left corner of the chip is displayed on the screen.
2. With your left mouse button, select **`Blk`** → **`EditBlk`** from the menu or select the appropriate function key, in this case, F8.

   This choice allows you to view the current setup of the chosen block and to edit the logic functions performed by that block. The command line displays:

   ```
   Select block:
   ```

3. Choose block P36 by either selecting it with the mouse or entering the block's name using the keyboard.

   The screen now displays the Block Editor display for I/O block P36, as shown in Figure 9-4.

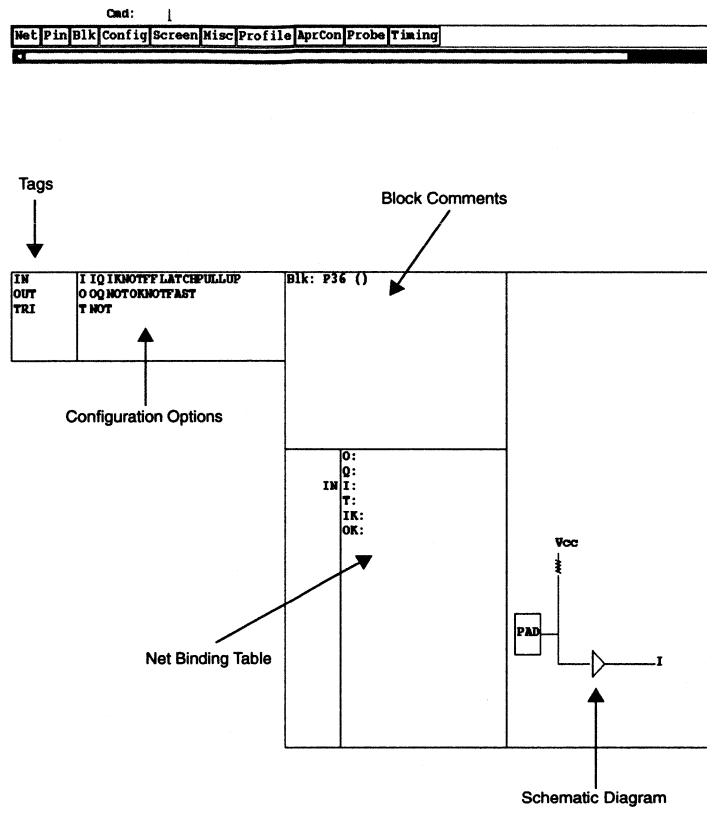


Figure 9-4 The EditBlk Screen (IOB)

You can switch the display between the EditBlk screen and the EditLCA screen by any of the following methods:

- Select **`Screen`** → **`Switch`** from the EditBlk menu.
- Enter `switch` on the command line.
- Press the right mouse button, which you defined as the Switch button.

4. Switch to the EditLCA screen. Block P36 is now colored red. This denotes that P36 is the active block — the block that appears when the switch command is executed.

To edit another block, perform the following steps:

5. With your left mouse button, select **Blk** → **EditBlk** from the menu. The command line displays

   ```
   Select block:
   ```

6. Type the desired block label, as follows:

   jc ↵

   The screen changes to display the Block Editor display for block JC, as shown in Figure 9-5.

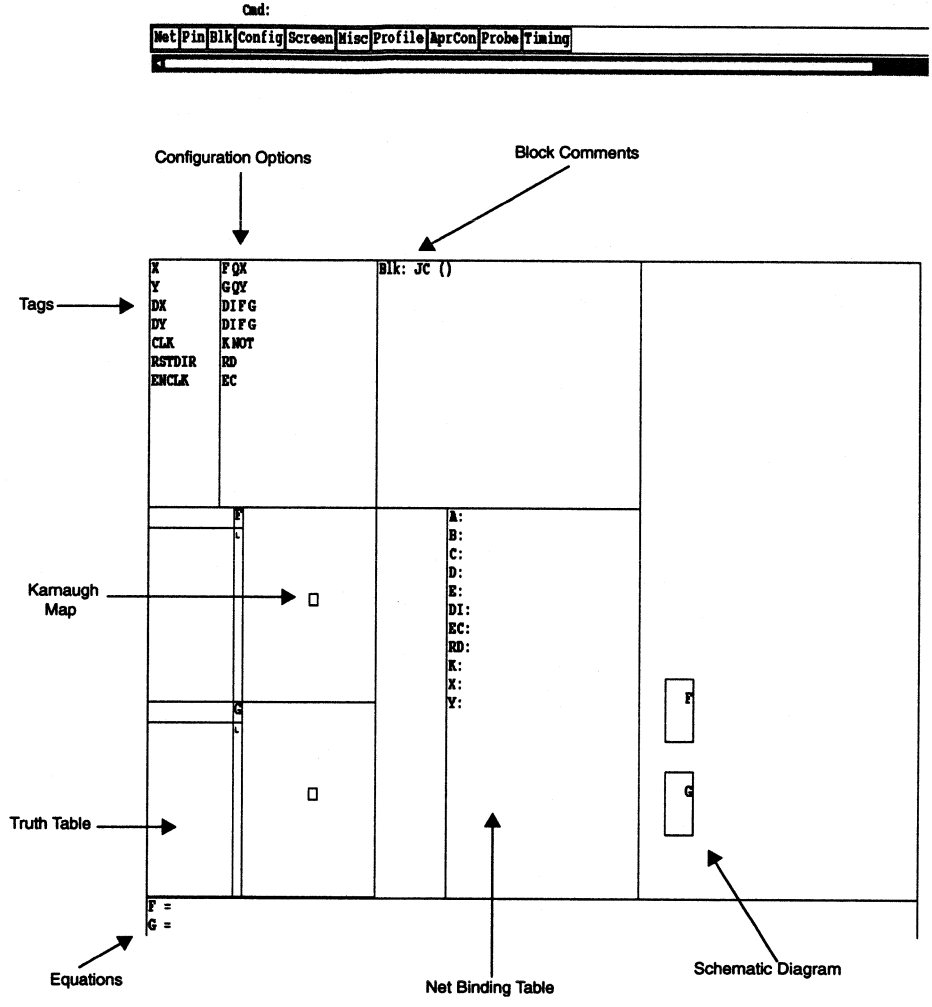


Figure 9-5 The EditBlk Screen (CLB)

7. Select each configuration option by clicking on it with the left mouse button.

   XDE updates the schematic diagram area to reflect the change. In the configuration area, the system highlights enabled tags in yellow and unused tags are shown in green. Not only can you

change the placement and routing of a design, but also edit the functionality of your design at the CLB level.

The following table describes the functions of these tags.

Table 9-2 Tag Definitions

Tag	Option	Description
X	F	The X tag refers to the top output from a CLB (refer to Figure 9-3). Choosing F sets X equal to the output of the F function generator.
	QX	Choosing QX sets X equal to the output of the top flip-flop (QX).
Y	G	The Y tag refers to the bottom output from a CLB. Choosing G sets Y equal to the output of the G function generator.
	QY	Choosing QY sets Y equal to the output of the bottom flip-flop (QX).
DX	DI	The DX tag refers to the input of the top flip-flop. Choosing DI sets DX equal to the direct input pin of the CLB.
	F	Choosing F sets DX equal to the output of the F function generator.
	G	Choosing G sets DX equal to the output of the G function generator.
DY	DI	The DY tag refers to the input of the top flip-flop. Choosing DI sets DY equal to the direct input pin of the CLB.
	F	Choosing F sets DY equal to the output of the F function generator.
	G	Choosing G sets DY equal to the output of the G function generator.
CLK	K	The CLK tag refers to the clock input of both storage elements. Choosing K sets the clock inputs of the flip-flops equal to the K pin of the CLB. The flip-flops are clocked on the rising edge of K.
	NOT	Choosing NOT sets the clock inputs of the flip-flops equal to the complement of the K pin.

Tag	Option	Description
RSTDIR	RD	The RSTDIR tag refers to the active-high reset direct input of both flip-flops. Choosing RD sets the reset input of both flip-flops equal to the RD pin of the CLB. If this tag is not selected, the RD input is disabled.
ENCLK	EC	The ENCLK tag refers to the active-high clock enable input of both flip-flops. Choosing EC sets the clock-enable input of both flip-flops equal to the EC pin of the CLB. If this tag is not selected, the EC input is disabled.

Figure 9-6 shows an example of a configured CLB with a user-assigned block name, comments in the top center box, and pin names in the lower center of the display.

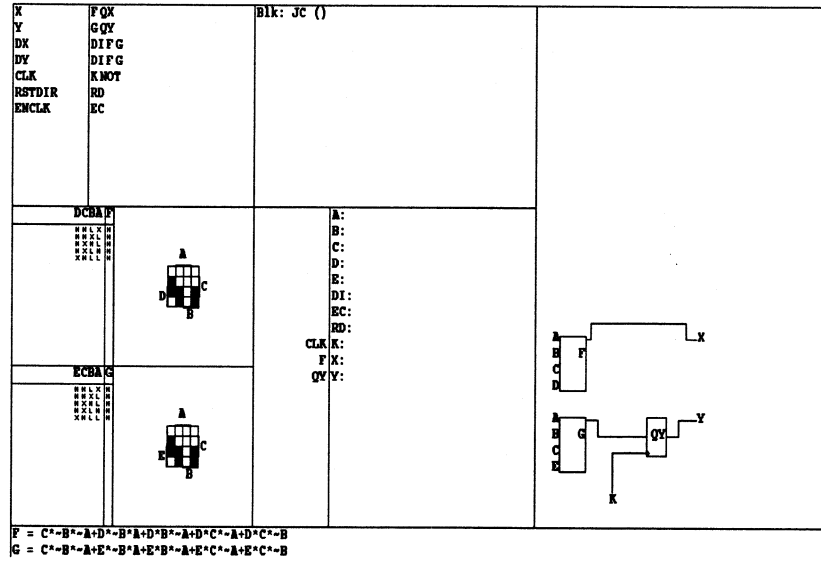


Figure 9-6 A Configured CLB

To configure CLB JC, perform the following steps:

1. Type the following equation on the command line:

   f=a*b@d ↵

   Both the truth table and Karnaugh map representation appear for equation F.

2. Move the cursor to the Karnaugh map, select a square, and click on it with the left mouse button. Notice that the equation reflects the change.
3. Type the following on the command line to terminate the Block Editor and return to the LCA Editor graphics display.

   endblk ↵

EndBlk is different from Switch. EndBlk ends the EditBlk sequence; it does not leave block JC colored red. If you try a switch now, nothing happens since there is no longer an active block.

High-level Editing

Now that you are familiar with the EditLCA and some of its features, you can now perform some editing. High-level editing involves using XDE's higher-level block and signal commands to structure or restructure a design. High-level editing requires knowledge of the following commands:

- SwapSig
- SwapBlk

SwapSig

The SwapSig command interchanges the net connections and block functionality of two pins. In other words, the attributes of the two pins are traded, but the function of the design remains unchanged.

The following figures show the results of performing a SwapSig.

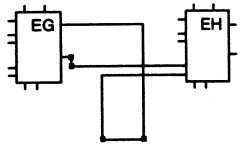


Figure 9-7 Before SwapSig

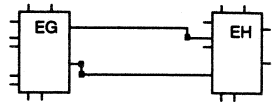


Figure 9-8 After SwapSig

You can easily reroute nets using this command, which can reduce timing delays and free interconnect resources. To understand the usefulness of SwapSig, perform the following steps:

1. Type the following on the command line:

   find bb ↵

   The system positions the cursor on CLB BB.

Note: You can also use Find to locate IOBs and nets.

2. Select **Net** → **QueryNet** from the EditLCA menu.

   The Querynet pop-up window appears. The command line displays the following:

   ```
   Select option(s) or net(s):
   ```

3. To select nets using the mouse, put the cursor on a pin of the net and press the left button. Click on the following pins:

BB.X	Selects net net0.
BB.Y	Selects net net1.

4. When finished select **Done**.

   The screen switches to text mode and displays information about net0 and net1.

5. Press any key to return to the graphics screen.

   Figure 9-9 shows the screen display. Note the timing of the nets.

```
– net0 . . . . .     BB.X . . . . . .  2.5 BC.E
– net1 . . . . .     BB.Y . . . . . .  2.5 BC.B
```

Figure 9-9 Querynet Display

6. Now swap the two signals by selecting **Pin → SwapSig** from the menu. The command line displays

   ```
   From pin:
   ```

7. Type the following on the command line:

   BB.X ↵

   The Status Line displays `Pin BB.X Selected`. The command line displays the following:

   ```
   To pin:
   ```

8. Choose the pin for which you want to swap net connections and block functionality. For this example, enter the BB.Y pin on the command line as follows:

   BB.Y ↵

   The net connections and the CLB functionality of the two pins are interchanged. The message flashes the three following messages:

   ```
   Pin BB.X and BB.Y: Signals swapped.
   Pin BB.X: Routed.
   Pin BB.Y: Routed.
   ```

9. Now perform another QueryNet, as illustrated by Figure 9-10, on the same two nets, and compare the new timing delays to the previous delays, as illustrated in Figure 9-9. In this case, exchanging the pins results in a shorter routing path for net1, reducing its delay time.

   ```
   – net0 . . . . .     BB.Y . . . . . .  2.5 BC.E
   – net1 . . . . .     BB.X . . . . . .  1.0 BC.B
   ```

Figure 9-10 QueryNet Display

You can swap most pins except, for example, the .K (clock) pin and the .DI (direct input) pin. You cannot swap input pins with output pins; and depending on how the CLB is configured, you cannot directly exchange some of the general input pins with each other.

Figure 9-11 shows all the configurations possible in a CLB.

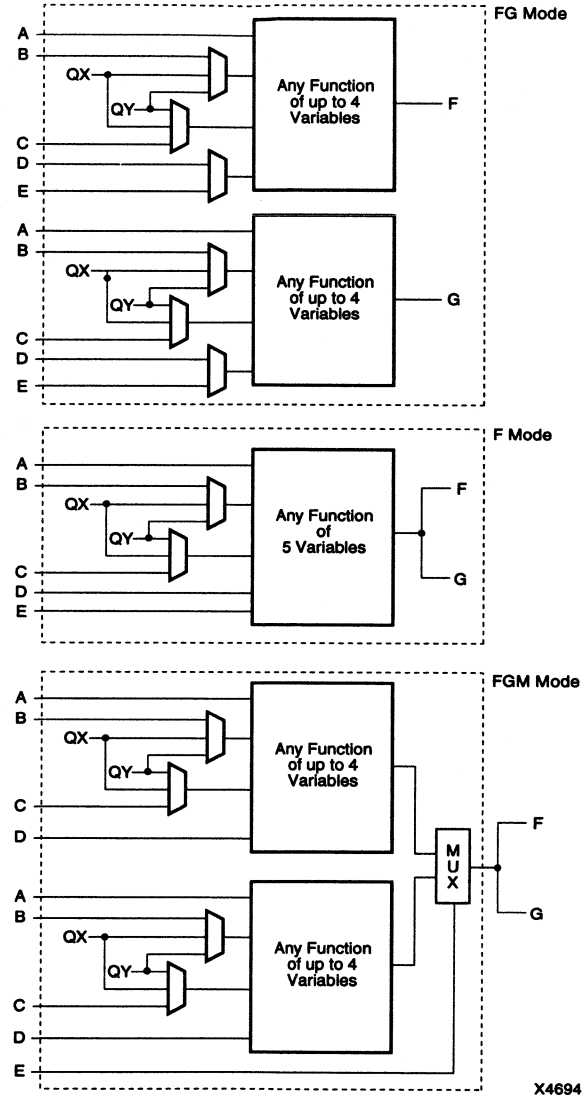


Figure 9-11 CLB Configurations

SwapBlk

The SwapBlk command interchanges the configurations and net connections of two blocks; for example, two blocks switch positions without affecting the functionality of the design. Because of certain design issues (delays through critical nets, routing resource usage), you might want to have some blocks in certain relative positions.

You can use the SwapBlk command to change the positioning of the blocks. To practice using SwapBlk, perform the following steps:

1. Type the following on the command line:

   `find da` ↵

2. Select **`Blk`** → **`SwapBlk`** from the EditLCA menu. The command line displays:

   ```
   Select block 1:
   ```

3. Click once on block **`DA`** with the left mouse button or enter **`da`** ↵ on the command line.

   The message line and command line display:

   ```
   Block DA: Selected
   ```

   ```
   Select block 2:
   ```

4. Click once on block **`DB`** with the left mouse button. The message line displays:

   ```
   Block DA and DB: Swapped.
   ```

The following figures illustrate the net assignments before and after using SwapBlk.

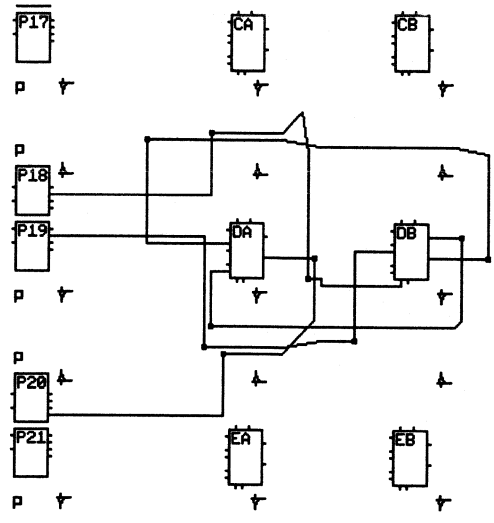


Figure 9-12 Net Assignments Before SwapBlk

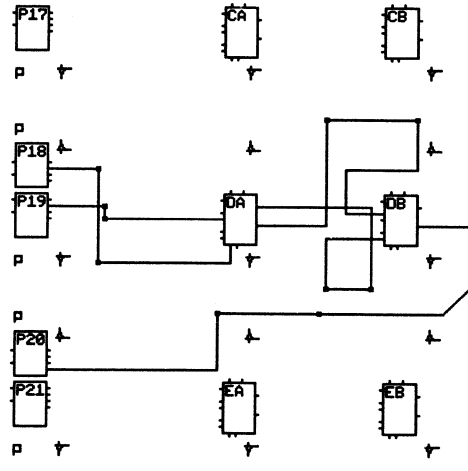


Figure 9-13 Net Assignments After SwapBlk

The following figures illustrate the CLB configuration before and after using SwapBlk.

```
X: Y:G F: G:B:E DX: DY: ENCLK: RSTDIR: CLK:
G = -~E+B
   A=                X=
   B=net4            Y=net5
   C=
   D=
   DI=
   E=net3
   EC=
   RD=
   K=

X: Y:G F:C:D G:D DX: DY: ENCLK: RSTDIR: CLK:
F = -~C+D
G = ~D
   A=                X=net3
   B=                Y=net4
   C=net2
   D=net6
   DI=
   E=
   EC=
   RD=
   K=
```

Figure 9-14 CLB Configuration Before SwapBlk

```
X: Y:G F:C:D G:D DX: DY: ENCLK: RSTDIR: CLK:
F = ~~C+D
G = ~D
   A=                X=net3
   B=                Y=net4
   C=net2
   D=net6
   DI=
   E=
   EC=
   RD=
   K=

X: Y:G F:C:D G:D DX: DY: ENCLK: RSTDIR: CLK:
F = ~~C+D
G = ~D
   A=                X=net3
   B=                Y=net4
   C=net2
   D=net6
   DI=
   E=
   EC=
   RD=
   K=
```

Figure 9-15 CLB Configuration After SwapBlk

Low-level Editing

Low-level editing is a very powerful feature of XDE. It gives you absolute control over net routing paths. This is very useful when trying to meet timing constraints, or when trying to route the last net in a packed FPGA. This type of editing requires detailed knowledge of both the FPGA architecture and the available routing resources. These resources consist of programmable interconnection points (PIPs) and switch matrices, which interact to make the connections between metal segments and block pins.

This section discusses the following topics:

- Screen Options
- Switch Matrices
- Programmable Interconnection Points
- Routing Resources

- Manual Routing
- Routing through a Switch Matrix
- Routing Nets Using a Longline

Figure 9-16 is an example of a routed net with the PIPs and matrices shown.

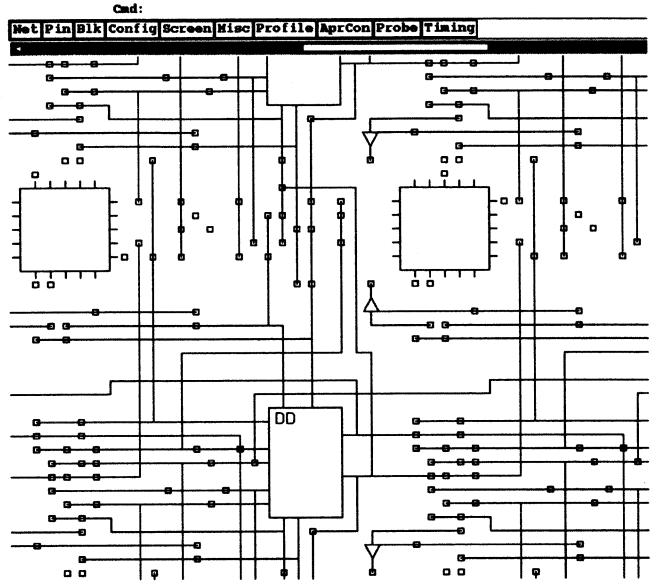


Figure 9-16 EditLCA Screen with PIPs Shown

To manually route nets at this level, you must be able to see the PIPs and switch matrices. The Show option on the Screen menu selects the different options for the screen display.

The first group of options determines what items are displayed. Refer to the following table.

Table 9-3 Show Options

Option	Description
Pips/NoPips	Turn on/off display of PIPs and switching matrices.
World/NoWorld/ NeverWorld	Turn on/off display of world view.
Used/Available	Show the interconnect segments that are used (connected to a programmed PIP or switching matrix pin) or show the interconnect that is not currently used.
Bidis/NoBidis	Turn on/off display of the position of the bidirectional buffers for the general purpose interconnect.
Matrix/NoMatrix	Turn on/off display of routing options through the switching matrices.

A second group of options determines the zoom level of the Editor Screen:

- XSmall/Small/Medium/Large/Xlarge
- In — Zoom in one level
- Out — Zoom out one level

Screen Options

Set the screen options as follows:

1. Select **Screen** → **Show** from the menu. The screen options pop-up window appears.
2. From the pop-up window, enable the following options:

   Pips
   Matrix
   In

3. Select **Done** to close the zoom level pop-up window. The message line displays:

```
Drawing screen...
```

When the redraw is complete, the screen shows programmable interconnection points (PIPs) and switch matrices in addition to the CLBs and IOBs.

4. Save this profile by typing the following on the command line:

   saveprofile ↵

Switch Matrices

Switch matrices join the ends of metal segments and allow interconnections between adjoining rows and columns of interconnect. You can establish the connections through the switch matrix by the automatic router, or by using the EditNet command to manually select the desired pairs of matrix pins.

Place the cursor on a pin of a switch matrix.

Since the Show Matrix option is enabled, XDE highlights all legitimate switching matrix combinations for that pin. Figure 9-17 shows the possible connections for all of the pins in a switch matrix.

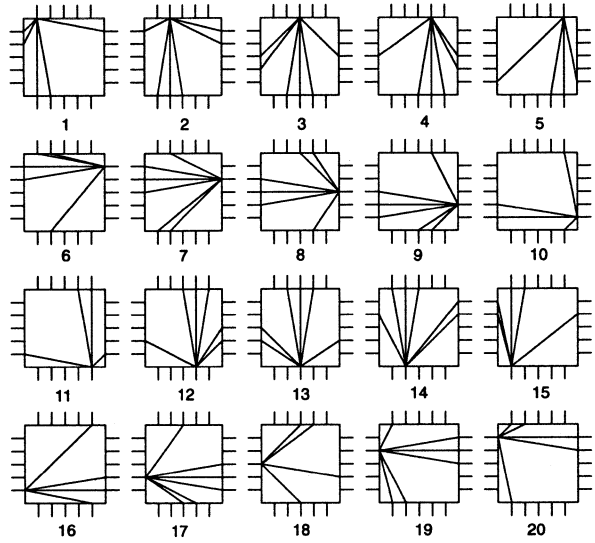


Figure 9-17 Possible Connections in a Switch Matrix

Programmable Interconnection Points

PIPs are the only places where distinct wire segments can connect. You can think of them as on/off switches that pass a signal only when enabled. However, some of these switches are directional — they conduct signals in only one direction.

Place the cursor on any PIP. The status line indicates the PIP's directionality, as follows:

Table 9-4 PIP Directionality

Direction	Description
ND	Nondirectional interconnection
D:H->V	PIP that drives from a horizontal to a vertical line
D:V->H	PIP that drives from a vertical to a horizontal line
D:C->T	PIP that drives from the horizontal segment of a T to the vertical segment
D:T->C	PIP that drives from the vertical segment of a T to the horizontal segment
D:CW	Corner PIP that drives in the clockwise direction
D:CCW	Corner PIP that drives in a counter-clockwise direction
P0	Non-conducting (OFF) PIP
P1	Conducting (ON) PIP

Since these definitions can be unclear for new users, XDE has a command to change the square PIPs into arrows that denote directionality.

On the command line, type:

```
show directional ↵
```

PIP arrows that point to the right are PIPs that drive horizontally (both right and left). PIP arrows that point up are PIPs that drive vertically (both up and down). PIPs that still appear as squares are non-directional PIPs that drive in any direction.

Routing Resources

For efficient routing at the lower level, you should have a good understanding of the different routing resources available, and how best to use them. The programmable interconnect consists of three distinct types:

- Direct interconnect
- General purpose interconnect
- Longlines

Direct Interconnect

Direct interconnect, shown in Figure 9-18, provides the most efficient way to connect nets that run between adjacent blocks. Signals routed using direct interconnects have very short delays and use no general interconnect resources. For each CLB, the .X output can be directly connected to the .B input of the CLB immediately to its right and to the .C input of the CLB directly to its left. The .Y output can use direct interconnect to drive the .D input of the block immediately above it and the .A input of the block below it.

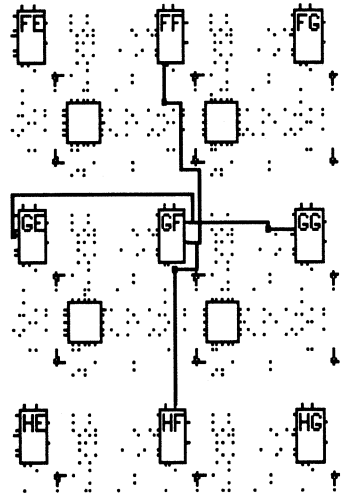


Figure 9-18 Direct Interconnect Routing Resources

The following exercise shows the benefits of using direct interconnect wherever possible.

1. At the command line, type the following:

   querynet ↵

   The system displays a pop-up window with options. The command line displays:

   ```
   Select net(s) or option(s):
   ```

2. At the command line, type the following:

   net3 ↵

   net4 ↵

   ↵

Note: The final carriage return indicates to the system that you are finished entering net names.

3. Make a note of the net delays, and then swap the outputs of block DA. At the command line, enter the following:

   swapsig

   The command line displays the following:

   ```
   From pin:
   ```

4. Enter the first pin on the command line:

   da.x ↵

   The command line displays the following:

   ```
   To pin:
   ```

5. Enter the next pin on the command line:

   da.y

6. Note that one of the nets has been routed using direct interconnect, while the other net has not.

7. Select **Net** → **QueryNet** from the menu.

8. Enter **da.x** on the command line.

9. Enter **da.y** on the command line.

10. Select **Done**. The system displays a pop-up window that lists the delays for the specified nets.

The delay of net4, which uses direct interconnect, is now 1.0 ns, while the delay for the other net is 2.5ns.

It is beneficial to place blocks and choose pin positions to take advantage of direct interconnect, since direct interconnect exhibits the shortest routing delays and uses no general purpose interconnect resources, thus reducing routing density and congestion.

General Purpose Interconnect

General purpose interconnect consists of a grid of horizontal and vertical wire segments that can be interconnected through the use of PIPs and switching matrices. In Figure 9-19, the horizontal segment is a piece of general purpose interconnect.

Special buffers within the general interconnect area provide signal restoration for the improved performance of lengthy nets. The interconnect buffers propagate signals in the required direction on a given interconnect segment. These bidirectional buffers (BIDIs) are found adjacent to the switching matrices and are highlighted in red by the use of the ShowBidis command, which you enabled at the beginning of this section.

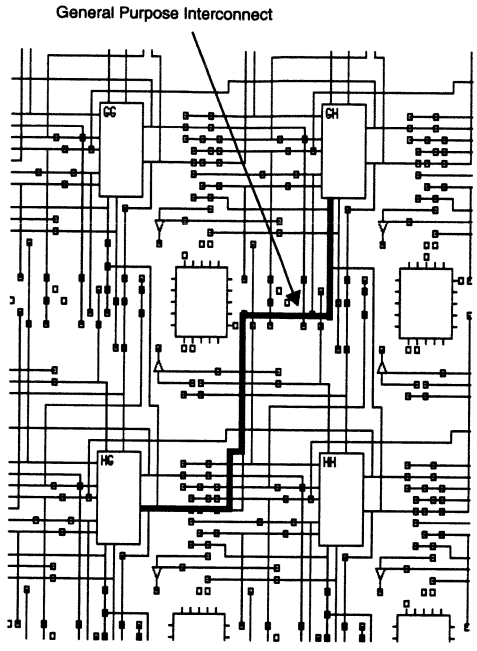


Figure 9-19 General Purpose Routing Resources

Longlines

Longlines avoid switch matrices and are intended primarily for signals that must travel a long distance, or must have minimum skew between multiple destinations. Longlines, shown in Figure 9-20, run vertically and horizontally the height or width of the interconnect area. In devices larger than the XC3020, two of the vertical longlines are connectable half-length lines.

The global clock buffer (GCLK) in the upper-left corner of the chip drives one reserved longline in each column, which connects to the .K inputs of the logic blocks. Using the global buffer for a clock signal provides a skew-free, high fan-out, synchronized clock available to all CLBs and IOBs. Direct access to this buffer is available at the second IO pad from the top on the left side of the chip. Remember to use GCLK in all of your designs; since certain longlines are dedicated to the global clock buffer, they are only used if GCLK is specified.

The alternate clock buffer (ACLK) is similar to GLCK, and is located in the lower-right corner of the FPGA.

There is no command in XDE to show the longlines, so they must be located indirectly. Vertical longlines can be found to the left of a CLB. The left-most and right-most lines run through the PIPs on the .K clock pin. Horizontal longlines are located above and below switch matrices. These longlines cross through the PIPs that connect to the outputs of the TBUFs. Refer to Figure 9-20 for the positioning of the longlines.

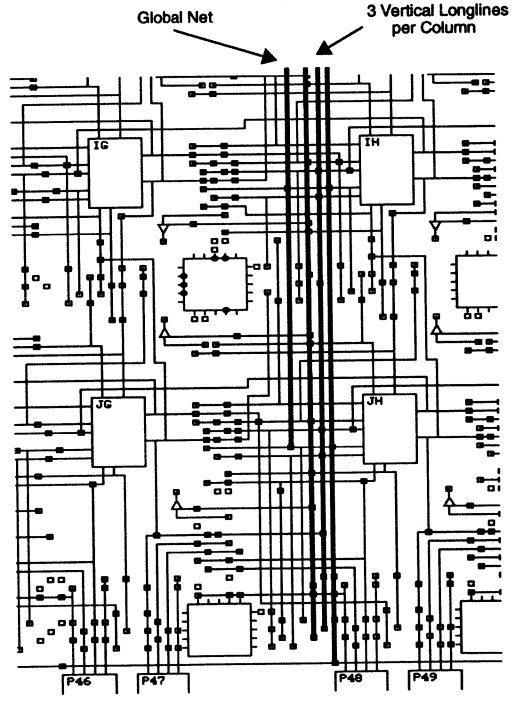


Figure 9-20 XC3000 Family Interconnect Resources

Manual Routing

Generally, when you modify a net, XDE automatically configures the corresponding interconnect. However, you can also route nets manually. The following subsections describe routing a net through a switch matrix:

- Disable Autoroute Option
- Route Nets Manually
- Connect Source and Load Pins
- Route through a Switch Matrix

Disable Autoroute Option

For the next set of examples, turn the autorouter off by performing the following steps:

1. Select **Profile** → **Autoroute** from the EditLCA menu. The system displays a pop-up window with the autoroute options.
2. Select **Off** with your left mouse button.

The XACT autoroute option is disabled. Now, when you create nets or move pins, you have to manually route the nets.

Route Nets Manually

To practice manually routing nets, perform the following tasks:

1. At the command line, type:

   find net8 ↵

   The cursor moves to the source pin of the net.

2. Select **Net** → **Hilight** from the EditLCA menu. The system displays a pop-up window with color options.
3. Select **Magenta** from the pop-up window. The command line displays:

   ```
   Select net(s):
   ```

4. Enter **net8** ↵
5. Select **Done**.

The system highlights the interconnect attached to the pins in magenta.

6. Select **Net → EditNet** from the EditLCA menu. The command line displays:

   ```
   Select net:
   ```

7. Enter **net8** ↵. The command line displays:

   ```
   Select PIPs:
   ```

Connect Source and Load Pins

Now use the mouse to select PIPs so that the routing connects the source and the load pins.

1. Place the cursor on the second PIP to the right of pin FA.X and click the left mouse button, which turns the PIP on.
2. Move the cursor down in a straight line until it is on top of the PIP that is on the other highlighted wire. Click on this PIP. The net is now routed.
3. Select **Done** from the pull-down menu, or click the middle mouse button to end the EditNet command.

If you now place the cursor on the destination pin of the net (pin FB.D), the message line displays the delay from the source to that pin. If you place your cursor on the FB.D while in EditNet, the net delay is shown as a question mark since delays are not calculated until EditNet has ended.

Route Through a Switch Matrix

In the next example, the source and load pins cannot be directly connected. Therefore, you must route the net through a switch matrix as follows:

1. At the command line, enter:

   find net10 ↵

2. Select **Net → Hilight** from the EditLCA menu. The system displays a pop-up window with color options.

3. Select **Magenta** from the pop-up window. The command line displays:

   ```
   Select net(s):
   ```

4. Enter **net10** ↵ ↵.

   The second carriage return sends you back to the main menu. The system highlights the interconnect attached to the pins in magenta.

5. Select **Net** → **EditNet** from the EditLCA menu. The command line displays:

   ```
   Select net:
   ```

6. Enter **net10** ↵. The command line displays:

   ```
   Select PIPs:
   ```

   The best way to route this net is through the switch matrix above row F and between columns C and D.

7. Click on the first PIP, as illustrated in Figure 9-21.

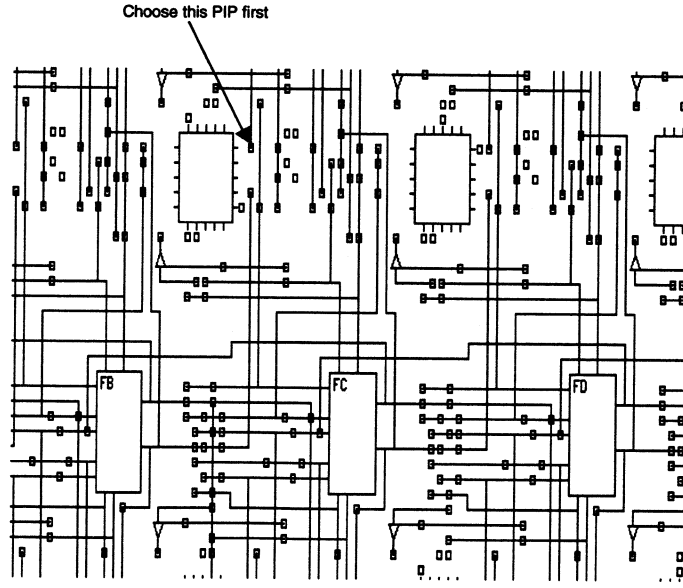


Figure 9-21 Manual Routing — Step 1

8. Click on the first pin, as illustrated in Figure 9-22.

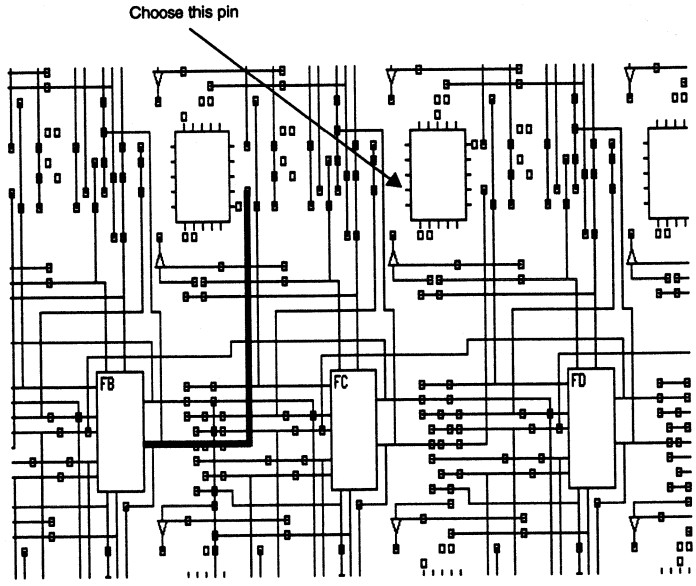


Figure 9-22 Manual Routing — Step 2

9. Check the cursor status line to make sure you are on the correct pin and then click the left mouse button. The command line shows:

```
Select other pin:
```

10. Select switch matrix pin 9 next, as illustrated in Figure 9-23.

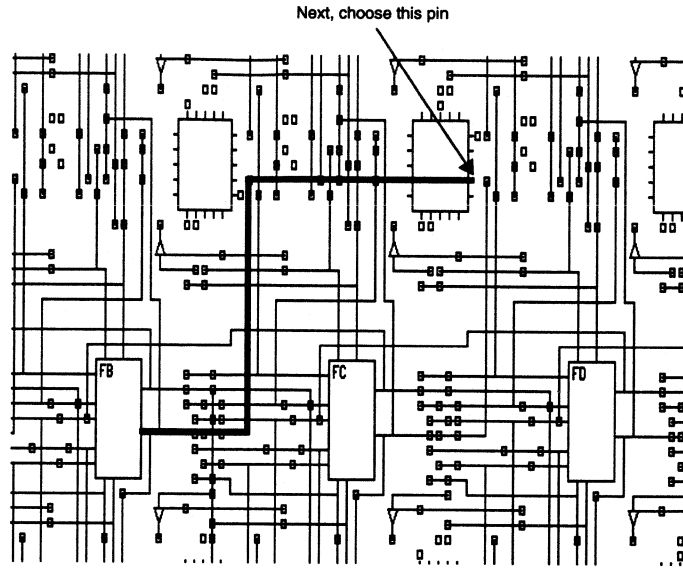


Figure 9-23 Manual Routing — Step 3

11. Finally, click on the PIP on the highlighted wire directly to the right of switch matrix pin 9, as indicated by Figure 9-24.

    The net is now routed. Figure 9-24 shows the complete routing path for net10.

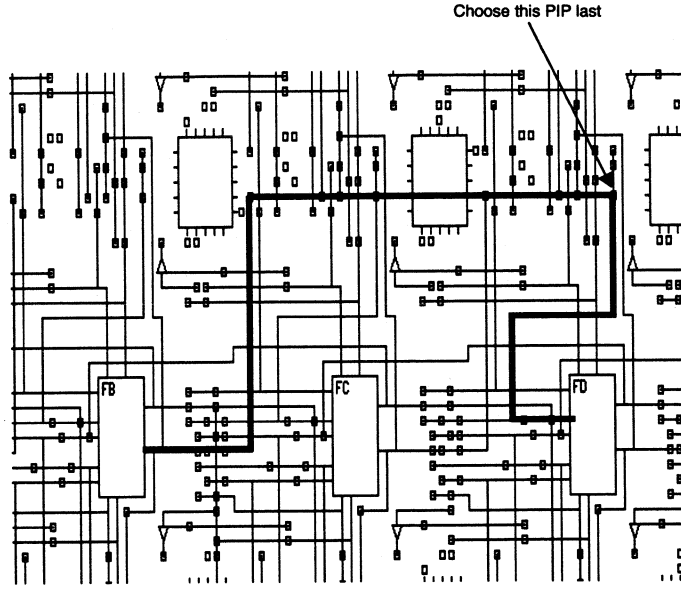


Figure 9-24 Manual Routing — Step 4

Routing Through a Switch Matrix

When routing through a switch matrix, you might need to use an output pin that is not a legitimate connection of the input. In this case, you can use a bank-shot to connect the pins through an intermediate pin that is a valid connection of both the input and output, as illustrated in Figure 9-25.

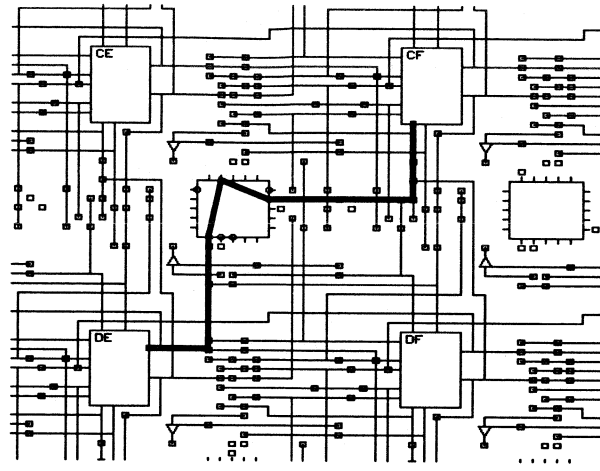


Figure 9-25 Switch Matrix Bank-shot

In the following example, the only way to connect the routed segments is by using a bank-shot, since there is no way to directly connect the routed segments of the net.

1. Redraw the screen to remove other highlighting. Select **Screen** → **Redraw** from the main menu.
2. Type the following on the command line:

   find af ↵
3. Highlight the text to be edited.

   hilight ↵

   The system displays a pop-up window with color options.
4. Select **Magenta**. The status line displays:

   `Select net(s):`
5. Enter the following:

   net13 ↵
6. Select **Done** to return to the main menu.

7. Select **Net** → **EditNet** from the main menu. The status line displays the following:

   ```
   Select net:
   ```

8. Enter **net13** ↵.
9. In switch matrix BH.20, click on pins 19, 14, 14, and 2, in that order. The status line displays the current switch box and pin numbers.
10. Choose **Done** to end the EditNet command. The net is now connected through a bank-shot.

Routing Nets Using a Longline

The following example shows how to route nets using a longline. To use longlines efficiently, arrange the CLBs in a column, so that only one longline is needed. Use the SwapSig command to arrange the load pins so that the same pin is used on all the CLBs. In this way, you can directly connect the inputs to the longline, and no general interconnect is necessary. The .B pin is the best choice to which to move the load pins, because the majority of the inputs already come in on this pin.

The following sections walk you through each procedure in detail:

- Swap the Blocks
- Swap the Pins
- Route the Net
- Perform a QueryNet

Swap the Blocks

To swap the blocks, follow these steps:

1. Type the following on the command line:

   find net15 ↵

2. Rearrange the blocks so that they line up in one column. Select **Blk** → **SwapBlk** from the main menu. The command line displays:

   ```
   Select block 1:
   ```

3. Enter **bi** ↵. The command line displays:

   ```
   Select block 2:
   ```

4. Enter **bj** ↵.

Swap the Pins

1. Move to block CJ.
2. Swap net15 from the C pin to the B pin to use a single longline. Type the following on the command line:

   swapsig cj.c cj.b ↵

3. Move net15 to the B pin for blocks DJ and JJ. Type the following on the command line:

   swapsig dj.c dj.b ↵

   swapsig jj.a jj.b ↵

4. Now unroute the net so it can be rerouted using a longline by entering this on the command line:

   unroute net15 ↵

5. Highlight the net so that you can easily find the pins.

   hilight magenta net15 ↵

Route the Net

To route the net using a longline, perform these steps:

1. Use EditNet to route the net as follows:

   editnet net15 ↵

   The command line displays the following:

   ```
   Select PIP(s):
   ```

2. Go to I/O pin P76. Click on the first PIP below P76.I, which is the highlighted pin.
3. Move to the left and click on the second PIP to the right of the PIP on the highlighted wire. This PIP drives the longline.
4. Moving downward, choose every PIP you cross that is on a highlighted wire segment. Stop when you reach block FJ.

5. Click on the longline PIP for FJ.B. The status line displays the following message:

   ```
   net net15: No sources reachable.
   ```

   This warning message means that the PIP you have just turned on is not connected to the source pin. As mentioned in the beginning of this section, longlines are only half the length of the chip. This halfway point is between rows E and F. To connect the longline halves, you must turn on the splitter PIP.

   Between CLBs EJ and FJ, and level with the top of the switch matrix, are a pair of adjacent PIPs with no wire segments attached to them. These are the splitter PIPs for the B and C longlines.

6. Click on the left one. The two halves of the longline are now connected.
7. Continue moving down, connecting the rest of the unrouted load pins.

Perform a QueryNet

Perform a Querynet to make sure that you have not missed any load pins.

1. On the command line, enter:

   querynet net15 ↵

   The system displays a pop-up window that lists all load pins. Three asterisks (***) indicates a non-connected load pin.

2. Go back to the schematic and connect the missing pin.

Commands that Perform Similar Functions

This section explains the differences between commands that perform similar functions. Complete the exercises in the following sections to illustrate the differences.

Unroute and UnroutePin

To remove all the routing from a net, use Unroute. To remove the routing from selected pins, use UnroutePin. Perform the following steps:

1. Enter:

   find net15 ↵

2. Change the color of the net:

   colornet ↵

3. Select the color from the pop-up window or enter it directly on the command line:

   magenta ↵

   The system displays the following:

   ```
   Select net(s):
   ```

4. Enter the specific net, in this case:

   net15 ↵

5. Try the UnroutePin command, as follows:

   unroutepin ↵

   BJ.B CJ.B ↵

6. Try the unroute command as follows:

   unroute net15 ↵

Route and Routepin

Route configures the interconnection for an entire net. RoutePin connects only the selected pin to the net's source. Perform the following steps:

1. Enter:

   find net26 ↵

2. Change the color of the net:

   hilight↵

3. Select the color from the pop-up window or enter it directly on the command line:

   magenta ↵

The system displays the following:

```
Select net(s):
```

4. Enter the specific net, in this case:

   net26 ↵

5. To route a single load pin, use the RoutePin command as follows:

   routepin cf.a ↵

Note: Performing a Routepin on the source pin routes all the load pins, which is the equivalent to specifying "route net26."

6. To route all load pins by specifying the source pin, enter the following:

   routepin cd.x

7. To route all load pins, enter:

   route net26 ↵

8. To unroute all load pins from the net, enter:

   unroute net26 ↵

Route, EditNet, and RoutePoint

The differences between these three commands are as follows:

- The Route command uses the autorouter to pick the entire routing path. You have no control over the path taken.
- The EditNet, command requires that you chose the entire path manually. You must turn on every PIP and choose the pins on every switch matrix.
- RoutePoint allows you to choose intermediate points, which the autorouter routes between. This command offers the speed of the Route command combined with the control of EditNet.

Hilight and ColorNet

Hilight colors highlights not only the routed interconnect, but also unused stubs that connect to routed lines. ColorNet changes only the part of the interconnect that contains routing.

Perform the following steps:

1. Type the following on the command line:

   hilight magenta net6 ↵

   colornet magenta net2 ↵

   Hilight draws over a net (slow and temporary), and ColorNet assigns a new color to the net (fast and permanent). If a net is unrouted, Hilight colors the pin stubs; however, ColorNet appears to do nothing since it colors only routed interconnect. However, when the net does get routed, it has the color assigned to it by the ColorNet statement.

2. Enter:

   unroute net2 net6 ↵

3. Use Hilight to color the routed interconnect as follows:

   hilight magenta net6 ↵

4. Use ColorNet to change the interconnect that contains routing:

   colornet brightblue net2 ↵

5. Route the net:

   route net2 ↵

Building a Four-Bit Multiplier

The first part of this tutorial introduced the commands needed for using the design editor. This section walks you through an actual design, as follows:

- Creating a new design
- Creating the mult0, mult1, and mult2 CLBs
- Configuring IOBs as input and output pins
- Giving blocks logical names
- Adding nets to the design
- Checking the nets in your design
- Routing the nets

- Using the Delay command
- Improving the routing

Figure 9-26 shows the design flow.

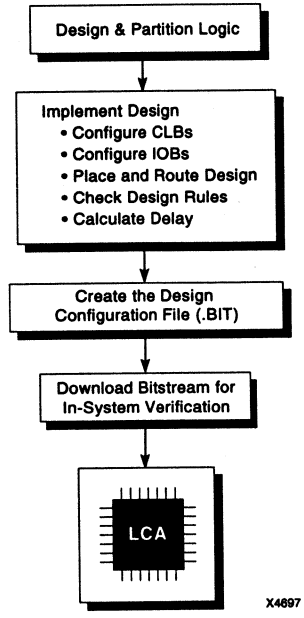


Figure 9-26 FPGA Design Flow Using XDE

The design to be implemented is a 4 x 4 bit combinatorial multiplier. The block diagram in Figure 9-27 shows the placement for the CLBs, their block names, and which of three possible ways each CLB should be configured. The inputs are labeled with X and Y; the outputs have a Z prefix.

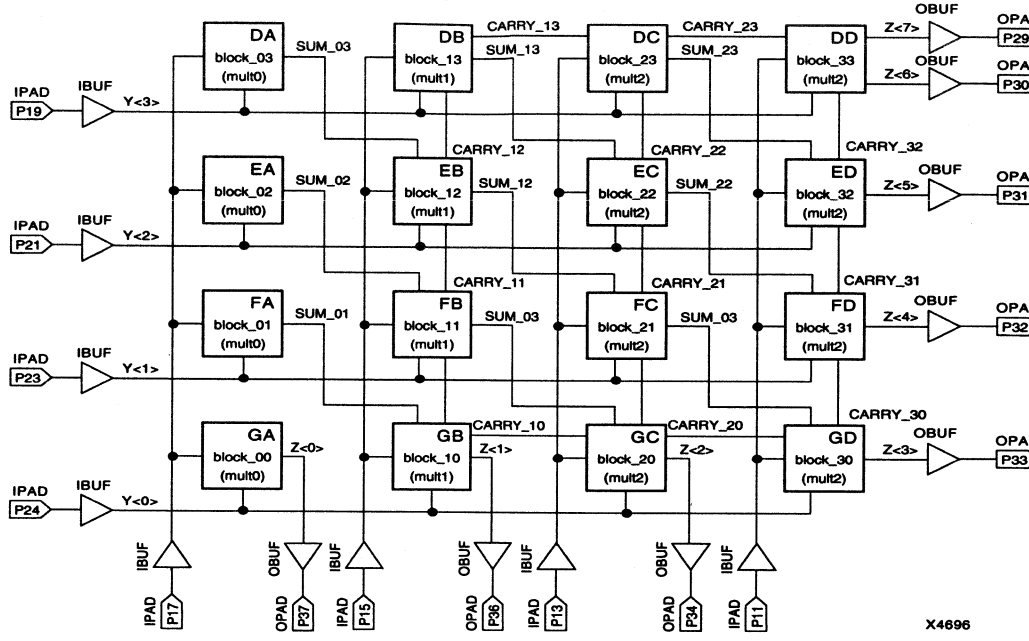


Figure 9-27 Combinatorial Multiplier Block Diagram/EditLCA Layout

A combinatorial multiplier uses one CLB per partial product. A 2-input AND gate generates each partial product, but additional circuitry is required to add together all partial products of equal weight.

Figure 9-28 gives the schematics for the three different types of CLBs that make up the multiplier that arbitrarily have been named mult0, mult1, and mult2. The mult2 CLB generates the partial products, adds two input values to it, and generates two outputs — Sum and Carry. Sum has the same binary weight as the partial product; Carry has the next higher weight. The blocks on the left and top edge have fewer inputs and are simpler, but perform the same fundamental function.

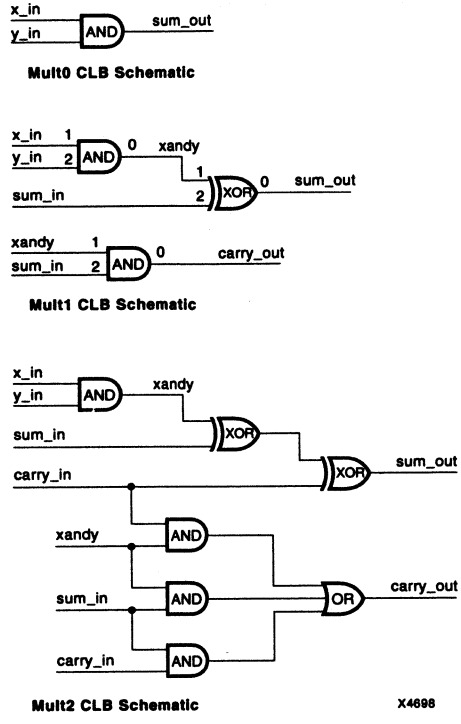


Figure 9-28 Schematics for Multiplier Cells

The following subsections describe step-by-step how to create the four-bit multiplier.

Creating a New Design

To exit EditLCA and prepare to edit the new design, perform the following steps:

1. Enter:

   `exit` ↵

   Exit sends you back to the main XDE screen.

2. Select **`Designs`** → **`Part`** from the main menu.

3. Change the part type to a 3020pc68.

4. Edit a new design called mult4.

   editlca new mult4 ↵

   The system displays a pop-up window asking you if you want to save or discard the changes to the design you are currently editing.

5. Select **Yes** to discard current changes.

   The system redraws the screen. Next, build the four CLBs.

Creating the mult0 CLB

Create the first CLB called mult0:

1. Enter the following:

   editblk ga ↵

   Use Figure 9-29 as a guide to configure block GA.

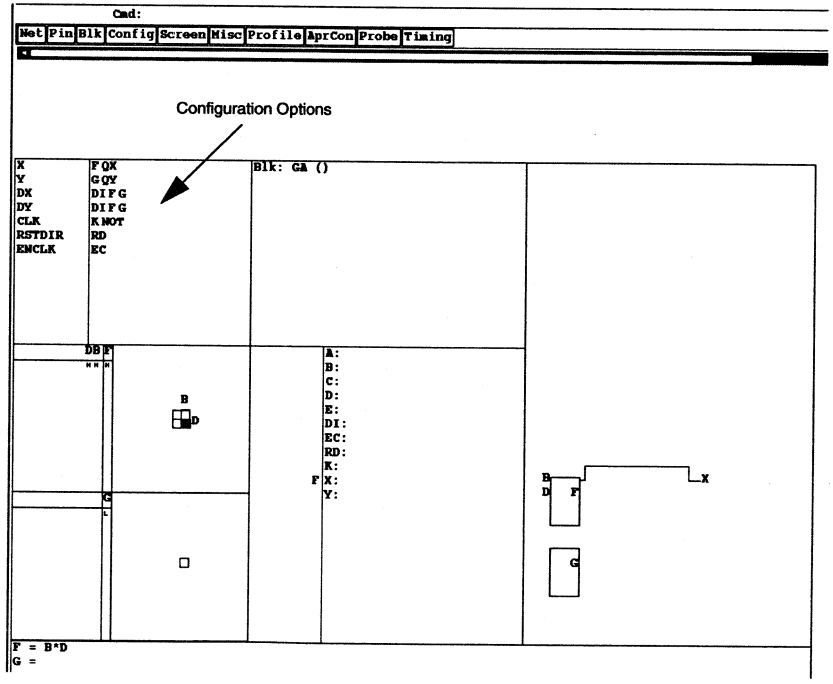


Figure 9-29 CLB Configured as mult0

2. Enter the following equation on the command line:

   f=b*d ↵

3. Select **F** from the Configuration Options area for X.

   Block GA is now configured. The F function generator ANDs the signals that come in on pins B and D.

4. Return to the editor screen from the Editblk screen:

   switch

   According to Figure 9-27, this block configuration is used in four CLBs. Rather than re-entering this information in each location, copy the configured CLB to the other blocks as follows:

5. Select **Blk** → **CopyBlk** from the EditLCA menu. The status line displays the following:

   `From block:`

6. Enter the block from which you want to copy its configuration, in this case, GA.

   ga ↵

   The status line displays the following:

   `To block(s):`

7. Enter the following blocks on the command line:

   fa ↵

   ea ↵

   da ↵

8. Select **Done** off the CopyBlk menu or click your middle mouse button.

Creating the mult1 CLB

Configure the next CLB, called mult1, as follows:

1. Enter:

   editblk gb ↵

2. Add the following equations on the command line:

   ```
   f=c@b*d
   g=c*b*d
   ```

3. Click on **F** for X and **G** for Y from the Configuration Options area. Make sure block GB resembles Figure 9-30.

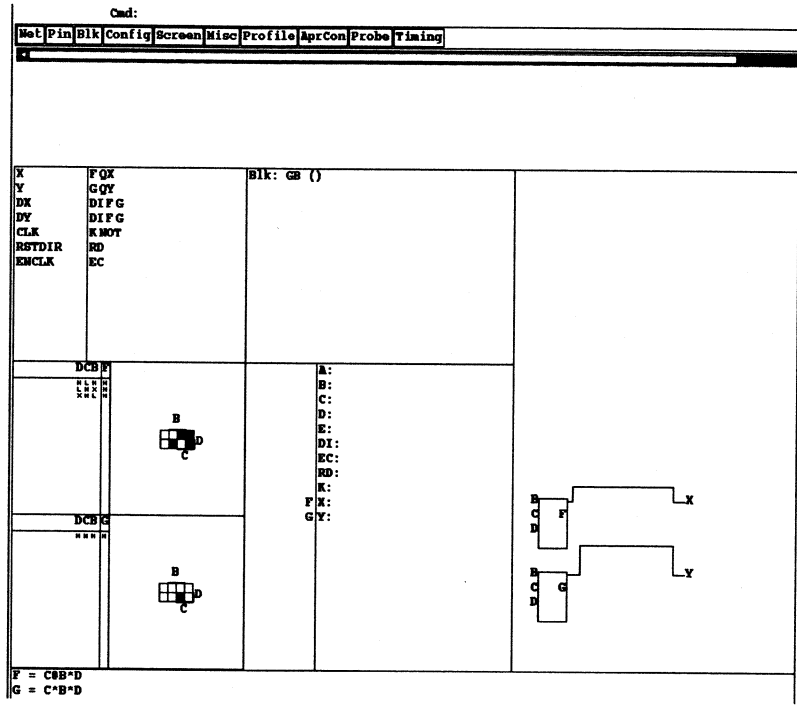


Figure 9-30 CLB Configured as mult1

4. Now copy the mult1 block to the other locations where it is used. Switch back to the EditLCA screen:

   ```
   switch
   ```

5. Enter:

   ```
   copyblk ↵
   ```

6. Enter the block from which you want to copy its configuration, in this case, GB.

   gb ↵

   The status line displays the following:

   `To block(s):`

7. Enter the following blocks on the command line:

   fb ↵

   eb ↵

   db ↵

8. Select **Done** or click your middle mouse button.

Creating the mult2 CLB

Create the next CLB, called mult2, as follows:

1. Enter:

   editblk gc ↵

2. Enter:

   ```
   f=(a@(c@(b*d)))
   g=((a*c)+(c*(b*d))+((b*d)*a))
   ```

3. Select **F** for X and **G** for Y from the Configuration Options area. Make sure block GC resembles Figure 9-31.

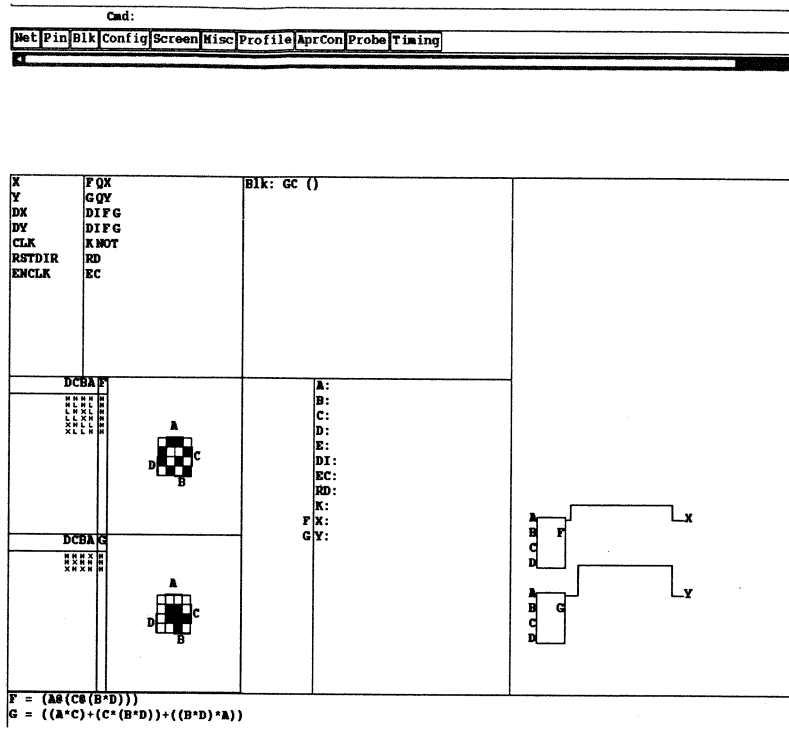


Figure 9-31 CLB Configured as mult2

4. Switch back to the EditLCA screen.
5. Copy GC to the other locations where it is used. Type the following on the command line:

   copyblk gc fc ec gd fd ed dc dd ↵

Configuring an IOB as an Input Pin

Configure an IOB as an input pin:

1. Enter:

   editblk p24 ↵

2. Configure block P24 as an input pin. Click on the **LATCH** and **PULLUP** tags to turn them off.

3. Click on the **I** tag to turn it on. This is the configuration for an input pin.
4. Make sure that block P24 resembles Figure 9-32 as follows:

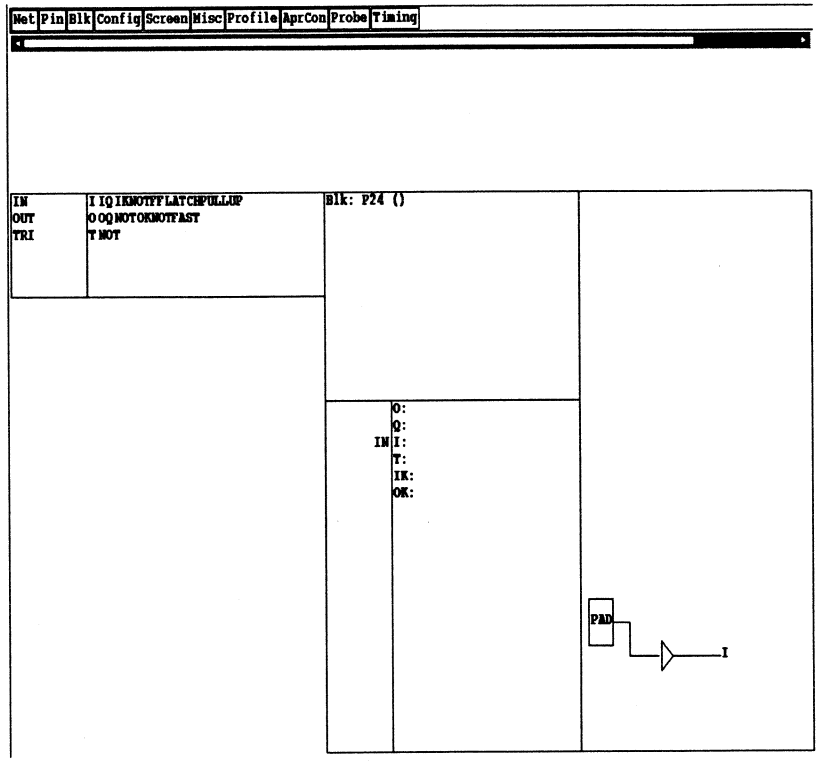


Figure 9-32 IOB Configured as an Input Pin

5. Copy the input pin to the other locations where it is used. Enter:

```
copyblk p24 p23 p21 p19 p17 p15 p13 p11 ↵
```

Configuring an IOB as an Output Pin

Configure an IOB as an output pin:

1. Enter:

```
editblk p37 ↵
```

2. Configure block P37 as an output pin.
3. Click on the **I**, **LATCH**, and **PULLUP** tags to turn them off.
4. Click on the **O** tag to turn it on. This is the configuration for an output pin.
5. Make sure your EditBlk screen resembles Figure 9-33 as follows:

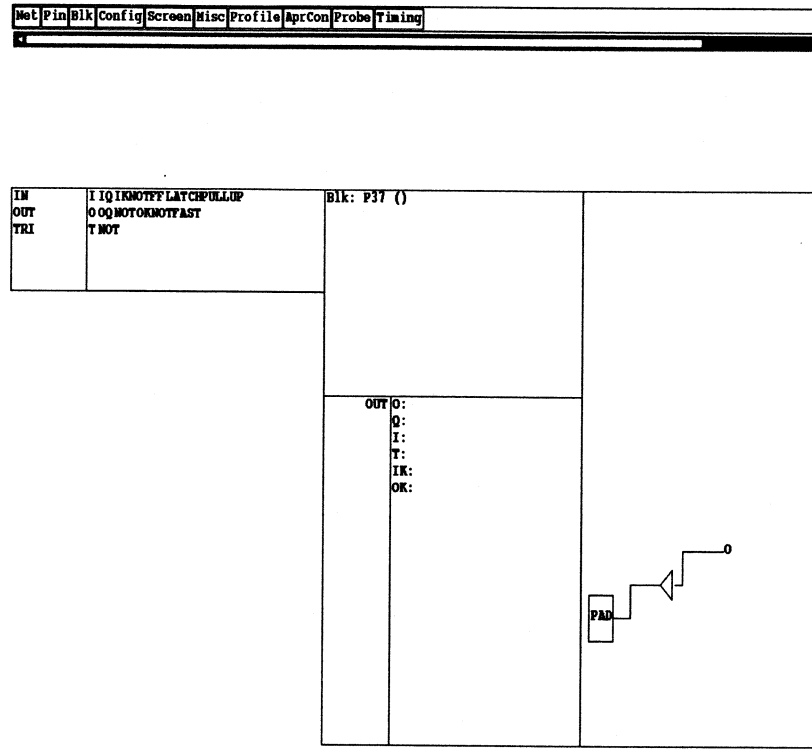


Figure 9-33 IOB Configured as an Output Pin

6. Copy this pin to the other locations where it is used. Enter:

   copyblk p37 p36 p34 p33 p32 p31 p30 p29 ↵

7. Save your changes. Enter:

   save ↵

   You should periodically save the design since XACT has no Undo command. The message line prompts for the design name.

8. Enter ↵ to indicate the default name, in this case, edit_lca.lca.

Giving Blocks Logical Names

To make the design easier to understand, give all the blocks logical names using the NameBlk command.

1. Enter:

   nameblk p24 yo_in ↵

   This command changes the name of block P24 to Y0_IN. You can enter either name when using commands that expect a location as input, such as Editblk or Find.

   You can enter the names in either of two ways:

   - Manually, as described above
   - Automatically, by creating a batch file that contains a set of NameBlk statements

2. Rename the rest of the I/O blocks and CLBs according to Figure 9-34.

IOB	NAME	IOB	NAME
P23	Y1_IN	GA	BLK_00
P21	Y2_IN	FA	BLK_01
P19	Y3_IN	EA	BLK_02
P17	X0_IN	DA	BLK_03
P15	X1_IN	GB	BLK_10
P13	X2_IN	FB	BLK_11
P11	X3_IN	EB	BLK_12
P37	Z7_OUT	DB	BLK_13
P36	Z6_OUT	GC	BLK_20
P34	Z5_OUT	FC	BLK_21
P33	Z4_OUT	EC	BLK_22
P32	Z3_OUT	DC	BLK_23
P31	Z2_OUT	GD	BLK_30
P30	Z1_OUT	FD	BLK_31
P29	Z0_OUT	ED	BLK_32
		DD	BLK_33

Figure 9-34 IOB and CLB Block Names

3. Type the following at the command line to create a file that automatically names the IOBs and CLBs:

   exec names ↵

   The system automatically renames all IOBs and CLBs.

   You can put any list of XDE commands into a file and execute them with the Exec command. In cases where you perform the same sequence of commands repeatedly or perform them on different designs, this method is faster than entering commands manually.

Adding Nets to the Design

1. Make sure that the autorouter is off. Select **Profile** → **Autoroute** from the menu. A pop-up window appears.
2. Select **Off** from the pop-up window.
3. Switch back to the EditLCA screen if you are still in EditBlk mode.
4. Select **Net** → **AddNet** from the menu.

   XACT prompts for the name of the new net.
5. Type the following on the command line:

   x<0> ↵
6. Move the mouse to each of the following pin locations and click on them with the left mouse button, which adds the pin to the net.

   P17.I
   GA.D
   FA.D
   EA.D
   DA.D
7. Choose **Done** to finish adding pins to the net.
8. Add net X<1> by typing:

   addnet x<1> p15.i gb.d fb.d eb.d db.d ↵

The rest of the nets can be added by typing the lines as they appear in the figure, using the mouse to select the individual pins, or using a supplied command file called "nets."

9. To use the command file, type:

   exec nets ↵

   Figure 9-35 shows all the pin connections for the multiplier.

NET NAME	SOURCE PIN	LOAD PIN(S)			
X<0>	P17.I	GA.D	FA.D	EA.D	DA.D
X<1>	15.I	GB.D	FB.D	EB.D	DB.D
X<2>	13.I	GC.D	FC.D	EC.D	DC.D
X<3>	11.I	GD.D	FD.D	ED.D	DD.D
Y<0>	24.I	GA.B	GB.B	GC.B	GD.B
Y<1>	23.I	FA.B	FB.B	FC.B	FD.B
Y<2>	21.I	EA.B	EB.B	EC.B	ED.B
Y<3>	19.I	DA.B	DB.B	DC.B	DD.B
Z<0>	GA.X	29.O			
Z<1>	GB.X	30.O			
Z<2>	GC.X	31.O			
Z<3>	GD.X	32.O			
Z<4>	FD.X	33.O			
Z<5>	ED.X	34.O			
Z<6>	DD.X	36.O			
Z<7>	DD.Y	37.O			
SUM_01	FA.X	GB.C			
SUM_02	EA.X	FB.C			
SUM_03	DA.X	EB.C			
SUM_11	FB.X	GC.C			
SUM_12	EB.X	FC.C			
SUM_13	DB.X	EC.C			
SUM_21	FC.X	GD.C			
SUM_22	EC.X	FD.C			
SUM_23	DC.X	ED.C			
CARRY_10	G.Y	GC.A			
CARRY_11	F.Y	FC.A			
CARRY_12	E.Y	DB.C			
CARRY_13	D.Y	DC.C			
CARRY_20	G.Y	GD.A			
CARRY_21	F.Y	EC.A			
CARRY_22	E.Y	DC.A			
CARRY_23	D.Y	DD.C			
CARRY_30	G.Y	FD.A			
CARRY_31	F.Y	ED.A			
CARRY_32	E.Y	DD.A			

Figure 9-35 Net Connection Information

Checking the Nets in Your Design

1. To obtain a list of all the nets in the design, select **Net** → **QueryNet** from the menu. The system displays a pop-up window with QueryNet options.
2. Select **-All**.
3. Select **Done**.

The system displays the results of your QueryNet. The QueryNet listing should resemble the one illustrated in the following figure:

```
- CARRY_10        U  GB.Y (BLK_10)       ***   GC.A (BLK_20)
- CARRY_11        U  FB.Y (BLK_11)       ***   FC.A (BLK_21)
- CARRY_12        U  EB.Y (BLK_12)       ***   DB.C (BLK_13)
- CARRY_13        U  DB.Y (BLK_13)       ***   DC.C (BLK_23)
- CARRY_20        U  GC.Y (BLK_20)       ***   GD.A (BLK_30)
- CARRY_21        U  FC.Y (BLK_21)       ***   EC.A (BLK_22)
- CARRY_22        U  EC.Y (BLK_22)       ***   DC.A (BLK_23)
- CARRY_23        U  DC.Y (BLK_23)       ***   DD.C (BLK_33)
- CARRY_30        U  GD.Y (BLK_30)       ***   FD.A (BLK_31)
- CARRY_31        U  FD.Y (BLK_31)       ***   ED.A (BLK_32)
- CARRY_32        U  ED.Y (BLK_32)       ***   DD.A (BLK_33)
- SUM_01          U  FA.X (BLK_01)       ***   GB.C (BLK_10)
- SUM_02          U  EA.X (BLK_02)       ***   FB.C (BLK_11)
- SUM_03          U  DA.X (BLK_03)       ***   EB.C (BLK_12)
- SUM_11          U  FB.X (BLK_11)       ***   GC.C (BLK_20)
- SUM_12          U  EB.X (BLK_12)       ***   FC.C (BLK_21)
- SUM_13          U  DB.X (BLK_13)       ***   EC.C (BLK_22)
- SUM_21          U  FC.X (BLK_21)       ***   GD.C (BLK_30)
- SUM_22          U  EC.X (BLK_22)       ***   FD.C (BLK_31)
- SUM_23          U  DC.X (BLK_23)       ***   ED.C (BLK_32)
- X<0>            U P17.I (X0_IN)        ***   GA.D (BLK_00)
                                         ***   FA.D (BLK_01)
                                         ***   EA.D (BLK_02)
                                         ***   DA.D (BLK_03)
- X<1>            U P15.I (X1_IN)        ***   GB.D (BLK_10)
                                         ***   FB.D (BLK_11)
                                         ***   EB.D (BLK_12)
                                         ***   DB.D (BLK_13)
- X<2>            U P13.I (X2_IN)        ***   GC.D (BLK_20)
                                         ***   FC.D (BLK_21)
                                         ***   EC.D (BLK_22)
                                         ***   DC.D (BLK_23)
- X<3>            U P11.I (X3_IN)        ***   GD.D (BLK_30)
                                         ***   FD.D (BLK_31)
                                         ***   ED.D (BLK_32)
                                         ***   DD.D (BLK_33)
```

```
- Y<0>          U P24.I (Y0_IN)        ***   GA.B (BLK_00)
                                       ***   GB.B (BLK_10)
                                       ***   GC.B (BLK_20)
                                       ***   GD.B (BLK_30)
- Y<1>          U P23.I (Y1_IN)        ***   FA.B (BLK_01)
                                       ***   FB.B (BLK_11)
                                       ***   FC.B (BLK_21)
                                       ***   FD.B (BLK_31)
- Y<2>          U P21.I (Y2_IN)        ***   EA.B (BLK_02)
                                       ***   EB.B (BLK_12)
                                       ***   EC.B (BLK_22)
                                       ***   ED.B (BLK_32)
- Y<3>          U P19.I (Y3_IN)        ***   DA.B (BLK_03)
                                       ***   DB.B (BLK_13)
                                       ***   DC.B (BLK_23)
                                       ***   DD.B (BLK_33)
- Z<0>          U  GA.X (BLK_00)       ***   P29.O(Z0_OUT)
- Z<1>          U  GB.X (BLK_10)       ***   P30.O(Z1_OUT)
- Z<2>          U  GC.X (BLK_20)       ***   P31.O(Z2_OUT)
- Z<3>          U  GD.X (BLK_30)       ***   P32.O(Z3_OUT)
- Z<4>          U  FD.X (BLK_31)       ***   P33.O(Z4_OUT)
- Z<5>          U  ED.X (BLK_32)       ***   P34.O(Z5_OUT)
- Z<6>          U  DD.X (BLK_33)       ***   P36.O(Z6_OUT)
- Z<7>          U  DD.Y (BLK_33)       ***   P37.O(Z7_OUT)
```

Figure 9-36 QueryNet Output

Compare your listing to Figure 9-36. If a mistake has been made or if the Design Rules Checker program (DRC) issues a warning about a net or pin, use the DelNet command to delete the offending net and redo that net.

Routing the Nets

The design is now functionally correct. The only thing left to do is route the nets.

1. Type the following on the command line:

   `route *` ↵

   XDE uses the autorouter to route all the nets.

2. Select **`Misc`** → **`Drc`** from the menu to run DRC.

   This program checks the routing and the block configuration for any possible mistakes. The system displays a pop-up window with DRC options.

3. Select **Done**.

   DRC checks to make sure that no fatal mistakes have been made in the design. The system displays a pop-up window with informational message, warnings, and fatal errors. There should be no errors or warnings. If any occur, go back and fix them. Alternately, you can reload the last-saved version of the design and start again at that point.

Using the XDelay Command

Now use the XDelay command to find the worst-case path delay through the multiplier.

1. Select **Timing** → **XDelay** from the menu. The system displays a pop-up window with XDelay options.
2. Select **Analyze** from the pop-up window. This options shows an overview of the worst-case timing delays.

Improving the Routing

Improve the routing of the four-bit multiplier by using the AlignSig command to swap multiple pins at once, as follows:

1. Unroute the design by typing:

   unroute * ↵

   A dialogue box asks:

   ```
   Unroute 36 nets. Are you sure?
   ```

2. Select **Yes**.
3. Select **Pin** → **AlignSig** from the EditLCA menu.
4. Select the first net. At the command line, enter:

   x<0> ↵

   The command line displays:

   ```
   Select pin:
   ```

5. Enter the following:

   c ↵

The command line displays:

```
Select block(s):
```

6. Enter the following:

   ***a** ↵

   The system displays a pop-up windowing verifying that you want to perform this function.

7. Select **Yes**.

   XDE puts net X<0> on the C pin of all the CLBs in column A. Now a longline can drive these CLBs.

8. Enter the following on the command line to swap the pins for the specified nets:

   alignsig x<1> c *b ↵

   alignsig x<2> c *c ↵

   alignsig x<3> c *d ↵

   alignsig y<0> a g* ↵

   alignsig y<1> a f* ↵

   alignsig y<2> a e* ↵

   alignsig y<3> a d* ↵

9. Now route X<0:3> and Y<0:3> onto the longlines. You do not have to route the nets completely, just turn on the PIPs that connect the loads to the longlines.

   The system displays the following the warning message:

   ```
   No sources reachable.
   ```

   Ignore it or turn off DRC by typing:

   autodrc off ↵

10. Now refer back to Figure 9-18 and find where you can use direct connections to improve the routing. Net CARRY_10 can use the X-to-B direct connect, and CARRY_22 can use the Y-to-D direct connect.

11. Find the other nets that can use direct connects and use SwapSig to put the net on the correct pins. If a direct connect can be used but is not, routing resources are wasted.
12. Now route the design by typing:

    route * ↵

    Use the delay calculator to again find the worst-case delay through the part.
13. Select **Timing** → **XDelay** from the menu. The system displays a pop-up window with XDelay options.
14. Select **Analyze** from the pop-up window. This options shows an overview of the worst-case timing delays.

    The worst-case delay time is lower now since the design is less congested and uses the longlines more efficiently.

Downloading a Bitstream

Now that you entered the design, it can be downloaded to the demo board. Perform the following steps:

1. Save the design using the default name.

   save ↵

   ↵
2. Go back to the Executive Screen.

   exit ↵
3. Select **Programs** → **MakeBits** from the XDE Executive menu:
4. Select **Config** → **Makebits** from the MakeBits menu.
5. Select **Done**.

   This translates the design into a bitstream that can be loaded into the part.
6. Select **Config** → **Writebits**.
7. Enter ↵ to indicate the default file name.
8. Make sure that the download cable is attached to both the demo board and parallel port.

On the demo board, turn switches 2, 3, and 4 to the off position so that the FPGA configures in slave mode. Now turn the demo board on.

9. Next, initialize your parallel port.

Note: If you are working on a Sun Workstation, you can skip this step.

For PC users, select **Misc** → **Port** from the MakeBits menu. The system displays a pop-up window with port options.

10. Select **LPT1**.
11. Now download the bitstream. Select **Download** → **DownLoad** from the MakeBits menu.
12. Press the reset button on the demo board and then press ↵. The following message appears on the screen:

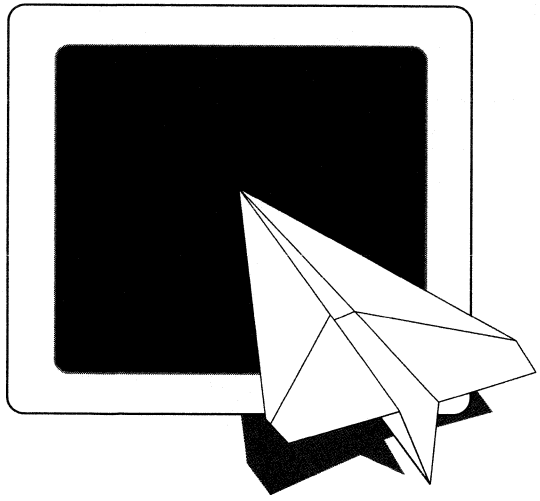
```
'done' signal now high
```

13. Enter **quit** on the command line to exit the MakeBits screen.

The demo board is now configured as a four-bit multiplier. Switches one through four control X<3:0>. Switches five through eight control Y<3:0>. The output appears on the LEDs.

Quitting XDE

Enter `quit` or `exit` on the XDE command line to exit XDE. A pop-up window appears verifying you want to quit XDE and save any changes.



Index

XACT User Guide

XACT User Guide — April, 1994 (0401128 01)

Printed in U.S.A.

Index

T

U

V

X